UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

DISEÑO E IMPLEMENTACIÓN DE UN COMPILADOR PARA UN SUBCONJUNTO DE PYTHON

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A:

ANGEL FRANCISCO ZÚÑIGA CHÁVEZ

DIRECTOR DE TESIS:
DRA. ELISA VISO GUROVICH

2013

Datos del Jurado

1. Datos del alumno
   Zúñiga
   Chávez
   Angel Francisco
   55 91 39 89 02
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   301019400

2. Datos del tutor
   Dra
   Elisa
   Viso
   Gurovich

3. Datos del sinodal 1
   Dra
   Amparo
   López
   Gaona

4. Datos del sinodal 2
   Dr
   José de Jesús
   Galaviz
   Casas

5. Datos del sinodal 3
   Dra
   Sofía Natalia
   Galicia
   Haro

6. Datos del sinodal 4
   L en C.C
   Francisco Lorenzo
   Solsona
   Cruz

7. Datos del trabajo escrito
   Diseño e implementación de un compilador para un subconjunto de Python
   316 p
   2013

Agradecimientos:

En primer lugar quiero expresar mi más sincero agradecimiento a cada uno de mis sinodales, los cuales, además de formar parte de mi jurado, también fueron mis maestros durante el transcurso de mi carrera. Agradezco al L. en C.C. Francisco Solsona por el interesante y divertido curso de sistemas operativos que tuve el gusto de tomar con él, además de su apoyo y consejos para concluir este trabajo; a la Dra. Sofía Natalia Galicia por su generosidad, paciencia y confianza que tuvo al revisar este trabajo y por su buena atención cuando al tomar su curso de inteligencia artificial la visitaba en su cubículo para mostrarle mis prácticas en prolog; al Dr. José Galaviz le agradezco el interés que despertó en mí (tal vez sin darse cuenta) al tomar su curso de arquitectura de computadoras (fue desde entonces que me dediqué a investigar cómo programar un ensamblador), además de sus graciosas bromas que lo caracterizan al dar clase; a la Dra. Amparo López tengo mucho que agradecerle, en primer lugar por el curso de estructuras de datos, el cual tuve el placer de tomar con ella, un curso que desde el primer día que tuve el gusto de conocerla como maestra al frente de un pizarrón atrapó inmediatamente mi atención y me sentí identificado con su persona; por otra parte le agradezco por la persona que es más allá de las aulas, por el calor humano que la caracteriza, por los buenos consejos y charlas que tuvimos y, finalmente, por la revisión y los comentarios que me hizo acerca de este trabajo; he dejado en último lugar a la Dra. Elisa Viso por ser una persona tan importante para mí en mi formación académica. En primer lugar le agradezco que haya aceptado ser mi directora de tesis, por compartir conmigo su enorme cantidad de conocimientos, por los excelentes cursos de teoría de la computación y de compiladores que tomé con ella, que sembraron en mí, una pasión por el tema de compiladores que desde entonces ha crecido día con día, siempre con la fortuna de contar con su excelente guía. Elisa es un ejemplo de constancia, esfuerzo, trabajo y excelencia, es quizás esta última cualidad con la que me siento más identificado con ella ya que es algo que mi persona siempre está en busca de alcanzar; le estoy muy agradecido también por sus rigurosas revisiones durante el desarrollo de este trabajo, por su inagotable paciencia, por su gran respeto por mis ideas aún cuando éstas no eran siempre buenas, por sus correcciones y comentarios siempre atinados respecto a este trabajo, por brindarme su amistad sincera y confiar en mí, por todas las largas pláticas que sostuvimos en las cuales aprendí mucho de ella, por su inconmensurable apoyo, por todo esto gracias Elisa.

Por otra parte deseo agradecer profundamente al Dr. Sergio Rajsbaum por enseñarme a ver la computación teórica desde un enfoque diferente e intrigante, por su grandioso curso de complejidad computacional, por enseñar siempre con rigor haciendo enfásis en los puntos críticos, finos e importantes del tema, enfoque que es el que considero disfrutar más y que es el que más me gusta seguir. Por hacerme ver la importancia de la

computación teórica y originar así en mí, una búsqueda (casi insaciable) de conocimientos, en los libros y artículos de investigación de computación.

Quiero agradecer también a mi alma máter, la Universidad Nacional Autónoma de México, por brindarme todo lo necesario para que fuera posible el desarrollo integral de mi persona, me siento muy orgulloso de formar parte de ella, primero como deportista defendiendo sus colores como parte del equipo de fútbol americano (etapa en la cual juntos ganamos 3 campeonatos); después como estudiante y finalmente como académico; me siento pues orgulloso de ser universitario; una tradición que comenzó en mi familia con el Dr. Higinio Zúñiga y que después continuó con mi abuelo el Dr. Luis Zúñiga, mi padre el Lic. Luis Zúñiga y ahora mi hermano el Lic. Luis Zúñiga, a cada uno de los cuales les estoy profundamente agradecido, por haber forjado en mí (de manera directa o indirecta) a través de sus conocimientos y ejemplo la persona que soy. Agradezco también a mi tío el Dr. Darío Nuñez Zúñiga, investigador del Instituto de Ciencias Nucleares de nuestra Universidad, por su gran solidaridad y apoyo en los momentos difíciles, por estar siempre al tanto de los avances de mi tesis y por sus excelentes consejos para llegar a la conclusión de ésta; aprovecho la oportunidad también para felicitarle por su próxima promoción al nivel III del Sistema Nacional de Investigadores; a mi tía la M. en C. Mariana Nuñez Zúñiga por contarme la historia acerca de la familia, por su solidaridad, empatía y palabras de aliento.

A mis amigos y compañeros de carrera a los cuales les estoy profundamente agradecido por todo su apoyo, por su gran ayuda, por escucharme, tomar en cuenta mi opinión y confiar en mí en los momentos críticos. Juan Germán, con el que siempre estuve compartiendo mi pasión por los conocimientos profundos y, desde mi punto de vista, interesantes de la computación; Alejandro Cruz, el cual, además de ser mi amigo, lo considero una excelente persona que siempre va en busca del bien para los demás; David Rodriguez por compartir conmigo mi enorme pasión por la filosofía y la psicología, y por haberme orillado a estudiar el interesante mundo de la filosofía analítica. A ellos los considero mis mejores amigos.

A la familia de Germán, a su esposa Donají y en especial a los pequeños Ian y Xoan por jugar conmigo y llenar de alegría mi ser al ver en ellos un futuro prometedor para el mundo.

A mi amiga Circe por haberme mencionado en su tesis de maestría y por esos momentos tan emocionantes que compartimos durante el transcurso de mi carrera; a Casandra por haber propiciado en mí un cambio tan positivo en mi estado de ánimo, cuando durante nuestros estudios siempre que teníamos oportunidad de hacer algo juntos eran momentos de diversión y profunda alegría. A mi amiga Karla por escucharme y estar siempre ahí en los momentos más difíciles.

Al Act. Jaime Vázquez por brindarme su amistad y buenos deseos. A la Dra. Lucy Gasca y al Mat. Adrián Girard por ser grandes amigos y por brindarme siempre su apoyo y estar a mi lado.

A mi amigo Alfonso Zamora, por enseñarme con su gran ejemplo a ser humilde, por siempre buscar el bien común y en particular por siempre estar al tanto de mí y de este trabajo; por intentar siempre hacer de mí una mejor persona y ver en mí un hombre con muchas cualidades y, finalmente pero quizás lo más importante, por brindarme su ayuda incondicional en las situaciones y momentos más críticos de mi vida.

A mis compañeros y amigos del xfc por su compresión durante la conclusión del presente trabajo. A la Mat. Martha Rico Diener por su compresión, gran apoyo y buenos consejos en los momentos difíciles de mi vida.

A mis alumnos del curso de compiladores del semestre 2011-2 por mostrar interés en el curso y por permitirme guiarlos en el tema de compiladores.

A mi abuela la maestra Amparo Gómez, por su ejemplo de dedicación, constancia y compromiso con la educación; a mi tía Maritza Chávez por su apoyo incondicional hasta el último en todos los sentidos de mi vida.

A mi madre la profesora Alma Lilia Chávez, mujer fuerte y capaz que nunca ha permitido que vacile en mi caminar por la vida y quien siempre ha esperado y exigido lo mejor de mí.

A mi padre por haber sido y ser el mejor padre que un hijo pueda tener.

Al creador de la vida.

Índice

Prólogo

Este trabajo corresponde a los aspectos de implementación que se deben desarrollar en un curso introductorio de Compiladores. En este contexto, desarrollamos un compilador para un subconjunto del lenguaje Python y pretendemos que el diseño e implementación de este compilador transmita a los alumnos los aspectos importantes que se deben tomar en cuenta hoy en día para la producción de un compilador "real". Esto último involucra muchísimas horas hombre en equipos grandes de programadores con un alto nivel educativo, algo imposible de lograr en un primer curso de compiladores.

Dado que el curso de compiladores es, en general, un curso avanzado, damos por hecho que los estudiantes conocen el material de un curso de autómatas y lenguajes formales, indispensable para las primeras dos etapas del compilador, el análisis léxico y sintáctico, por lo que se insistirá sólo en aquellos aspectos que son particulares del tema de compiladores. También suponemos que el nivel de programación es avanzado y que el estudiante debe ser capaz de migrar a nuevos lenguajes de programación, lo que facilitamos con un apéndice sobre cómo hacerlo de Java a C++, dado que es común que Java sea el lenguaje más familiar para muchos estudiantes.

Las primeras etapas de un compilador han sido muy estudiadas y automatizadas, por lo que consideramos primordial en este material introducir herramientas que se usan para este efecto, pero con un enfoque que penetra en el cómo y por qué de las herramientas utilizadas, no nada más en el para qué. También utilizaremos herramientas menos conocidas en la etapa de generación de código, siguiendo el mismo enfoque que acabamos de mencionar. De esta manera liberamos tiempo del curso para aspectos donde las herramientas son menos versátiles o inexistentes, como el análisis semántico, el alojamiento en registro y la planificación de instrucciones.

El tono del trabajo es en todo momento didáctico, pero haciendo énfasis en las decisiones prácticas que se toman en la elaboración de un compilador, exponiendo de ma-

nera amplia alternativas novedosas y las ventajas y desventajas de éstas, y justificando las elecciones particulares que se toman.

En el capítulo 1 justificamos la elección del lenguaje anfitrión, C++, y el lenguaje fuente a compilar, Python, aunque dependiendo de la etapa de compilación con la que estemos trabajando, restringiremos al subconjunto de Python que consideraremos para nuestro desarrollo.

En el capítulo 2 revisamos la etapa del análisis léxico y presentamos la herramienta flex, cómo trabaja, por qué lo hace así y cómo usarla en el contexto de nuestro compilador. Por supuesto que no se muestra el trabajo completo pues, como ya mencionamos, nuestro objetivo es que los estudiantes, a partir de los ejemplos dados, terminen la definición del analizador léxico del lenguaje. Sin embargo, como parte de este trabajo se desarrolló el analizador léxico para todo el lenguaje Python.

En el capítulo 3 revisamos la etapa del análisis sintáctico, integrando nuevamente herramientas para su automatización, con el mismo paradigma que en el análisis léxico. También en este caso, aunque no aparezca en el texto, el analizador sintáctico contempla a todo el lenguaje Python.

En el capítulo 4 ilustramos la etapa del análisis semántico y mencionamos la importancia del sistema de tipos y las distintas formas de vigilarlo. Introducimos diversas variaciones sobre el tema de código intermedio. Se justifica la selección del código de tres direcciones para poder implementar la forma SSA, que facilita la inferencia de tipos. Finalmente se realizan las tareas tradicionales del análisis semántico, restringiendo la implementación del analizador semántico exclusivamente a funciones (aunque esta no se encuentra concluida).

En el capítulo 5 enfrentamos la etapa de generación de código, revisando para ello diversos sistemas en tiempo de ejecución. Para la selección de instrucciones utilizamos la herramienta burg con el mismo enfoque que se dio para las otras herramientas ya presentadas. Adicionalmente describimos un algoritmo de alojamiento en registros con algunas de sus variantes. Aunque no hay implementación de esta etapa en este trabajo, dejándosela a los estudiantes.

En el Apéndice A, como ya mencionamos, aparece una hoja de ruta para migrar del lenguaje Java al lenguaje C++. En el apéndice B aparece la gramática completa de Python usada en este material.

Por supuesto que éste no es un trabajo que pretende cubrir todos los enfoques existentes en esta materia, ni siquiera todo el material disponible para este enfoque particular, sino simplemente proveer a los estudiantes con un proyecto realizable en un semestre y que corresponda, en cierta manera, a un producto completo, aunque limitado, de un compilador real.

Prefacio

El presente trabajo es de carácter introductorio y tiene como objetivo principal ser un texto que dote al lector con los conocimientos tanto teóricos como prácticos para que sea potencialmente capaz de desarrollar un compilador de un lenguaje de alto nivel como C, C++ o Java, además de servir como punto de partida para que el lector pueda convertirse (si así lo desea) en un futuro investigador en el área de compiladores. Adicionalmente presenta un panorama de los temas necesarios en los que se debe profundizar en el caso que se desee desarrollar un compilador para un lenguaje funcional como Haskell o ML.

Con este fin, la estrategia que se sigue es presentar de forma lineal cada una de las etapas que conforman el proceso de compilación, mostrando para cada una de éstas, en primer lugar los conocimientos teóricos necesarios e indispensables para comprender el objetivo de la etapa y cómo implementar la estrategia o diferentes estrategias que se pueden seguir para implementarla y, en segundo lugar (inmediatamente después), se presenta el diseño y los pasos necesarios para desarrollar una implementación concreta de dicha etapa. Con lo anterior se busca, por una parte, que el lector ponga inmediatamente en práctica los conocimientos adquiridos, para de esta forma reforzarlos y, por otra, que vislumbre cómo es que la parte práctica se beneficia enormemente de los resultados teóricos y cómo es que ésta motiva el desarrollo de nuevos resultados teóricos.

Teniendo lo anterior presente se decidió realizar el diseño de un compilador para un subconjunto de Python, para que de esta forma el lector contara con un diseño fijo de un compilador que pudiese implementar y, al mismo tiempo, pudiese experimentar con él. El diseño de dicho compilador es modular; cada uno de los módulos corresponde a una de las etapas que se presentan en este texto y que forman parte del proceso del compilación. Adicionalmente se realizó la implementación de (algunas de) las etapas del compilador con el objetivo que cuando la implementación de alguna de las etapas realizada por el lector no fuese correcta o se hubiese decidido por algún motivo no rea-

lizarla, pudiese utilizar nuestra implementación y de esta forma no estar impedido de continuar con el estudio e implementación de algún otra etapa.

La elección de Python como lenguaje fuente obedece fuertemente a que este es un lenguaje que cuenta tanto con características imperativas como funcionales, cuenta con tipado dinámico y a que es un lenguaje que se utiliza ampliamente en la vida real. Aunque este texto se centra principalmente en estudiar el proceso de compilación para las características imperativas de un lenguaje de programación, también menciona los puntos en los que se debe seguir otro camino para realizar la compilación de las características funcionales de un lenguaje de programación.

Cabe resaltar que los conocimientos teóricos que se presentan son abstractos y por tanto se pueden utilizar para realizar la implementación de un compilador de diferentes de lenguajes de alto nivel. Por otra parte, el diseño del compilador con el que lector trabajará es claramente concreto y se eligió así para que el lector pueda utilizar y observar de manera concreta los conocimientos adquiridos.

El diseño y objetivo del presente trabajo hace que sea un texto ideal para que se utilice en un curso de compiladores y ha sido escrito teniendo presente a los estudiantes.

Por último, este trabajo tiene como objetivo mostrar el por qué los traductores de lenguajes con tipado dinámico (como Scheme) que se implementan tradicionalmente son intérpretes y no compiladores. Se muestra que existen compiladores para este tipo de lenguajes y las diferentes estrategias que éstos utilizan y en este contexto se resalta la importancia del sistema de tipos con el que cuenta el lenguaje fuente y los diferentes avances que se han realizado en este tema.

CAPÍTULO 1

Introducción

1.1. Antecedentes

Desde la aparición de las primeras herramientas de cómputo utilizadas por la humanidad, se ha tenido la inquietud de poder automatizar el proceso de realizar los cálculos deseados para un determinado fin, sin jugar la persona otro rol más que el de "decirle" a la herramienta en cuestión cómo debe "actuar" o qué es lo que debe "hacer" con la información que se le presenta; más precisamente, lo anterior se plantea de la siguiente manera: dado un conjunto de datos de entrada, la máquina (herramienta) debe generar un conjunto de datos de salida. Aun cuando esto, en principio, se puede ver como una transformación meramente sintáctica de datos de entrada a datos de salida, puede tener semánticas diferentes; lo que es usual en computación es dar un conjunto de datos de entrada que representan un ejemplar de un problema específico para que dicha herramienta, con base en reglas bien definidas establecidas previamente, realice un proceso que genere datos de salida que representan la resolución de dicho problema para ese ejemplar.

Un problema típico que ha preocupado desde que se generalizó el uso de computadoras es el proceso automático de lenguajes y, en particular, lenguajes de programación o formales.

Veamos ahora cómo se dio el surgimiento de las primeras computadoras digitales y la manera en que fue evolucionando la forma en que se programaban. La primera computadora digital de propósito general se construyó en la Universidad de Pennsylvania en 1946 y se le nombró *ENIAC*; para programarla era necesario desconectar y conectar sus cables de forma adecuada para que realizara lo que el programador esperaba;

ésta, desde luego, era una labor lenta, tediosa y en donde era muy fácil cometer errores. Cabe señalar que estaba construida mediante bulbos y realizaba aritmética decimal. El 23 de diciembre de 1947 se desarrolló en los laboratorios *Bell* el primer transistor, lo que dio origen a que más tarde, en 1958, Jack Bilby de *Texas Instruments* realizara el primer circuito integrado. El matemático húngaro nacionalizado estadounidense John Von Neuman observó el problema que se presentaba cuando se requería programar la ENIAC y desarrolló lo que hoy se conoce como el modelo de Von Neuman (y que se utiliza ampliamente en las computadoras actuales) en el que se guardan tanto las instrucciones (el programa) como los datos en la memoria de la computadora para su posterior ejecución. La primer computadora en poner en práctica este modelo fue la *EDVAC* (*Electronic Discrete Variable Computer*), también construida en la Universidad de Pennsylvania; cabe notar que ésta realizaba aritmética binaria y no decimal como su antecesora. Ya en los años cincuentas con la llegada de nuevas computadoras digitales como la *UNIVAC*, la forma común de programarlas era escribir código binario, en el que se codificaba tanto las instrucciones a realizar, como los datos sobre los cuales se llevaban a cabo tales operaciones; a este lenguaje binario con el que trabajan las computadoras digitales binarias se le conoce como lenguaje de máquina (pues es el lenguaje con el que operan las máquinas). Si bien se había dado un paso importante en la forma en que se programaban las computadoras (el paso desde recableado a escribir código binario), ésta aún era bastante precaria y susceptible a que se dieran errores de programación; fue entonces que se dieron los primeros indicios de poder establecer un lenguaje más "humano" para poder proporcionar las instrucciones que debía ejecutar la computadora: surgió el lenguaje ensamblador, que consistía en asignar un código mnemónico a cada una de las instrucciones que podía realizar la computadora; esto hacía más fácil programar pues es más sencillo recordar ese código que una cadena binaria. De esta manera el programador escribía códigos mnemónicos y un programa (al que se le llamó ensamblador) se encargaba de realizar la traducción de los códigos mnemónicos al lenguaje de máquina.

Aunque el uso del lenguaje ensamblador marcó un avance en la manera de programar, los problemas de aquél eran casi los mismos que para el lenguaje de máquina, pues había una correspondencia casi[1] uno a uno entre ambos.

La forma natural de formular problemas y de pensar los algoritmos que nos llevan a lo solución de aquéllos aún distaba en gran medida de cómo se debían programar en el lenguaje ensamblador, pues los algoritmos se presentan en su manera natural como una serie de órdenes con un nivel de abstracción mayor. Es a partir de esta idea que surgen los llamados lenguajes de alto nivel, haciendo referencia a un alto nivel de abstracción y que por tanto resultan más cómodos para el programador, es decir, le es más cómodo expresar el programa en este tipo de lenguajes. Los lenguajes de alto nivel resultaron ser de gran ayuda para el programador y un salto importante en la forma de programar

[1] Algunos ensambladores permitían el uso de macros.

computadoras pero, sin duda al mismo tiempo, resultaron ser un reto importante que impulsó el desarrollo de ciertas áreas en la computación, pues al escribir un programa en un lenguaje con enunciados de alto nivel, al igual que sucedía con el lenguaje ensamblador, se requería un programa que llevase a cabo una traducción de esos enunciados (programas escritos en un lenguaje de alto nivel) al lenguaje de máquina; a este traductor se le dio el nombre de *compilador* , pues el proceso de traducción se veía en ese entonces como la *compilación* de una secuencia de subprogramas escritos en lenguaje de máquina seleccionados de una biblioteca.[2] En 1957 Grace Hopper diseñó el primer lenguaje de programación de alto nivel de nombre *FLOW-MATIC*, un antecesor de *COBOL*, aunque en el mismo año un equipo dirigido por John Backus en la empresa *IBM* desarrollaron *FORTRAN* (*Formula Translator*), un lenguaje de alto nivel enfocado a la realización de cálculos matemáticos; generalmente FORTRAN se conoce como el primer lenguaje de programación de alto nivel debido a que fue el primero que se utilizó ampliamente.

Sin lugar a dudas el desarrollar un compilador resultó ser un reto difícil y en algún momento considerado imposible; el primer compilador de FORTRAN, por ejemplo, tomó 18 años en ser implementado y, por supuesto, surgieron varias cuestiones al desarrollar los primeros compiladores; por ejemplo, es fácil notar que para que se lleve a cabo la traducción primero se debe reconocer que la estructura de los enunciados del programa sea correcta y esto es en esencia reconocer un lenguaje; surgieron preguntas como el tipo de lenguajes que puede reconocer una computadora, lo que propició el estudio, clasificación y formalización de lenguajes, un tema fundamental dentro de las ciencias de la computación y en particular en el desarrollo de compiladores, donde destacan las contribuciones que realizó Noam Chomsky.

1.1.1. Desarrollo de lenguajes de programación y arquitecturas de computadoras

Miles de lenguajes de programación han aparecido desde entonces. Los lenguajes de programación ofrecen abstracciones, principios de organización y estructuras de control a los programadores para que escriban código legible. La historia de los lenguajes de programación modernos inicia con la aparición de FORTRAN y a partir de ese momento comenzó el surgimiento de nuevos lenguajes de programación que aportaban nuevos conceptos y que dieron lugar a varios paradigmas de programación. Algunos de los lenguajes más destacados e influyentes por su aportación de nuevos conceptos son:

1. FORTRAN. El primer lenguaje de alto nivel cuyo principal objetivo era facilitar el cálculo de operaciones matemáticas complejas.

2. LISP. El segundo lenguaje de programación más antiguo, fue el primer lenguaje

[2] En ese tiempo el proceso de compilación se conocía como programación automática.

funcional, tenía soporte para funciones recursivas; además introdujo las funciones de orden superior y fue el primero en incorporar un recolector de basura.

3. COBOL. Originalmente desarrollado por el departamento de defensa de los Estados Unidos a final de los años cincuentas y principios de los sesentas por un equipo a cargo de Grace Murray Hopper. Su utilizaba principalmente en los negocios y el procesamiento de datos. Introdujo el concepto de estructuras y facilitó el mecanismo de entrada/salida.

4. Algol. Está basado en una estructura de bloques, incluye estructuras y uniones, un mejor sistema de tipos y también soporta funciones recursivas. Prácticamente todos los lenguajes modernos de programación tienen la estructura de bloques introducida por Algol.

5. Simula. Diseñado a mediados de los sesenta en Noruega es el primer lenguaje orientado a objetos; basado en Algol, introdujo las clases y corrutinas.

A la par del desarrollo de los lenguajes de programación, se llevó a cabo el desarrollo de las diferentes arquitecturas de cómputo, dándose entre éstas (las arquitecturas de cómputo) y éstos (los lenguajes de programación y sus respectivos compiladores), una influencia mutua en el diseño de ambas partes, pues en todo momento un compilador de un lenguaje de programación tiene como objetivo sacar el mayor provecho de la arquitectura subyacente; y al revés, el diseño de una arquitectura siempre tiene como objetivo ofrecer a nivel hardware las mejores características que puedan ser aprovechadas por el compilador.[3] Así, a finales de los años 60 y principios de los 70 se observó que el costo del software crecía más rápido que el del hardware; se argumentó que los sistemas operativos y los compiladores eran muy grandes y complejos y llevaba mucho tiempo desarrollarlos. Dada la inferioridad de los compiladores y la memoria limitada de las computadoras, la mayoría de los programas de la época seguían siendo escritos en lenguaje ensamblador. Algunos investigadores propusieron aliviar la crisis de software con arquitecturas más poderosas orientadas al software y esta estrategia funcionó bien por más de diez años; la *Burroughs B5000* (que apareció en 1961) fue diseñada para simplificar la compilación de los lenguajes de alto nivel; más tarde la VAX hizo su aparición en 1970 con el mismo enfoque, cuando la VAX comenzó a diseñarse, surgió un enfoque aún más radical, que se denominó (*HLLCA*) (*High Level Language Computer Architecture*), el cual comenzó a ganar adeptos en las comunidad de investigación; este movimiento estaba enfocado a eliminar la barrera entre los lenguajes de alto nivel y el hardware, es decir, la tendencia era que la arquitectura de la computadora soportara instrucciones lo más similares posible a las instrucciones de un lenguaje de programación

[3]Por ejemplo, debido al desarrollo de LISP y por el ferviente deseo de ejecutar los programas escritos en este lenguaje con una mayor eficiencia, se crearon computadoras que ejecutaban operaciones básicas de LISP a nivel hardware.

de alto nivel, siendo sin duda éstas mucho más complejas y difíciles de implementar a nivel hardware. El enfoque HLLCA (por fortuna) tuvo poco impacto comercial (en particular la Burroughs).

Contrariamente a este movimiento de diseño de arquitecturas de computadoras, surgió en los años 80 un nuevo paradigma de diseño de arquitecturas de computadoras, el diseño *RISC* (*Reduced Instruction Set Computer*) que, a diferencia de las arquitecturas que se habían venido desarrollando hasta entonces, planteaban tener un número reducido de instrucciones que se ejecutaran en un solo ciclo de reloj; cada una de sus instrucciones tenía un tamaño fijo; utilizaban pocos modos de direccionamiento y utilizaban un *pipeline* para realizar varias instrucciones al mismo tiempo; en tanto, las arquitecturas *CISC* (*Complex Instruction Set Computer*, el antiguo diseño) ofrecían una amplia variedad de modos de direccionamiento; el tamaño de cada una de sus instrucciones era variable y los ciclos que tomaba realizar cada una de sus instrucciones era variable también. El objetivo de la VAX era tener compiladores simples, modos de direccionamiento complejos, instrucciones complejas, codificación eficiente de instrucciones y pocos registros; el objetivo de una computadora con una arquitectura *MIPS* (que fue una de las primeras arquitecturas RISC) era tener alto desempeño por medio del pipeline (el cual explicaremos más adelante), fácil implementación de hardware y que sus compiladores fuesen compiladores altamente optimizadores (es decir, que generaran código lo más eficiente posible), por lo que contaba con instrucciones simples, modos de direccionamiento simples, formato de instrucciones de tamaño fijo y un extenso número de registros. La arquitectura MIPS resultó tener a lo más tres veces el desempeño de la VAX. La compañía *Sun Microsystems* se basó en el diseño RISC para fabricar sus procesadores *SPARC*.

Un pipeline es una técnica de implementación de hardware en donde múltiples instrucciones se traslapan en ejecución; se utiliza en la mayoría de los procesadores desde mediados de los 80, siendo una pieza fundamental para que los procesadores sean rápidos. Sin embargo, aun cuando desearíamos que varias instrucciones pudiesen tratarse en paralelo, no siempre es posible ya que, por ejemplo, una instrucción puede depender del resultado de otra que aún no ha sido completada; a este tipo de situaciones se les llama obstáculos (*hazards*). Teniendo en mente la idea de ejecutar simultáneamente varias instrucciones en un ciclo de reloj, se añadieron más unidades aritmético-lógicas y de cargar/guardar (*load/store*) al procesador, para tener simultáneamente trabajando dos pipelines al mismo tiempo; a esto se le conoce como una arquitectura superescalar. Las etapas comunes en un pipeline para una instrucción son:

- IF: Cargar la instrucción (*Instruction Fetch*)
- ID: Decodificar la instrucción (*Instruction Decode*)
- EX: Ejecutar o calcular la dirección (*Execution or Address Calculation*)

- MEM: Acceder a memoria (*Data Memory Access*)
- WB: Escribir resultado (*Write Back*)

Algunas de las arquitecturas superescalares cuentan con la posibilidad de reordenar las instrucciones de un programa para evitar el mayor número de obstáculos posibles; esto se conoce como una planificación dinámica del pipeline o como ejecución fuera de orden y está presente, por ejemplo, en la arquitectura *x86* de *Intel* desde el *Pentium III*; a diferencia de éstas, existen arquitecturas que sólo ejecutan las instrucciones en orden, tal como las generó el compilador, es decir, realizan una planificación estática del pipeline. Hay otro tipo de paralelismo a nivel de instrucción, llamado *VLIW* (*Very Long Instruction Word*), en el cual el compilador es el encargado de verificar las dependencias entre instrucciones y de ordenarlas de la mejor manera posible, para que haya el menor número posible de obstáculos; una vez realizado esto, las empaqueta en una cadena grande de instrucciones; el procesador se encarga entonces de decodificarlas y ejecutarlas en paralelo, sin realizar éste ningún tipo de verificación, pues esta tarea debió haber sido realizada por el compilador. Un ejemplo de este tipo de arquitectura lo tenemos en el procesador *Intel Itanium*.

Hemos dado ya una breve introducción acerca del interés tanto teórico como práctico de programar computadoras, así como del desarrollo de los estilos de programación, los lenguajes de programación y las arquitecturas de computadoras.[4]

1.2. Traductores

Mencionamos anteriormente la necesidad de desarrollar programas que sean capaces en última instancia de ejecutar programas escritos en un lenguaje de alto nivel. Si bien un camino para lograr lo anterior es mediante la traducción de los enunciados de alto nivel a lenguaje de máquina, existen otros enfoques para llevarlo a cabo. Analicemos con detenimiento cada una de las opciones que podemos desarrollar.

- Intérpretes. Un intérprete es un programa que cuando se le alimenta con una expresión del código fuente realiza las acciones requeridas para evaluar dicha expresión. Un intérprete actúa como una capa de software entre el programa (escrito en un lenguaje de alto nivel) que se desea ejecutar y la máquina (el hadware) en donde realmente se ejecuta. Generalmente las instrucciones en el lenguaje de alto nivel se ejecutan una por una, esto es, la implementación de un intérprete generalmente se presenta mediante un ciclo lee-evalúa-imprime (*read-evaluate-print loop*, *REPL*),

[4]Para obtener un conocimiento más profundo del diseño y clasificación de lenguajes de programación se recomienda [Sco09]; para una mayor referencia acerca del diseño y evolución de las arquitecturas de computadoras se recomienda [PH08] y [HP06].

en donde se lee un enunciado en el lenguaje de alto nivel, se lleva a cabo un proceso de verificación, (análisis léxico, análisis sintáctico, análisis semántico),[5] se evalúa y está listo para el siguiente enunciado; este ciclo se repite hasta que finalmente se ha ejecutado todo el programa. Es importante señalar que el intérprete **no** genera explícitamente código de máquina que corresponda al enunciado que está en proceso de evaluación, sino que es el intérprete quien ejecuta el enunciado; en este sentido podemos ver el lenguaje de implementación como nuestro lenguaje objetivo en lugar de lenguaje de máquina. Por supuesto que el intérprete es en sí mismo un programa que contiene instrucciones que dictan qué instrucciones utilizar para evaluar el enunciado que acaba de leer; aunque esto pueda parecer difícil a primera vista, en realidad no lo es como veremos más adelante con un ejemplo. Los intérpretes son programas cuyos datos son los enunciados, por lo que una vez que el intérprete ha sido traducido a lenguaje de máquina, procesa sus datos reaccionando frente a ellos.

- Compiladores. Un compilador es un programa que traduce el código escrito en un lenguaje de alto nivel a un código generalmente de bajo nivel[6] que usualmente es código de máquina,[7] al que se le nombra código objeto; a la máquina para la cual se genera este código se le llama máquina objetivo. En general al código de entrada para un traductor se le llama código fuente; en el caso de los compiladores e intérpretes es un código escrito en un lenguaje de alto nivel. Un compilador, a diferencia de un intérprete, realiza al menos una lectura de inicio a fin del código fuente, lleva a cabo varias etapas de análisis y reescritura, y finalmente genera código de máquina que corresponde a cada una de las instrucciones del código fuente.

- Máquinas virtuales. Una máquina virtual es una aplicación que simula el comportamiento de un procesador real, esto es, desde el punto de vista del programador puede observarse como si se tratara de un procesador real, sólo que cuando deben ejecutarse instrucciones en ella, en lugar de que se ejecuten en hardware como sucedería en un procesador físico, la aplicación realiza una traducción para que dichas instrucciones se ejecuten en el procesador real sobre el que se está ejecutando la máquina virtual. Para elaborar una máquina virtual, de la misma forma que sucede con una máquina real, primero debe diseñarse; en este proceso el diseñador

[5] Estudiaremos estos temas con más detenimiento posteriormente.

[6] Esto no siempre es así pues existen compiladores que traducen código escrito en un lenguaje de alto nivel a código de otro lenguaje de alto nivel; así tenemos por ejemplo compiladores que traducen de Java a C.

[7] Frecuentemente es código ensamblador que posteriormente es ensamblado por un programa ensamblador; también se puede generar código de una máquina virtual, para la cual no hay una correspondencia directa en hardware.

de la máquina elige, entre otras cosas, los modos de direccionamiento y las instrucciones que formarán parte de ella. Las arquitecturas de las máquinas virtuales se clasifican en:

- Arquitectura orientada a registros (*register-oriented architecture*). Se llama orientada a registros debido a que todas o la mayor parte de sus instrucciones toman al menos un operando de algún registro, es decir, en este tipo de arquitecturas las instrucciones usualmente involucran el uso de uno o varios registros.
- Arquitectura orientada a la pila (*stack-oriented architecture*). En este tipo de arquitectura los operandos de las instrucciones se toman a partir del tope de la pila (y no de los registros) y el resultado se pone en el tope de la pila; en estas máquinas todas o la mayor de parte de sus instrucciones se realizan haciendo uso únicamente de la pila.[8]

Por supuesto la implementación de una máquina virtual se realiza mediante software, aunque si se desea puede implementarse en hardware (en este punto dejaría de ser una máquina virtual y se convertiría en una máquina real).

Estudiemos ahora algunas características, ventajas y desventajas de cada uno de estos modelos de traducción.

1.2.1. Intérpretes

Los intérpretes tiene varias ventajas y son generalmente más fáciles de implementar ya que contamos y podemos hacer uso de todas la características del lenguaje de implementación –al que se conoce como lenguaje anfitrión– para proveer características del lenguaje a ser interpretado; es por tanto importante la elección del lenguaje anfitrión, por ejemplo, si el lenguaje fuente debe contar con un recolector de basura y evaluación perezosa, y el lenguaje anfitrión no cuenta con un recolector y tiene un mecanismo de evaluación diferente, la implementación del intérprete conllevará por supuesto un mayor trabajo; si en cambio el lenguaje de implementación cuenta ya con un recolector de basura y evaluación perezosa, puede resultar realmente sencillo delegar sobre ellas la implementación de estas características en el lenguaje fuente. Sin embargo Sussman y Steele en [SS98] nos dicen que es mejor pensar en el lenguaje de implementación como un lenguaje de máquina y la razón de esta manera de proceder es que si el lenguaje de implementación no soporta algunas características presentes en el lenguaje fuente será difícil interpretarlas; en particular si no soporta ciertas estructuras de control, entonces

[8]Un ejemplo de máquina que cuenta con esta arquitectura es la máquina virtual Java y las arquitecturas Burroughs de los años 60 y 70.

no será posible interpretarlas efectivamente. Por ejemplo es común que los lenguajes implementen las llamadas a funciones como marcos (*frames*) instalados en una pila y puede ser que el lenguaje a ser interpretado tenga un mecanismo más general de estructuras de control. Así entonces, podemos no hacer uso del mecanismo de recursión del lenguaje de implementación para implementar la recursión en el lenguaje fuente y en lugar de ello podemos pensar que contamos con ciertas operaciones codificadas en el lenguaje de implementación y contamos además con ciertos registros (virtuales) sobre los cuales se pueden aplicar las operaciones, para de esta forma implementar la recursión explícitamente. Hay una característica particular sobresaliente en los intérpretes y es que en éstos se cuenta con la información del programa en tiempo de ejecución, tanta como se requiera; esta es tal vez la característica que los hace más importantes.

Por ejemplo, supongamos que estamos realizando la implementación de un intérprete de *Scheme* en *C* y al intérprete se le presenta el enunciado `(+ 5 7)`; el intérprete primero realiza un análisis sintáctico para verificar que la expresión esté bien formada; luego realiza un análisis basado en casos (mediante un `switch`) para determinar de qué tipo de expresión se trata, que en este caso es una aplicación de función. Por tanto, primero se evalúa el elemento a la izquierda, al código de la función `+`, y como es una función primitiva el intérprete debe implementarla; posteriormente se evalúan los operandos (Scheme utiliza paso de parámetros por valor), después se crea un nuevo entorno con los parámetros formales establecidos con los valores de los operandos (argumentos) y finalmente se ejecuta la función (es decir, se ejecuta cada uno de los enunciados contenidos en el cuerpo de la función, remplazando las presencias de los parámetros formales con los valores de los argumentos). Hay que aclarar que dentro del cuerpo de la función `+` se debe realizar verificación de tipos, pues de otra forma se tendría que suponer que los argumentos tienen el tipo esperado; para la función `+` los argumentos deben ser de tipo numérico, pues en Scheme la aritmética se realiza sólo sobre tipos numéricos y no sobre otros tipos; en nuestro ejemplo la función se aplicaría de la manera esperada. Sin embargo, si en lugar de ello presentamos el enunciado `(+ 5 ``hola'')` y no se realiza verificación de tipos, el intérprete aborta inesperadamente sin emitir ningún mensaje de error; si se realiza verificación de tipos, el intérprete detecta que uno de los argumentos no tiene el tipo esperado y puede mostrar un mensaje de error, o lanzar una excepción con un mensaje asociado, para que después se maneje ya sea por el usuario o por el intérprete. Lo que queremos decir es que el intérprete actúa como un supervisor que verifica que el programa se comporte de manera adecuada en tiempo de ejecución. Esto representa una ventaja sobre los compiladores, ya que éstos no pueden llevar a cabo esta verificación pues se ejecutan antes de que se ejecute el programa. De lo anterior, tenemos dos opciones, realizar o no verificación de tipos. Si se realiza verificación de tipos el intérprete es más robusto y seguro; pero a cambio de esto el intérprete se vuelve más lento debido a la sobrecarga que se genera por las instrucciones que realizan la verificación de tipos. Además si no realizamos estas verificaciones en tiempo de ejecución

(*runtime-checks*) el intérprete es más rápido pues no tiene sobrecarga, pero a cambio de ello nuestro intérprete se vuelve inseguro y frágil ante los posibles errores en tiempo de ejecución. Aunque esto no pareciera tener mucha importancia, en la práctica sí la tiene, pues los intérpretes gastan un tiempo "considerable" en la realización de estas verificaciones en tiempo de ejecución (algunos intérpretes además realizan otras actividades como verificar que el número de argumentos pasados a una función sea el correcto); en particular; en [SH87] se analiza LISP y se tiene que en promedio el 25 % del tiempo de ejecución de los programas se gasta en verificaciones en tiempo de ejecución y fue precisamente este tipo de resultados los que motivaron a que se desarrollaran las máquinas LISP que contaban con soporte en hardware para la realización de estas verificaciones. Volveremos a este tema cuando estudiemos el análisis semántico, donde nos daremos cuenta que la semántica del lenguaje y, en particular, el sistema de tipos de un lenguaje, juega un papel fundamental en la determinación del tiempo que toma la verificación de tipos, ya sea en tiempo de ejecución o en tiempo de compilación.

Existen dos estrategias generales para la implementación de un intérprete:

1. Intérpretes dirigidos por la sintaxis. En este tipo de intérpretes se lee un enunciado, se verifica que sea un enunciado válido y se realizan las acciones semánticas relacionadas con éste (se evalúa). Los intérpretes de este tipo tienen una visión local del código fuente y puede ser necesario que reanalicen un enunciado una o más veces; generalmente se implementan asociando acciones (que realizan las acciones semánticas) al analizador sintáctico. No generan representaciones intermedias, son fáciles de implementar e intuitivos, aunque tienen como característica ser lentos.

2. Intérpretes basados en árbol. Este tipo de intérpretes primero realizan una lectura de inicio a fin del código fuente, llevando a cabo un análisis sintáctico de todo el programa y generando una representación intermedia en forma de árbol; debido a lo anterior soportan referencias hacia adelante y evitan el reanálisis de los enunciados del programa; además pueden realizar optimizaciones basadas en representaciones intermedias de alto nivel. Típicamente los intérpretes de este tipo son más rápidos que los anteriores.

1.2.2. Compiladores

Los compiladores, por su parte, suelen ser más complejos (ya que por lo general realizan una mayor cantidad de etapas como la generación de código y varias etapas de optimización) y por tanto su implementación es más laboriosa y requiere de más tiempo. Podemos ver a un compilador como una caja negra que recibe como entrada un programa escrito en un lenguaje de alto nivel y produce como salida la traducción del programa en código de máquina, listo para ser ejecutado directamente por la máquina (ya sea física o virtual). Pero ¿qué es lo que sucede con una visión más cercana dentro

del funcionamiento de un compilador? Un compilador usualmente realiza varias fases o etapas que son las siguientes:

1. **Análisis léxico:** Su principal objetivo es descomponer el código fuente en elementos o átomos que sirvan como unidades indivisibles (símbolos terminales de la gramática) al analizador sintáctico, además de eliminar todos aquellos caracteres que no tengan sentido para el analizador sintáctico como son espacios en blanco, saltos de línea y comentarios.

2. **Análisis sintáctico:** Una vez descompuesto el código fuente en átomos, el analizador sintáctico se encarga de verificar que la estructura sintáctica del programa sea válida, basado en la gramática del lenguaje. Además, dependiendo del tipo de compilador, puede generar una representación intermedia (siendo ésta usualmente un árbol de sintaxis abstracta) o generar código al vuelo.

3. **Análisis semántico:** Se encarga de verificar la estructura semántica del programa, siendo una parte destacada de ésta la verificación de tipos; también se realizan otras actividades más como son: verificar que una variable esté definida antes de que se utilice; que se pase el número de argumentos correcto en una llamada a una función; que la función o método que se llama esté definido; entre otras. Esta etapa hace uso de una estructura de datos conocida como tabla de símbolos que puede ser compartida por diferentes etapas del compilador.

4. **Generación de código intermedio:** En esta etapa se genera una representación intermedia de bajo nivel independiente de la máquina destino, que puede ser árboles de bajo nivel, código de tres direcciones, cuádruplas o alguna otra. Aunque esta etapa no es estrictamente necesaria para la traducción del programa, es fundamental si se desea desarrollar un compilador que genere código óptimo y que sea reobjetivizable.

5. **Optimización independiente de la arquitectura:** Toma como entrada la representación intermedia de bajo nivel y realiza sobre ella varias transformaciones de acuerdo a las optimizaciones que se apliquen.

6. **Generación de código:** Toma como entrada la representación intermedia de bajo nivel y genera código de máquina o código ensamblador. Usualmente se divide en las siguientes subetapas: selección de instrucciones, alojamiento de registros y planificación de código.

7. **Optimización dependiente de la arquitectura:** Se realizan optimizaciones basadas en las características presentes en la arquitectura objetivo.

Estas etapas representan la organización lógica de un compilador, aunque algunas de ellas pueden no estar presentes en algunos casos y otras pueden ser incluidas en otros, sobre todo aquellas que realizan otro tipo de optimizaciones. Durante la ejecución de un compilador es común que se realicen una o varias pasadas; una pasada toma como entrada una representación del código y hace una lectura de inicio a fin de ésta con el objetivo de obtener información para realizar una transformación; dependiendo del diseño del compilador, una etapa puede realizarse llevando a cabo una o varias pasadas (lo que crea un compilador más modular) o, en otro caso, varias etapas pueden agruparse en una única pasada. Por ejemplo, los compiladores que generan código al vuelo dirigidos por la sintaxis agrupan las fases de análisis léxico, análisis sintáctico, análisis semántico y generación de código en una sola pasada.

El código que generaban los primeros compiladores no era tan eficiente como el código que generan los compiladores de nuestros días, pues en aquel tiempo no se habían desarrollado todas las técnicas de optimización que se conocen ahora.

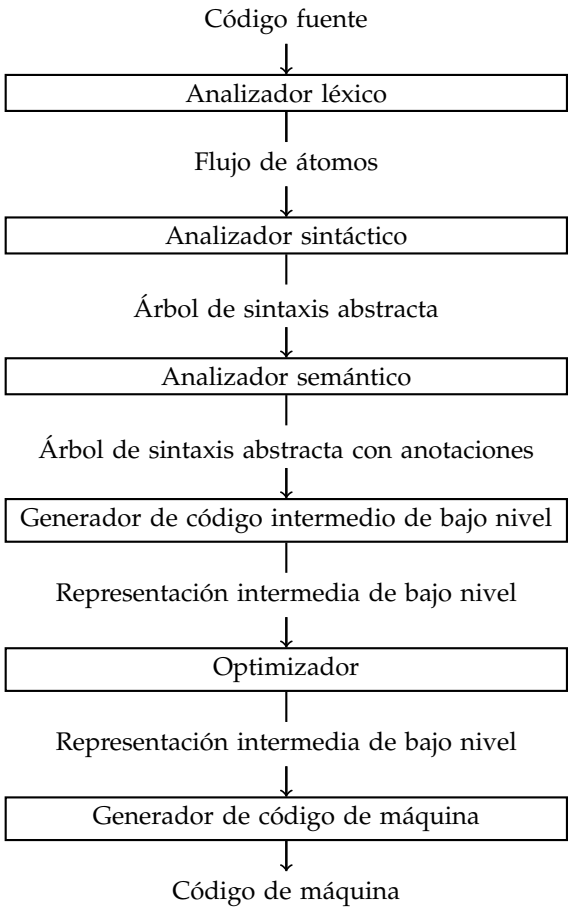
Diseño de compiladores

Existen dos diseños principales para la elaboración de compiladores optimizadores, que se muestran en la Figura 1.1 de la página 13.

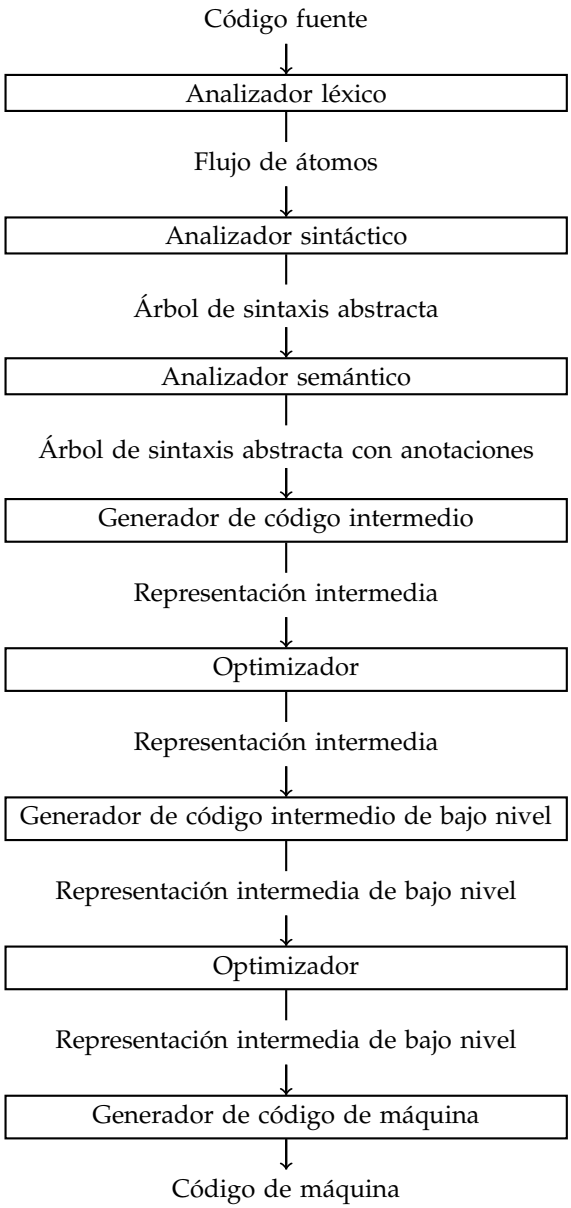
Como se puede observar en el diseño que se muestra en la Figura 1.1(a) todas las optimizaciones se llevan a cabo sobre una representación intermedia de bajo nivel, mientras que en el diseño que se muestra en la Figura 1.1(b) se aplica un conjunto distinto de optimizaciones a cada una de las representaciones intermedias con base en su nivel de abstracción.

Así, hay optimizaciones que únicamente se realizan sobre la representación intermedia de alto nivel pero no sobre la de bajo nivel y viceversa, aunque hay ciertos tipos de optimizaciones que se pueden realizar sobre ambas.

Uno de los diseños que concuerda con el diseño de la Figura 1.1(a) es el descrito en [DF84], donde el compilador cuenta con un *front end* que genera código para una máquina abstracta (diseñada por el desarrollador del compilador); además cuenta con un expansor de código que toma como entrada las instrucciones de la máquina abstracta y genera una representación intermedia de bajo nivel de nombre *RTL* (*Register-Transfer Lists*). Este expansor de código es dependiente de la máquina pero su función es generar código ingenuo (es decir, sin que se haya intentado ningún tipo de optimización sobre él), por lo que es relativamente fácil de implementar. Una vez que se cuenta con el código ingenuo, se realizan optimizaciones sobre él, las cuales se llevan a cabo de la siguiente forma: se realiza una posible optimización sobre el código RTL, posteriormente se presenta el nuevo código RTL mejorado a un autómata finito (también conocido como reconocedor) -que se construye automáticamente a partir de una descripción de la máquina- y éste decide si las nuevas instrucciones son legales, es decir, si hay una



(a) Diseño de compilador que optimiza únicamente la representación intermedia de bajo nivel.



(b) Diseño de compilador que optimiza las diversas representaciones intermedias.

Figura 1.1: Principales diseños de compiladores optimizadores.

correspondencia entre el código RTL y el código de máquina; de ser así el RTL viejo se remplaza por el nuevo; en caso contrario se descarta el nuevo y se continúa trabajando con el viejo; en ambos casos se intenta una nueva optimización. Ya que se han realizado todas las optimizaciones posibles, el asignador se encarga de hacer el alojamiento de registros en la máquina física y generar el código ensamblador haciendo uso del autómata finito descrito anteriormente. Cada descripción de máquina es una gramática adecuada para una traducción dirigida por la sintaxis, entre el lenguaje ensamblador y RTL y es, por supuesto, dependiente de la máquina. Este esquema hace uso de RTL porque permite optimizar código de máquina específico mediante código independiente de la máquina. Benitez y Davidson en [BD94] mostraron que un optimizador con esta estructura no solamente soporta optimizaciones de mirilla (*peephole optimizations*), sino además todas las optimizaciones escalares y de ciclos estándar.

Originalmente *GCC* en su primer versión se desarrolló con base en este diseño, aunque a partir de la versión 4 se incorporaron dos nuevas representaciones intermedias basadas en árbol (*GENERIC* Y *GIMPLE*), con un mayor nivel de abstracción que RTL, para poder realizar sobre ellas otros tipos de optimizaciones.

Un ejemplo del segundo diseño es el compilador *Open64*, cuyo nombre original era *Pro64*, que es un compilador altamente optimizador, desarrollado originalmente por la compañía *SGI* y posteriormente liberado como código libre. Open64 es un compilador modular, diseñado para hacer un gran número de optimizaciones en diferentes niveles. Su diseño se basa en *WHIRL* (*Winning Hierarchical Intermediate Representation Language*), una representación intermedia que cuenta con varios niveles de abstracción; así, el *front end* genera *Very High WHIRL* y el código se va transformando en los siguientes niveles de WHIRL: *Very High WHIRL*, *High WHIRL*, *Mid WHIRL*, *Low WHIRL* y *Very Low WHIRL*, hasta que se genera código de máquina y sobre cada uno de ellos se hacen diferentes optimizaciones, con base en la información del código que contiene cada nivel.

Así, en la actualidad tenemos compiladores en producción que generan código altamente optimizado, a diferencia del código que generaban los primeros compiladores que era código ingenuo e ineficiente en algunos casos. En general, una ventaja de los compiladores sobre los intérpretes es la eficiencia de un programa en ejecución ya que, a diferencia de los intérpretes, el compilador sólo se debe ejecutar una vez, genera código eficiente (en el mejor de los casos) del programa y una vez realizado lo anterior, el programa está listo para ser ejecutado tantas veces como se requiera sin la intervención del compilador. Es claro que no se tiene la sobrecarga del intérprete que traduce en tiempo de ejecución y que el código que se ejecuta es más eficiente pues está optimizado. A cambio de ello, el desarrollo de un compilador optimizador requiere de un grupo de desarrolladores con un conocimiento profundo del tema y toma un período de desarrollo más largo.

1.2.3. Máquinas virtuales

Las máquinas virtuales tienen la ventaja que el código que se ejecuta en ellas es portátil, es decir, si un programa p se ejecuta en la máquina virtual m, entonces p se puede ejecutar en cualquier máquina física que tenga una implementación de m; de otra forma p se tendría que compilar a código nativo de las diferentes máquinas físicas en las que se necesita ejecutar y estaríamos obligados a que un compilador genere código de máquina para cada una de las máquinas físicas. Además, como en el caso de los intérpretes, la máquina virtual actúa como una capa de software entre el código que se está ejecutando y la máquina física, lo que permite que se tenga un mayor control de errores y seguridad en tiempo de ejecución, características que son muy valiosas en ciertos entornos. El diseñador de la máquina virtual tiene la libertad de elegir qué instrucciones formarán parte de ella, por lo que puede incluir instrucciones tan complejas o simples como desee; además debe elegir los otros componentes de su arquitectura, decisiones que impactan sobre los traductores que vayan a generar código para ella. *Scheme 48* es un traductor de Scheme basado en una máquina virtual de pila cuya primera versión se desarrolló en 48 horas (de ahí su nombre). Scheme 48 tiene una interfaz interactiva con el usuario como la de un intérprete; por ejemplo, cuando se le presente el enunciado `(lambda (x)(+ 10 x)` lo compila a código de su máquina virtual produciendo las siguientes instrucciones:

```
(check-nargs= 1)
(make-env 1)
(literal '10)
(push)
(local 0 1)
(+)
(return)
```

Código 1.1: Instrucciones de la máquina virtual correspondientes al enunciado `(lambda (x)(+ 10 x)`.

que el intérprete de la máquina virtual ejecuta. Generalmente la implementación del intérprete lee una instrucción y con base en un análisis de casos mediante un `switch` determina de qué instrucción se trata y actúa en consecuencia. Este análisis de casos cuando se realiza mediante un switch puede degradar la eficiencia del intérprete y por ello se han desarrollado métodos que lo realizan de manera más eficiente como los que se analizan en [EG03].

Quizá la máquina virtual más popular de nuestros días es la que interpreta programas escritos originalmente en *Java*, la *JVM* (*Java Virtual Machine*). Java es por sí mismo un lenguaje de programación de alto nivel diseñado en los noventas por James Gosling

en la compañía Sun Microsystems.[9] La implementación de este lenguaje desarrollada por Sun consiste de un compilador[10] que traduce el código fuente a código objeto (*bytecode*) de la máquina virtual, además de una máquina virtual que interpreta el (bytecode).[11] Java tiene como una de sus características ser seguro, por tanto la máquina debe realizar verificaciones en tiempo de ejecución. Podríamos preguntarnos ¿por qué debe hacer este tipo de verificaciones si contamos con un compilador que realiza el análisis semántico en tiempo de compilación?. Como Java es un lenguaje fuertemente tipado, el analizador semántico del compilador debe detectar errores como los que se presentan en el enunciado `int x = 5 +''hola''`, por lo que la máquina virtual no tendría la necesidad de verificar que los operandos para la adición aritmética tuvieran el tipo correcto; y en efecto así es, pues en este caso el compilador detecta que el segundo operando es de tipo cadena y por tanto no tiene el tipo correcto y muestra un mensaje de error en tiempo de compilación; el problema está en que no podemos asegurar que se haya utilizado el compilador de Sun para generar el bytecode, y un compilador experimental o malicioso podría generar bytecode que se comportara de manera incorrecta o maliciosa para dejar el sistema en estado vulnerable. También un desarrollador tiene la libertad de escribir, si así lo desea, programas en ensamblador para la máquina virtual y si es un programador novato o malicioso de igual forma puede generar bytecode inseguro.

Es claro en este punto que la máquina virtual debe realizar verificaciones en tiempo de ejecución, pero hemos visto ya que éstas degradan el tiempo de ejecución de las aplicaciones, por lo que los desarrolladores de la máquina virtual decidieron implementar un verificador, que es una especie de demostrador de teoremas (*theorem prover*), que verifica que el comportamiento del programa en tiempo de ejecución sea seguro e implantaron políticas de seguridad que deciden en qué casos se debe ejecutar el verificador y cuándo no debe ejecutarse. Generalmente no se ejecuta cuando las clases son de la máquina local y se ejecuta cuando se recibe bytecode de otro equipo por medio de una red. En las primeras versiones del intérprete de la máquina virtual, Java se hizo famoso por ser un lenguaje lento. Para mejorar el rendimiento de las aplicaciones en Java sus desarrolladores decidieron que si ya tenían el bytecode podían generar a partir de éste, viéndolo como representación intermedia, la última parte de un compilador nativo, es decir traducir el bytecode a código de máquina y así mejorar el rendimiento de las aplicaciones; de esta forma cuando el intérprete va a ejecutar un método, primero lo convierte a código nativo, lo que sucede justo antes de que el método se ejecute por primera vez y es por esto que este tipo de compiladores se llaman compiladores *JIT* (*Just-In-Time*). Los compiladores JIT realizan la traducción a código nativo durante la ejecución del bytecode. Existen compiladores que realizan la traducción a código nativo antes de que se lleve a cabo la ejecución del bytecode, con el objetivo de hacer los

[9] Ahora parte de Oracle.

[10] De nombre `javac` en la mayoría de los sistemas.

[11] De nombre `java` en la mayoría de los sistemas.

programas más eficientes en tiempo de ejecución; este tipo de compiladores se conocen como compiladores *AOT* (*Ahead-Of-Time*).

1.3. Nuestro compilador

Nosotros nos enfocaremos únicamente en los compiladores y no hay mejor manera de explicar la estructura interna y los algoritmos que se realizan dentro de un compilador que desarrollando uno. La mayor parte de los textos actuales en el tema de compiladores pueden clasificarse en aquellos que son meramente teóricos y en otros más prácticos, en los que se suele desarrollar un compilador para un lenguaje de juguete; en el primer caso el lector interesado en desarrollar un compilador puede sentirse lejano del conocimiento necesario para realizar la implementación de uno y en el segundo puede resultar insatisfecho debido a que puede sentir que no conoce las técnicas necesarias para implementar un compilador de un lenguaje con características más avanzadas que se utilice en aplicaciones de la vida real. Alternativamente existen algunos textos en los que se desarrolla un compilador para un subconjunto de algún lenguaje ampliamente utilizado en la vida real como es el caso de C. Los compiladores históricamente se han desarrollado en grupos de investigación en universidades o en grandes compañías como Intel, IBM (entre otras) y su tiempo de desarrollo va desde algunos meses hasta varios años. Tradicionalmente los compiladores se desarrollan primero implementando las funciones mínimas para que sea un compilador funcional y a partir de este momento se le hacen mejoras, corrección de errores y se implementan nuevas características como una nueva optimización o el remplazo de alguno de sus módulos por otro más eficiente (por ejemplo se puede remplazar el recolector de basura por otro con un algoritmo diferente que sea más eficiente).

El compilador que desarrollaremos aquí tiene un enfoque didáctico, está diseñado para que los alumnos que tomen un primer curso de compiladores puedan experimentar e implementar en él los conocimientos teóricos adquiridos. Hemos elegido *Python* como lenguaje fuente debido a que es un lenguaje que incluye diferentes paradigmas de programación, lo que hace un reto interesante construir un compilador para él; además es por supuesto un lenguaje ampliamente utilizado en aplicaciones de la vida real en diferentes dominios. Así el lector quedará complacido de desarrollar un compilador para un lenguaje que se usa en la vida real. Quizá las características más interesantes con las que cuenta Python y que representan un mayor reto en la implementación del compilador son su sistema de tipos dinámico y que cuenta con características funcionales como son las funciones de primer orden y la función `eval` al estilo LISP, que evalúa código en tiempo de ejecución.

A continuación damos una descripción más detallada de Python.

1.3.1. Python

Python es un lenguaje de propósito general con múltiples capacidades y que se usa ampliamente en aplicaciones de la vida real. Está diseñado para que su sintaxis sea simple y entendible y los programas escritos en él reusables y mantenibles, en mayor medida que en los lenguajes de *script* tradicionales. Python sigue una filosofía minimalista, haciendo que los programadores logren más con menos esfuerzo: según el *Zen de Python*, explícito es mejor que implícito y simple es mejor que complejo. A diferencia de otros lenguajes como *C++* o Java, un programa equivalente en Python es por lo general más pequeño y por tanto se cometen menos errores; es un lenguaje en el que se disfruta programar. Python se diseñó para permitir un rápido desarrollo a los programadores. Algunas de las capacidades de Python se listan a continuación.

- Sintaxis simple.
- Tipado dinámico.
- Manejo automático de memoria.
- Orientación a objetos.
- Estructuras de control variadas y sofisticadas.
- Recursión.
- Funciones de orden superior.
- Enunciados imperativos.

A veces se dice que Python es un lenguaje script orientado a objetos. Python se usa y es popular hoy en día en los siguientes escenarios:

- La integración que tiene Python con los servicios provistos por el sistema operativo hace que se utilice para desarrollar herramientas de mantenimiento del sistema.
- La simplicidad y rapidez de Python hace que se utilice para desarrollar interfaces gráficas (*GUI*).
- La capacidad de Python para trabajar con *sockets* hace que sea una herramienta útil en internet y en las redes en general; permite desarrollar programas que se comuniquen por medio de la red ya sea como servidor o cliente y que se realice cualquier función deseada, por ejemplo como un cliente de correo, o tomar la información importante de una página web.

- Permite un rápido desarrollo con bases de datos, ya que tiene una API que realiza una transformación de manera transparente a los diversos manejadores de bases de datos que existen hoy en día, lo cual lo hace ideal para desarrollar programas en el mundo de los negocios y administración.

- Existe una extensión de Python que provee el uso de funciones y diversas herramientas matemáticas, que por ende hace de Python un lenguaje útil en aplicaciones de cómputo científico y análisis numérico.

- Existen también extensiones para campos como la inteligencia artificial, graficación y videojuegos, que hacen posible que Python sea utilizado en esos campos.

1.3.2. UltraSPARC T2

Ya que hemos elegido nuestro lenguaje fuente, es hora de elegir la máquina objetivo. Nuestro compilador está diseñado para ser reobjetivizable, es decir, que sea fácilmente extendido para soportar una nueva máquina objetivo. Hemos decidido enfocarnos principalmente en dos arquitecturas, la primera de ellas es la arquitectura *x86-64*[12] originalmente desarrollada por *AMD* –como *AMD64*–; la segunda es la arquitectura SPARC desarrollada por Sun. La razón de elegir la arquitectura x86-64 es que es un arquitectura ampliamente utilizada en las computadoras de escritorio y estaciones de trabajo, por lo que tenemos la esperanza que la mayoría de las personas puedan realizar pruebas del compilador en su computadora personal; x86-64 es una arquitectura que es compatible hacia atrás con la arquitectura x86 de Intel y de la que hereda el diseño CISC, aunque x86-64 introduce características de una arquitectura RISC como es un mayor número de registros. La arquitectura SPARC es una arquitectura RISC, que ha sido utilizada desde su surgimiento en los ochentas en estaciones de trabajo y servidores, y utilizada para la investigación en el campo de compiladores y diseño de arquitecturas de computadoras; es una arquitectura diseñada para trabajar conjuntamente con compiladores altamente optimizadores. La elección de SPARC obedece a que es una arquitectura con la que resulta interesante experimientar diferentes diseños, optimizaciones y técnicas involucradas en el desarrollo de un compilador. En particular los alumnos de la Facultad de Ciencias de la U.N.A.M tienen acceso a una computadora que cuenta con un procesador *UltraSPARC T2 codename Niagara* del cual a continuación se mencionan algunas de sus características.

El procesador UltraSPARC T2 desarrollado por Sun es un procesador basado en la arquitectura *UltraSPARC 2007*, que cuenta con ocho núcleos. Cada núcleo tiene soporte para ocho hilos, dos pipelines de ejecución para enteros (cada pipeline para enteros tiene

[12]También conocida como *x64*.

ocho etapas), un pipeline de ejecución de punto flotante (que tiene 12 etapas) y un pipeline de memoria. Los pipelines de memoria y de punto flotante se comparten entre los ocho hilos. Los ocho hilos están divididos en dos grupos de cuatro; cada grupo comparte un único pipeline para enteros. Mientras los ocho hilos se ejecutan simultáneamente en cualquier tiempo específico, a lo más dos hilos pueden estar activos en un núcleo y éstos pueden estar ejecutando alguna de las siguientes combinaciones: un par de operaciones de pipeline para enteros; una operación con enteros y una de punto flotante; una operación con enteros y una operación de memoria; o una operación de punto flotante y una operación de memoria.

Históricamente los procesadores se han diseñado con el objetivo que un programa cualquiera que se ejecute en el procesador tenga el mejor rendimiento posible, poniendo énfasis en las aplicaciones que se utilizan hoy en día con mayor frecuencia, que son las que conocemos como aplicaciones de escritorio. Por lo tanto los diseñadores se han enfocado en desarrollar arquitecturas que ejecuten eficientemente un programa secuencial, esto es, un programa que trabaja con un solo hilo de ejecución. La mejora en el desempeño en ese tipo de procesadores se ha dado mediante una técnica de hardware que ya mencionamos, que consiste en la adición de un pipeline e ir aumentando el número de etapas dentro de él, lo que ha dado como resultado que se ejecuten múltiples instrucciones en paralelo (*ILP, Instruction Level Parallelism*). El principio básico de explotar el paralelismo a nivel de instrucción ha dado como resultado un pipeline con demasiadas etapas, hasta llegar el punto en que el rendimiento ha disminuido en lugar de incrementarse y, como consecuencia, los procesadores actuales no utilizan el hardware de manera eficiente. Se ha observado que en la mayoría de las aplicaciones el procesador pasa la mayor parte de su tiempo de vida ocioso, incluso cuando está en ejecución, y sólo utiliza una pequeña fracción de su capacidad de cómputo. Por ello los ingenieros de Sun, junto con la comunidad del software libre, en vez de diseñar un procesador complejo con ILP que se encuentre en descanso la mayor parte del tiempo, diseñaron un nuevo procesador con múltiples núcleos en un único chip, cuyo objetivo es que cada uno de ellos realice tareas sencillas, en programas que hagan uso de múltiples hilos en ejecución. Cada uno de estos núcleos, a su vez, tiene la capacidad de dividir su trabajo en varios hilos de ejecución independientes. La colaboración de múltiples hilos en la ejecución de un programa permite un alto desempeño en aplicaciones que dividen su trabajo en múltiples hilos. Este nuevo enfoque se llama *paralelismo a nivel de hilo* (*Thread Level Paralelism*, *TLP*). TLP permite que mientras se esté realizando una operación que involucre un acceso a la memoria, otro hilo del programa puede ejecutarse completamente en paralelo, a diferencia de las arquitecturas con ILP, que tienen que esperar a que la operación en la memoria termine, lo que involucra que estén sujetos a la latencia de la memoria. La desventaja de esta arquitectura es que los programas que estén escritos para ser ejecutados en un único hilo tendrán un mejor desempeño en una arquitectura con ILP, mientras que las aplicaciones que utilicen múltiples hilos de ejecución tendrán

un mejor desempeño en un procesador con TLP. El procesador UltraSPARC T2 es un procesador que tiene TLP. En marzo de 2006 Sun Microsystems liberó el procesador *UltraSPARC T1* como hardware libre con el nombre de *OpenSPARC T1*, poniéndolo al alcance de grupos de investigación o de cualquier persona interesada en el desarrollo de arquitecturas; posteriormente desarrollaron el procesador UltraSPARC T2 y su versión libre *OpenSPARC T2*. Como mencionamos, el UltraSPARC T2 está basado en una arquitectura descendiente de la arquitectura SPARC. Primero apareció la arquitectura SPARC versión 7 (V7) de 32 bits, liberada en 1987; un poco más tarde se anunció la versión 8 y fue publicada en un libro. La arquitectura SPARC versión 9 de 64 bits se liberó en 1994. Y ahora la arquitectura *UltraSPARC* marca la primer actualización significativa desde la versión 9. Algunas de las características que comparte la arquitectura UltraSPARC 2007 con su predecesora, la arquitectura SPARC V9, son las siguientes:

- Un espacio lineal de direcciones de 64 bits.
- Instrucciones de 32 bits.
- Pocos modos de direccionamiento.
- Predicción de saltos.
- Instrucciones de sincronización entre múltiples procesadores.

Algunas de las características que tiene la arquitectura UltraSPARC 2007 y que no están presentes en la arquitectura SPARC V9 son:

- Modo hiper-privilegiado.
- Múltiples niveles de registros globales.
- Conjunto extendido de instrucciones.
- Multihilos a nivel de chip (*Chip Level Multithreading*, *CMT*).

La arquitectura UltraSPARC 2007 cumple completamente con el nivel 1 (no privilegiado) de SPARC V9 y parcialmente con el nivel 2 (privilegiado) e incluye numerosas extensiones. Trataremos a detalle la arquitectura SPARC en la etapa de generación de código de nuestro compilador.

Por último toca el turno de elegir el lenguaje de implementación. Para la elección del lenguaje de implementación hemos puesto atención en que sea un lenguaje eficiente, que se use en compiladores en producción, que sea modular y que permita utilizar técnicas de ingeniería de software. Todo esto nos llevó a elegir C++. En el camino a esta decisión hemos tomado en cuenta otros lenguajes, por ejemplo C que es un lenguaje eficiente,

pero no es tan amigable para utilizar técnicas de ingeniería de software; Java puede ser una buena elección en este sentido, pero su problema es la eficiencia; hay varios libros en la literatura que utilizan Java en un sentido didáctico al desarrollar compiladores, pero no lo hemos elegido debido a que no es un lenguaje que se utilice en compiladores en producción y hemos tenido el cuidado de construir un compilador con las características esenciales de un compilador en producción. C++ resulta ser un lenguaje que cumple con nuestras expectativas: es eficiente, modular y orientado a objetos; permite acceso a instrucciones de bajo nivel (que resultan ser necesarias en algunos puntos de la construcción de un compilador); y hay varios compiladores altamente optimizadores en producción que están escritos en C++.

1.3.3. Organización y arquitectura

Describiremos ahora la organización y arquitectura del compilador que construiremos.

Podemos clasificar el diseño e implementación de compiladores en dos escuelas: la funcional y la imperativa.

A la funcional pertenecen las implementaciones de compiladores para lenguajes como Scheme, *ML* y *Haskell*; comúnmente el diseño de los compiladores para estos lenguajes está basado en una representación intermedia conocida como *CPS* (*Continuation Passing Style*) que se presentó por primera vez en la descripción de Scheme (en la cual también se menciona la descripción del primer intérprete de Scheme) [SS98]. Posteriormente se usó en la implementación del primer compilador de Scheme (*Rabbit*) descrito en [Ste78]. CPS es una representación intermedia que expresa explícitamente el control de flujo y de datos en un programa y está estrechamente apegada al cálculo lambda; por lo anterior CPS es una representación que permite realizar diversas optimizaciones y puede utilizarse como preámbulo para la generación de código. En la escuela imperativa en cambio se encuentran los compiladores para lenguaje como Java, C, C++, que hacen uso (algunos de ellos) de una representación intermedia llamada *SSA* (*Static Single Assigment*) que es la base para realizar diferentes tipos de optimizaciones. Existe una correspondencia entre ambas (CPS y SSA) descrita en [Kes95] y en [App98]: todo programa en SSA se puede convertir a CPS y la mayor parte de los programas en CPS se pueden convertir a SSA (pero no todos). Además, en la escuela imperativa generalmente se hace uso de la pila con la que cuenta la arquitectura para implementar las llamadas a funciones, mientras que en la escuela funcional generalmente se hace uso de técnicas más sofisticadas que no necesariamente hacen uso de la pila.

Cabe señalar que, por supuesto, los compiladores no están obligados a utilizar alguna de estas representaciones, y que podría haber compiladores para lenguajes funcionales que utilicen SSA como representación intermedia y compiladores para lenguajes imperativos que utilicen CPS o, incluso, compiladores (para lenguajes funcionales o im-

perativos) que utilicen ambas (aunque esto no es común).

Python cuenta tanto con características imperativas como funcionales. Hemos elegido seguir el modelo imperativo, debido a que Python tiene principalmente características imperativas y es orientado a objetos, así que en ese sentido se conjuga de manera natural con el modelo imperativo. Por otra parte, el modelo imperativo resulta ser más intuitivo y es ampliamente utilizado en muchos de los compiladores en producción. Sin embargo, en una segunda fase a futuro del desarrollo de nuestro compilador deseamos utilizar CPS al estilo funcional para implementar las características funcionales de Python (como son las funciones de primer orden).

Nuestro compilador cuenta con un diseño principal que tiene algunas variantes llegada la hora de generar código; estas variantes se incluyeron para darle la libertad al lector (o los estudiantes de un curso de compiladores) de elegir la que mejor se adecúe a sus necesidades con base en sus conocimientos, aptitudes e intereses.

El diseño de nuestro compilador es modular, está compuesto por diferentes etapas, donde a cada etapa le corresponde un módulo; las diferentes etapas de un compilador se pueden comunicar mediante estructuras de datos en memoria o mediante archivos de texto o binarios que tengan una cierta codificación. La primera opción es más eficiente y la segunda es más modular y permite que el programador pueda depurar de manera más clara e independiente cada una de las etapas del compilador. El problema con esta estrategia es que cada etapa del compilador debe releer el programa del archivo que generó la etapa inmediata anterior, lo que representa un problema de eficiencia. Hemos seleccionado una estrategia mixta que nos permite beneficiarnos de ambos enfoques: las etapas del compilador se comunicarán por medio de estructuras de datos en memoria, pero opcionalmente cada uno de las etapas generará un archivo de salida (que contiene el programa con la correspondiente transformación realizada por la etapa); de esta forma el compilador es eficiente y opcionalmente permite al programador ver (y corregir, en su caso) la salida que está generando la etapa en cuestión, además que permite ver a los estudiantes explícitamente la nueva representación del programa.

En el diseño principal de nuestro compilador se realizarán las siguientes etapas:

1. Análisis léxico.
2. Análisis sintáctico.
3. Análisis semántico.
4. Optimizaciones (opcional).
5. Generación de código intermedio de bajo nivel.
6. Optimizaciones (opcional).
7. Generador de código dependiente de la arquitectura.

8. Optimizaciones dependientes de la arquitectura (opcional).

La separación de nuestro compilador en las etapas anteriores obedece a que de esta forma se logra un diseño modular y limpio y por tanto es más fácil depurar, corregir errores, y agregar, eliminar o reemplazar alguno de sus módulos. En este sentido existen compiladores que realizan todas sus fases en una única pasada y por tanto en un único módulo, que se conocen como compiladores dirigidos por la sintaxis y que generan código "al vuelo"; usualmente no realizan optimizaciones y comparten su diseño con los intérpretes dirigidos por la sintaxis (descritos anteriormente), sólo que en lugar de ejecutar un enunciado generan el código de máquina correspondiente a dicho enunciado. A diferencia de este tipo de compiladores existen compiladores que separan cada una de sus etapas en diferentes módulos y que además agregan varias etapas para realizar optimizaciones; como hemos descrito antes, existen dos diseños que siguen esta estrategia –nuestro compilador está basado en el segundo que se muestra en la Figura 1.1(b) en la página 13, es decir, nuestro compilador realiza opcionalmente optimizaciones sobre una representación intermedia de nivel intermedio SSA y en una segunda etapa del compilador se le puede agregar opcionalmente un optimizador que tome como entrada la representación intermedia de bajo nivel que en nuestro compilador serán árboles de expresiones, además de otra etapa opcional de optimizaciones que tome como entrada ya sea el código ensamblador y el código binario que se generó para una arquitectura específica y realice sobre éste optimizaciones dependientes de la arquitectura y optimizaciones de mirilla–.

A continuación describiremos brevemente las tareas que realizará cada una de las etapas de nuestro compilador.

El analizador léxico es nuestro primer componente en el proceso de compilación; su función es tomar como entrada el código fuente, limpiarlo (quitar todos aquellos caracteres que no sean relevantes) y transformarlo en unidades léxicas indivisibles que le sean útiles al analizador sintáctico. Nuestro analizador léxico lo desarrollaremos mediante una herramienta llamada `flex`, un generador de analizadores léxicos. El analizador sintáctico toma como entrada las unidades léxicas generadas por el analizador léxico y verifica que los enunciados del programa estén bien formados (i.e. que el código fuente cumpla con la sintaxis del lenguaje, en este caso Python); además, el analizador sintáctico se encarga de construir un árbol de sintaxis abstracta (*Abstract Syntax Tree*, *AST*), una representación intermedia de alto nivel estrechamente apegada a la sintaxis del programa) siendo ésta la salida de nuestro analizador; este módulo del compilador (el analizador sintáctico) lo construiremos haciendo uso de `bison`, un generador de analizadores sintácticos. Posteriormente el analizador semántico trabajará sobre el AST e implementaremos dentro de él una adaptación del algoritmo del producto cartesiano[Age96b], un algoritmo de inferencia de tipos que nos evita hacer verificaciones de tipos en tiempo de ejecución (aunque siempre que no sea posible hacer esta inferencia se realizarán las correspondientes verificaciones en tiempo de ejecución). Implementaremos además

las restantes tareas encargadas de verificar que el programa sea semánticamente válido. Luego tendremos una etapa opcional, la cual toma como entrada el AST decorado que entrega el analizador semántico y ésta se encargará de construir la forma SSA y realizar todo tipo de optimizaciones posibles con base en la información con la que se cuenta en ese nivel de abstracción. Esta etapa tendrá como salida un AST decorado, por lo que será transparente si es que ésta se lleva a cabo o no.

En este punto el diseño de nuestro compilador podría tomar diferentes caminos, que se mencionan a continuación:

1. Generar código ensamblador para la máquina virtual java (JVM).
2. Generar código de máquina para una arquitectura específica.
3. Generar código de bajo nivel en forma de árboles de bajo nivel (*low-level trees*), una representación intermedia de bajo nivel que utilizaremos en nuestro compilador.

Si lo que se desea es construir el compilador en el menor tiempo posible y sin involucrarse en la elaboración de un sistema en tiempo de ejecución (*run-time system*), se debe elegir la primera opción, ya que en ella se genera directamente código ensamblador de la máquina virtual java; las instrucciones con las que cuenta la máquina virtual nos liberan de varias tareas que debemos realizar en las otras opciones; por ejemplo, si debemos crear un objeto, la máquina virtual cuenta con las instrucción *new* que crea un nuevo objeto en memoria de la clase que especifiquemos; en cambio, en las otras alternativas debemos incluir instrucciones explícitas que asignen espacio en memoria para el nuevo objeto y esto (y muchas otras tareas) se realizan a través de un sistema en tiempo de ejecución, que involucra conocer a fondo la arquitectura objetivo y la interfaz provista por el sistema operativo; por otra parte su pueden realizar únicamente algunas pocas optimizaciones sobre el código ensamblador de la máquina virtual.

Si apremia el tiempo pero se desea construir un compilador que genere código nativo (que no dependa de una máquina virtual) y sólo interesa generar código para una única arquitectura, la segunda opción debe ser la elección. Este es el esquema que se sigue en varios de los textos didácticos. En él se genera directamente código de máquina y es necesario construir un sistema en tiempo de ejecución. Este esquema resulta ser didáctico porque se implementan todas las etapas necesarias de un compilador funcional nativo y en el menor tiempo posible; normalmente no se realizan optimizaciones excepto quizás algunas sobre el código de máquina generado, esto es, se implementan solo el mínimo número necesario de etapas y solo se genera código para una única arquitectura; sin embargo, este esquema no es el que se sigue en los compiladores en producción, ya que generalmente no cuenta con optimizaciones o no son suficientes y resulta ser difícilmente reobjetivizable; así que este esquema queda en el campo de los compiladores "reales" pero meramente didácticos.

La última opción consiste en generar código de bajo nivel, lo que representa diversas ventajas sobre las opciones anteriores. La principal ventaja de generar código de bajo nivel es que nuestro compilador se vuelve fácilmente reobjetivizable. A cambio de ello se necesita implementar un sistema en tiempo de ejecución (uno por cada arquitectura soportada); además implica agregar al menos una etapa más en el compilador, es decir, se necesita primero generar código de bajo nivel y a partir de éste realizar una etapa que genere código de máquina específico, o sea un generador de código dependiente de la plataforma y se debe construir uno de éstos para cada arquitectura distinta que soporte el compilador. Una vez que se haya generado código de máquina[13] se pueden realizar optimizaciones dependientes de la arquitectura sobre este código; lo anterior, a priori, representa un mayor trabajo de implementación; sin embargo, nosotros haremos uso de una herramienta de nombre `burg` para implementar la etapa de generación de código[14] dependiente de la plataforma a partir de árboles de bajo nivel, que es el código de bajo nivel que usaremos. `burg` es un generador de generadores de código basado en la teoría *BURS* (*Buttom-Up Rewrite System*); la idea detrás de BURS consiste en que dada la representación del programa en árboles de bajo nivel se encuentra el enlosetado de mínimo costo tal que cubra por completo el árbol; esto se lleva cabo dando como entrada al generador de generadores de código una gramática (con atributos) con lenguaje de máquina, que contiene los patrones de árboles a empatar junto con un costo asociado para cada patrón (enlosetado local). El generador de generadores de código utiliza empate de patrones de árboles (*tree-pattern matching*) y programación dinámica, realiza dos pasadas sobre el árbol de bajo nivel, la primer pasada es de abajo hacia arriba (*bottom-up*) y encuentra el conjunto de patrones (losetas) de costo mínimo que cubren el árbol; la segunda pasada se realiza de arriba hacia abajo (*top-down*) y va ejecutando las acciones semánticas asociadas con cada una de las losetas, que son generalmente emitir el código de máquina que corresponde a cada una de las losetas; así, por medio de `burg` la generación de código de máquina se vuelve más clara, limpia y fácil de modificar; por supuesto que de esta forma el compilador es fácilmente reobjetivizable, esto es, para soportar una nueva arquitectura debemos construir una nueva gramática de dicha arquitectura con sus respectivas acciones semánticas y un sistema en tiempo de ejecución. Esto, comparado con la opción anterior, incluso puede llegar a resultar en un tiempo de implementación menor, ya que corregir y hacer modificaciones en el esquema anterior resulta más tedioso y complicado. Este es el esquema principal que seguiremos en nuestro compilador.

Hay varios aspectos importantes que debemos resaltar del diseño de nuestro compi-

[13] En realidad nosotros podemos generar código de máquina o código ensamblador para cada una de las máquinas objetivo.

[14] En realidad en la etapa de generación de código se realizan tres tareas principales: selección de instrucciones, alojamiento de registros y planificación de instrucciones. En este párrafo utilizamos el término *generación de código* para referirnos únicamente a selección de instrucciones.

lador. Primero hemos decidido utilizar, siempre que sea posible, una herramienta que automatice la etapa en cuestión, debido a que, por una parte, implementar una etapa "a pie" normalmente toma más tiempo, esfuerzo y es más difícil de corregir, de depurar y modificar; por otra parte, las herramientas que usaremos descansan sobre una base teórica sólida y en la mayoría de los casos generan código igual o más eficiente que una implementación de la etapa hecha a pie . Estas herramientas han sido previamente utilizadas en el desarrollo de otros compiladores; la motivación que ha hecho que se desarrollen este tipo de herramientas es precisamente hacer el periodo de implementación de un compilador más corto, hacer más fácil la tarea de corregir y depurar errores, y facilitar la modificación y ampliación del compilador, por ejemplo para agregar nuevos enunciados en la sintaxis. Como el lector habrá notado ya, nuestro diseño no hace uso de una herramienta en el analizador semántico; esto es debido a que si bien ha habido varios intentos de automatizar esta etapa, no consideramos que exista una herramienta en amplio uso que automatice el análisis semántico; además, nuestro analizador semántico cuenta con características particulares (como el algoritmo de inferencia de tipos) que nos conducen a hacer una implementación a pie.

Una parte fundamental de nuestro compilador son las tareas involucradas en el analizador semántico. Python, al igual que Scheme, cuenta con tipado dinámico,[15] esto es, la clase de los valores se conoce únicamente en tiempo de ejecución, (es decir, hasta que se ejecuta el programa) y no antes; por esta razón, en principio, el analizador semántico no puede realizar verificación de tipos en tiempo de compilación y debe delegar la verificación de las clases de valores al sistema en tiempo de ejecución, o sea que el compilador debe generar código que haga verificación de clases de valores en tiempo de ejecución; es por lo anterior que generalmente los lenguajes que cuentan con tipado dinámico se implementan por medio de intérpretes y no de compiladores, donde en lugar de generar código que realice la verificación de tipos en tiempo de ejecución, es el intérprete quien funge como un supervisor que realiza esta función. Esta implementación del lenguaje es más sencilla, aunque hemos visto ya que la verificación de tipos en tiempo ejecución aumenta de manera considerable el tiempo de ejecución de un programa y esto, aunado a las tareas que realiza el intérprete en tiempo de ejecución, da como resultado que las implementaciones[16] de lenguajes como Python y Scheme degraden el desempeño en tiempo de ejecución de los programas que se ejecutan en ellas. Por esto,

[15] Suele mencionarse en la literatura que en un lenguaje con tipado dinámico la verificación de tipos se realiza en tiempo de ejecución; sin embargo esto no es correcto, como lo estudiaremos en el capítulo de análisis semántico. A pesar de ello hemos utilizado y utilizaremos el enunciado verificación de tipos en tiempo de ejecución, con el que queremos decir (debe leerse) verificación de clases de valores en tiempo de ejecución.

[16] Aquí nos referimos a la mayoría de las implementaciones, no a todas.

a este tipo de lenguajes usualmente se les cataloga como lenguajes lentos en comparación con implementaciones de lenguajes estáticamente y explícitamente tipados como FORTRAN, C o C++. Para remediar un poco esta situación, existen compiladores (para lenguajes con tipado dinámico) que optan por el camino de generar código que realiza la verificación de tipos en tiempo de ejecución. De esta forma, el tiempo de ejecución de un programa se libera de las otras tareas para tiempo de ejecución realizadas por el intérprete (por ejemplo el análisis léxico y sintáctico); en su lugar estas tareas se realizan en tiempo de compilación y la única sobrecarga que se tiene en tiempo de ejecución es la verificación de tipos, con lo que se logran implementaciones por medio de compiladores que generan programas que se ejecutan con un mejor desempeño; algunos de estos compiladores van más allá y ofrecen una interfaz interactiva con el usuario al estilo de un intérprete. En particular, cuando se les presenta un enunciado lo compilan y después lo ejecutan, aunque también pueden compilar un programa completo almacenado en un archivo, para después ejecutarlo una o cuantas veces se requiera. Algunos de ellos generan incluso de forma explícita un archivo binario ejecutable.[17] Con este tipo de compiladores tenemos la ventaja que podemos manejar programas que hagan uso de la función `eval` al estilo LISP, ya que en tiempo de ejecución podemos realizar la compilación y ejecución "al vuelo" del código que sea el argumento de la función `eval`; a este tipo de compiladores se les conoce como *compiladores incrementales*. Y precisamente por las razones anteriores hemos decidido seguir este modelo, por lo que nuestro compilador será un compilador incremental; pero a diferencia de compiladores como *Ikarus* [Ghu] y *Twobit* [CH94], que generan en todos los casos código que realiza la verificación de tipos en ejecución, nosotros hemos elegido incluir un algoritmo de inferencia de tipos en tiempo de compilación; de esta forma llevamos a cabo, siempre que sea posible, la verificación de tipos en tiempo de compilación y sólo en aquellos enunciados o fragmentos de código en los que no sea posible realizar inferencia de tipos, el compilador generará código que verifique tipos en tiempo de ejecución. Con esta estrategia buscamos obtener una implementación con un mejor desempeño en tiempo de ejecución de los programas escritos en Python y con el objetivo de que sea un compilador que se use en la vida real.

Por otra parte, el análisis semántico usualmente se implementa realizando un análisis de casos en base al tipo de nodo del AST: dependiendo del tipo de categoría sintáctica que represente un nodo en el AST son las acciones que realiza sobre él; por ejemplo, si se trata de un nodo que represente a un `if`, entonces el analizador semántico verifica que el primero de sus hijos se trate de un enunciado booleano, y si no es así emite un error. Para llevar a cabo esta labor el analizador sintáctico debe recorrer el árbol y realizar un análisis de casos para cada uno de los nodos del árbol, que en algunos casos se

[17] Que suele depender del sistema en tiempo de ejecución incrustado en el compilador; es por esta razón que usualmente no se pueden ejecutar directamente en el sistema operativo y se deben ejecutar con ayuda del sistema en tiempo de ejecución del compilador.

implementa como un `switch` enorme; en lugar de esto, nosotros implementaremos el patrón Visitante (*Visitor*) que nos sirve para recorrer el árbol y que cada nodo acepte un visitante que realice acciones específicas en base al tipo de nodo. De esta manera, conforme se va recorriendo el árbol, el visitante de cada uno de los nodos actúa en consecuencia de forma limpia y transparente sin hacer uso del ineficiente `switch`.

CAPÍTULO 2

Análisis léxico

Conocimientos supuestos

Para la etapa del análisis léxico estamos suponiendo que el lector conoce lo siguiente:

1. Definiciones de:
   - Alfabeto, cadena, lenguaje.
   - Gramáticas y su clasificación.
   - Autómata finito determinista (AFD).
   - Autómata finito no–determinista (AFN).
   - Autómata finito con transiciones-ε (AF$_\varepsilon$).
   - Expresiones regulares (ER).
2. Las operaciones de unión, concatenación y cerradura de Kleene entre cadenas y entre lenguajes.
3. Los algoritmos para:
   - Transformar un AFN en un AFD.
   - Transformar un AF$_\varepsilon$ en un AFN.
   - Minimizar y reducir autómatas finitos.
   - El algoritmo de Thompson y de McNaughton y Yamada para construir autómatas finitos a partir de expresiones regulares.

4. Cómo implementar las siguientes estructuras de datos:
   - Listas.
   - Colas.
   - Pilas.
   - Árboles.
   - Funciones y tablas de dispersión (*hash*).

Los primeros temas se pueden encontrar en cualquier libro de texto sobre autómatas y lenguajes formales, como [HU79] y [Vis08]. El último tema se puede consultar en libros sobre la implementación de estructuras de datos, como [Bud00] y [AHU83].

2.1. Introducción

El análisis léxico es la primer fase que se lleva a cabo en el proceso de compilación y el analizador léxico es el módulo del compilador que se encarga de realizarla; la labor de este último consiste en leer el código fuente (que puede ser visto como un flujo de caracteres), ir agrupándolo en cadenas (a las cuales se les denomina *lexemas*) e indicar a qué unidad léxica válida pertenece cada uno de los lexemas; en caso de que no pertenezca a ninguna, reportar un error léxico; las unidades léxicas válidas también se conocen como *átomos*.[1] Un átomo es un par *(nombre,atributo)*, donde *nombre* corresponde a un símbolo terminal de la gramática que especifica la sintaxis del lenguaje y *atributo* es un campo opcional, que almacena información útil acerca del átomo encontrado y que es necesaria (cuando está presente) en las subsecuentes etapas del proceso de compilación.

Un átomo a su vez representa un conjunto de cadenas válidas que conforman una misma entidad o categoría léxica; este conjunto resulta ser un conjunto regular por lo que se puede denotar mediante una expresión regular.

Hemos mencionado ya que el analizador léxico forma lexemas y tiene la tarea de determinar a qué átomo corresponde cada lexema. Para determinar si un lexema pertenece al conjunto representado por un átomo se necesita un reconocedor que sea capaz de reconocer el conjunto, esto es, para cada átomo debemos tener una máquina reconocedora tal que al alimentarle un lexema nos indique si dicho lexema pertenece al conjunto que representa el átomo; de ser así el analizador léxico reporta una presencia del átomo y en caso de que el átomo cuente con un atributo, el analizador léxico lo almacena;[2] en caso contrario se debe probar con el siguiente reconocedor (que corresponde al siguiente átomo). Si se agotan todas las posibilidades y no hay aceptación, se reporta un error léxico.

Es natural hacer uso de los autómatas finitos como máquinas reconocedoras (pues se trata de reconocer conjuntos regulares) para llevar a cabo esta tarea.

Si bien podemos construir a pie cada uno de los autómatas finitos correspondientes a

[1] El equivalente al término en inglés *tokens* que nosotros usaremos.

[2] En un tabla de símbolos que estudiaremos a profundidad más adelante.

cada uno de los átomos, sabemos que cada conjunto de cadenas (lenguaje) representado por un átomo se puede expresar mediante una expresión regular, de ésta obtener algorítmicamente un autómata finito con transiciones–ε, de acá obtener el AFD equivalente, y por último obtener el autómata reducido correspondiente.

2.2. Construcción algorítmica de analizadores léxicos

Es fácil notar que, usando los algoritmos para realizar cada una de las tareas que mencionamos en el párrafo anterior, podemos automatizar el proceso de creación de un analizador léxico mediante los siguientes pasos:

1. Expresar cada uno de los cojuntos de cadenas representados por su respectivo átomo mediante una expresión regular.
2. Obtener el AFN correspondiente a cada una de las expresiones regulares del paso anterior.
3. Sean $N_1, \ldots, N_i, \ldots, N_n$ los AFN obtenidos en el paso anterior y $s_1, \ldots, s_i, \ldots, s_n$ sus correspondientes estados iniciales; construimos un nuevo AFN N con estado inicial s, que represente a la unión de los AFN $N_1, N_2, \ldots, N_n$, tal que para cada s_i exista una $transición$–ε de s a s_i, como se muestra en la figura 2.1.
4. Aplicar el algoritmo de conversión de AFN a AFD con entrada N, para obtener el AFD M correspondiente.
5. Aplicar el algoritmo de minimización con entrada M obteniendo así M', el AFD mínimo.

Sin embargo, debemos estar atentos a ciertas situaciones que surgen en el procedimiento anterior. Por ejemplo, consideremos los átomos `LT`, `GT`, `LE`, `GE`, que representan a los lenguajes cuyas expresiones regulares son: `<`, `>`, `<=`, `>=`, respectivamente, correspondientes a algunos de los operadores de comparación en Python, y supongamos que tenemos en la entrada actual la cadena `<= 5`; es claro que cuando se consuma el carácter `<` el AFD del paso 4 se encontrará en un estado final; ¿debemos en este punto reportar que hemos encontrado el átomo `LT`? La respuesta es no; en lugar de ello debemos continuar consumiendo caracteres de la entrada hasta que no exista una transición del estado actual bajo el carácter de entrada actual; en este punto se dice que hemos alcanzado la condición de *terminación*; de esta forma, en nuestro ejemplo se continúa leyendo el carácter `=` que deja al AFD en otro estado final y posteriormente se encuentra un carácter espacio en blanco en la entrada, bajo el cual no existe una transición en el estado actual y por tanto se debe revisar cuál fue el último estado final que se visitó y reportar una presencia del átomo correspondiente a tal estado, en este caso `LE`. Con esta estrategia, garantizamos que siempre se empata el lexema de mayor longitud válido que pertenezca al lenguaje representado por un átomo dado.

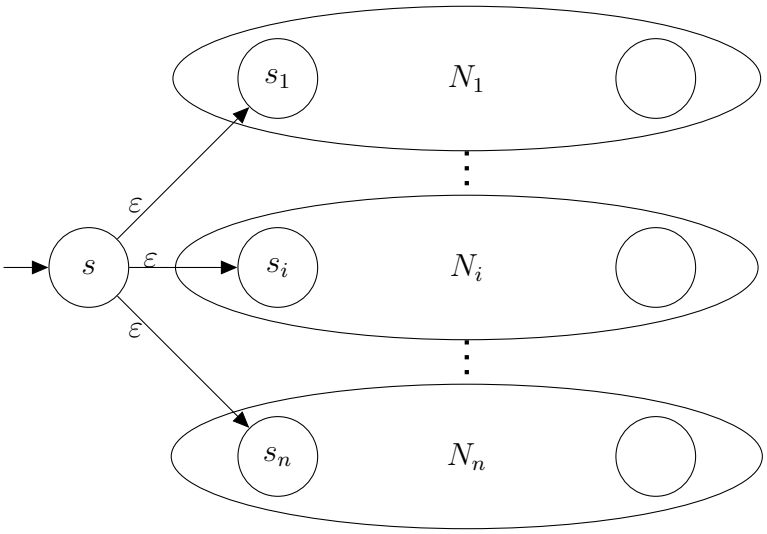
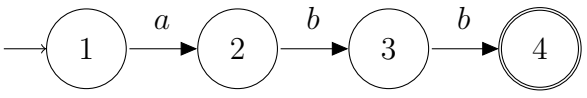


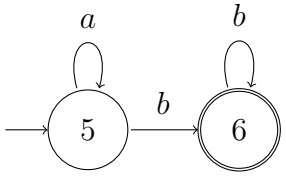
Figura 2.1: AFN que acepta múltiples átomos.

Por otra parte, es claro que en los AFN del paso 2, los estados finales de cada uno de ellos indican la presencia de su respectivo átomo, pero cuando se lleva a cabo la conversión a AFD del AFN del paso 3, se realizan construcciones de subconjuntos de estados; de esta forma, un subconjunto q_f obtenido puede contener dos o más estados finales correspondientes a diferentes AFN y por tanto a diferentes átomos. El subconjunto, por supuesto, es un estado final del AFD; el problema aquí radica en que cuando se deba reportar la presencia de un átomo en el estado q_f ¿qué átomo debemos reportar? Para solucionar este problema estableceremos la convención de reportar el átomo según el orden en que se listan sus respectivas expresiones regulares; de esta manera, si q_f representa a dos estados finales q_{f_1} y q_{f_2} de diferentes AFN N_1 y N_2 y con r_1, r_2 sus respectivas expresiones regulares, si r_1 se lista primero que r_2 se debe reportar el átomo del estado q_{f_1}.

Veamos un ejemplo sencillo pero ilustrativo. Supongamos que tenemos dos átomos t_1, t_2 con sus respectivas expresiones regulares $r_1 = abb$ y $r_2 = a^*b^+$; es claro que los siguientes autómatas N_1 y N_2 reconocen a los lenguajes $L(r_1)$ y $L(r_2)$ respectivamente.



(a) N_1 que reconoce el lenguaje denotado por abb.



(b) N_2 que reconoce el lenguaje denotado por a^*b^+.

Figura 2.2: AFN que reconocen los lenguajes denotados por r_1 y r_2 respectivamente.

A partir de estos autómatas construimos el AFN que se muestra a en la figura 2.3.

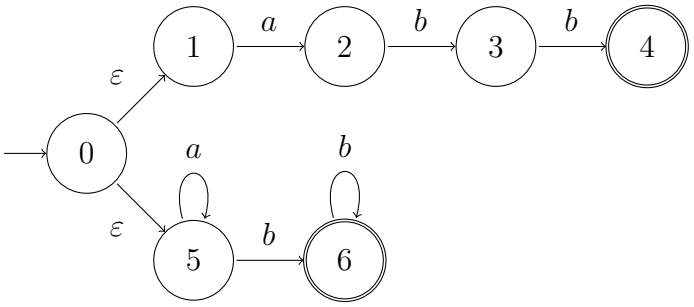


Figura 2.3: AFN N que reconoce $L(r_1)$ y $L(r_2)$.

A partir del AFN de la figura 2.3 obtenemos el AFD de la figurea 2.4.

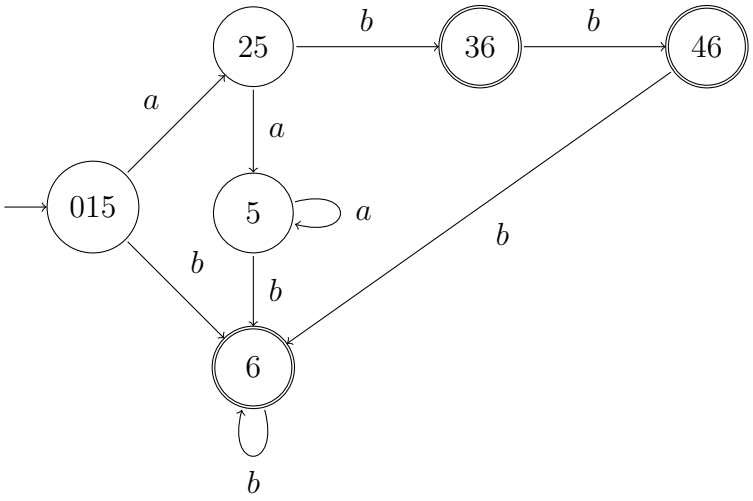


Figura 2.4: Autómata finito M que se construyó a partir de N.

Debemos notar que el autómata $M = (Q, \Sigma, \delta, q_0, F)$ que acabamos de presentar no presenta un estado pozo p que definiría completamente las transiciones del autómata para cada $a \in \Sigma$ para el que no hubiese transición definida en el autómata original. Estas transiciones, que completarían la función de transición en el autómata determinista, no se agregan en este caso, debido por una parte a razones de eficiencia y por otra a que de esta forma podemos implementar de forma natural la regla de empatar el lexema de mayor longitud, en la que debemos consumir caracteres de la entrada hasta que no haya una transición del estado actual con el carácter actual de la entrada.

Supongamos que tenemos la cadena de entrada abb; al consumirse todos los caracteres de esta cadena el AFD se encuentra en el estado 46; debemos resaltar que el estado

46 se formó a partir del estado final 4 del AFN N_1 y del estado final 6 del AFN N_2; de esta manera nos preguntamos ¿qué átomo debemos reportar? ¿t_1 o t_2? Como dijimos anteriormente la convención que seguiremos es reportar el átomo correspondiente a la expresión regular que se liste primero; en este caso, como r_1 se listó primero que r_2, se debe reportar t_1. Nótese que en otros casos, como con la entrada ab, el autómata alcanza un estado en el que no hay conflictos a resolver; en este caso el autómata alcanza el estado 36 donde solo el estado 6 corresponde a un estado final y por tanto es claro que se debe reportar el átomo t_2.

De lo anterior podemos concluir que debemos seguir las siguientes reglas:

1. Empatar el lexema de longitud máxima que pertenezca a algún lenguaje representado por un átomo determinado. Esto se realiza consumiendo caracteres de la entrada hasta que no haya una transición del estado actual bajo el carácter actual de la entrada; una vez que se llega a esta condición, se debe verificar cuál fue el último estado final que se visitó y reportar el átomo asociado con él (por supuesto los caracteres que no formen parte del lexema empatado y que hayan sido consumidos temporalmente deben devolverse a la entrada). En el caso en que no se haya visitado ningún estado final debe reportarse un error.
2. En caso de que exista más de un átomo a reportar, se debe reportar el átomo que representa a aquella expresión regular que se listó antes que todas aquellas que representan a los otros átomos.

Debemos además prestar atención cuando se lleve a cabo el algoritmo de minimización, pues éste comienza con una partición donde todos los estados finales pertenecen a una misma clase y todos aquellos no finales pertenecen a otra distinta. Aquí debemos hacer una ligera modificación y poner en clases diferentes aquellos estados finales que no compartan el mismo átomo a reportar, para de esta manera garantizar que el AFD mínimo que obtengamos no contenga algún estado que se haya formado a partir de dos o más estados finales del AFD de entrada y debido a esto se generen ambigüedades al reportar átomos. En nuestro ejemplo anterior, debemos comenzar colocando los estados $015, 25$ y 5 en una misma clase pues no son estados finales. En cuanto a los estados finales, los estados 36 y 6 reportan el átomo t_2 por lo que deben permanecer en la misma clase, no así el estado 46 que aunque originalmente contenía un conflicto entre los átomos t_1 y t_2 éste se resolvió en favor de t_1 y por tanto debe colocarse en una clase diferente. De esta forma el algoritmo debe comenzar con una partición de los estados en las clases: $C_0 = \{015, 25, 5\}$, $C_1 = \{36, 6\}$ y $C_3 = \{46\}$.

En nuestro ejemplo el algoritmo termina con el mismo AFD como salida, lo que indica que el AFD de la figura 2.4 es el AFD mínimo.

2.2.1. Detalles de implementación

Veamos ahora algunas estrategias para llevar a cabo la implementación de un analizador léxico. Cuando se trata de un número finito de lexemas es conveniente colocarlos en una tabla, ya sea mediante una función de dispersión u ordenados lexicográficamente –hablaremos más extensamente de esto– para poder utilizar una comparación eficiente.

Si suponemos que el número de lexemas es infinito, lo que sucede en la mayoría de los lenguajes de programación que tienen tamaños no acotados para sus identificadores, debemos fijarnos en cómo implementar un AFD y existen diversas estrategias para este fin. Quizás la más ingenua es realizar la implementación mediante enunciados **`switch`** anidados, donde contamos con una variable `entrada` para manejar el símbolo leído, y una variable `estado` que maneja el estado actual. Siguiendo esta estrategia terminamos con algo parecido a lo que mostramos a continuación:

```
state actual = S0
char entrada
while haya caracteres en la entrada
  entrada = sgtChar()
  switch state
    case 0:
      switch entrada
        case a:
          actual = S1
          continue
        case b:
          actual = S2
          continue
    case 1:
      swtich entrada
        case a:
          actual = S0
          continue
        case b:
          actual = S2
          continue
    case 2:
       switch entrada
         case a:
           continue
         case b:
           actual = S1
           continue
```

Código 2.1: Implementación de un AFD manejando explícitamente el estado actual en una variable

De esta forma tenemos un único **switch** para los estados y un **switch** en cada estado para las transiciones (alternativamente podemos utilizar el **switch** externo para las transiciones y los **switch** internos para los estados) con lo que obtenemos enunciados **switch** anidados con profundidad dos. La estrategia presentada es una estrategia hecha a la medida (*ad hoc*) para cada autómata.

Una estrategia más general es aquella dirigida por una tabla, en la que las transiciones del autómata se representan mediante una matriz de $|Q| \times |\Sigma|$ donde hay un reglón por cada estado y una columna por cada símbolo, la cual se puede implementar de manera inmediata mediante un arreglo bidimensional; el arreglo para el autómata de nuestro ejemplo es:

```
state tabla [3][2] ={ /* a   b  */
             /* S0 */   {S1, S2},
             /* S1 */   {S0, S2},
             /* S2 */   {S2, S1},
             }
```

Código 2.2: Arreglo bidimensional que representa las transiciones de un AFD

Y el código genérico para un analizador dirigido por tabla es:

```
state actual = S0
char entrada
while haya caracteres en la entrada
   entrada = sgtChar()
   actual = tabla[actual][entrada]
```

Código 2.3: Manejador general en una implementación de un AFD basada en tabla

Cabe señalar que `state` debe ser una enumeración (subconjunto de $\mathbb{N}$ que empiece en 0), para que en el enunciado `tabla[actual][entrada]`, `actual` se maneje internamente como un entero; en cambio `entrada` es un carácter y aunque podemos utilizar su representación binaria como entero, éste tendrá el valor de la codificación de caracteres correspondiente al carácter. Actualmente, los conjuntos de caracteres tienden a ser muy grandes, como en Unicode (65,536) o UTF8, por lo que no debemos utilizar directamente la representación binaria de `entrada` como un índice para el arreglo. En lugar de eso primero debemos establecer una correspondencia entre los caracteres del alfabeto y los índices del arreglo[3] y lo podemos hacer de la siguiente manera:

[3]En el peor de los casos son 256 posibilidades utilizando ASCII en su versión de 8 bits, pero si la codificación es *Unicode* en su versión de 16 bits, tendremos 65536 columnas en la tabla -hoy en día se usa Unicode- ambas opciones (ASCII y Unicode) son malas pues en la mayoría de los casos se van a usar muy pocas columnas.

```
int ientrada
switch entrada
  case a:
    ientrada = 1
  case b:
    ientrada = 2
      .
      .
      .
  default:
    ientrada = 0
```

Código 2.4: Implementación de la correspondencia entre los caracteres del AFD y los índices del arreglo.

De esta forma ahora podemos hacer uso del enunciado `tabla[actual][ientrada]`, aunque nuevamente con la característica de que resulta ser una codificación *ad hoc* para un alfabeto particular.

Otra manera de resolver esta situación es haciendo uso de una tabla de dispersión cuyos elementos sean los renglones de la matriz, es decir, arreglos unidimensionales, en la cuál tendríamos pares (llave,valor), donde la llave sería un carácter y el valor un arreglo de las transiciones sobre ese carácter, como se muestra en la figura 2.5.

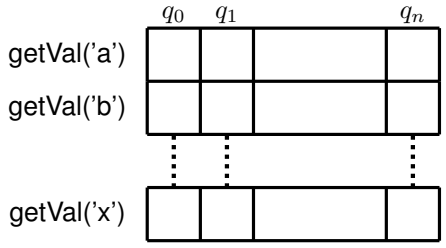


Figura 2.5: Implementación de conversión de caracteres a enteros

La función `getVal` recibiría un carácter y ésta (dependiendo de su implementación) calcula una función de dispersión correspondiente al carácter, obteniendo así el índice del arreglo que utiliza internamente la tabla hash para posteriormente devolvernos el arreglo correspondiente a ese carácter. Esta estrategia tiene la ventaja que no tenemos que codificar a pie la correspondencia entre los caracteres y los índices del arreglo, pero tiene la desventaja que generalmente utilizaría más espacio y dependemos de contar con una buena función de dispersión para que sea eficiente; generalmente las implementaciones de las tablas hash contienen una función de dispersión genérica, pero usualmente

el usuario puede proveer una específica que se comporte de mejor manera para un problema particular.

Si se realiza la implementación de la tabla directamente mediante un arreglo bidimensional tendrá n renglones y m columnas donde n es el número de estados, que usualmente son varios cientos; mientras que m es el número de caracteres que son 128 en el caso del código ASCII original (que solo utilizaba 7 de los 8 bits para representar caracteres); esto da como resultado tablas que ocupan menos de un megabyte de memoria, lo cual no representa un problema significativo para las computadoras de escritorio y estaciones de trabajo hoy en día, pero sí para los compiladores de dispositivos móviles que cuentan con una cantidad de recursos limitada; dado esto existen técnicas de compactación para la tabla que utilizan menos espacio, como la que se describe en [Aho+07].

La estrategia de utilizar una tabla para dirigir un programa va más allá de su uso en los analizadores léxicos: hay muchas otras aplicaciones en las que se puede utilizar y usualmente es una buena estrategia para diseñar programas, ya que mantiene en claro el comportamiento del programa y hace que sea más fácil modificar y añadir nuevas funcionalidades en él.

Por otra parte debemos señalar que aunque podemos leer los caracteres de la entrada uno a uno y regresarlos de igual forma, es más conveniente hacer uso de un *buffer* que es un espacio temporal de almacenamiento; de esta forma, en lugar de traer uno a uno los caracteres de la entrada leemos un bloque de un tamaño fijo, por ejemplo 512 bytes, y podemos manejar el carácter actual mediante un apuntador; con esto, avanzar y devolver caracteres de la entrada simplemente significa mover hacia adelante y hacia atrás respectivamente un apuntador; algunas técnicas de implementación de buffers se describen en [Knu97].

Por último debemos mencionar que a todas las estrategias anteriores debemos agregar el código necesario que implemente el comportamiento descrito en las reglas 1 y 2 en la página 36.

2.3. Flex

Aunque podemos implementar a pie un analizador léxico, para lo cual primero tendríamos que elaborar manualmente el AFD que reconozca todos los átomos con los que cuenta nuestro lenguaje fuente, ya sea siguiendo el método descrito antes -denotando los átomos con expresiones regulares, convertirlas a un AFN, convertirlo a AFD y por último a AFD mínimo- o a pie, y posteriormente implementarlo utilizando las técnicas que acabamos de estudiar; es claro que podemos automatizar este proceso mediante un programa que construya analizadores léxicos, que tome como entrada las expresiones regulares correspondientes a cada uno de los átomos del lenguaje a implementar

y entregue como salida el código de un analizador léxico con todas la tareas que éste debe realizar. Este tipo de programas se conocen como generadores de analizadores léxicos y han sido desde hace tiempo herramientas muy utilizadas en la construcción de compiladores y en diversas aplicaciones que requieren realizar una manipulación léxica de su entrada; se ha observado que los generadores de analizadores léxicos de hoy en día generan código igual o más eficiente que el de los analizadores léxicos escritos a pie. Existen numerosos generadores de analizadores léxicos que trabajan con diferentes lenguajes de programación; nosotros utilizaremos flex un generador de analizadores léxicos que trabaja con C y C++, que está basado y es compatible con `lex` [LS79], uno de los primeros generadores de analizadores léxicos.

flex es un generador de analizadores léxicos que toma como entrada las expresiones regulares que denotan a los lenguajes, representados por cada uno de los átomos que forman parte del lenguaje fuente del compilador a implementar, junto con las acciones que deben realizarse en caso de que un lexema (de longitud máxima) pertenezca a alguno de los lenguajes. Estas acciones son generalmente reportar la presencia del átomo correspondiente al analizador sintáctico, junto con información opcional del átomo que le sea útil a las subsecuentes etapas del proceso de compilación. En otras aplicaciones se puede utilizar un analizador léxico para hacer únicamente transformaciones léxicas sobre la entrada; en este caso las acciones consisten solamente en realizar una transformación sobre el flujo de entrada y escribir la salida, ya sea en un archivo o en pantalla. Un ejemplo sencillo de este tipo de aplicaciones sería convertir todas las palabras de un texto de entrada de minúsculas a mayúsculas o quitar todos los espacios en blanco.

flex en sí mismo es un compilador. Su lenguaje fuente son las expresiones regulares junto con las acciones y su lenguaje objetivo es código en C del AFD que se construye a partir de las expresiones regulares.

flex recibe como entrada un archivo con extensión `.l` y produce como salida un archivo de nombre `lex.yy.c`.[4] Alternativamente puede generar código en C++, en cuyo caso produce el archivo `lex.yy.cc`.[5]

Estructura de un archivo de entrada de flex

Los archivos de entrada de flex se dividen en tres secciones, que se describen a continuación:

1. **Definiciones.** En esta sección se encuentra un apartado delimitado por %{ y %}; su contenido será puesto al principio del archivo de salida y por tanto aquí se deben escribir todos los enunciados que es común que un programa en C contenga al inicio, como son las directivas del preprocesador y la declaración de funciones

[4] El usuario puede especificar un nombre de archivo diferente si así lo desea.

[5] De igual forma el usuario puede dar un nombre diferente a este archivo si así lo desea.

mediante sus firmas (prototipos). Se debe incluir también la declaración de constantes[6] y de variables globales que se utilizarán a lo largo del programa, que en este caso es el analizador léxico; al término de este apartado (aun dentro de la primera sección) se deben definir las expresiones regulares correspondientes a cada uno de los átomos y se les asigna un nombre a cada expresión regular para poder referirse a ellas en la siguiente sección (también se pueden definir expresiones regulares auxiliares que se utilicen dentro de otras expresiones regulares más complejas y que no necesariamente correspondan a un átomo, como pudiese ser la definición de caracteres alfanuméricos o de dígitos).

2. **Reglas de traducción.** Dentro de esta sección, en cada renglón se escribe una expresión regular seguida de un espacio y los enunciados en lenguaje C a ejecutar, en caso de que el analizador léxico encuentre un lexema (de longitud máxima) que empate con la expresión regular. Generalmente la acción que se realiza es reportar al analizador sintáctico de qué tipo de átomo se trata y, si tiene alguna información útil para las siguientes fases, comunicar esta información a través de una variable o proceder a guardarla en una estructura de datos auxiliar como la tabla de símbolos.

3. **Funciones auxiliares.** En esta sección deben definirse todas aquellas funciones auxiliares que sean de utilidad para el programador; estas funciones se pueden llamar desde las acciones escritas en la sección de reglas de traducción. Esta sección se escribe sin modificaciones al final del archivo de salida generado por flex.

Dentro del archivo de entrada de flex, el cambio de una sección a otra se denota mediante los caracteres `%%` que deben colocarse al inicio de una línea. De esta manera la estructura general de un archivo de entrada de flex es la siguiente:

Definiciones

`%%`

Reglas de traducción

`%%`

Funciones auxiliares

A continuación damos una descripción más detallada de estas secciones.

[6] C originalmente no contaba con el modificador **`const`** por lo que las "constates" se manejaban a través de la directiva `define` del preprocesador.

2.3.1. Definiciones

En la sección de definiciones, como hemos mencionado anteriormente, debemos escribir pares *(nombre, definición)*, donde *nombre* es un identificador que comienza con una letra o un "_" (guion bajo), seguido de cero o más letras, dígitos, "_" o "-" (guion); *definición* es una expresión regular. De esta forma le hemos dado un nombre a una expresión regular que nos servirá como alias de ésta dentro de otras expresiones regulares o para referirnos a ella dentro de la sección de reglas de traducción. Es importante señalar que debemos escribir el nombre al inicio de la línea (sin dejar espacios en blanco) y al final de éste dejar al menos un espacio en blanco (pero sin saltos de línea) y escribir la definición. Para hacer uso de nuestra definición dentro de otras expresiones regulares o en la sección de reglas de traducción debemos escribir su nombre entre `{}`, como se muestra en el siguiente ejemplo:

```
DIGITO [0-9]
ENTERO {DIGITO}+
```

Aquí hemos definido `ENTERO` haciendo uso de la definición previa `DIGITO [0-9]`, así cuando flex encuentre el enunciado `{DIGITO}` lo reemplazará por `[0-9]`, con lo que escribir

```
ENTERO {DIGITO}+
```

será lo mismo que escribir

```
ENTERO [0-9]+
```

Dentro de esta sección podemos hacer uso de comentarios. Los comentarios empiezan con `/*` y terminan con `*/` y deben comenzar al inicio de una línea (sin dejar espacios en blanco antes); los comentarios se escriben sin modificación en el archivo de salida. Por otra parte, cualquier texto con sangría (con al menos un espacio en blanco al inicio de la línea) se escribe sin modificación en el archivo de salida; de igual forma, como se señaló anteriormente, el texto contenido dentro de un bloque que comience con `%{` y termine con `%}` se escribirá sin modificación en el archivo de salida (sin los delimitadores); nótese que tanto `%{` como `%}` deben escribirse al inicio de una línea; lo anterior nos sirve para poder definir directivas del preprocesador, variables globales y constantes. Un bloque `top` está delimitado por `{` y `}` y es similar al bloque descrito anteriormente, sólo que en este caso el texto dentro de este bloque se pone al inicio del archivo generado, antes de cualquier definición de flex y es útil cuando debemos asegurarnos que algo debe aparecer antes de cualquier enunciado generado por flex. Veamos el ejemplo en la siguiente página.

```
% top{
     /*Este comentario aparecera al principio del archivo
        generado */
     //Otras declaraciones
     #include<stdio.h>
    }
```

Es posible utilizar más de un bloque `top` y se respeta el orden en el que aparecen.

2.3.2. Reglas de traducción

Esta sección contiene un serie de reglas de la forma:

```
patrón acción
```

donde antes de patrón no debe haber sangría y la acción debe comenzar en la misma línea. El patrón es una expresión regular y se pueden utilizar las definiciones hechas en la sección anterior, como hemos descrito previamente. Al final del patrón se debe dejar al menos un espacio en blanco y escribir las acciones que se realizarán cuando el lexema empate con la expresión regular; estas acciones se enuncian en lenguaje C; si se trata de acciones que ocupan una sola línea basta con dejar al menos un espacio en blanco después del patrón y escribirlas. Si las acciones ocupan más de una línea entonces se deben poner dentro de un bloque que comienza con `{` y termina con `}`. Como hemos mencionado anteriormente, lo que se realiza en general es reportar la presencia de un átomo al analizador sintáctico y, opcionalmente, entregar información adicional relevante, lo que implica tener una interfaz de comunicación entre el analizador léxico y el sintáctico. En nuestro caso haremos uso de bison como herramienta para generar nuestro analizador sintáctico, así que el analizador léxico que genera flex se comunica con el analizador sintáctico que genera bison mediante la función principal que genera flex de nombre `yylex` y que regresa un entero correspondiente al identificador único de cada átomo. De esta manera bison llama a al función `yylex` y dependiendo del identificador que reciba sabe de qué átomo se trata; para el programador es transparente el uso de estos identificadores pues bison es quien se encarga de asignarlos internamente, ya que el programador sólo trabaja con nombres de los átomos. Por otra parte, la información adicional relevante de algunos átomos se comunica por medio de una variable llamada `yylval`. Veremos con más detenimiento cómo se lleva a cabo esta comunicación cuando estudiemos bison en el siguiente capítulo.

En esta sección, cualquier texto con sangría o dentro de un bloque `%{ %}` (con las mismas restricciones que en la sección anterior) que se encuentre antes de la primera regla, se puede utilizar para declarar variables locales a la función de escaneo generada por flex (`yylex`). Si hay texto con sangría o dentro de un bloque `%{ %}` después de la

primera regla, se pasará sin modificación al archivo de salida, pero su uso no estará bien definido y puede causar errores, por lo que no debe utilizarse.

2.3.3. Funciones Auxiliares

Esta sección se usa para definir funciones auxiliares que se pueden llamar entre ellas mismas o desde la función `yylex`, es decir, dentro de las acciones (de la sección anterior). Es opcional; si no está presente su pueden omitir los `%%` en el archivo de entrada que denotan el comienzo de esta sección.

Por último debemos señalar que flex soporta comentarios al estilo C, es decir aquellos delimitados por `/*` y `*/`, como vimos anteriormente. Siempre que flex encuentra un comentario lo pasa sin modificación al archivo de salida. En principio los comentarios se pueden encontrar en cualquier parte, excepto en los siguientes lugares:

- No pueden aparecer en la sección de reglas de traducción cuando flex esté esperando una expresión regular; esto es, no pueden aparecer al comienzo de una línea o inmediatamente después de una lista de estados.[7]
- No pueden aparecer después de una declaración `%option` en la sección de definiciones.

Expresiones regulares extendidas

Como vimos tanto en la sección de definiciones como en la sección de reglas de traducción, flex hace uso de expresiones regulares. Las expresiones regulares básicas se han extendido a lo largo de los años en el campo de la implementación para hacer más clara y sencilla su habilidad para representar conjuntos de cadenas. Algunas de estas extensiones son:

1. Una o más presencias. El operador unario posfijo + representa la cerradura positiva de un conjunto regular. Sea L un conjunto regular sobre un alfabeto Σ; definimos $L^+ = \bigcup_{k=1}^{\infty} L^k$. Sea r una expresión regular, r^+ denota el lenguaje $L(r)^+$. El operador + tiene la misma precedencia y asociatividad que el operador $*$ y se tiene que $r^* = r^+|\varepsilon$ y $r^+ = rr^* = r^*r$.

2. Cero o una presencia. El operador unario posfijo $?$ denota cero o exactamente una presencia; esto es, sea r una expresión regular $r?$ es equivalente a $r|\varepsilon$ y denota al lenguaje $L(r?) = L(r) \cup \{\varepsilon\}$. El operador $?$ tiene la misma precedencia y asociatividad que los operadores $*$ y +.

[7] Estudiaremos estados cuando veamos cómo manejar dependencias del contexto más adelante.

3. Clases de caracteres. Sea r una expresión regular sobre un alfabeto Σ; si $r = a_1|\ldots|a_i|\ldots|a_n$, con $a_i \in \Sigma$ y $s = [a_1 \ldots a_i \ldots a_n]$, decimos que $L(s) = L(r)$. Cuando $a_1, a_2, \ldots, a_n$ forman una secuencia consecutiva, por ejemplo letras mayúsculas, letras minúsculas o dígitos, podemos denotar dicha secuencia con la expresión regular t compuesta por el primer símbolo de la secuencia seguido de un guion y el último símbolo de la secuencia; esto es, si $t = [a_1 - a_n]$ definimos $L(t) = L(r) = L(s)$; por ejemplo $[aeiou]$ es equivalente a $a|e|i|o|u$, y $[a - z]$ es equivalente a $a|b|\ldots|z$.

flex, como es natural esperar, hace uso de expresiones regulares extendidas para representar los conjuntos correspondientes a cada uno de los átomos.

Algunas de las expresiones regulares extendidas presentes en flex se muestran en la tabla 2.1, a continuación.

Expresión	Empata
`x`	El carácter `x`.
`.`	Cualquier carácter excepto nueva línea.
`[xyz]`	Clase de caracteres, en este caso empata ya sea con una `x`, una `y` o una `z`.
`[abj-oZ]`	Clase de caracteres con un rango dentro de ella, empata ya sea con una `a`, una `b`, cualquier letra en el rango de `j` a `o` o una `Z`.
`[\^A-Z]`	Clase de caracteres negada, denota cualquier carácter excepto los que están en la clase. En este caso empata cualquier carácter excepto una letra mayúscula.
`[a-z]{-}[aeiou]`	Diferencia de clases de caracteres. En este caso empata las consonantes minúsculas.
`r*`	Cero o más presencias de la expresión regular `r`.
`r+`	Una o más presencias de la expresión regular `r`.
`r?`	Cero o exactamente una presencia de la expresión regular `r`.
`r{4,7}`	De cuatro a siete presencias de la expresión regular `r`.
`r{4,}`	Cuatro o más presencias de la expresión regular `r`.
`r{4}`	Exactamente cuatro presencias de la expresión regular `r`.
`{nombre}`	Reemplaza `{nombre}` por su definición como se explicó en la sección 2.3.1.
`"[a-z]\"fin"`	Toda cadena entre `""` se interpreta literalmente. En este caso empata la cadena `[a-z]"fin`. Nótese que para que `"` forme parte de la cadena se debe anteponer una `\`.

Tabla 2.1: Expresiones regulares en flex (continúa en la siguiente página)

Expresión	Empata
`\C`	Si `C` es uno de los siguientes caracteres: `a`, `b`, `t`, `n`, `v`, `f` o `r`, `\C` se interpreta igual que en ANSI-C. En otro caso se interpreta como `C` y se usa para que los meta-caracteres de las expresiones regulares sean caracteres, por ejemplo `\*` empata un `*`.
`\123`	El carácter con valor octal `123` .
`\xaf`	El carácter con valor hexadecimal `af`.
`(r)`	La expresión regular `r`. Los paréntesis se usan para agrupar expresiones regulares.
`rs`	La concatenación de las expresiones regulares `r` y `s`.
`r|s`	La expresión regular `r` o la expresión regular `s`.
`^r`	La expresión regular `r` pero solamente cuando ésta se encuentra al inicio de una línea.
`r$`	La expresión regular `r` pero solamente cuando ésta se encuentra al final de una línea.
`<<EOF>>`	Fin de archivo.

Tabla 2.1: Expresiones regulares en flex (continúa de la página anterior...)

Nótese que dentro de una clase de caracteres -entre `[` y `]`-, los metacaracteres se convierten en caracteres excepto `\` y los operadores que delimitan a las clases de caracteres `[`, `]`, `-`. El símbolo `^` cuando está presente al inicio de la clase de caracteres, indica que se trata de una clase de caracteres negada.

La precedencia de los operadores es de acuerdo a su presencia en la lista anterior, donde el primero que se lista tiene la mayor precedencia y el último la menor.

2.3.4. Condiciones iniciales

Como vimos, en la sección de reglas de traducción se encuentran las reglas que dictan el comportamiento del analizador léxico. Cada una de estas reglas está formada por una expresión regular y por las acciones que se deben realizar en caso de que un lexema empate con la expresión regular. En algunas situaciones resulta conveniente que sólo algunas de estas reglas estén activas, es decir, que el lexema a empatar sólo pueda empatar con alguna de las expresiones regulares de las reglas que se encuentran activas, siendo de esta manera imposible que empate con las expresiones regulares de las reglas inactivas; de esta forma logramos, entre otras cosas, que ciertos átomos se reconozcan sólo si se cumplen ciertas condiciones, o que el analizador tenga un comportamiento diferente al encontrar un ejemplar de un átomo determinado, dependiendo de la situación actual en la que se encuentra el analizador. Para realizar esta tarea flex cuenta con lo que denomina *condiciones iniciales*.

Para hacer uso de una condición inicial primero debemos declararla. Las condiciones iniciales constan tanto de un tipo como de un identificador y se declaran en la sección de definiciones de la siguiente manera:

```
% s PRIMERA
% x SEGUNDA TERCERA
```

como podemos observar, una declaración tiene que comenzar al inicio de una línea (sin dejar sangría); primero se escribe el tipo, a continuación se deja al menos un espacio en blanco (sin pasar a la siguiente línea) y por último se escribe el identificador; podemos escribir más de un identificador en una misma línea, dejando al menos un espacio en blanco entre ellos. Así, si tenemos una declaración con un tipo t y escribimos n identificadores distintos enseguida, habremos declarado n condiciones iniciales, todas con el tipo t. Existen dos tipos de condiciones iniciales, las inclusivas que se declaran con `s` y las exclusivas que se declaran con `x`. En nuestro ejemplo anterior hemos declarado una condición inicial inclusiva con nombre `PRIMERA` y dos exclusivas con nombres `SEGUNDA` y `TERCERA` respectivamente.

Hay varias reglas que se cumplen cuando trabajamos con condiciones iniciales y se listan a continuación:

- El analizador se encuentra en una única condición inicial en todo momento.
- Existe una única condición inicial por omisión de nombre `INITIAL` que es la condición en la que se encuentra al empezar a funcionar el analizador producido28.
- Si tenemos una condición inicial inclusiva `INCL`, siempre que el analizador se encuentre en la condición `INCL`, todas aquellas reglas en las que esté presente `INCL` y todas aquellas reglas sin condiciones iniciales explícitas estarán activas.
- Si tenemos una condición inicial exclusiva `EXCL`, siempre que el analizador se encuentre en la condición `EXCL` únicamente todas aquellas reglas en las que esté presente `EXCL` estarán activas y sólo éstas.

Para utilizar una condición inicial, estando en la sección de reglas de traducción, debemos escribir el identificador de la condición inicial que deseamos usar encerrado entre los caracteres < y > al inicio de la línea de la regla en la que la vamos a usar, justo antes de la expresión regular, como se muestra a continuación:

```
%s PRIMERA
%%
<PRIMERA>{integer}  { printf("Encontre un entero estando en la"
                                          "condicion PRIMERA"); }
```

En este ejemplo la regla que tiene como expresión regular a `integer` (que se supone definida en la sección de definiciones) estará activa sólo cuando el analizador se encuentre en la condición `PRIMERA`.

Por otra parte, podemos utilizar más de una condición inicial en una regla, escribiendo los identificadores de las condiciones iniciales separados por comas (,) como se muestra en el siguiente ejemplo:

```
%s PRIMERA
%x SEGUNDA TERCERA
%%
<PRIMERA,SEGUNDA,TERCERA>{integer} {printf("Encontre un entero"
                                    "estando en la condicion"
                                    "PRIMERA o SEGUNDA o TERCERA");}
```

Esta vez la regla estará activa cuando el analizador se encuentre en cualquiera de las condiciones PRIMERA, SEGUNDA o TERCERA. De esta manera, podemos agregar a una regla tantas condiciones iniciales como sea necesario.

Mostremos ahora con un ejemplo la diferencia entre las condiciones iniciales inclusivas y exclusivas:

```
%s PRIMERA
%x SEGUNDA
%%
<PRIMERA>{integer}  { printf("Encontre un entero estando en la"
                             "condicion PRIMERA"); }
<SEGUNDA>{string}   { printf("Encontre una cadena estando en la"
                             "condicion SEGUNDA"); }
{identifier}        { printf("Encontre un identificador"); }
```

Si el analizador se encuentra en la condición inicial PRIMERA entonces es claro que estará activa la primer regla, pero además estará activa la tercer regla (y todas las demás reglas sin condiciones iniciales explícitas, si las hubiera), debido a que, por un lado, PRIMERA es inclusiva y, por otro, la tercer regla no tiene condiciones iniciales explícitas; de esta forma, si estando el analizador en la condición PRIMERA encuentra un lexema en la entrada que empate con la expresión regular identifier, se imprimirá en pantalla Encontre un identificador. En cambio, si el analizador se encuentra en la condición inicial SEGUNDA, únicamente estará activa la segunda regla, ya que SEGUNDA es exclusiva. También podemos hacer uso explícito de la condición inicial INITIAL:

```
%s PRIMERA
%x SEGUNDA
%%
<PRIMERA>{integer}  { printf("Encontre un entero estando en la"
                             "condicion PRIMERA"); }
<SEGUNDA>{string}   { printf("Encontre una cadena estando en la"
                             "condicion SEGUNDA"); }
```

(continúa...)

```
{identifier}          { printf("Encontre un identificador"); }

<INITIAL>"if"         { printf("Encontre una palabra reservada"
                               "if"); }
"else"                { printf("Encontre una palabra reservada"
                               "else"); }
```

Una posible ambigüedad se presenta cuando tenemos dos reglas que tienen una misma expresión regular, sólo que una de ellas tiene al menos una condición inicial inclusiva explícita y la otra no tiene condiciones iniciales explícitas, como lo ilustra el siguiente ejemplo:

```
%s PRIMERA
%%
<PRIMERA>{integer}  { printf("Encontre un entero estando en la"
                                  "condicion PRIMERA"); }
{integer}           { printf("Encontre un entero"); }
```

En este ejemplo, si el analizador se encuentra en la condición inicial `PRIMERA`, en principio deberían estar activas ambas reglas; es aquí donde tenemos la ambigüedad que se resuelve tomando la primera de las dos que se liste, por lo que lo que tiene sentido es poner siempre primero la regla que tiene la condición inicial explícita, pues en otro caso nunca se ejecutaría esta regla.

Como hemos mencionado, el analizador se encuentra en una única condición inicial en cualquier instante del tiempo de ejecución. Por omisión, al inicio de su ejecución está en la condición `INITIAL` establecida por flex. Para llevar a cabo un cambio de condición se debe utilizar el enunciado `BEGIN(CONDITION)`, donde `CONDITION` es el identificador de la condición a la que se desea cambiar. Este enunciado debe escribirse dentro dentro de aquellas acciones en las que se desee utilizar. Veamos un ejemplo:

```
%x STRING

%%

<STRING>[^"]*  {}
<STRING>\"     { printf("Termina la cadena\n"); BEGIN(INITIAL);}
\"             { printf("Comienza la cadena\n"); BEGIN(STRING);}
```

Al inicio, la única regla que está activa es la tercera, así que si el analizador lee unas comillas (") de la entrada, indica que una cadena va a comenzar y cambia de estado a la condición `STRING`; por otra parte, si el analizador encuentra unas comillas estando

en la condición `STRING`, entonces indica que la cadena terminó y cambia a la condición en la que inicia por omisión el analizador. Este es un ejemplo de cómo el analizador puede realizar acciones diferentes para la misma expresión regular dependiendo de la condición en la que se encuentre.

Si en algún momento es necesario consultar el estado actual en el que se encuentra el analizador podemos utilizar la macro `YY_START` que flex ofrece para este fin. Internamente las condiciones iniciales son enteros, así que las podemos manipular como enteros en el código C o C++ de las acciones.

Como señalamos anteriormente, podemos utilizar las condiciones iniciales para empatar ciertos átomo sólo bajo ciertas condiciones, pero además podemos tener condiciones anidadas; para ayudar con esta tarea flex provee una pila de condiciones iniciales, manipulable por medio de las siguientes funciones:

1. `void yy_push_state (int estadoNuevo)`
   Hace un push del estado actual en la pila y establece el estado `estadoNuevo` como el nuevo estado actual; esta última operación equivale a hacer `BEGIN( estadoNuevo)`.
2. `void yy_pop_state()`
   Hace un pop en la pila y establece como nuevo estado actual el elemento que acaba de sacar de la pila.
3. `int yy_top_state()`
   Regresa el tope de la pila (sin alterar su contenido y sin realizar cambios al estado actual).

Es claro que a través de esta pila podemos seguir el rastro de las condiciones anidadas, aunque no debe olvidarse que en todo momento únicamente hay una condición inicial activa.

2.4. Analizador léxico de Python

Describiremos ahora el diseño e implementación de nuestro analizador léxico para Python.

Lo primero que tenemos que tener claro para desarrollar nuestro analizador léxico es qué átomos forman parte del lenguaje a implementar, por lo que hemos recurrido a la especificación oficial disponible en [Fou]. En particular debemos fijarnos en la gramática de Python, pues como lo hemos estudiado, los átomos que nuestro analizador léxico debe reportar son los símbolos terminales de la gramática. Por ende presentamos a continuación algunas de las producciones[8] de la gramática de Python que nos servirán para identificar los átomos presentes en Python.

[8]En estas producciones los metacaracteres "[]"denotan cero o exactamente una presencia.

SIMPLE_STMT	$\longrightarrow$	SMALL_STMT (**;** SMALL_STMT)* [**;**] ***newline***
IF_STMT	$\longrightarrow$	**if** TEST **:** SUITE (**elif** TEST **:** SUITE)* [**else :** SUITE]
SUITE	$\longrightarrow$	SIMPLE_STMT | ***newline indent*** STMT$^+$ ***dedent***
ATOM	$\longrightarrow$	**(** [TESTLIST_GEXP] **)** | **[** [LISTMAKER] **]** | **{** [DICTMAKER] **}**
ATOM	$\longrightarrow$	**‘** TESTLIST1 **‘** | ***name*** | ***number*** | ***string***$^+$

Producciones que muestran algunos de los átomos más relevantes de Python

Hemos utilizado la siguiente notación en la gramática:

- MAYÚSCULAS. Denotan los símbolos no terminales.
- **Negritas**. Denotan los átomos correspondientes a las palabras reservadas del lenguaje. La expresión regular correspondiente a estos átomos es simplemente la cadena correspondiente a la palabra reservada, no tienen información adicional relevante.
- ***Negritas itálicas***. Denotan los átomos que no son palabras reservadas y que representan mayor trabajo para el analizador léxico. Además, varios de estos átomos tienen información adicional útil necesaria en las siguientes fases del proceso de compilación y sus respectivas expresiones regulares son más complejas que el caso anterior.

Comenzaremos mostrando cómo implementar los átomos correspondientes a las palabras reservadas del lenguaje. Como sabemos, en el archivo de entrada de flex en la sección de reglas de traducción debemos escribir primero la expresión regular correspondiente al átomo; en este caso tomaremos como ejemplo la palabra reservada **if**, presente en la segunda producción. Dicha expresión regular es simplemente la cadena **if**, esto es, en general, para cada uno de los átomos que representen una palabra reservada, el único lexema que forma parte del conjunto de cadenas que representa cada uno de estos átomos es simplemente la cadena correspondiente a la palabra reservada; debemos ahora prestar atención a qué acciones debemos ejecutar cuando un lexema empate con alguna de las palabras reservadas. En este ejemplo, si se lee una cadena **if** de la entrada, en principio debemos reportar el átomo **if**. En flex y bison los átomos se escriben con mayúscula, por lo que la regla correspondiente a nuestro ejemplo sería:

```
"if"    {return IF;}
```

Aún cuando esta regla ha cumplido con nuestro objetivo principal, como mencionamos en el capítulo 1, nuestro analizador léxico también debe generar una representación basada en texto con fines de depuración, por lo que debemos agregar lo siguiente:

```
"if"    { salida << "IF"; return IF;}
```

Sin embargo, lo anterior representa una desventaja, ya que no siempre queremos generar la salida basada en texto para depuración, por lo que tendríamos que agregar

enunciados condicionales y banderas para que, con base en los valores de éstas, se generara la representación basada en texto, el flujo de átomos para el analizador sintáctico o ambas, según las necesidades del usuario. Esta estrategia representa varias desventajas, quizá la más evidente es que siempre que el analizador léxico ejecute una acción debe evaluarse una condición, lo que representa una degradación en el desempeño del analizador; pero no sólo eso, sino que además hace el código menos legible. Por esta razón nosotros hemos optado por utilizar el patrón Constructor (*Builder*), un patrón de creación. El patrón Builder nos permite generar diferentes representaciones del objeto que vamos a construir. En este caso, el objeto a construir es un flujo de átomos y deseamos generar dos representaciones: una basada en texto y otra que genere átomos (codificados internamente por flex y bison como enteros) para el analizador sintáctico; adecuadamente, el hacer uso de este patrón podemos en un futuro generar otras representaciones de manera clara y elegante, como podría ser una representación gráfica. En el patrón Builder existen cuatro participantes:

- Constructor (*Builder*). Especifica una interfaz abstracta para construir partes de un ejemplar de Producto.
- ConstructorConcreto (*ConcreteBuilder*). Construye y ensambla partes del Producto implementando la interfaz definida por Constructor.
- Director (*Director*). Construye un objeto haciendo uso de la interfaz provista por el Constructor.
- Producto (*Product*). Representa el objeto bajo construcción.

En nuestro analizador la clase STBuilder juega el papel de Constructor; en ella se encuentran declarados los diferentes métodos que se encargan de la construcción de átomos, por ejemplo:

```
virtual token_type bTok(token_type ct,string name,location_type* lval,
                        int lnum,ostream* os) = 0;
```

Las clases CSTBuilder y TSTBuilder juegan el papel de ConstructorConcreto, y son las que en realidad construyen los átomos, implementando la interfaz declarada en STBuilder, ya sea generando un entero para el analizador sintáctico en el caso de CSTBuilder o escribiendo su representación en texto en el caso de TSTBuilder. La clase Lexer es la abstracción de nuestro analizador léxico; su función es presentar al exterior métodos para comunicarnos con el analizador, como es `void setSFile(FILE* sf);`, para indicarle al analizador de dónde leer la entrada. Aunque su principal método es yylex, que regresa el siguiente átomo que el analizador encontró en la entrada, en realidad la clase Lexer es un envoltorio del analizador generado por flex; flex, por omisión, crea una analizador en C y se comunica al exterior por medio de variables globales; sus funciones, variables y estructuras internas no están protegidas del exterior, lo cual viola el principio de encapsulamiento, por lo que hemos utilizado la opción reentrant de flex

que genera un analizador que protege su estructura interna del exterior y provee un conjunto de funciones como interfaz para manipularla. A su vez, hemos encapsulado este analizador a través de la clase Lexer, mostrando al exterior sólo los métodos necesarios para nuestros fines; de esta forma, el método yylex de la clase Lexer es solamente un envoltorio de la función yylex generada por flex; no obstante, esta última no puede ser accedida directamente desde el exterior porque la hemos protegido; a su vez la clase Lexer juega el papel de Director, por lo que mantiene internamente un apuntador a un ConstructorConcreto (ya sea CSTBuilder o TSTBuilder) que se le asigna en el momento de su creación y es quien la clase Lexer va a dirigir para que construya los átomos. De esta manera resulta transparente la representación que se vaya a construir para la clase Lexer, pues Lexer sólo dicta qué hacer, cómo construir un átomo y, dependiendo del tipo de ConstructorConcreto que le asignen, este último construirá la versión en texto, en caso de que sea un ejemplar de TSTBuilder, o regresará un átomo para el analizador sintáctico, en caso de CSTBuilder. No existe un clase que juegue el rol de Producto, debido a que acá al producto a construir es un flujo de átomos y no un objeto que se construya en memoria; en su lugar, simplemente se manda una secuencia de átomos, ya sea al analizador sintáctico o a algún flujo de salida como la pantalla o un archivo de texto.

Regresando a nuestro ejemplo, haciendo uso del patrón Builder el código que debemos escribir en las acciones es:

```
"if"        { return stb->bTok(token::IF,"IF",yylloc,yylineno,os); }
```

donde stb es un ejemplar de un STBuilder. De esta forma hemos resuelto el problema que se nos presentaba al inicio, pues si stb es un ejemplar de CSTBuilder, se ejecutará la implementación de bTok de la clase CSTBuilder que es:

```
token_type CSTBuilder::bTok(token_type ct,string name,
                            location_type* lval,int lnum,ostream* os){
  lval->begin.line = lnum;
  lval->end.line = lnum;
  return ct;
}
```

Código 2.5: Implementación del método bTok de la clase CSTBuilder

y si stb es un ejemplar de TSTBuilder, se ejecutará:

```
token_type TSTBuilder::bTok(token_type ct,string name,location_type*
    lval,int lnum,ostream* os){
  *os << name << " ";
  return token_type(-1);
}
```

Código 2.6: Implementación del método bTok de la clase TSTBuilder

Con esto el código de las acciones queda fijo y libre de enunciados condicionales y desde su perspectiva es transparente la representación concreta de los átomos que se vaya a crear; en caso de que se desee agregar una nueva representación, basta con generar otra clase que implemente la interfaz impuesta por STBuilder, con lo que tenemos como resultado una solución limpia y elegante. En el caso de un curso de compiladores se sugiere dar a los alumnos el código fijo de las acciones y que los alumnos implementen la interfaz provista por STBuilder en las clases CSTBuilder, TSTBuilder. Si se desea una o más representaciones adicionales, se generarán una o más clases respectivamente, que de igual forma implementen dicha interfaz.

Continuemos nuestro estudio mostrando cómo implementar los átomos que no son palabras reservadas, analizando primero aquellos que no poseen información adicional. Tomemos como ejemplo el átomo ***newline***; el lector podría pensar a primera vista que siempre que se encuentre un carácter de salto de línea (\n) en la entrada debe generarse un átomo ***newline***; sin embargo esto no siempre debe ser así y el ejemplo más claro es cuando tenemos un comentario #Esto es un comentario, el salto del línea al final del comentario no debe generar un átomo ***newline***. Pero más allá de este caso, en Python existe lo que se conoce como *implicit join*, descrito en la especificación de Python [Fou], que consiste en que cuando hay uno o varios saltos de línea dentro de la definición de alguna de las estructuras de datos soportadas nativamente por Python, éstos simplemente deben ignorarse y por tanto el analizador léxico no debe generar átomos ***newline***. Por ejemplo, en Python es un programa válido lo siguiente:

```
1 x = [1,2,
2
3       3,[
4 4,
5 7]
6           ,9]
7 print x
```

Código 2.7: Ejemplo de una declaración válida de una lista en Python

Aquí, los saltos de línea presentes en las líneas de la 1 a la 5 deben ignorarse, debido a que se encuentran dentro de la definición de una lista; en cambio, el salto de línea presente al final de la línea 6 sí debe generar un átomo ***newline***, pues no se encuentra dentro de la declaración de la lista. Así el programa anterior es equivalente al siguiente:

```
1 x = [1, 2, 3, [4, 7], 9]
2 print x
```

Código 2.8: Ejemplo de una declaración válida de una lista en Python

Pensemos ahora cómo implementar lo anterior en nuestro analizador. Intuitivamente podemos pensar que siempre que veamos un carácter que denota el inicio de la defini-

ción de una estructura de datos, ("[" en el caso de una lista) debemos seguir generando átomos de manera normal, excepto que sea un salto de línea, en cuyo caso simplemente lo ignoramos hasta que encontremos un carácter que denote el final de la definición de la estructura de datos ("]" en el caso de nuestra lista), con lo que en este punto el analizador debe volver a su funcionamiento original en el que, cuando empate un carácter de salto de línea (que no forme parte de un comentario), genera un átomo ***newline***, por lo que podemos utilizar una variable booleana para que cuando leamos un "[", además de generar el átomo "[", establezcamos su valor en verdadero, y cuando encontremos un carácter "]" y generemos el átomo "]", el valor sea falso. El problema con esta primera aproximación es que podemos tener estructuras de datos anidadas, como en el caso de nuestro ejemplo, en el que la lista [4,7] es un elemento de la lista que estamos definiendo. Es claro que no nos basta con una bandera; quizá podríamos construir una expresión regular que empatara la definición de la estructura de datos externa y posteriormente pasar el lexema empatado para ser procesado por el analizador, generando átomos de manera usual pero ignorando los saltos de línea presentes en el lexema. Este es un problema en el que debemos utilizar algunos resultado teóricos. En los libros de texto de lenguajes formales (consúltese por ejemplo [HU79]) se suele presentar el lenguaje $L = \{(^n)^n \mid n \geq 0\}$ como el primer ejemplo de un lenguaje que no es regular; hasta este momento sólo hemos trabajado con lenguajes regulares, pues mencionamos anteriormente que los átomos representan lenguajes regulares. Con el resultado anterior es claro que no podemos construir la expresión regular en la que habíamos pensado, pero ha surgido la inquietud de por qué, si estamos trabajando en el analizador léxico, tenemos que trabajar con un lenguaje que no es regular y que se ha presentado de forma inesperada; debemos prestar atención y tener claro que es el analizador sintáctico (como veremos en el siguiente capítulo) el que verifica que haya paréntesis bien balanceados en la definición de las estructuras de datos, así que el analizador léxico no debe llevar a cabo esta verificación: simplemente debe generar los átomos:

```
IDENTIFIER EQ [ INTEGER , INTEGER , INTEGER , [ INTEGER , INTEGER ] ,
    INTEGER ] NEWLINE
PRINT IDENTIFIER NEWLINE
```

donde cada lenguaje representado por cada uno de los átomos anteriores es un lenguaje regular y por tanto, cada uno de estos átomos cuenta con su correspondiente expresión regular. Posteriormente el analizador sintáctico procesará los átomos anteriores para verificar que se respete la sintaxis del lenguaje (en particular los paréntesis bien balanceados en las definiciones de estructuras de datos). El problema estriba en que el analizador léxico necesita información para saber cuándo se encuentra dentro de una definición de una estructura de datos y, con base en ello, decidir cómo actuar bajo la expresión regular

\n, esto es, el analizador debe realizar acciones diferentes para la misma expresión regular (en este caso \n) dependiendo de si se encuentra o no en ese momento dentro de una definición de una estructura de datos. Es claro que es un problema que podemos resolver haciendo uso de condiciones iniciales y la pila ofrecidas por flex para manipularlas. De esta manera obtenemos lo siguiente:

```
%s IMPLICITJ

newline                 \n
%%

"("             { yy_push_state(IMPLICITJ,yyscanner);
                  return stb->bTok(token_type('('),"(",yylloc,yylineno,os);
                }
"["             { yy_push_state(IMPLICITJ,yyscanner);
                  return stb->bTok(token_type('['),"[",yylloc,yylineno,os);
                }
"{"             { yy_push_state(IMPLICITJ,yyscanner);
                  return stb->bTok(token_type('{'),"{",yylloc,yylineno,os);
                }

<IMPLICITJ>newline   { }

")"             { yy_pop_state(yyscanner);
                  return stb->bTok(token_type(')'),")",yylloc,yylineno,os);}
"]"             { yy_pop_state(yyscanner);
                  return stb->bTok(token_type(']'),"]",yylloc,yylineno,os);}
"}"             { yy_pop_state(yyscanner);
                  return stb->bTok(token_type('}'),"}",yylloc,yylineno,os);}
{newline}       { yy_push_state(INDENT,yyscanner);
                  return stb->bTokNL(token::NEWLINE,"NEWLINE",yylloc,yylineno,os
                      );
                }
```

Como podemos observar, IMPLICIT es una condición inicial inclusiva que metemos a la pila (y por ende se activa) siempre que encontramos un carácter que indica el comienzo de una estructura de datos; es inclusiva porque todos los átomos se deben seguir generando de manera normal, excepto cuando encontremos un salto de línea, que es cuando se empata la expresión regular newline y es precisamente donde hemos definido que no se realice acción alguna cuando el analizador se encuentre en la condición IMPLICIT. En cambio, cuando el analizador se encuentre en la condición INITIAL (la condición por omisión) y se empate la expresión regular newline, el analizador debe meter a la pila la condición INDENT (que permite verificar la sangría de la siguiente línea que va a comenzar y generar átomos ***indent*** o ***dedent*** según corresponda) y generar el átomo ***newline***; por otra parte, siempre que se encuentra un carácter que indica el fin de

la definición de una estructura de datos, se reporta el átomo correspondiente y se hace un pop a la pila. De esta manera, si hemos visto n caracteres de inicio y m caracteres de fin hasta el momento, si $n > m$ estará activa la condición IMPLICITJ; en cambio, si $n = m$ habremos regresado a la condición INITIAL, pues se habrán realizado igual número de operaciones pop que de push(IMPLICITJ). En caso de que $m = n + 1$, el analizador de manera normal devolverá el átomo correspondiente y el analizador sintáctico es quien enviará un error sintáctico. Por último señalamos que también podemos resolver este problema utilizando una pila propia.

Finalmente, veamos cómo implementar los átomos cuyas expresiones regulares son más complejas y que tienen información adicional útil para las siguientes fases del compilador; tomemos como ejemplo los identificadores en Python representados por el átomo ***name***;[9] la implementación de este átomo en nuestro analizador es la siguiente:

```
identifier      ({letter}|_)({letter}|{digit}|_)*
letter          {lowercase}|{uppercase}
lowercase       [a-z]
uppercase       [A-Z]
digit           [0-9]

%%
{identifier}   { return stb->bTok(token::IDENTIFIER,"IDENTIFIER",yytext
   ,yylval,yylloc,yylineno,os);}
```

Hasta ahora no habíamos mencionado la función de cada uno de los parámetros del método bTok, por lo que a continuación mostramos la firma de este método de la clase abstracta STBuilder:

```
virtual token_type bTok(token_type ct,string name,string lexem,
   semantic_type* sval,location_type* lval,int lnum,ostream* os) =0;
```

Podemos notar que este método regresa un elemento de tipo token_type, donde token_type es una enumeración definida por bison donde asigna un entero a cada átomo y está envuelta dentro de una estructura de nombre token, como se muestra a continuación:

[9] ***name*** es el nombre otorgado al átomo que representa a los identificadores en la gramática oficial de Python. Nosotros lo hemos renombrado como ***identifier*** por fines didácticos.

```
struct token {
      /* Tokens.  */
   enum yytokentype {
     END = 0,
     IF = 258,
     NEWLINE = 259,
     IDENTIFIER = 260,
    };
};
```

Es por eso que el analizador sintáctico, en nuestro caso implementado en bison, espera que el analizador léxico genere átomos de tipo token_type; además, bTok cuenta con 8 parámetros formales donde ct es el código del átomo (definido por bison); name es la representación en cadena del átomo; lexem es el lexema que empató con la expresión regular actual; sval es donde el método bTok espera que se pase la variable definida por bison, donde el analizador sintáctico espera recibir la información adicional del átomo; lval es donde el método bTok espera que se pase la variable definida por bison la información sobre la posición del átomo en el código fuente; lnum el número de línea en el que el analizador léxico encontró el lexema actual; os el flujo de salida donde, en su caso, se mostrará el átomo construido. Analicemos ahora las dos implementaciones con las que contamos de este método, comenzando con la implementación de la clase CSTBuilder:

```
token_type CSTBuilder::bTok(token_type ct,string name,string lexem,
   semantic_type* sval,location_type* lval,int lnum,ostream* os){
  lval->begin.line = lnum;
  lval->end.line= lnum;
  sval->st = new string(lexem);
  return ct;
}
```

Como mencionamos, la clase CSTBuilder es la clase que construye la representación del flujo de átomos que espera el analizador sintáctico, por lo que regresa el código correspondiente al átomo encontrado, que es el que el analizador léxico le pasó como argumento. Pero además, el analizador sintáctico le pasa como argumento al analizador léxico las variables yylval y yylloc definidas por bison, donde bison espera recibir la información adicional del átomo y la información de su posición en el código fuente respectivamente. Esta información debe ser establecida por el analizador léxico. A su vez, el analizador léxico, que juega el rol de Director, delega esta responsabilidad a

los ConstructoresConcretos (CSTBuilder y TSTBuilder); es por eso que pasa como argumento yylval y yylloc, pero además pasa yylineno, que es la línea donde encontró el átomo actual. Por último, pasa un apuntador del flujo de salida donde se mostrará el átomo construido. Con esto, el ConstructorConcreto (cualquiera que éste sea) tiene toda la información necesaria para construir un átomo. Por eso, la implementación de la clase CSTBuilder que acabamos de ver asigna a sval una cadena que es la información adicional que necesita el analizador sintáctico; así, si encontramos el identificador x en la entrada no nos basta sólo con saber que es identificador, sino además que necesitamos almacenar x; adicionalmente se establece la posición del átomo en el número de línea donde el analizador léxico lo encontró.

Presentamos ahora la implementación de la clase TSTBuilder:

```
token_type TSTBuilder::bTok(token_type ct,string name,string lexem,
   semantic_type* sval,location_type* lval,int lnum,ostream* os){
  *os << name << "(" << lexem << ") ";
  return token_type(-1);
}
```

En este caso no es relevante el código de átomo que se regrese y por ello, por omisión regresa el valor -1. Finalmente se imprime en el flujo de salida la representación en cadena del átomo, junto con el lexema como información adicional. Como se observa, este último es un caso más general del que teníamos para las palabras clave, en el que no contábamos con información adicional.

2.4.1. Evaluación de nuestra implementación

A continuación se presenta un conjunto de programas de entrada junto con la respectiva salida que genera nuestra implementación del analizador léxico. Los programas fueron tomados de un ejercicio de programación de los alumnos, el cual consiste en implementar una máquina de pila que sirve para evaluar expresiones aritméticas y cuyo funcionamiento se parece a la pila para operaciones de punto flotante con la que cuenta la arquitectura x86 de Intel.

```
class Pila(object):
        def __init__(self, pila):
                self.pila = pila

        def push(self,elemento) :
                self.pila.append(elemento)
```

Código 2.9: Primer programa de prueba (continúa en la siguiente página)

```
        def pop(self) :
                return self.pila.pop()

        def isEmpty(pila) :
                return (self.pila == [])

        def suma(pila) :
                elem1 = pila.pop()
                elem2 = pila.pop()
                suma = elem1 + elem2
                pila.push(suma)

        def swap(pila) :
                elem1 = pila.pop()
                elem2 = pila.pop()
                pila.push(elem1)
                pila.push(elem2)

        def imprime(self) :
                self.pila.reverse()
                for i in self.pila:
                        if i==-1:
                                sys.stdout.write("+")
                        elif i == -2:
                                sys.stdout.write("s")
                        else:
                                print i
                self.pila.reverse()

entrada = '1'
p = Pila([])
while entrada != 'x' :
        sys.stdout.write("#")
        entrada = sys.stdin.readline()
        entrada = entrada[:-1]
        if entrada == '+':
                p.push(-1)
        elif entrada == 's':
                p.push(-2)
```

Código 2.9: Primer programa de prueba (continúa en la siguiente página)

```
        elif entrada == 'e':
                elem = p.pop()
                if elem == -1:
                        p.suma()
                elif elem == -2:
                        p.swap()
                else:
                        p.push(elem)
                        sys.stdout.write("No se puede evaluar un
                           entero")
        elif entrada == 'd':
                p.imprime()
        elif entrada == 'x':
                break;
        else:
                p.push(int(entrada));
```

Código 2.9: Primer programa de prueba

```
CLASS IDENTIFIER(Pila) ( IDENTIFIER(object) ) : NEWLINE
INDENT DEF IDENTIFIER(__init__) ( IDENTIFIER(self) , IDENTIFIER(pila)
    ) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(pila) EQ IDENTIFIER(pila) NEWLINE
DEDENT DEF IDENTIFIER(push) ( IDENTIFIER(self) , IDENTIFIER(elemento)
    ) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(pila) . IDENTIFIER(append) (
    IDENTIFIER(elemento) ) NEWLINE
DEDENT DEF IDENTIFIER(pop) ( IDENTIFIER(self) ) : NEWLINE
INDENT RETURN IDENTIFIER(self) . IDENTIFIER(pila) . IDENTIFIER(pop) (
    ) NEWLINE
DEDENT DEF IDENTIFIER(isEmpty) ( IDENTIFIER(pila) ) : NEWLINE
INDENT RETURN ( IDENTIFIER(self) . IDENTIFIER(pila) EQC [ ] ) NEWLINE
DEDENT DEF IDENTIFIER(suma) ( IDENTIFIER(pila) ) : NEWLINE
INDENT IDENTIFIER(elem1) EQ IDENTIFIER(pila) . IDENTIFIER(pop) ( )
    NEWLINE
IDENTIFIER(elem2) EQ IDENTIFIER(pila) . IDENTIFIER(pop) ( ) NEWLINE
IDENTIFIER(suma) EQ IDENTIFIER(elem1) PLUS IDENTIFIER(elem2) NEWLINE
IDENTIFIER(pila) . IDENTIFIER(push) ( IDENTIFIER(suma) ) NEWLINE
DEDENT DEF IDENTIFIER(swap) ( IDENTIFIER(pila) ) : NEWLINE
INDENT IDENTIFIER(elem1) EQ IDENTIFIER(pila) . IDENTIFIER(pop) ( )
    NEWLINE
```

Salida del analizador léxico correspondiente al Código 3.3 (continúa en la siguiente página)

```
IDENTIFIER(elem2) EQ IDENTIFIER(pila) . IDENTIFIER(pop) ( ) NEWLINE
IDENTIFIER(pila) . IDENTIFIER(push) ( IDENTIFIER(elem1) ) NEWLINE
IDENTIFIER(pila) . IDENTIFIER(push) ( IDENTIFIER(elem2) ) NEWLINE
DEDENT DEF IDENTIFIER(imprime) ( IDENTIFIER(self) ) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(pila) . IDENTIFIER(reverse) ( )
    NEWLINE
FOR IDENTIFIER(i) IN IDENTIFIER(self) . IDENTIFIER(pila) : NEWLINE
INDENT IF IDENTIFIER(i) EQC MINUS INTEGER(1) : NEWLINE
INDENT IDENTIFIER(sys) . IDENTIFIER(stdout) . IDENTIFIER(write) (
    STRING(+) ) NEWLINE
DEDENT ELIF IDENTIFIER(i) EQC MINUS INTEGER(2) : NEWLINE
INDENT IDENTIFIER(sys) . IDENTIFIER(stdout) . IDENTIFIER(write) (
    STRING(s) ) NEWLINE
DEDENT ELSE : NEWLINE
INDENT PRINT IDENTIFIER(i) NEWLINE
DEDENT DEDENT IDENTIFIER(self) . IDENTIFIER(pila) . IDENTIFIER(reverse
    ) ( ) NEWLINE
DEDENT DEDENT IDENTIFIER(entrada) EQ STRING(1) NEWLINE
IDENTIFIER(p) EQ IDENTIFIER(Pila) ( [ ] ) NEWLINE
WHILE IDENTIFIER(entrada) NEQN STRING(x) : NEWLINE
INDENT IDENTIFIER(sys) . IDENTIFIER(stdout) . IDENTIFIER(write) (
    STRING(#) ) NEWLINE
IDENTIFIER(entrada) EQ IDENTIFIER(sys) . IDENTIFIER(stdin) .
    IDENTIFIER(readline) ( ) NEWLINE
IDENTIFIER(entrada) EQ IDENTIFIER(entrada) [ : MINUS INTEGER(1) ]
    NEWLINE
IF IDENTIFIER(entrada) EQC STRING(+) : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(push) ( MINUS INTEGER(1) ) NEWLINE
DEDENT ELIF IDENTIFIER(entrada) EQC STRING(s) : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(push) ( MINUS INTEGER(2) ) NEWLINE
DEDENT ELIF IDENTIFIER(entrada) EQC STRING(e) : NEWLINE
INDENT IDENTIFIER(elem) EQ IDENTIFIER(p) . IDENTIFIER(pop) ( ) NEWLINE
IF IDENTIFIER(elem) EQC MINUS INTEGER(1) : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(suma) ( ) NEWLINE
DEDENT ELIF IDENTIFIER(elem) EQC MINUS INTEGER(2) : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(swap) ( ) NEWLINE
DEDENT ELSE : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(push) ( IDENTIFIER(elem) ) NEWLINE
IDENTIFIER(sys) . IDENTIFIER(stdout) . IDENTIFIER(write) ( STRING(No
    se puede evaluar un entero) ) NEWLINE
```

Salida del analizador léxico correspondiente al Código 3.3 (continúa en la siguiente página)

```
DEDENT DEDENT ELIF IDENTIFIER(entrada) EQC STRING(d) : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(imprime) ( ) NEWLINE
DEDENT ELIF IDENTIFIER(entrada) EQC STRING(x) : NEWLINE
INDENT BREAK ; NEWLINE
DEDENT ELSE : NEWLINE
INDENT IDENTIFIER(p) . IDENTIFIER(push) ( IDENTIFIER(int) ( IDENTIFIER
    (entrada) ) ) ; NEWLINE
DEDENT DEDENT
```

Salida del analizador léxico correspondiente al Código 3.3

```
'''
Created on 15/02/2011

Clase que implementa la funcionalidad basica
de un interprete de pila

@author: Ricchy Alain Perez Chevanier
'''

class Interprete:

    '''
    Constructor
    '''
    def __init__(self):
        self.stack = []

    '''
    Inserta una cadena en el stack que representa al inter-
    prete o realiza acciones segun la cadena proporcionada
    al interprete. Regresa un boolean que indica si el inter
    prete debe de continuar recibiendo cadenas.
    '''
    def insertWord(self, word):
        if word=="x" :
            return False
        elif word=="d" :
            for x in self.stack:
                print x
        elif word=="e" :
```

Código 2.10: Segundo programa de prueba (continúa en la siguiente página)

```
            op = self.stack.pop(0)
            if op== "s" :
                op1 = self.stack.pop(0)
                op2 = self.stack.pop(0)

                self.stack = [op2, op1] + self.stack
            elif op=="+" :
                op1 = int(self.stack.pop(0))
                op2 = int(self.stack.pop(0))
                res = op1 + op2
                self.stack = [res] + self.stack
            else :
                self.stack = [op] + self.stack
        else :
            self.stack = [word] + self.stack
        return True
```

Código 2.10: Segundo programa de prueba

```
STRING(
Created on 15/02/2011

Clase que implementa la funcionalidad basica
de un interprete de pila

@author: Ricchy Alain Perez Chevanier
) NEWLINE
CLASS IDENTIFIER(Interprete) : NEWLINE
INDENT STRING(
    Constructor
    ) NEWLINE
DEF IDENTIFIER(__init__) ( IDENTIFIER(self) ) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(stack) EQ [ ] NEWLINE
DEDENT STRING(
    Inserta una cadena en el stack que representa al inter-
    prete o realiza acciones segun la cadena proporcionada
    al interprete. Regresa un boolean que indica si el inter
    prete debe de continuar recibiendo cadenas.
    ) NEWLINE
DEF IDENTIFIER(insertWord) ( IDENTIFIER(self) , IDENTIFIER(word) ) :
  NEWLINE
```

Salida del analizador léxico correspondiente al Código 3.4 (continúa en la siguiente página)

```
INDENT IF IDENTIFIER(word) EQC STRING(x) : NEWLINE
INDENT RETURN IDENTIFER(False) NEWLINE
DEDENT ELIF IDENTIFIER(word) EQC STRING(d) : NEWLINE
INDENT FOR IDENTIFIER(x) IN IDENTIFIER(self) . IDENTIFIER(stack) :
    NEWLINE
INDENT PRINT IDENTIFIER(x) NEWLINE
DEDENT DEDENT ELIF IDENTIFIER(word) EQC STRING(e) : NEWLINE
INDENT IDENTIFIER(op) EQ IDENTIFIER(self) . IDENTIFIER(stack) .
    IDENTIFIER(pop) ( INTEGER(0) ) NEWLINE
IF IDENTIFIER(op) EQC STRING(s) : NEWLINE
INDENT IDENTIFIER(op1) EQ IDENTIFIER(self) . IDENTIFIER(stack) .
    IDENTIFIER(pop) ( INTEGER(0) ) NEWLINE
IDENTIFIER(op2) EQ IDENTIFIER(self) . IDENTIFIER(stack) . IDENTIFIER(
    pop) ( INTEGER(0) ) NEWLINE
IDENTIFIER(self) . IDENTIFIER(stack) EQ [ IDENTIFIER(op2) , IDENTIFIER
    (op1) ] PLUS IDENTIFIER(self) . IDENTIFIER(stack) NEWLINE
DEDENT ELIF IDENTIFIER(op) EQC STRING(+) : NEWLINE
INDENT IDENTIFIER(op1) EQ IDENTIFIER(int) ( IDENTIFIER(self) .
    IDENTIFIER(stack) . IDENTIFIER(pop) ( INTEGER(0) ) ) NEWLINE
IDENTIFIER(op2) EQ IDENTIFIER(int) ( IDENTIFIER(self) . IDENTIFIER(
    stack) . IDENTIFIER(pop) ( INTEGER(0) ) ) NEWLINE
IDENTIFIER(res) EQ IDENTIFIER(op1) PLUS IDENTIFIER(op2) NEWLINE
IDENTIFIER(self) . IDENTIFIER(stack) EQ [ IDENTIFIER(res) ] PLUS
    IDENTIFIER(self) . IDENTIFIER(stack) NEWLINE
DEDENT ELSE : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(stack) EQ [ IDENTIFIER(op) ] PLUS
     IDENTIFIER(self) . IDENTIFIER(stack) NEWLINE
DEDENT DEDENT ELSE : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(stack) EQ [ IDENTIFIER(word) ]
    PLUS IDENTIFIER(self) . IDENTIFIER(stack) NEWLINE
DEDENT RETURN IDENTIFIER(True) NEWLINE
```

Salida del analizador léxico correspondiente al Código 3.4

```
class Token:
    """Esta clase representa un elemento de StackMachine"""

    def __init__(self, val):
        self.value = val
```

Código 2.11: Tercer programa de prueba (continúa en la siguiente página)

```
    def __str__(self):
        return str(self.value)

class Operator(Token):
    """Token que puede servir como función"""
    def __init__(self, val, representation):
        Token.__init__(self, val)
        self.representation = representation

    def __call__(self,stack):
        self.value(stack)

    def __str__(self):
        return self.representation

class StackMachine:
    """Esta clase representa una calculadora de pila"""
    def __init__(self):
        self.stack = []

    def push(self, token):
        """Agrega un elemento a la parte superior de la pila"""
        self.stack.append(token)

    def pop(self):
        """Quita y regresa el elemento superior de la pila"""
        return self.stack.pop()

    def execute(self):
        """Ejecuta el elemento superior"""
        if len(self.stack) == 0:
            return

        top = self.pop()
        if callable(top):
            top(self)
        else:
            self.push(top)
```

Código 2.11: Tercer programa de prueba (continúa en la siguiente página)

```
    def __str__(self):
        concatenator = lambda accum, x: str(x) + '\n' + accum
        return reduce(concatenator, self.stack, '')


class CommandLineInterpreter:
    """
    Clase que representa una linea de comandos que maneja un
    StackMachine.

    """
    def __init__(self):
        """Crea una linea de comandos con un stack vacío"""
        self.calc = StackMachine()
        self.directives = tables.directives
        self.operators = tables.operators

    def start(self):
        """Entra en un loop a la espera de input de usuario"""
        while True:
            try:
                input_string = raw_input('# ')
                if input_string in self.directives:
                    self.directives[input_string](self.calc)
                elif input_string in self.operators:
                    token = Operator(self.operators[input_string],
                       input_string)
                    self.calc.push(token)
                else:
                    try:
                        self.calc.push(Token(int(input_string)))
                    except ValueError:
                        print 'Entrada invalida'
            except EOFError:
                # Si el cacacter es EOF (^D en la consola) imprimimos
                # un blanco y salimos del ciclo
                print
                break
```

Código 2.11: Tercer programa de prueba (continúa en la siguiente página)

```
if __name__ == '__main__':
    cli = CommandLineInterpreter()
    cli.start()
```

Código 2.11: Tecer programa de prueba

```
CLASS IDENTIFIER(Token) : NEWLINE
INDENT STRING(Esta clase representa un elemento de StackMachine)
    NEWLINE
DEF IDENTIFIER(__init__) ( IDENTIFIER(self) , IDENTIFIER(val) ) :
    NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(value) EQ IDENTIFIER(val) NEWLINE
DEDENT DEF IDENTIFIER(__str__) ( IDENTIFIER(self) ) : NEWLINE
INDENT RETURN IDENTIFIER(str) ( IDENTIFIER(self) . IDENTIFIER(value) )
    NEWLINE
DEDENT DEDENT CLASS IDENTIFIER(Operator) ( IDENTIFIER(Token) ) :
    NEWLINE
INDENT STRING(Token que puede servir como función) NEWLINE
DEF IDENTIFIER(__init__) ( IDENTIFIER(self) , IDENTIFIER(val) ,
    IDENTIFIER(representation) ) : NEWLINE
INDENT IDENTIFIER(Token) . IDENTIFIER(__init__) ( IDENTIFIER(self) ,
    IDENTIFIER(val) ) NEWLINE
IDENTIFIER(self) . IDENTIFIER(representation) EQ IDENTIFIER(
    representation) NEWLINE
DEDENT DEF IDENTIFIER(__call__) ( IDENTIFIER(self) , IDENTIFIER(stack)
    ) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(value) ( IDENTIFIER(stack) )
    NEWLINE
DEDENT DEF IDENTIFIER(__str__) ( IDENTIFIER(self) ) : NEWLINE
INDENT RETURN IDENTIFIER(self) . IDENTIFIER(representation) NEWLINE
DEDENT DEDENT CLASS IDENTIFIER(StackMachine) : NEWLINE
INDENT STRING(Esta clase representa una calculadora de pila) NEWLINE
DEF IDENTIFIER(__init__) ( IDENTIFIER(self) ) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(stack) EQ [ ] NEWLINE
DEDENT DEF IDENTIFIER(push) ( IDENTIFIER(self) , IDENTIFIER(token) ) :
    NEWLINE
INDENT STRING(Agrega un elemento a la parte superior de la pila)
    NEWLINE
IDENTIFIER(self) . IDENTIFIER(stack) . IDENTIFIER(append) ( IDENTIFIER
    (token) ) NEWLINE
```

Salida del analizador léxico correspondiente al Código 3.5 (continúa en la siguiente página)

```
DEDENT DEF IDENTIFIER(pop) ( IDENTIFIER(self) ) : NEWLINE
INDENT STRING(Quita y regresa el elemento superior de la pila) NEWLINE
RETURN IDENTIFIER(self) . IDENTIFIER(stack) . IDENTIFIER(pop) ( )
    NEWLINE
DEDENT DEF IDENTIFIER(execute) ( IDENTIFIER(self) ) : NEWLINE
INDENT STRING(Ejecuta el elemento superior) NEWLINE
IF IDENTIFIER(len) ( IDENTIFIER(self) . IDENTIFIER(stack) ) EQC
    INTEGER(0) : NEWLINE
INDENT RETURN NEWLINE
DEDENT IDENTIFIER(top) EQ IDENTIFIER(self) . IDENTIFIER(pop) ( )
    NEWLINE
IF IDENTIFIER(callable) ( IDENTIFIER(top) ) : NEWLINE
INDENT IDENTIFIER(top) ( IDENTIFIER(self) ) NEWLINE
DEDENT ELSE : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(push) ( IDENTIFIER(top) ) NEWLINE
DEDENT DEDENT DEF IDENTIFIER(__str__) ( IDENTIFIER(self) ) : NEWLINE
INDENT IDENTIFIER(concatenator) EQ LAMBDA IDENTIFIER(accum) ,
    IDENTIFIER(x) : IDENTIFIER(str) ( IDENTIFIER(x) ) PLUS STRING(
) PLUS IDENTIFIER(accum) NEWLINE
RETURN IDENTIFIER(reduce) ( IDENTIFIER(concatenator) , IDENTIFIER(self
    ) . IDENTIFIER(stack) , STRING() ) NEWLINE
DEDENT DEDENT CLASS IDENTIFIER(CommandLineInterpreter) : NEWLINE
INDENT STRING(
     Clase que representa una linea de comandos que maneja un
     StackMachine.

     ) NEWLINE
DEF IDENTIFIER(__init__) ( IDENTIFIER(self) ) : NEWLINE
INDENT STRING(Crea una linea de comandos con un stack vacío) NEWLINE
IDENTIFIER(self) . IDENTIFIER(calc) EQ IDENTIFIER(StackMachine) ( )
    NEWLINE
IDENTIFIER(self) . IDENTIFIER(directives) EQ IDENTIFIER(tables) .
    IDENTIFIER(directives) NEWLINE
IDENTIFIER(self) . IDENTIFIER(operators) EQ IDENTIFIER(tables) .
    IDENTIFIER(operators) NEWLINE
DEDENT DEF IDENTIFIER(start) ( IDENTIFIER(self) ) : NEWLINE
INDENT STRING(Entra en un loop a la espera de input de usuario)
    NEWLINE
WHILE IDENTIFIER(True) : NEWLINE
INDENT TRY : NEWLINE
```

Salida del analizador léxico correspondiente al Código 3.5 (continúa en la siguiente página)

```
INDENT IDENTIFIER(input_string) EQ IDENTIFIER(raw_input) ( STRING(# )
    ) NEWLINE
IF IDENTIFIER(input_string) IN IDENTIFIER(self) . IDENTIFIER(
    directives) : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(directives) [ IDENTIFIER(
    input_string) ] ( IDENTIFIER(self) . IDENTIFIER(calc) ) NEWLINE
DEDENT ELIF IDENTIFIER(input_string) IN IDENTIFIER(self) . IDENTIFIER(
    operators) : NEWLINE
INDENT IDENTIFIER(token) EQ IDENTIFIER(Operator) ( IDENTIFIER(self) .
    IDENTIFIER(operators) [ IDENTIFIER(input_string) ] , IDENTIFIER(
    input_string) ) NEWLINE
IDENTIFIER(self) . IDENTIFIER(calc) . IDENTIFIER(push) ( IDENTIFIER(
    token) ) NEWLINE
DEDENT ELSE : NEWLINE
INDENT TRY : NEWLINE
INDENT IDENTIFIER(self) . IDENTIFIER(calc) . IDENTIFIER(push) (
    IDENTIFIER(Token) ( IDENTIFIER(int) ( IDENTIFIER(input_string) ) )
    ) NEWLINE
DEDENT EXCEPT IDENTIFIER(ValueError) : NEWLINE
INDENT PRINT STRING(Entrada invalida) NEWLINE
DEDENT DEDENT DEDENT EXCEPT IDENTIFIER(EOFError) : NEWLINE
INDENT PRINT NEWLINE
BREAK NEWLINE
DEDENT DEDENT DEDENT DEDENT IF IDENTIFIER(__name__) EQC STRING(
    __main__) : NEWLINE
INDENT IDENTIFIER(cli) EQ IDENTIFIER(CommandLineInterpreter) ( )
    NEWLINE
IDENTIFIER(cli) . IDENTIFIER(start) ( ) NEWLINE
DEDENT
```

Salida del analizador léxico correspondiente al Código 3.5

Como se puede observar a partir de lo anterior, hemos elegido que en este caso el analizador léxico genere la salida basado en texto, con la que contamos gracias a la utlilización del patrón Constructor. Es claro aquí que el diseño del compilador, en particular de la implementación del ConstructorConcreto que genera la salida basado en texto, es de gran ayuda para propósitos de depuración y para ver de qué forma se está comportando cada módulo del compilador, en este caso el analizador léxico.

En todos los casos la salida que genera nuestra implementación del analizador es correcta, ya que genera exactamente los átomos que se esperaba que generara.

Con esto damos por concluido el análisis léxico.

CAPÍTULO 3

Análisis sintáctico

Conocimientos supuestos

Para la etapa del análisis sintáctico estamos suponiendo que el lector conoce lo siguiente:

1. Definiciones de:
   - Gramáticas y su clasificación.
   - Lenguajes generados por una gramática.
   - Gramáticas libres del contexto.
   - Formas Normales de gramáticas libres del contexto.
   - Relación entre tipo de gramáticas y tipo de un lenguaje.
   - Derivaciones, derivación por la izquierda, derivación por la derecha.
   - Forma sentencial.
   - Árboles de derivación total y parcial.
   - Frontera de un árbol de derivación.
   - Autómatas de pila deterministas (APD) y no–deterministas (APN).
   - Reconocimiento en lenguajes libres del contexto.
   - Ambigüedad en lenguajes libres del contexto.
   - Complejidad de algoritmos.
2. Los algoritmos para:
   - Obtener formas normales de gramáticas libres del contexto, que incluyen:
     - Eliminación de recursividad izquierda en gramáticas libres del contexto.
     - Fusión de producciones en gramáticas libres del contexto.

4. Resultados:
    - Equivalencia entre autómatas de pila y gramáticas libres del contexto.
    - Existencia de lenguajes libres del contexto inherentemente ambiguos.

Estos temas se pueden encontrar en las mismas referencias que dimos en el capítulo anterior.

3.1. Introducción

Una definición completa de un lenguaje de programación debe incluir la especificación tanto de su sintaxis (estructura) como de su semántica (significado).

La labor del análisis sintáctico es verificar que el programa de entrada que va a traducir el compilador sea sintácticamente válido de acuerdo a las reglas establecidas en la especificación de la sintaxis del lenguaje; esto es, que la secuencia de enunciados que conforman el código fuente tenga la estructura correcta.

La sintaxis de un lenguaje de programación se define mediante una gramática (que nos permite capturar la estructura del código fuente); no así la semántica en la que se han utilizado diferentes mecanismos para definirla como veremos en el siguiente capítulo.

Es claro que para llevar a cabo su labor el analizador sintáctico puede leer directamente el programa de entrada e ir realizando la verificación. No obstante tendría que encargarse al mismo tiempo de ir obteniendo los átomos (símbolos terminales) que forman parte de la gramática que define su sintaxis, para que, una vez que éstos hayan sido obtenidos, verificar si una secuencia de ellos empata con alguna producción. En lugar de lo anterior, generalmente este trabajo se separa en dos fases, una en la que se forman los átomos y otra en la que se verifica que empaten con alguna de las producciones, lo que da como resultado un compilador modular donde el trabajo de cada una de estas fases es más claro, limpio e independiente. Por ejemplo, la primera de estas fases puede detectar errores que son exclusivamente léxicos, como átomos mal formados, mientras que la segunda puede concentrarse en reportar errores sintácticos como enunciados incorrectos. Como estudiamos en el capítulo anterior, nuestro compilador sigue este diseño. De esta manera en nuestro análisis sintáctico debemos preocuparnos por verificar la estructura sintáctica a partir de los átomos que genera nuestro analizador léxico.

Analicemos ahora qué tipo de gramática podemos utilizar en la definición de la sintaxis de un lenguaje de programación. Con este fin, traeremos a colación algunos de los conceptos que ya conocemos sobre gramáticas, pero adaptados al análisis sintáctico, que es el que nos ocupa.

3.2. Ambigüedad

Como sabemos, el problema de presencia de ambigüedad es importante en los lenguajes de programación, y en sus procesadores, pues pueden provocar ejecuciones distintas del mismo código.

Definición 3.1. Sea $G = (N, T, P, S)$ una gramática libre del contexto y supongamos que las producciones en P están numeradas $1, 2, \ldots, p$. Sea $\alpha \in (N \cup T)^*$. Entonces:

1. Un *análisis sintáctico izquierdo* de α es una secuencia de producciones utilizada en una derivación por la izquierda de α a partir de S.

2. Un *análisis sintáctico derecho* de α es la reversa de una secuencia de producciones utilizadas en una derivación por la derecha de α a partir de S.

Podemos representar estos análisis sintácticos con una secuencia de números desde 1 hasta p.

Utilizaremos el símbolo $\underset{n}{\Longrightarrow}$ para indicar que la producción número n se aplicó en ese paso de la derivación.

Recordemos que una gramática libre del contexto es ambigua si, para una misma palabra del lenguaje generado, existen dos o más árboles de derivación estructuralmente distintos.

Un ejemplo clásico e ilustrativo de la presencia de ambigüedad se muestra a continuación y corresponde a una gramática sencilla que genera expresiones aritméticas sin precedencia para sus operadores.

Ejemplo: Sea $G = (N, T, P, S)$ una gramática libre del contexto con las siguientes producciones:

(1) $E \longrightarrow E + E$
(2) $E \longrightarrow E * E$
(3) $E \longrightarrow (E)$
(4) $E \longrightarrow \mathbf{id}$

Ahora consideremos el enunciado $5 + 7 * 2$, que tiene las siguientes derivaciones por la izquierda:

1. $E \underset{1}{\Longrightarrow} E + E \underset{4}{\Longrightarrow} \mathbf{id} + E \underset{2}{\Longrightarrow} \mathbf{id} + E * E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} * E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}$
2. $E \underset{2}{\Longrightarrow} E * E \underset{1}{\Longrightarrow} E + E * E \underset{4}{\Longrightarrow} \mathbf{id} + E * E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} * E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}$

y sus correspondientes árboles de derivación:

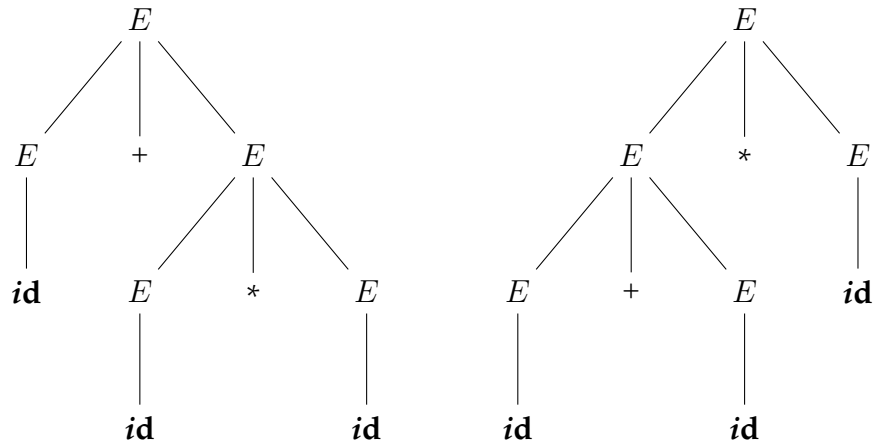


Figura 3.1

Es fácil notar que cada uno de ellos tiene semánticas diferentes;[1] y por tanto, la ambigüedad es una propiedad de las gramáticas que se desea evitar siempre.[2]

Veamos otro ejemplo considerando ahora el enunciado $5 + 2 + 4$, que tiene las siguientes derivaciones por la izquierda:

1. $E \underset{1}{\Longrightarrow} E + E \underset{4}{\Longrightarrow} \mathbf{id} + E \underset{1}{\Longrightarrow} \mathbf{id} + E + E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} + E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} + \mathbf{id}$.
2. $E \underset{1}{\Longrightarrow} E + E \underset{1}{\Longrightarrow} E + E + E \underset{4}{\Longrightarrow} \mathbf{id} + E + E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} + E \underset{4}{\Longrightarrow} \mathbf{id} + \mathbf{id} + \mathbf{id}$.

y sus respectivos árboles de derivación son:

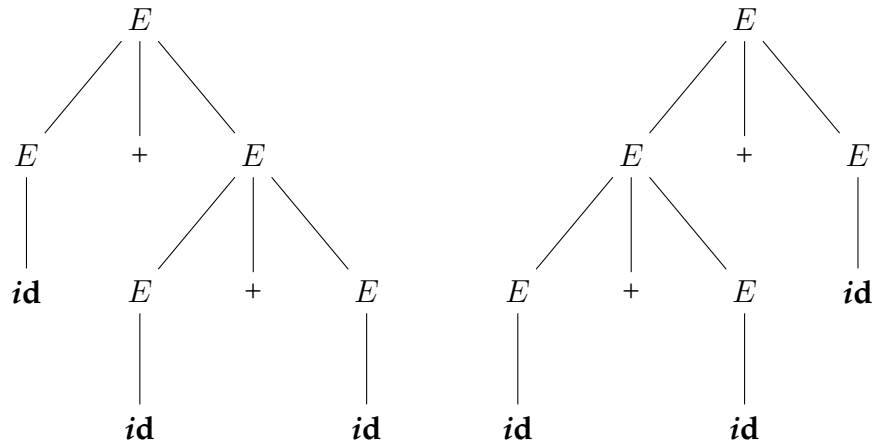


Figura 3.2

[1] La asociatividad y precedencia son distintas lo que no calculan el mismo valor.
[2] Sin embargo esto no es siempre posible como veremos más adelante.

En el segundo ejemplo, aunque existen dos derivaciones por la izquierda para el enunciado $5 + 2 + 4$, ambas tienen la misma semántica; a este tipo de ambigüedad, que se presenta entre las derivaciones 1 y 2 de este ejemplo, se le llama **ambigüedad espuria** (*spurius ambiguity*), mientras que la que se presenta entre las derivaciones 1 y 2 del primer ejemplo se denomina **ambigüedad esencial** (*essential ambiguity*), siendo esta última la más fuerte e indeseable. En otras palabras, podemos decir que la ambigüedad espuria se presenta cuando un enunciado tiene dos o más derivaciones (ya sea por la derecha o por la izquierda) pero en todas éstas el resultado de realizar sus respectivas acciones semánticas es el mismo, mientras que en la ambigüedad esencial el resultado de realizar sus respectivas acciones semánticas es distinto.

En general siempre se busca trabajar con gramáticas que no presenten ningún tipo de ambigüedad.

Como hemos mencionado, debemos evitar la ambigüedad siempre que sea posible, pues a ningún programador le gustaría que uno o más de los enunciados de su código fuente, se evaluaran de dos o más formas distintas.

En nuestros ejemplos, con base en nuestro conocimiento previo de las reglas de precedencia y asociatividad habituales de las operaciones aritméticas, sabemos que la derivación 1 es la derivación correcta en el primero de nuestros ejemplos y que la derivación 2 es la derivación correcta en el segundo. El problema radica en que la gramática no refleja dichas reglas, por lo que es necesario reescribir la gramática de forma en que las reglas de asociatividad y precedencia de las expresiones aritméticas se vean reflejadas. La siguiente gramática remedia esta situación pues genera el mismo lenguaje, reflejando las reglas de precedencia y asociatividad habituales, y no presenta ambigüedades.

1) $E \longrightarrow T$
2) $E \longrightarrow E + T$
3) $T \longrightarrow F$
4) $T \longrightarrow T * F$
5) $F \longrightarrow (E)$
6) $F \longrightarrow \boldsymbol{i}\mathbf{d}$

Ahora veamos la derivación por la izquierda del enunciado $5+7*2$ en esta gramática (que utilizamos en el primero de nuestro ejemplos).

$$1.\ E \underset{2}{\Longrightarrow} E + T \underset{1}{\Longrightarrow} T + T \underset{3}{\Longrightarrow} F + T \underset{6}{\Longrightarrow} \boldsymbol{i}\mathbf{d} + T \underset{4}{\Longrightarrow} \boldsymbol{i}\mathbf{d} + T * F \underset{3}{\Longrightarrow} \boldsymbol{i}\mathbf{d} + F * F \underset{6}{\Longrightarrow}$$
$$\Longrightarrow \boldsymbol{i}\mathbf{d} + \boldsymbol{i}\mathbf{d} * F \underset{6}{\Longrightarrow} \boldsymbol{i}\mathbf{d} + \boldsymbol{i}\mathbf{d} * \boldsymbol{i}\mathbf{d}.$$

donde su correspondiente árbol de derivación es:

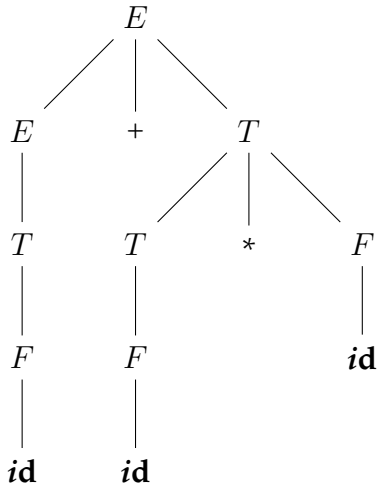


Figura 3.3

En este caso tenemos una única derivación por la izquierda y un único árbol de derivación para el enunciado $5 + 7 * 2$.

De este ejemplo podemos ver que la ambigüedad es una propiedad de la gramática y no del lenguaje, pues como hemos visto hemos expuesto dos gramáticas que generan el mismo lenguaje, sólo que una de ellas es ambigua y la otra no lo es. Sin embargo, como ya sabemos, hay lenguajes inherentemente ambiguos, cuya ambigüedad no depende de la gramática.

Como mencionamos, la ambigüedad de una gramática es una característica indeseable para el proceso de compilación. Desafortunadamente, es indecidible determinar cuando una gramática libre del contexto G es ambigua [**pp.203**; AU72]. También es indecidible determinar si un lenguaje libre del contexto es inherentemente ambiguo [**pp.206**; HU79].

3.3. Gramáticas en la sintaxis de un lenguaje de programación

Lo que nos interesa en el análisis sintáctico es: dado un programa de entrada w encontrar un análisis sintáctico (ya sea izquierdo o derecho) del programa w; como se puede observar no sólo nos basta con decidir si w pertenece al lenguaje generado por la gramática que especifica la sintaxis del lenguaje de programación, sino que además debemos conocer la secuencia de producciones que participan en la derivación de w; pero, más aún; en la práctica, cada vez que destapamos un símbolo no terminal (es decir,

utilizamos una producción) se realiza una acción[3] asociada con dicha producción (que en el caso de un compilador suele ser construir el nodo del árbol de sintaxis abstracta que corresponde a la categoría sintáctica presente en el lado izquierdo de la producción); en otras palabras, nos interesa conocer la estructura de w, por lo que es indispensable trabajar con gramáticas que permitan en primer instancia que podamos obtener un análisis sintáctico a partir de un programa de entrada, pero que además lo obtengamos de manera eficiente.

Hagamos una breve justificación para la elección de gramáticas libres del contexto. Las gramáticas tipo 0 quedan rápidamente descartadas dado que el problema de aceptación es indecidible para los lenguajes tipo 0. El segundo tipo de gramáticas que quedan descartadas son las gramáticas tipo 1, debido a que, aunque en principio el problema de aceptación es decidible, esto es, siempre podemos obtener un análisis, todos los algoritmos de análisis que se conocen para este tipo de gramáticas toman tiempo exponencial, lo que sin duda las deja sin posibilidades de uso dentro de un compilador. Afortunadamente la situación para las gramáticas libres del contexto es completamente diferente, ya que para ellas existen algoritmos que nos permiten encontrar un análisis determinado en tiempo cúbico; en otras palabras, se conocen al menos dos algoritmos, el algoritmo CYK (*Cocke-Younger-Kasami*) y el algoritmo de Earley, que para cualquier gramática libre del contexto nos permiten encontrar un análisis en tiempo $O(n^3)$.

Aun cuando esto representa una mejora significativa respecto de los casos anteriores, todavía resulta ser ineficiente su uso en un compilador. La causa principal de que estos algoritmos sean de $O(n^3)$ y no tengan un mejor desempeño es el no determinismo que se genera en el autómata de pila construido a partir de la gramática libre del contexto; a diferencia de los autómatas finitos en los que existe una equivalencia entre los AFN y AFD, no existe una equivalencia entre los APN y APD, lo que da como resultado una nueva clasificación de lenguajes, los lenguaje libres del contexto deterministas (que forman un subconjunto propio de los lenguajes libres del contexto), que son aquellos lenguajes aceptados por un APD y son el principal objeto de estudio en el análisis sintáctico en un curso de compiladores, ya que existen algoritmos para este tipo de lenguajes que nos permiten obtener el análisis de un programa en tiempo $O(n)$.

La siguiente pregunta relevante en este contexto es qué tipo de gramáticas (mínimas) generan a los lenguajes libres del contexto deterministas y éstas resultan ser las gramáticas LR (que estudiaremos con detenimiento más adelante). Además, (como se esperaba) tenemos que dado un APD que acepta el lenguaje L podemos obtener una gramática LR que genere a L; en sentido contrario, si tenemos una gramática LR que genera el lenguaje L entonces podemos construir a partir de ella un APD que acepte a L, por lo que es equivalente definir un lenguaje libre del contexto determinista como aquel que es generado por una gramática mínima LR.

[3]Llamada acción semántica en la literatura.

Por último, las gramáticas tipo 3 o regulares son un subconjunto propio de las gramáticas libres del contexto deterministas, con lo que resulta que con base en ellas podemos obtener análisis sintácticos eficientes. Sin embargo, dada la limitación en la forma de sus producciones, no son capaces de generar varias de las estructuras presentes en los lenguajes de programación comunes. Podemos concluir, entonces, que las gramáticas que generalmente se utilizan en la especificación de un lenguaje de programación son las gramáticas LR, también llamadas libres del contexto deterministas, y que son el tema de estudio en lo que resta del presente capítulo.

Estudiemos ahora cómo dada una gramática podemos construir un analizador sintáctico que genere análisis sintácticos para los enunciados de entrada que le presentemos.

En la práctica lo que nos interesa es que cuando presentemos un programa de entrada al analizador sintáctico, éste encuentre una derivación de w, pero que además, conforme vaya encontrando los pasos (correctos) de la derivación, vaya realizando las acciones semánticas correspondientes al símbolo no–terminal del paso actual.

3.4. Analizadores sintácticos

Existen dos estrategias generales que un analizador sintáctico puede seguir para encontrar una derivación. La primera de ellas es comenzar por el símbolo inicial e ir sustituyendo los símbolos no terminales con los lados derechos de sus producciones, con la esperanza de obtener una derivación del programa de entrada, que se conoce como de arriba hacia abajo o descendente (*top-down*), dado que en el árbol de derivación se comienza en la raíz y va creciendo hacia abajo hasta llegar a las hojas; en cambio, en la segunda estrategia, a partir del enunciado w se trata de reconstruir la derivación que nos llevó a w (suponiendo que existe) siguiendo la dirección opuesta en los pasos de la derivación, es decir se van haciendo sustituciones (conocidas como reducciones) de las cadenas presentes en el cuerpo de las producciones por su respectivo lado izquierdo, con la esperanza de llegar al símbolo inicial. Esta estrategia se conoce como de abajo hacia arriba o ascendente (*bottom-up*), pues se comienza en las hojas del árbol de derivación que corresponden al enunciado de entrada w y va subiendo, compactando la forma sentencial hasta llegar a la raíz. Existen además dos formas de ir leyendo la entrada, una es de izquierda a derecha y la otra de derecha izquierda, aunque generalmente la única que se utiliza en los analizadores sintácticos es de izquierda a derecha.

Vamos a estudiar cada uno de los analizadores que presentemos, primero analizando su funcionamiento abstracto y después dando detalles para llevar a cabo su implementación; además comenzaremos relajando la condición de que los analizadores tomen tiempo lineal. Para estudiar el funcionamiento abstracto de los analizadores utilizaremos como herramienta un autómata que denominaremos *autómata de análisis*, que en

principio cuenta con una pila de análisis y una pila de predicción, pero que puede tener tantas de cada una de éstas según se requiera y que toma como entrada un enunciado w; a la configuración de este autómata en un instante de tiempo determinado la llamamos *descripción instantánea* y se representa gráficamente de la siguiente manera:

entrada empatada	entrada por empatar
pila de análisis$_1$	pila de predicción$_1$
$\vdots$	$\vdots$
pila de análisis$_n$	pila de predicción$_n$

Figura 3.4: Representación gráfica de una descripción instantánea del autómata de análisis.

En estas descripciones instantáneas se divide a la cadena en 2 partes, lo que ya se vio y lo que falta por ver. Lo que ya se vio -y se logró empatar- tiene aparejada su pila de análisis, mientras lo que falta por ver tiene aparejada su pila de predicción.

Mostraremos ahora el funcionamiento del autómata de análisis mediante un ejemplo. Supongamos que tenemos la siguiente gramática:

$$\begin{aligned} S &\longrightarrow DC \mid AB \\ A &\longrightarrow a \mid aA \\ B &\longrightarrow bc \mid bBc \\ D &\longrightarrow ab \mid aDb \\ C &\longrightarrow c \mid cC \end{aligned}$$

Figura 3.5

y tenemos como enunciado de entrada $aabc$; lo primero que haremos es agregar el símbolo de fin de cadena $\#$ al final de la cadena de entrada, además lo agregaremos al inicio del análisis después del símbolo inicial S en la pila de predicción; lo anterior lo hemos agregado sólo por comodidad para detectar que el análisis ha terminado; de esta forma al inicio el autómata de análisis se encuentra así:

1)

	$aabc\#$
	$S\#$

Figura 3.6

y los subsecuentes pasos en el análisis se muestran a continuación.

Cada línea en la figura representa una pila dado que hay más de una elección para expandir el símbolo no terminal.

2)

	$aabc\#$
S_1	$DC\#$
S_2	$AB\#$

3)

	$aabc\#$
S_1D_1	$abC\#$
S_1D_2	$aDbC\#$
S_2A_1	$aB\#$
S_2A_2	$aAB\#$

4)

a	$abc\#$
S_1D_1a	$bC\#$
S_1D_2a	$DbC\#$
S_2A_1a	$B\#$
S_2A_2a	$AB\#$

5)

a	$abc\#$
S_1D_1a	$bC\#$
$S_1D_2aD_1$	$abbC\#$
$S_1D_2aD_2$	$aDbbC\#$
$S_2A_1aB_1$	$bc\#$
$S_2A_1aB_2$	$bBc\#$
$S_2A_2aA_1$	$aB\#$
$S_2A_2aA_2$	$aAB\#$

6)

aa	$bc\#$
$S_1D_2aD_1a$	$bbC\#$
$S_1D_2aD_2a$	$DbbC\#$
$S_2A_2aA_1a$	$B\#$
$S_2A_2aA_2a$	$AB\#$

7)

aa	$bc\#$
$S_1D_2aD_1a$	$bbC\#$
$S_1D_2aD_2aD_1$	$abbbC\#$
$S_1D_2aD_2aD_2$	$aDbbbC\#$
$S_2A_2aA_1aB_1$	$bc\#$
$S_2A_2aA_1aB_2$	$bBc\#$
$S_2A_2aA_2aA_1$	$aB\#$
$S_2A_2aA_2aA_2$	$aAB\#$

8)

aab	$c\#$
$S_1D_2aD_1ab$	$bC\#$
$S_2A_2aA_1aB_1b$	$c\#$
$S_2A_2aA_1aB_2b$	$Bc\#$

9)

aab	$c\#$
$S_1D_2aD_1ab$	$bC\#$
$S_2A_2aA_1aB_1b$	$c\#$
$S_2A_2aA_1aB_2bB_1$	$bcc\#$
$S_2A_2aA_1aB_2bB_1$	$bBcc\#$

10)

$aabc$	$\#$
$S_2A_2aA_1aB_1bc$	$\#$

Figura 3.7

Tal como se muestra en las descripciones instantáneas del autómata de análisis en la figura 3.7, tanto los respectivos topes de las pilas de análisis como los de las pilas de pre-

dicción están marcado por la línea horizontal en la representación gráfica; no obstante, los respectivos fondos de las pilas de análisis se encuentran a la izquierda mientras que los de las pilas de predicción se encuentran a la derecha. Como se muestra en el inciso 4) de la figura 3.6, el autómata comienza con una única pila de análisis vacía y una pila de predicción que contiene la cadena $S\#$, es decir, el símbolo inicial de la gramática junto con el símbolo de fin de cadena; además, en la parte superior del lado izquierdo se muestra la parte del enunciado de entrada que ha sido procesado (empatado) hasta el momento (al inicio desde luego esta parte se encuentra vacía), mientras que del lado superior derecho se encuentra la parte del enunciado que aún no ha sido procesada (empatada).

Por otra parte, siempre que tengamos un símbolo no terminal N en el tope de la pila de predicción, debemos elegir una producción que tenga como lado izquierdo el símbolo N para poder utilizarla dentro del análisis. Como veremos con detalle más adelante, quisiéramos siempre elegir la producción correcta, es decir, aquella que forme parte del análisis (y por tanto que se utilice en la derivación) del enunciado de entrada (en caso de que exista). Sin embargo esto no es siempre posible, por lo que tenemos que hacer una predicción de cuál será la producción correcta. Es por esto que llamamos a este tipo de paso dentro del análisis *paso de predicción*.

Cuando tenemos dos o más producciones que tienen como lado izquierdo el símbolo N, como el caso de S en nuestro ejemplo que tenemos las producciones $S \longrightarrow DC \mid AB$, utilizaremos la notación N_i con $1 \leq i \leq n$ donde n es el número de producciones distintas que tienen como lado izquierdo N, y la i se refiere a la i-ésima producción que tiene como lado izquierdo N.

Al terminar el paso 4) debemos realizar nuestro paso de predicción. Como hasta el momento no hemos desarrollado técnicas para saber cuál es la predicción correcta y poder elegirla, debemos explorar todas las posibles producciones,[4] es decir, en este paso tenemos que explorar tanto la producción $S \longrightarrow DC$ como la producción $S \longrightarrow AB$; como tenemos que seguir los dos posibles caminos (las dos posibles derivaciones), necesitamos una nueva pila de análisis y predicción respectivamente para seguir el rastro de la segunda producción. En general, cuando tengamos n posibles producciones en un paso de predicción agregaremos $n-1$ pilas de análisis y predicción respectivamente. Una vez que contamos con el número adecuado de pilas hacemos un push del símbolo no terminal N con su respectivo subíndice (que indica la producción que se está utilizando) en la pila de análisis correspondiente a ese camino, y en la respectiva pila de predicción reemplazamos N con el lado derecho de la producción que estamos utilizando (es decir, hacemos un pop y después un push α, donde α es el cuerpo de la producción), con lo que nuestro autómata queda como se muestra en el paso 2). En el paso 2) hacemos de nuevo un paso de predicción; cuando tenemos un símbolo terminal t en el tope de la pila de predicción, debemos empatar con el símbolo terminal actual de la entrada (que en

[4] Es claro que por seguir esta estrategia, nuestro analizador es un analizador ingenuo general.

nuestra representación es el que está en la parte superior inmediatamente a la derecha de la línea vertical que marca el tope de las pilas); es por eso que llamamos a este tipo de paso *paso de empate*: si ambos caracteres empatan entonces hacemos un push t en la pila de análisis correspondiente y movemos t a la parte de la entrada empatada; en caso de que los caracteres no empaten, concluimos que estos respectivos caminos fracasan y dejan de existir. En el paso 3) se realiza un paso de empate donde todos los caminos empatan. Nótese que en el paso 4) el primer camino está listo para realizar un paso de empate; no obstante, no se debe realizar un paso de empate hasta que todos los posibles caminos estén listos para realizarlo, es decir, siempre que haya al menos un camino con un símbolo no terminal en el tope de la pila de predicción debemos realizar un paso de predicción; una vez que todos los posibles caminos tienen un símbolo terminal en el tope de su pila de predicción debemos realizar un paso de empate; por lo que en el paso 5) realizamos un paso de empate, pero esta vez no todos los caminos tienen éxito como lo muestra el paso 6). Al final, en el paso 10), sólo nos resta realizar un paso de empate con el carácter de fin de cadena, el cuál tiene éxito y por lo tanto hemos encontrado un análisis. Para obtener el análisis que hemos encontrado simplemente tomamos la cadena que quedó dentro de la pila de análisis y eliminamos los símbolos terminales que forman parte de ella, con lo que en nuestro ejemplo obtenemos el análisis: $S_2A_2A_1B_1$, que corresponde a la derivación por la izquierda $S \Longrightarrow AB \Longrightarrow aAB \Longrightarrow aaB \Longrightarrow aabc$.

3.4.1. Analizadores sintácticos descendentes (*top–down*)

El analizador que acabamos de presentar como ejemplo es un analizador de arriba hacia abajo general, en amplitud, que encuentra la derivación por la izquierda. En general, los analizadores de arriba hacia abajo encuentran la derivación por la izquierda; es claro que, al menos en principio, este analizador pareciera funcionar para cualquier tipo de gramática libre del contexto, aunque uno de sus inconvenientes (que podemos intuir a partir del ejemplo) es que toma tiempo y espacio exponencial en el peor de los casos. El que tome espacio exponencial es debido a que explora todos los posibles caminos en amplitud, es decir, cuando tiene que hacer un paso de predicción crea tantas pilas como sean necesarias; si en lugar de esto explorara un único posible camino, entonces tomaría espacio lineal. Desearíamos poderlo modificar para que en cada paso de predicción explorara un único posible camino y en caso de que éste fracasara hacer retroceso (*backtrack*) y explorar el siguiente, hasta encontrar un análisis o, en su defecto, hasta que no haya más caminos por explorar y todos hayan fracasado (en cuyo caso no existe un análisis); es decir, explorar los caminos en profundidad, con lo que obtenemos un analizador descendente general en profundidad que toma espacio lineal y tiempo exponencial.

Recursividad izquierda

Aunque los métodos anteriores parecieran a primera vista que funcionan para cualquier gramática libre del contexto tienen un serio inconveniente. Analicemos como funcionan con la siguiente gramática:

$$\begin{aligned} S &\longrightarrow aB \\ B &\longrightarrow Bb \\ B &\longrightarrow b \end{aligned}$$

Figura 3.8

Veamos por ejemplo los primeros pasos para el analizador en profundidad con la cadena de entrada abb

1)

	$abb\#$
	$S\#$

2)

	$abb\#$
S	$aB\#$

3)

a	$bb\#$
Sa	$B\#$

4)

a	$bb\#$
SaB_1	$Bb\#$

5)

a	$bb\#$
SaB_1B_1	$Bbb\#$

6)

a	$bb\#$
$SaB_1B_1B_1$	$Bbbb\#$

7) .....

Figura 3.9

Es claro que a partir del paso 3, se realizan sólo pasos de predicción y no hay forma de que se llegue a un paso de empate, lo que ocasiona que el analizador nunca pare. El problema se genera debido a la forma de la producción $B \longrightarrow Bb$, que presenta lo que se conoce como *recursividad izquierda*. Si una gramática tiene al menos un símbolo no terminal recursivo izquierdo o derecho, se dice que es *recursiva izquierda* o *recursiva derecha* según sea el caso.

Los analizadores de arriba hacia abajo que encuentran la derivación por la izquierda pueden entrar en un ciclo infinito cuando trabajan con gramáticas recursivas izquierdas. Este es el caso que vimos de los analizadores anteriores. Afortunadamente, tenemos un algoritmo –ver [**pp.213**; Aho+07]– que elimina la recursividad por la izquierda en una gramática.

De esta forma siempre que necesitemos hacer un analizador de arriba hacia abajo para una gramática con recursividad izquierda, debemos primero transformarla en una gramática sin recursividad izquierda aplicando el algoritmo mencionado.

Analizador de descenso recursivo

Continuemos nuestro estudio con la implementación de un analizador de arriba hacia abajo. En principio podemos escribir un programa que imite el comportamiento del analizador en profundidad que hemos estudiado anteriormente, tomando como entrada una gramática G y un enunciado w, y nos devuelva (en caso de que exista) el análisis sintáctico de w. Esta implementación resulta ser general en el sentido que no está hecha específicamente para una gramática, sino que trabaja con la gramática que se le dé como entrada. Aunque en principio ésta pareciera ser una buena estrategia, en el contexto de compiladores, como ya lo hemos mencionado, generalmente se deben realizar acciones semánticas asociadas con las producciones durante el análisis, así que tendríamos que modificar nuestro programa para que sea capaz de realizarlas; pero además, el hecho de que esta implementación sea general conlleva el costo que en principio es menos eficiente, en comparación con una implementación de un analizador hecho a la medida para una única gramática. Es por esto que cuando se implementan analizadores de arriba hacia abajo que encuentran la derivación por la izquierda, se cuenta con una técnica de implementación muy utilizada que permite realizar acciones semánticas durante el análisis y que está hecho a la medida para una única gramática; esta técnica se conoce como *descenso recursivo*, la cuál, en su forma más general, hace uso de retroceso para probar las diferentes producciones hasta encontrar las producciones correctas que nos lleven a la derivación del enunciado de entrada w. Los analizadores predictivos de arriba hacia abajo generalmente se implementan como un caso especial de descenso recursivo sin retroceso (más adelante veremos este tipo de analizadores). En un analizador de descenso recursivo hay una función por cada símbolo no terminal A, donde en el cuerpo de cada una de estas funciones se debe probar cada una de las producciones que tengan como lado izquierdo el símbolo no terminal A correspondiente; además se cuenta con un apuntador que señala el carácter actual a ser consumido en la cadena de entrada. Cada llamada a alguna de las funciones descritas previamente corresponde a un paso de predicción, mientras que si en el cuerpo de una producción se encuentra un símbolo terminal entonces se debe verificar que éste corresponda con el símbolo actual de la cadena de entrada; de ser así se realiza un paso de empate y el apuntador avanza al siguiente carácter de la cadena de entrada; en caso contrario, la producción ha fallado y se debe probar otra. Debemos notar que si una producción falla tendremos que restablecer el apuntador en la posición en la que se encontraba antes de probar la producción que falló. Consideremos la siguiente gramática:

$$
\begin{array}{lcl}
S & \longrightarrow & AB \mid DC \\
A & \longrightarrow & a \mid aA \\
B & \longrightarrow & bc \mid bBc \\
D & \longrightarrow & ab \mid aDb \\
C & \longrightarrow & c \mid cC
\end{array}
$$

Figura 3.10

Veamos una porción del código en C correspondiente al analizador de descenso recursivo de la gramática que acabamos de presentar:

```
1  char* input;
2  int ind;
3  nodep* last;
4
5  int S(){
6    int li = ind;
7    nodep* pcu = last;
8    addprod("S->AB");
9    if(A() && B()){
10     return TRUE;
11   }
12   removecu(pcu);
13   ind = li;
14   addprod("S->DC");
15   if(D() && C()){
16     return TRUE;
17   }
18   removecu(pcu);
19   return FALSE;
20 }
21
22 int A(){
23   int li = ind;
24   nodep* pcu = last;
25   addprod("A->aA");
26   if(input[ind]=='a'){
27     ind++;
28     if(A()){
29       return TRUE;
```

Código 3.1: Fragmento del analizador de descenso recursivo para la gramática de la figura 3.10

(continúa en la siguiente página)

```
30        }
31      }
32      removecu(pcu);
33      addprod("A->a");
34      ind = li;
35      if(input[ind]=='a'){
36        ind++;
37        return TRUE;
38      }
39      removecu(pcu);
40      return FALSE;
41    }
```

Código 3.1: Fragmento del analizador de descenso recursivo para la gramática de la figura3.10

En el código anterior, en input almacenamos la cadena de entrada, ind es un índice que nos sirve como apuntador al carácter actual de la cadena de entrada y last es un apuntador al elemento en el tope de la pila donde almacenamos las producciones correspondientes al camino actual del analizador. Veamos ahora el código correspondiente al símbolo inicial S: tenemos una función con nombre S que regresa un entero;[5] en el cuerpo de la función primero se prueba la producción $S \longrightarrow AB$ a partir de la línea 8; si la llamada a la función A regresa verdadero significa que el paso de predicción es correcto y ahora debemos llamar a B; si ésta regresa verdadero significa que también este paso de predicción es correcto y por tanto la producción $S \longrightarrow AB$ es correcta y regresamos verdadero; en caso contrario, si A regresa falso, concluimos que la producción $S \longrightarrow AB$ no es correcta y debemos intentar la siguiente, lo mismo que si A regresa verdadero pero B falla; en la línea 12 eliminamos de la pila el camino que falló, restableciendo su contenido como se encontraba antes de que iniciara la predicción incorrecta. De la misma manera restablecemos el apuntador al carácter actual de la cadena de entrada. Nótese que para llevar a cabo lo anterior tenemos que guardar en variables locales los valores tanto del apuntador al tope de la pila, como del apuntador a la cadena actual, que en el código anterior corresponde a las variables pcu y li respectivamente. A partir de la línea 14, intentamos ahora la producción $S \longrightarrow DC$; si esta producción también falla, entonces no tenemos más alternativas y debemos regresar falso. Observemos ahora la función A donde tenemos el mismo comienzo que en el caso anterior: en la línea 25 comenzamos a probar la producción $A \longrightarrow aA$. Como el primer símbolo en el cuerpo de la producción (a) es un símbolo terminal, se debe verificar que sea el mismo al carácter actual de la cadena de entrada; de ser así se realiza un movimiento de empate y el apuntador avanza un carácter en la cadena de entrada. Por tanto la estrategia general es la siguiente:

[5] Que representa un valor booleano, ya que C no contaba originalmente con tipos booleanos.

1. Hacer una función por cada símbolo no terminal.

2. En el cuerpo de la función se deben probar todas las producciones correspondientes a dicho símbolo no terminal.

3. Si se va a probar una producción de la forma $A \longrightarrow X_1 \dots X_i \dots X_n$, entonces para cada símbolo X_i en el cuerpo de la producción, se debe realizar lo siguiente:

   a) Si X_i es un símbolo no terminal se debe llamar la función de nombre X_i, que corresponde a un paso de predicción.

   b) Si X_i es un símbolo terminal entonces se debe verificar que empate con el símbolo actual de la cadena de entrada y de ser así el apuntador de la cadena de entrada se avanza al siguiente símbolo; lo anterior corresponde a un movimiento de empate.

   c) Terminamos cuando la función S correspondiente al símbolo inicial regresa verdadero y hemos consumido toda la cadena de entrada (para verificar que se haya consumido toda la cadena de entrada, añadimos un símbolo de fin de cadena a la cadena de entrada y cuando S regresa verdadero verificamos que el símbolo actual de la cadena de entrada empate con el de fin de cadena). De otra forma tras haber agotado todas las posibles alternativas, paramos anunciando que no hay análisis posible para la cadena de entrada.

Los analizadores de descenso recursivo presentan algunas restricciones en las gramáticas con las que pueden trabajar; es claro en este punto que no pueden trabajar con gramáticas recursivas izquierdas, pero hay una restricción mayor que no se ve a primera vista. Analicemos el funcionamiento del analizador anterior cuando se le presenta la cadena de entrada $w = abcc$; comienza probando la producción $S \longrightarrow AB$, enseguida prueba la producción $A \longrightarrow a$, la cual regresa verdadero y después se prueba la producción $B \rightarrow bc$ que también regresa verdadero, en este punto la función S regresa verdadero, y entonces se procede a verificar que se haya agotado la cadena de entrada, lo cual no se cumple pues aún resta una c por consumir. En este punto el analizador se detiene diciendo que no existe análisis para w, aunque w se puede derivar mediante $S \Longrightarrow DC \Longrightarrow abC \Longrightarrow abcC \Longrightarrow bccc$. El problema surge porque el analizador nunca prueba la producción $S \longrightarrow DC$, debido a que la producción $S \longrightarrow AB$ tiene éxito y esto último sucede debido a que la producción $S \longrightarrow AB$ genera cadenas que son prefijos de las cadenas generadas por la producción $S \longrightarrow DC$. Esto nos indica que debemos poner atención al problema de los prefijos propios de una cadena.

Definición 3.2. Sea $G = (N, T, P, S)$ una gramática libre del contexto y sea $A \in N$, se dice que A es *libre de prefijo* si $A \overset{*}{\Longrightarrow} w_1$ y $A \overset{*}{\Longrightarrow} w_2$ con $w_1 \in T^*$ y $w_2 \in T^*$ implica que $w_2 = \varepsilon$.

Definición 3.3. Se dice que una gramática libre del contexto es *libre de prefijo* si todos sus símbolos no terminales son libres de prefijo.

Definición 3.4. Se dice que un lenguaje L es *libre de prefijo* si y sólo si para todo par de cadenas (α, β) con $\alpha \in L$ y $\alpha\beta \in L$ implica que $\beta = \varepsilon$, es decir, se dice que un lenguaje L tiene la propiedad del prefijo si siempre que $w \in T^*$ está en L, no hay prefijo propio de w que esté en L.

Por tanto, podemos notar que los analizadores de descenso recursivo como el de nuestro ejemplo anterior, no trabajan de manera correcta con gramáticas que no son libres de prefijo. Sin embargo, hay estrategias de implementación (que dependen de las características ofrecidas por el lenguaje de implementación del analizador) que permiten modificar los analizadores de descenso recursivo para que sean capaces de reconocer gramáticas que no sean libres de prefijo.

3.4.2. Analizador descendente predictivo

Es claro que los analizadores presentados hasta el momento, incluyendo el analizador de descenso recursivo de la sección anterior, pueden llegar a tomar tiempo exponencial, y esto es debido a que deben probar cada una de las alternativas en los pasos de predicción. Lo ideal sería que sólo existiera una única alternativa por tomar en los pasos de predicción, y así obtendríamos un analizador determinista que tome tiempo lineal. Como primer paso hacia nuestro objetivo podemos poner la restricción de que el primer símbolo del lado derecho de las producciones sea un símbolo terminal, ya que de esta forma nuestra decisión en un paso de predicción quedaría determinada por el símbolo no terminal por destapar y el símbolo terminal actual por consumir de la cadena de entrada; así cuando nos encontremos en un paso de predicción en donde debemos destapar un símbolo no terminal A y el símbolo por consumir en la cadena de entrada sea a, debemos buscar una producción $A \longrightarrow a\alpha$, es decir, una producción que tenga como lado izquierdo A y su lado derecho comience con el símbolo terminal a; podemos incluso hacer este proceso más eficiente calculando con anterioridad una tabla que tenga como columnas los símbolos no terminales de la gramática y como renglones los símbolos terminales que aparecen al comienzo del lado derecho de alguna producción; con esto, los elementos de la entrada (A, a) de la tabla deben ser todas aquellas producciones de la forma $\underline{A} \longrightarrow \underline{a}\alpha$; así, en un paso de predicción basta con que el analizador consulte la entrada (A, a) para saber qué producciones elegir. Llamaremos a la tabla que acabamos de describir *tabla de análisis*. Es fácil notar que las entradas de esta tabla pueden contener más de una producción y por tanto el analizador sería no determinista. Así que nuestro objetivo es que la tabla contenga a lo más una producción por entrada,[6] y para ello debemos establecer la restricción de que el símbolo terminal con el que comienza el cuerpo

[6]Puede haber entradas vacías que indican un error en el análisis.

de una producción sea distinto en cada una de las producciones que tengan el mismo lado izquierdo. De esta forma, las entradas de nuestra tabla contendrán a lo más una producción y nuestro analizador será determinista; a las gramáticas que cumplen esta restricción se les conoce como gramáticas simples o gramáticas-s.

Definición 3.5. Una gramática libre del contexto $G = (N, T, P, S)$ se dice que es una *gramática simple* o *gramática-s* si todas sus producciones son de la forma $A \longrightarrow a\alpha$, donde $A \in N$, $a \in T$, $\alpha \in (N \cup T)^*$ y para cualquier par (A, a) existe a lo más una producción de la forma $A \longrightarrow a\alpha$.

Por ejemplo, la gramática

$$\begin{array}{lcl} S & \longrightarrow & aA \\ A & \longrightarrow & aA \\ A & \longrightarrow & bB \\ B & \longrightarrow & bB \\ B & \longrightarrow & c \end{array}$$

es una gramática simple y su correspondiente tabla de predicción es:

	a	b	c
S	$S \longrightarrow aA$		
A	$A_1 \longrightarrow aA$	$A_2 \longrightarrow bB$	
B		$B_1 \longrightarrow bB$	
B			$B_2 \longrightarrow c$

El analizador se comporta de la siguiente manera cuando se le presenta la cadena de entrada abc:

Comienza en el símbolo inicial S en la pila de predicción y se encuentra mirando al principio de la cadena de entrada. Como el primer símbolo en la entrada es a, consulta la entrada (S, a) en la tabla; dicha entrada tiene como elemento la producción $S \longrightarrow aA$. Una vez realizado esto, se lleva a cabo un paso de empate; después un paso de predicción en el que se consulta la entrada (A, b) y se obtiene el elemento $A_2 \longrightarrow bB$; a continuación se realiza un paso de empate; después un paso de predicción consultando la entrada (B, c) que regresa la producción $B_2 \longrightarrow c$; y por último se realizan dos pasos de empate con lo que finaliza el análisis. Lo anterior se muestra gráficamente en la siguiente figura:

1)

	$abc\#$
	$S\#$

2)

	$abc\#$
S	$aA\#$

3)

a	$bc\#$
Sa	$A\#$

4)

a	$bc\#$
SaA_2	$bB\#$

5)

ab	$c\#$
SaA_2b	$B\#$

6)

ab	$c\#$
SaA_2bB_2	$c\#$

7)

abc	$\#$
SaA_2bB_2c	$\#$

8)

$abc\#$	
$SaA_2bB_2c\#$	

Figura 3.11

Al analizador de nuestro de ejemplo se le denomina analizador predictivo, pues en cada paso de predicción predice la producción correcta de forma determinista. Nótese que aún podemos mejorar su eficiencia si juntamos un paso de predicción con el siguiente paso de empate que estamos seguros que tendrá éxito. Es fácil observar que este tipo de analizadores toman tiempo lineal. Sin embargo, exigir que la gramática con la que trabajen sea simple resulta ser una restricción fuerte, ya que generalmente los lenguajes que generan este tipo de gramáticas son regulares, que como ya mencionamos, no incluyen los enunciados que son comunes encontrar en los lenguaje de programación. En la siguiente sección estudiaremos cómo establecer restricciones menos fuertes y así trabajar con gramáticas más generales, utilizando aún el mismo tipo de analizador, es decir, predictivo determinista. Nuestro problema consiste en encontrar una manera de calcular la tabla de predicción (cuidando que ésta cumpla que a lo más haya una producción en cada una de sus entradas), sobre gramáticas menos restringidas que las gramáticas simples.

3.4.3. Analizador LL(1)

Vimos en la sección anterior que la restricción de usar gramáticas simples nos servía para elegir la producción correcta en un paso de predicción, pues dado que en el cuerpo de las producciones que tienen el mismo lado izquierdo comienzan siempre con un símbolo terminal distinto del lado derecho, la elección en paso de predicción estaba determinada por el símbolo actual en la cadena de entrada y el primer símbolo del cuerpo de las producciones que tenían como lado izquierdo el símbolo no terminal por sustituir en la pila de predicción.

Buscaremos ahora una forma más general de tomar nuestra decisión. Lo que tenemos en este momento es que siempre que tenemos un paso de predicción sobre el símbolo no terminal A, en la pila de predicción tenemos la cadena $A\alpha$ y como carácter de entrada actual a; la forma de las gramáticas simples nos permite conocer qué producción nos lleva a $A \Longrightarrow a\beta$. Con lo anterior en mente podemos generalizar esta idea y buscar un método que nos permita conocer cuándo $A \overset{*}{\Longrightarrow} a\beta$ y así no tendríamos que pedir que necesariamente el cuerpo de las producciones comiencen con un símbolo terminal, es decir, lo que estamos buscando es un camino que nos asegure que en algún momento obtendremos a en el tope de la pila de predicción para poder realizar un paso de empate con el carácter que estamos mirando en la entrada. Esta información la podemos obtener definiendo los siguientes conjuntos.

Definición 3.6. Sea $G = (N, T, P, S)$ una gramática libre del contexto. Definimos

$$FIRST(\alpha) = \{a \mid \alpha \overset{*}{\Longrightarrow} a\beta\}$$

donde $\alpha \in (N \cup T)^*$, $\beta \in (N \cup T)^*$ y $a \in T$. Además, si $\alpha \overset{*}{\Longrightarrow} \varepsilon$ entonces se agrega ε a $FIRST(\alpha)$.

Para calcular $FIRST(X)$ $\forall X \in (N \cup T)$, debemos realizar los siguientes pasos hasta que no haya más símbolos terminales o ε que puedan agregarse a cualquier conjunto $FIRST$.

1. Si X es un símbolo terminal, entonces $FIRST(X) = \{X\}$.

2. Si X es un símbolo no terminal y $X \longrightarrow Y_1 \dots Y_i \dots Y_k$ es una producción para algún $k \geq 1$, entonces agregar a en $FIRST(X)$ si a está en $FIRST(Y_1)$ o para algún i, a está en $FIRST(Y_i)$, y ε está en todos los conjuntos $FIRST(Y_1), \dots FIRST(Y_{i-1})$; esto es, $Y_1 \dots Y_{i-1} \overset{*}{\Longrightarrow} \varepsilon$. Si ε está en $FIRST(Y_j)$ para toda $j = 1, 2, \dots, k$, entonces agregar ε a $FIRST(X)$. Nótese que todo los elementos de $FIRST(Y_1)$ están en $FIRST(X)$.

3. Si $X \longrightarrow \varepsilon$ es una producción entonces agregar ε a $FIRST(X)$.

Ahora podemos calcular $FIRST$ para cualquier cadena $\alpha = X_1 \dots X_i \dots X_n$ con $X_i \in (N \cup T)$ de la siguiente manera:

1. Agregar a $FIRST(\alpha)$ todos los símbolos diferentes de ε que estén en $FIRST(X_1)$, es decir, $\forall a \in FIRST(X_1)$ con $a \neq \varepsilon$ agregar a en $FIRST(\alpha)$.

2. Si ε está en $FIRST(X_1)$; agregar todos los símbolos diferentes de ε que estén en $FIRST(X_2)$, si ε está en $FIRST(X_1)$ y en $FIRST(X_2)$ agregar todos los símbolos diferentes de ε que estén en $FIRST(X_3)$ y así sucesivamente, es decir, si

$\varepsilon \in \bigcap_{i=1}^{j-1} FIRST(X_i)$ con $1 < j \leq n$ entonces $\forall a \in FIRST(X_j)$ con $a \neq \varepsilon$ agregar a en $FIRST(\alpha)$.

3. Finalmente, si $\forall i$, ε está en $FIRST(X_i)$ agregar ε a $FIRST(\alpha)$.

Los conjuntos $FIRST$ nos ayudan a tomar la elección correcta en un paso de predicción, pues nos dicen cuál es el primer símbolo terminal de la primer forma sentencial γ que comienza con símbolo terminal y que se deriva a partir de α , es decir, $\alpha \overset{*}{\Longrightarrow} \gamma = a\beta$; o en otro caso, nos dicen que la cadena α deriva ε, es decir, $\alpha \overset{*}{\Longrightarrow} \gamma = \varepsilon$. Cuando en un paso de predicción tenemos la cadena $A\gamma$ en la pila de predicción y el símbolo actual de al entrada es a, es decir, tenemos la situación que se muestra en la figura 3.12.

...	$a \ldots \#$
...	$A\gamma\#$

Figura 3.12

necesitamos hacer una predicción sobre el símbolo A como ya lo habíamos notado anteriormente. Supongamos que A tiene las producciones $A \longrightarrow \alpha$ y $A \longrightarrow \beta$, entonces lo que nos importa conocer es si $\alpha\gamma\# \overset{*}{\Longrightarrow} \eta = a\delta$ (el que hayamos considerado $\alpha\gamma\#$ es debido a que se puede dar el caso que $\alpha \overset{*}{\Longrightarrow} \eta = \varepsilon$ por lo que nos basta considerar únicamente α), o si $\beta\gamma\# \overset{*}{\Longrightarrow} \eta = a\delta$. Es aquí donde hacemos uso de los conjuntos $FIRST$ que nos dicen exactamente si a es el primer símbolo terminal de η (o si $\eta = \varepsilon$), con lo que en el primer caso tendríamos que calcular $FIRST(\alpha\gamma\#)$ mientras que en el segundo $FIRST(\beta\gamma\#)$. En caso de que a esté en únicamente $FIRST(\alpha\gamma\#)$ se elige la primer producción; en cambio, si a está únicamente en $FIRST(\beta\gamma\#)$ se elige la segunda; nótese que puede darse el caso en que a esté en ambos conjuntos con lo que se pierde el determinismo; bajo esta perspectiva podemos dar la definición de una gramática $LL(1)$:

Definición 3.7. Sea $G = (N, T, P, S)$ una gramática libre del contexto. Decimos que G es $LL(1)$ si siempre que hay dos derivaciones por la izquierda

1. $S \overset{*}{\Longrightarrow} wA\alpha \Longrightarrow w\beta\alpha \overset{*}{\Longrightarrow} wx$ y

2. $S \overset{*}{\Longrightarrow} wA\alpha \Longrightarrow w\gamma\alpha \overset{*}{\Longrightarrow} wy$

tal que $FIRST(x) = FIRST(y)$, se sigue que $\beta = \gamma$.

Su conexión con nuestro primer enfoque se muestra con el siguiente teorema, cuya demostración se puede ver en [**pp.342**; AU72].

Teorema 3.1. Sea $G = (N, T, P, S)$ una gramática libre del contexto. Entonces G es $LL(1)$ si y sólo si la siguiente condición se cumple: Si $A \longrightarrow \beta$ y $A \longrightarrow \gamma$ son producciones distintas en P, entonces $FIRST(\beta\alpha) \cap FIRST(\gamma\alpha) = \varnothing$ para toda $wA\alpha$ tal que $S \overset{*}{\Longrightarrow} wA\alpha$.

Con lo anterior nos queda claro que siempre que tengamos que hacer un paso de predicción tenemos que calcular los conjuntos $FIRST$ correspondientes que nos guiarán a elegir la producción correcta siempre que nuestra gramática sea $LL(1)$; pero esto aún presenta un inconveniente, pues siempre que nos encontremos ante un paso de predicción durante el análisis tenemos que calcular los conjuntos $FIRST$ correspondientes dinámicamente, es decir, en tiempo de ejecución, lo que provoca que se degrade el desempeño del analizador. Lo que nosotros quisiéramos es tener precalculada la información necesaria durante la construcción del analizador (en tiempo de compilación) y almacenarla en un tabla de análisis, como la que hemos utilizado con las gramáticas simples. El problema que tenemos es que no tenemos información suficiente en tiempo de compilación de qué cadenas se encontrarán en la pila de análisis en tiempo de ejecución, particularmente en el siguiente caso: supongamos que durante un análisis tenemos la cadena $A\gamma\#$ en la pila de predicción y tenemos la producción $A \longrightarrow \alpha$ y $\alpha \overset{*}{\Longrightarrow} \varepsilon$, es decir, α es o deriva ε; entonces debemos calcular $FIRST(\alpha\gamma\#)$ que, como sabemos, contendrá los elementos en $FIRST(\alpha)$, pero además como α es o deriva ε, también contendrá los elementos en $FIRST(\gamma\#)$. Es precisamente ésta la información que no conocemos en tiempo de compilación, ya que no sabemos qué cadena $\gamma\#$ se presentará en el análisis; esto se origina debido a que α es o deriva ε. Sin embargo, lo que sí podemos hacer en tiempo de compilación es calcular la unión de todos los conjuntos $FIRST(\gamma\#)$ tal que $\gamma\#$ pueda seguir (*follow*) después de A en cualquier forma sentencial que se derive a partir de $S\#$; al conjunto que resulta de este cálculo se le denomina $FOLLOW$ de A.

Definición 3.8. Sea $G = (N, T, P, S)$ una gramática libre del contexto. Definimos

$$FOLLOW(\beta) = \{w \mid S \overset{*}{\Longrightarrow} \alpha\beta\gamma \text{ con } w \in FIRST(\gamma)\}$$

donde $\beta \in (N \cup T)^*$.

Para calcular $FOLLOW(A)$ $\forall A \in N$, debemos realizar los siguientes pasos hasta que ningún elemento más pueda ser agregado a cualquier conjunto $FOLLOW$.

1. Agregar $\#$ a $FOLLOW(S)$, donde S es el símbolo inicial y $\#$ es el símbolo de fin de cadena.
2. Si existe una producción $A \longrightarrow \alpha B\beta$, entonces todo elemento en $FIRST(\beta)$, excepto ε, está en $FOLLOW(B)$.
3. Si existe una producción $A \longrightarrow \alpha B$, o una producción $A \longrightarrow \alpha B\beta$, donde $FIRST(\beta)$ contiene ε, entonces todo elemento en $FOLLOW(A)$ está en $FOLLOW(B)$.

Con los conjuntos $FIRST$ y $FOLLOW$ descritos anteriormente estamos listos para poder construir una tabla de análisis.

Algoritmo 3.1. Construcción de una tabla de análisis predictivo.
Entrada: Una gramática $G = (N, T, P, S)$.
Salida: Una tabla de análisis M para G.
Método:

1. $\forall A \longrightarrow \alpha \in P$:

   1.1 $\forall a \in T$ tal que $a \in FIRST(A)$, agregar $A \longrightarrow \alpha$ en $M[A, a]$.

   1.2 Si $\varepsilon \in FIRST(\alpha)$, entonces $\forall b \in T$ tal que $b \in FOLLOW(A)$,
   agregar $A \longrightarrow \alpha$ en $M[A, b]$.
   Si $\varepsilon \in FIRST(\alpha)$ y $\# \in FOLLOW(A)$,
   agregar $A \longrightarrow \alpha$ en $M[A, \#]$.

Notemos que puede ser que una entrada de la tabla contenga más de una producción; en cuyo caso se pierde el determinismo. Si todas las entradas de la tabla contienen a lo más una producción entonces se dice que la gramática es $LL(1)$ fuerte (*strong* $LL(1)$).Formalmente:

Definición 3.9. Sea $G = (N, T, P, S)$ una gramática libre del contexto tal que se cumple la siguiente condición:
$\forall A \in N$ si $A \longrightarrow \beta$ y $A \longrightarrow \gamma$ son A-producciones distintas, entonces

$$FIRST(\beta FOLLOW(A)) \cap FIRST(\gamma FOLLOW(A)) = \varnothing$$

entonces se dice que G es $LL(1)$ *fuerte*.

De esta forma podemos seguir utilizando el analizador que usamos en la sección anterior, pero trabajando con gramáticas $LL(1)$ fuertes, cuidando que si durante el análisis se consulta una entrada vacía de la tabla, indicar un error.

Con lo anterior contamos por una parte con un analizador que trabaja con gramáticas $LL(1)$, que no hace uso de conjuntos $FOLLOW$ y que calcula los conjuntos $FIRST$ en tiempo de ejecución; y por el otro con un analizador que trabaja con gramáticas $LL(1)$ fuertes, que hace uso de conjuntos $FIRST$ y $FOLLOW$ y que los calcula en tiempo de compilación (desde luego este último es el que deseábamos encontrar); por lo que es natural preguntarnos si toda gramática $LL(1)$ es $LL(1)$ fuerte, lo que resulta ser cierto; pero, más aún, toda gramática $LL(1)$ fuerte es $LL(1)$ como se afirma en el siguiente teorema –ver [**pp.343**; AU72]–.

Teorema 3.2. Una gramática libre del contexto $G = (N, T, P, S)$ es $LL(1)$ si y sólo si la siguiente condición se cumple: $\forall A \in N$, si $A \longrightarrow \beta$ y $A \longrightarrow \gamma$ son producciones distintas, entonces $FIRST(\beta FOLLOW(A)) \cap FIRST(\gamma FOLLOW(A)) = \varnothing$. En otras palabras una gramática G es $LL(1)$ si y sólo si es $LL(1)$ fuerte.

La única (pequeña) desventaja que tienen los analizadores que trabajan con gramáticas $LL(1)$ fuertes en comparación con los que trabajan con gramáticas $LL(1)$ es que puede ser que hagan predicciones ε antes de detectar un error.

A un analizador que utiliza la tabla correspondiente a una gramática $LL(1)$ fuerte se le denomina analizador $LL(1)$ fuerte[7] debido a que lee la cadena de entrada de izquierda a derecha y encuentra la derivación por la izquierda utilizado un átomo de adelanto (*lookahead*) (*Left to right Leftmost derivation*).

Mencionamos anteriormente que en la tabla de análisis puede darse el caso en que una entrada contenga más de una producción a lo que se le denomina un *conflicto LL(1)*; en la próxima sección estudiaremos una transformación que nos ayuda a resolver algunos de estos conflictos, conocida como factorización izquierda.

Factorización izquierda

Cuando en una gramática tenemos producciones de la forma $A \longrightarrow \alpha\beta_1|\alpha\beta_2$, es claro que podemos tener un conflicto $LL(1)$ debido a que ambas producciones comienzan con α. Una transformación que permite evitar los posibles conflictos que surgen de la situación anterior es la que se conoce como *factorización izquierda* y se realiza aplicando el siguiente algoritmo:

Algoritmo 3.2. Factorización izquierda de una gramática.
Entrada: Una gramática $G = (N, T, P, S)$ libre del contexto.
Salida: G', una gramática libre del contexto, con factorización izquierda tal que $L(G) = L(G')$.
Método:
Sean $A \longrightarrow \alpha\beta_1 \,|\, \alpha\beta_2 \,|\, \ldots, \alpha\beta_k \,|\, \gamma_1 \,|\, \ldots |\gamma_r$, $k \geq 2$, $\gamma_i, 0 \leq i \leq r$ los consecuentes que no empiezan con α, α el prefijo común más largo en el cuerpo de dos o más de las A–producciones. Si $\alpha \neq \varepsilon$, reemplazar todas las A–producciones por

$$\begin{aligned} A &\longrightarrow \alpha A'|\gamma \\ A' &\longrightarrow \beta_1|\beta_2|\ldots|\beta_n \end{aligned}$$

donde A' es un nuevo símbolo no terminal. Repetir esta transformación hasta que no haya dos producciones que tengan un prefijo común en sus cuerpos y tengan un mismo símbolo no terminal en su lado izquierdo.

Veamos un ejemplo de cómo funciona la factorización izquierda. La siguiente gramática genera la estructura **if-then-else** de un lenguaje de programación:

[7] En realidad en la literatura se le denomina simplemente analizador $LL(1)$, nosotros utilizamos el adjetivo "fuerte" para distinguirlo del analizador $LL(1)$ que calcula los conjuntos $FIRST$ en tiempo de ejecución y al que en algunos textos se le denomina analizador $LL(1)$ lleno (*full*).

$$\begin{aligned} S &\longrightarrow \textbf{if } E \textbf{ then } S \mid \textbf{if } E \textbf{ then } S \textbf{ else } S \mid a \\ E &\longrightarrow b \end{aligned}$$

donde **if**, **else**, **then** son átomos que entrega el analizador léxico, E genera enunciados booleanos y S genera enunciados (de todo tipo). Aplicando factorización izquierda obtenemos:

$$\begin{aligned} S &\longrightarrow \textbf{if } E \textbf{ then } SS' \mid a \\ S' &\longrightarrow \textbf{else } S \mid \varepsilon \\ E &\longrightarrow b \end{aligned}$$

Implementación de un analizador LL(1)

Aun cuando es claro que podemos implementar un analizador $LL(1)$ que imite el comportamiento del analizador basado en tabla que hemos visto, presenta los mismos inconvenientes que vimos con el analizador en profundidad que utilizaba retroceso, es decir, tendríamos que modificarlo para que pudiera realizar acciones semánticas durante el análisis y es, en principio, menos eficiente que una implementación hecha a la medida para una gramática específica; es por esto que generalmente la implementación de un analizador $LL(1)$ se realiza mediante un caso especial de descenso recursivo. Esta vez no necesitamos probar todas las posibles producciones porque ya sabemos cuál producción elegir en cada paso de predicción. Además, cuando tenemos conflictos $LL(1)$ por resolver (y que no se pudieron resolver mediante factorización izquierda), se pueden seleccionar a mano cuál producción es la correcta con base en el conocimiento del escritor del analizador.

3.4.4. Analizadores sintácticos ascendentes (*bottom–up*)

Los analizadores ascendentes comienzan con el enunciado de entrada w y a partir de éste reconstruyen (en caso de que exista) la derivación por la derecha que parte del símbolo inicial y nos lleva a w; es decir, si $S \Longrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \alpha_{n-1} \Longrightarrow \alpha_n = w$, la estrategia que se sigue es: partiendo del enunciado w, encuentra el cuerpo de la producción que participa en el respectivo paso de la derivación por la derecha y hace una sustitución del cuerpo por el símbolo no terminal del lado izquierdo de la producción, –a lo que se le denomina una *reducción*–, con lo que se está realizando lo contrario a un paso de la derivación. Al final del análisis (en caso de que exista) se obtendrá una secuencia de producciones utilizadas en las reducciones; la reversa de esta secuencia será el análisis de w correspondiente a la derivación por la derecha de w; en otras palabras, cuando se comienza con el enunciado w se encuentra el último paso de la derivación por la derecha; a continuación se encuentra el penúltimo paso y así sucesivamente, hasta llegar al

último paso del análisis que corresponde al primer paso de la derivación por la derecha. Por otro lado, este proceso puede ser visto como comenzar en las hojas del árbol de derivación, que corresponden al enunciado w, (suponiendo que existe), e ir encontrando los símbolos que pertenezcan a un asa (*handle*); estos símbolos forman parte de una forma sentencial de la derivación. Una vez que se encuentran estos símbolos, (que deben ser el cuerpo de una producción P), se construye el subárbol (el asa) poniendo aristas que van de los símbolos que forman el asa hacia el símbolo no terminal que es su padre, (el símbolo que está en el lado izquierdo de la producción P), con lo que hemos encontrado un subárbol del árbol de derivación; suponiendo que tenemos la forma sentencial $\gamma = \alpha\beta x = w$ y que se tiene la producción $A \longrightarrow \beta$, encontramos que los símbolos en β constituyen un asa que parte del símbolo no terminal A, por lo que construimos un subárbol con raíz A e hijos $X_1 \dots X_n$, donde $X_1 \dots X_n = \beta$. Una vez realizado esto obtenemos la forma sentencial $\delta = \alpha A\beta$, donde debemos repetir el proceso de encontrar un asa tomando $\gamma = \delta$ y construir el subárbol correspondiente. Se debe continuar este proceso hasta que se llegue a la raíz del árbol de derivación de w, es decir, cuando $\gamma = S$ y se haya construido por completo el árbol de derivación de w.

Formalmente, si $S \overset{*}{\Longrightarrow} \alpha Ax \Longrightarrow \alpha\beta x$, la producción $A \longrightarrow \beta$ en la posición siguiente de α es un *asa* de $\alpha\beta x$. Alternativamente, un asa de una forma sentencial derecha γ es una producción $A \longrightarrow \beta$ y una posición de γ donde la cadena β puede ser encontrada, tal que reemplazando β en esa posición por A produce la forma sentencial derecha previa en una derivación por la derecha de γ. Nótese que $x \in T*$. Nosotros utilizaremos informalmente el término asa para referirnos únicamente al cuerpo y la posición, no a toda la producción.

Nuestro analizador ascendente (de abajo hacia arriba) hará uso de un autómata con una pila de análisis y lo representaremos gráficamente de la siguiente forma:

$X_1 \dots X_n$	x

donde del lado izquierdo se muestra la pila de análisis y del lado derecho se muestra el sufijo de la cadena que falta procesar. El contenido de la pila de análisis en un momento determinado son n símbolos con $X_i \in N \cup T$. Cuando se trabaja con una gramática sin recursividad derecha n está acotado, mientras que en caso contrario no lo está.

El analizador puede realizar las siguientes operaciones:

1. Shift. Desliza el siguiente símbolo de entrada al tope de la pila de análisis.
2. Reduce. Realiza una reducción. Cabe mencionar que el fin del cadena a reducir estará en el tope de la pila, mientras que el comienzo no necesariamente estará en el fondo de la pila.
3. Accept. Termina anunciando que encontró un análisis de la cadena de entrada.
4. Error. Indica un error de sintaxis.

Al comienzo del análisis no tenemos otra opción que realizar un shift, después, en cada paso del análisis, en principio sólo tenemos dos opciones: realizar un shift o un reduce (ignoremos por el momento las otras dos operaciones). Dependiendo de la gramática con la que trabajemos (suponiendo que estamos trabajando con gramáticas libres del contexto), algunos de estos pasos serán deterministas, es decir, sólo será posible o bien hacer un shift o en otro caso hacer un reduce; sin embargo, habrá pasos en los que será posible hacer ambas, en cuyo caso siguiendo una estrategia general (análogamente a como lo hicimos en los analizadores descendentes) podemos seguir ambos caminos agregando tantas pilas como sean necesarias durante el análisis. Otro caso que puede presentase es que en un paso del análisis se puedan realizar dos reducciones diferentes. De igual forma se pueden seguir ambos caminos, y si al final del análisis en algún camino se consumió por completo la cadena de entrada y el único símbolo en la pila de análisis es el símbolo inicial, entonces hemos encontrado un análisis de la cadena de entrada w. Con lo anterior obtenemos un analizador ascendente shift-reduce en amplitud. Si en lugar de seguir ambos caminos en un paso no determinista únicamente seguimos uno, y utilizamos retroceso para que en caso de fracasar probar el siguiente camino, entonces obtenemos un analizador ascendente shift-reduce en profundidad. Es claro que ambos toman tiempo exponencial y el analizador en amplitud toma espacio exponencial, mientras que el analizador en profundidad toma espacio lineal.

Una vez más nuestro objetivo es encontrar una manera de elegir el paso correcto a realizar, es decir, una estrategia que nos permita, en cada paso del análisis, decidir si hacer un shift o un reduce, y en caso de que sea un reduce que nos indique qué producción utilizar.

Lo anterior lo podemos plantear como un problema de búsqueda, donde cada vez que se realice un shift durante el análisis, debemos verificar si algún sufijo de la cadena que está en la pila de análisis empata con algún lado derecho de una producción que interviene en el paso de derivación respectivo,[8] por lo que es natural pensar en hacer uso de un autómata finito que nos ayude a realizar la búsqueda de los lados derechos de producciones, ya que éstos conforman un conjunto finito de cadenas. Cada vez que realicemos un shift debemos alimentar la cadena que está en la pila de análisis y el autómata finito nos dirá qué reducción utilizar o, en otro caso que debemos realizar un shift.

A continuación revisamos la construcción y forma de operación del autómata finito que acabamos de mencionar.

[8]Esto último se dice para hacer notar que puede darse el caso que se encuentre un sufijo de la cadena que está en la pila de análisis que empate con el cuerpo de una producción y que no forme parte de la derivación que estamos buscando, pero esto sólo puede suceder en los analizadores generales que mencionamos arriba, no en la estrategia actual que estamos desarrollando.

Construcción del autómata finito

Un *elemento* (*item*) de una gramática G es una producción con un punto en alguna posición de su cuerpo; por ejemplo, si $A \longrightarrow XYZ$ es una producción, $A \longrightarrow X \bullet YZ$ es un elemento. Intuitivamente, un elemento nos dice la parte del cuerpo de la producción que hemos encontrando hasta el momento durante el análisis.

Comenzaremos construyendo un AFN que tenga un elemento en cada uno de sus estados; de esta manera un estado del AFN que tenga un elemento $A \longrightarrow X \bullet YZ$ en su interior representa que el AFN tiene la esperanza de encontrar la reducción $A \longrightarrow XYZ$ de la cual ha visto el símbolo X de su cuerpo. Para facilitar nuestra explicación introduciremos un tipo de estados llamados *estados de estación* que tendrán en su interior un símbolo no terminal A cualquiera con un $\bullet$ a su izquierda (por ejemplo $\bullet A$), y cuando el analizador se encuentre un uno de esos estados durante el análisis significa que comienza una búsqueda de un asa que se reduzca a A. Utilizaremos la siguiente gramática:

$$\begin{array}{lcl} S & \longrightarrow & E\$ \\ E & \longrightarrow & E - T \\ E & \longrightarrow & T \\ T & \longrightarrow & n \\ T & \longrightarrow & (E) \end{array}$$

Figura 3.13

donde n es un átomo que representa a los números enteros, para mostrar un ejemplo de cómo se construye el AFN.

El AFN que se muestra en la figura 3.14 es el AFN que se construye a partir de la gramática de la figura 3.13. Como se observa en la figura 3.14, hemos puesto los estados de estación en la parte superior. Suponiendo que tenemos un estado de estación $\bullet A$, debemos agregar transiciones ε del estado $\bullet A$ a los estados que contengan elementos con lado izquierdo A y que tengan el punto a la izquierda en el cuerpo. Por ejemplo, el estado $\bullet E$ tiene transiciones ε a los estados que tienen los elementos $E \longrightarrow \bullet E - T$ y $E \longrightarrow \bullet T$ respectivamente, debido a que se espera encontrar un asa que se reduzca a E; por otra parte cuando tenemos estados que tienen elementos de la forma $A \longrightarrow \alpha \bullet B\beta$, se agregan transiciones ε hacia el estado $\bullet B$, pues se tiene la esperanza de reconocer una B en ese punto del análisis; también del estado $E \longrightarrow E - \bullet T$ hay una transición al estado $\bullet T$; por último, para cada estado que tiene un elemento de la forma $A \longrightarrow \alpha \bullet X\beta$ con $X \in N \cup T$ hay que agregar una transición X hacia el estado $A \longrightarrow \alpha X \bullet \beta$, que significa que, como se esperaba, hemos encontrado X. Por ejemplo tenemos una transición E que va del estado $S \longrightarrow \bullet E\$$ al estado $S \longrightarrow E \bullet \$$.

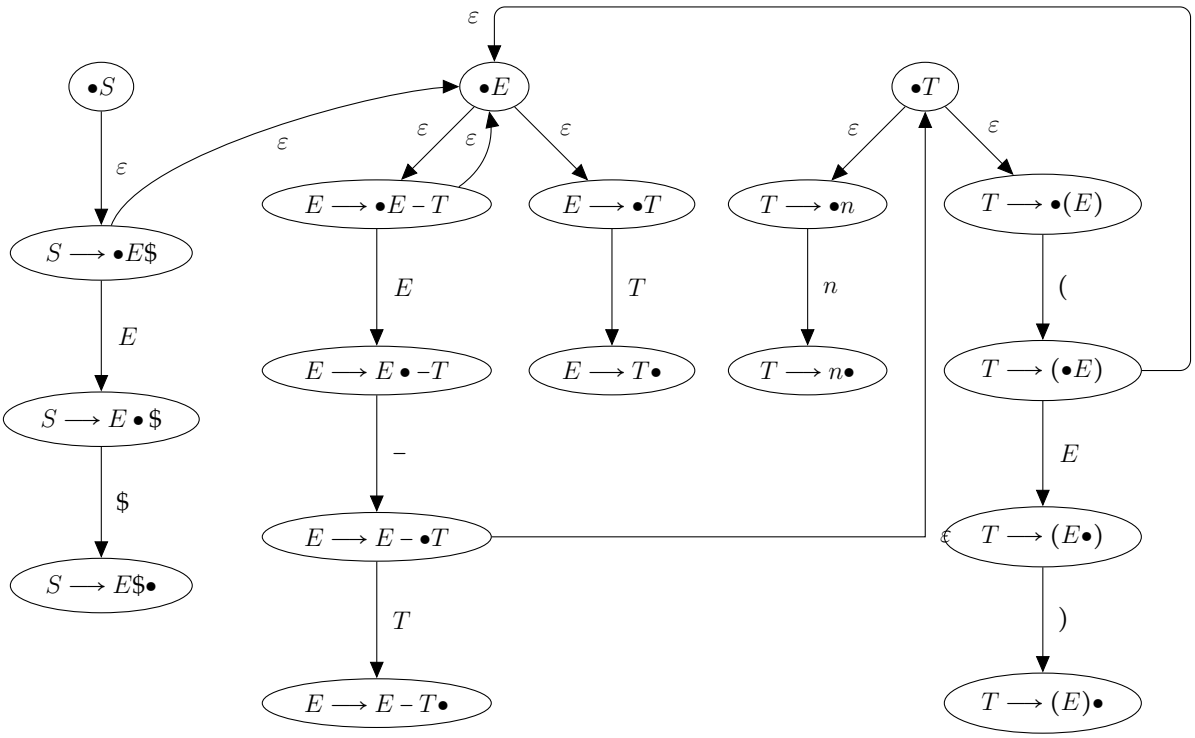


Figura 3.14

Como sabemos, podemos convertir nuestro AFN a un AFD, con lo que se obtiene el AFD que se muestra en la figura 3.20 de la siguiente página, en la que hemos numerado de forma arbitraria los estados del AFD para trabajar con ellos durante el análisis. Los estados que contienen un elemento con el punto al final, es decir, tienen elementos de la forma $A \longrightarrow X_1 \dots X_n\bullet$ indican que siempre que se alcancen durante el análisis se debe realizar una reducción al símbolo A, pues como se esperaba se encontró el cuerpo de la producción. Por otra parte, si un estado i tiene una transición X a un estado j, indica que se debe realizar un shift si durante el análisis estando en el estado i el símbolo actual en la entrada es X.

A continuación mostraremos, mediante un ejemplo, el funcionamiento de nuestro analizador.

Extenderemos la pila de análisis para que sea capaz de almacenar estados del AFD junto con los símbolos de la gramática durante el análisis. Supongamos que tenemos la cadena de entrada $5 - (7 - 3)$\$, para la cual el analizador léxico reportará: $n - (n - n)$\$. Entonces se realizan los pasos que se ven en la figura 3.16, y que procedemos a explicar.

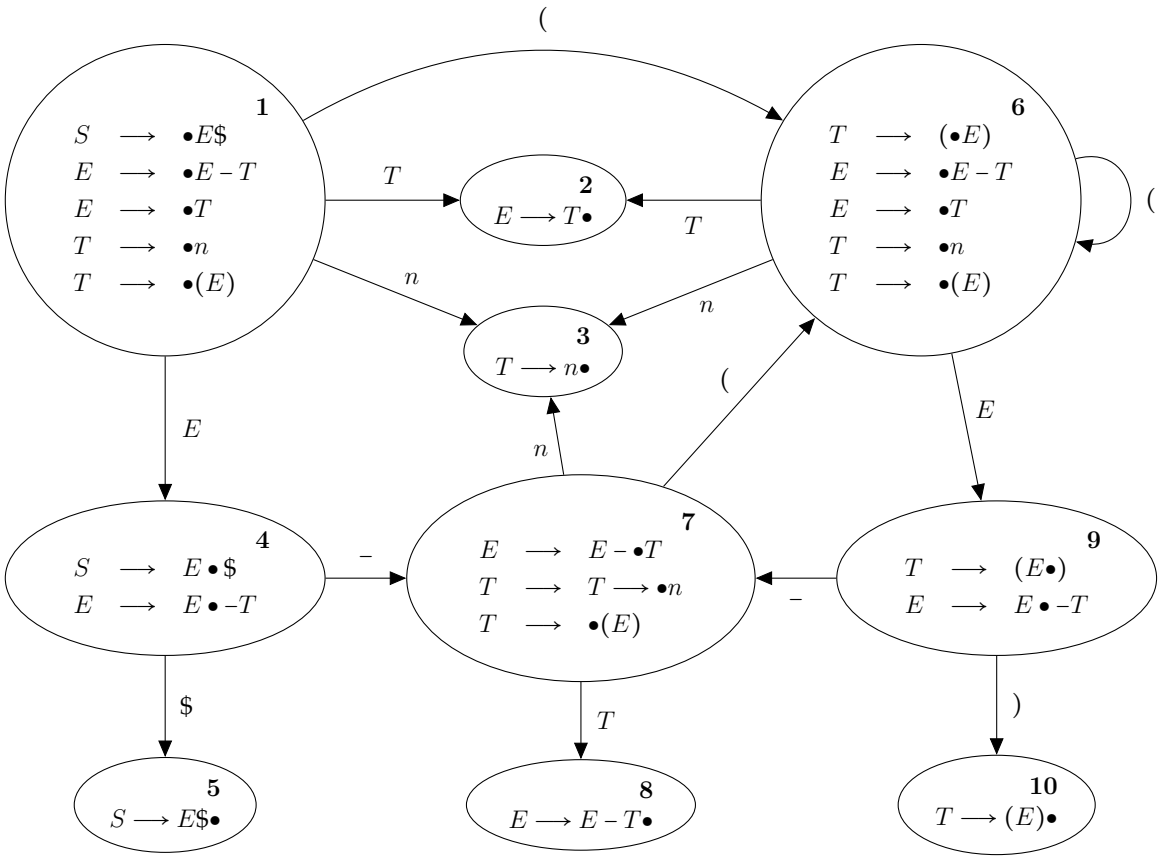


Figura 3.15

Se comienza con el estado 1 en la pila y se debe realizar un shift con el primer símbolo de la entrada, en este caso n, de la siguiente manera: como 1 es el estado que está en el tope de la pila debemos consultar qué estado se alcanza bajo la transición n, que en este caso es el estado 3, por lo que se hace un push n seguido de 3 a la pila; como el estado 3 nos indica que debemos realizar un reduce, entonces sacamos el estado 3 y el símbolo n y ahora tenemos T, por lo que debemos consultar qué estado se alcanza bajo la transición T a partir del estado 1, que era el que estaba nuevamente el tope de la pila. Dicho estado es 2 por lo que metemos T seguido de 2 y hemos realizado el paso correspondiente a un reduce. En general, cuando se alcance un estado que contiene un elemento de la forma $A \longrightarrow X_1 \dots X_n\bullet$, debemos sacar n símbolos de la pila y los n estados entre ellos, y meter el símbolo A seguido del estado al que nos lleve la transición con A a partir del estado que quedó en el tope de pila (antes de meter A). Cuando nos encontremos en un paso en el que hay una transición bajo el símbolo actual de la entrada a partir del estado que está en el tope de la pila, entonces se debe realizar un shift, como es el caso del paso 9,

en el que existe una transición del estado 9 bajo el símbolo actual "$-$", en caso de que no exista una transición del estado actual i, en el tope de la pila, con el símbolo actual de la entrada, entonces se espera que i indique una reducción y por tanto se debe realizar una reducción. Nótese que puede darse el caso en que se realicen dos reducciones seguidas, como en los pasos 2 y 3. En el paso 2 no existe una transición del estado 3 con "$-$", por lo que se espera que 3 indique una reducción, como en efecto lo hace. De igual forma, en el paso 3 no existe una transición del estado 2 bajo el símbolo "$-$", por lo que se espera que el estado 2 indique una reducción, como en efecto lo hace; si al final del análisis está únicamente el estado 1 seguido del símbolo inicial S en la pila, y no hay más símbolos por consumir en la entrada, entonces el análisis ha tenido éxito, como en el caso de nuestro ejemplo; en otro caso el análisis falla. En realidad, en la práctica únicamente se trabaja con estados en la pila y no se utilizan los símbolos; aquí los hemos incluido para facilitar la comprensión del analizador.

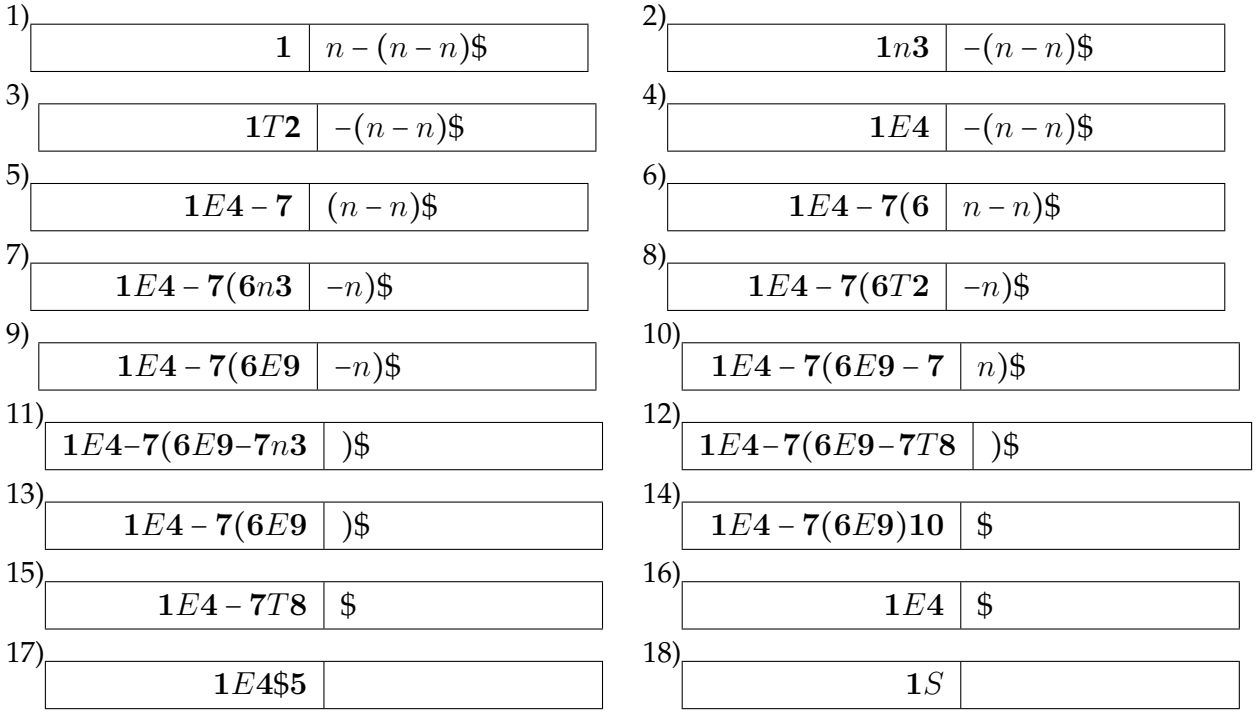


Figura 3.16

Nótese que podemos codificar nuestro autómata en dos tablas, la primera de nombre *ACTION* que nos indique, con base en el estado actual en el tope de la pila, cuando nos encontremos en un paso cualquiera del análisis, si debemos realizar un shift o un reduce; y la otra de nombre *GOTO*, que en caso de que la acción a realizar sea un shift, nos

indique el estado que se alcanza a partir del estado en el tope de la pila bajo el símbolo actual en la entrada.

Entonces, las tablas ACTION y GOTO correspondientes a nuestro autómata se muestran en la figura 3.17.

Estado	Acción
1	shift
2	$E \longrightarrow T$
3	$T \longrightarrow n$
4	shift
5	$S \longrightarrow E\$$
6	shift
7	shift
8	$E \longrightarrow E - T$
9	shift
10	$T \longrightarrow (E)$

(a) Tabla ACTION

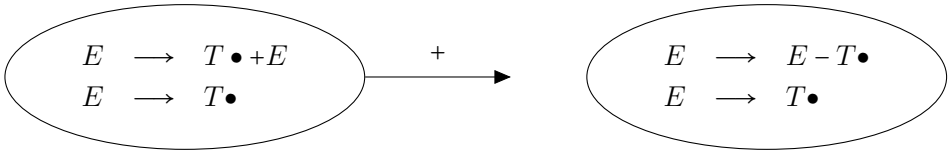
	Ir a						
	n	–	(	)	$	E	T
1	3	e	6	e	e	4	2
2							
3							
4	e	7	e	e	5		
5							
6	3	e	6	e	e	9	2
7	3	e	6	e	e		8
8							
9	e	7	e	10	e		
10							

(b) Tabla GOTO

Figura 3.17: Tablas ACTION y GOTO respectivamente del AFD de la figura 3.20.

Al AFD que construimos se le denomina AFD $LR(0)$ y al analizador que lo utiliza se le llama analizador $LR(0)$, que significa que hace una lectura de izquierda a derecha de la entrada (*Left to right*) y encuentra la derivación por la derecha (*Rightmost derivation*) utilizando 0 átomos de adelanto.

En principio podemos construir un AFD $LR(0)$ para cualquier gramática libre del contexto siguiendo los pasos que describimos anteriormente. El problema se presenta cuando para algunas gramáticas de este tipo su correspondiente AFD $LR(0)$ tiene estados como los que se muestran en la figura 3.18.



(a) Conflicto shift/reduce. (b) Conflicto reduce/reduce.

Figura 3.18: Ejemplo de los posibles conflictos en un AFD $LR(0)$.

Supongamos que durante el análisis nos encontramos en el estado que se muestra en la figura 3.18a y el símbolo actual de la entrada es "+"; entonces se tiene por una parte que debemos hacer un reduce, pues el estado actual en el tope de la pila contiene al elemento $E \longrightarrow T\bullet$; pero por otra parte, como el símbolo actual en la entrada es "+" y el estado actual en el tope de la pila tiene una transición bajo "+", entonces se debe hacer un shift; a esto se le denomina *conflicto shift/reduce*, pues tenemos ambas posibilidades: hacer un shift o un reduce. Por otra parte, si durante algún instante en el análisis se encuentra el estado que se muestra en la figura 3.18b en el tope de la pila, entonces se indica que se debe hacer un reduce utilizando la producción $E \longrightarrow E - T$, pues el estado contiene el elemento $E \longrightarrow E - T\bullet$; por otro lado, también se indica que se realice un reduce pero utilizando la producción $E \longrightarrow T$, pues el estado contiene el elemento $E \longrightarrow T\bullet$; a esto se le denomina *conflicto reduce/reduce*, pues tenemos dos posibles reducciones; a un estado que contiene al menos un conflicto shift/reduce o reduce/reduce se le llama *estado inadecuado*. Es claro que en presencia de cualquiera de los conflictos anteriores el determinismo se pierde. Podemos decir que una gramática G tal que su AFD $LR(0)$ no tiene estados inadecuados es una *gramática* $LR(0)$.

Desafortunadamente, en la práctica, las gramáticas $LR(0)$ generalmente no permiten expresar algunos de los enunciados comunes de los lenguajes de programación sin que se pierda la estructura interna de sus categorías sintácticas, por lo que una vez más estamos un busca de una caracterización que nos permita trabajar con gramáticas menos restringidas que las gramáticas $LR(0)$.

Por ejemplo, cuando en la gramática de la figura 3.13 quitamos el símbolo de fin de cadena \$ en el cuerpo de la producción $S \longrightarrow E\$$, obtenemos la siguiente gramática:

$$\begin{array}{lcl} S & \longrightarrow & E \\ E & \longrightarrow & E - T \\ E & \longrightarrow & T \\ T & \longrightarrow & n \\ T & \longrightarrow & (E) \end{array}$$

Figura 3.19

la cual no es una gramática $LR(0)$. Observemos por qué, viendo su correspondiente AFD $LR(0)$ que se presenta en la figura 3.20.

El problema es que el estado 4 se convirtió en un estado inadecuado, pues presenta un conflicto shift/reduce; así que si nos encontramos en un paso durante el análisis en el que el estado 4 esté en el tope de la pila, por una parte se debe realizar un reduce utilizando la producción $S \longrightarrow E\bullet$ y por otra se debe realizar un shift si el símbolo actual de la entrada es "−". Es claro que podemos resolver este conflicto viendo el siguiente

símbolo en la entrada: si se trata de un "–", entonces el paso correcto es realizar un shift, pues nunca debemos realizar un reduce como lo indica el elemento $S \longrightarrow E\bullet$, ya que de hacerlo estaríamos suponiendo que es el fin del enunciado de entrada; pero es obvio que no es así, porque tenemos como siguiente símbolo en la entrada un "–". Por otra parte, una vez más agregaremos el símbolo $\# \notin T$ al final del enunciado de entrada para facilitar la detección de su fin. De este modo, si el siguiente símbolo de la entrada es $\#$, es claro que se debe hacer un reduce y no un shift.

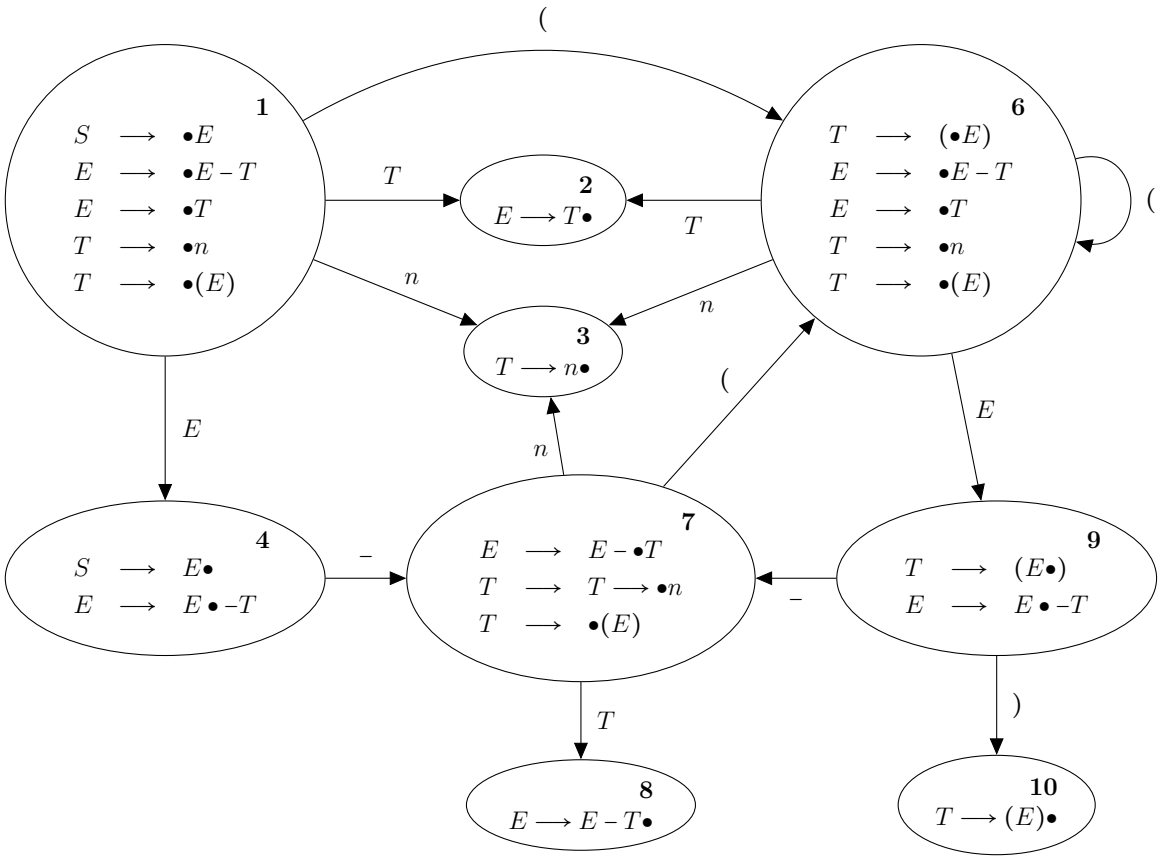


Figura 3.20: AFD LR(0) de la gramática de la figura 3.13 modificada.

Podemos generalizar este principio y observar un átomo de adelanto en cada paso del análisis que nos lleve a la elección correcta y de esta forma resolver los posibles conflictos que surgen en un analizador $LR(0)$.

A cada elemento le agregaremos un átomo, separado por una ",", para que nos queden elementos de la forma $[A \longrightarrow \alpha \bullet \beta, a]$ donde $a \in T$ corresponde al símbolo de adelanto. Para construir el AFD correspondiente a la gramática 3.19, comenzaremos con el

elemento $[S \longrightarrow \bullet E, \#]$ que indica que se tiene la expectativa de encontrar un E seguido de un "#". Esta vez construiremos el AFD directamente sin construir el AFN primero, pero es claro que se puede construir el AFN y a partir de este último el AFD. Tendremos que ir calculando la ε-cerradura al vuelo. Una vez que tenemos el elemento $[S \longrightarrow \bullet E, \#]$ en el estado 1 debemos agregar todos aquellos estados que nos lleven reconocer E seguido de "#", por lo que agregamos $[E \longrightarrow \bullet E - T, \#]$ y $[E \longrightarrow \bullet T, \#]$. Es fácil notar con lo anterior cómo el átomo de adelanto se propaga; por otra parte; $[E \longrightarrow \bullet E - T, \#]$ nos indica que debemos agregar al elemento $[E \longrightarrow \bullet E - T, -]$ pues se espera encontrar un E seguido de un "–", con lo que se genera un nuevo átomo de adelanto; en general se tiene que el átomo de adelanto se genera o propaga como en los ejemplos que acabamos de ver; además debe notarse que pudiera ser que en lugar de que E estuviese seguido del símbolo terminal "–", estuviese seguido de un símbolo no terminal cualquiera N, con lo que tendríamos que calcular $FIRST(N)$ para determinar el átomo de adelanto. Por lo demás, la construcción de este AFD es análoga a la construcción del AFD $LR(0)$. Siguiendo la estrategia anterior obtenemos el siguiente AFD.

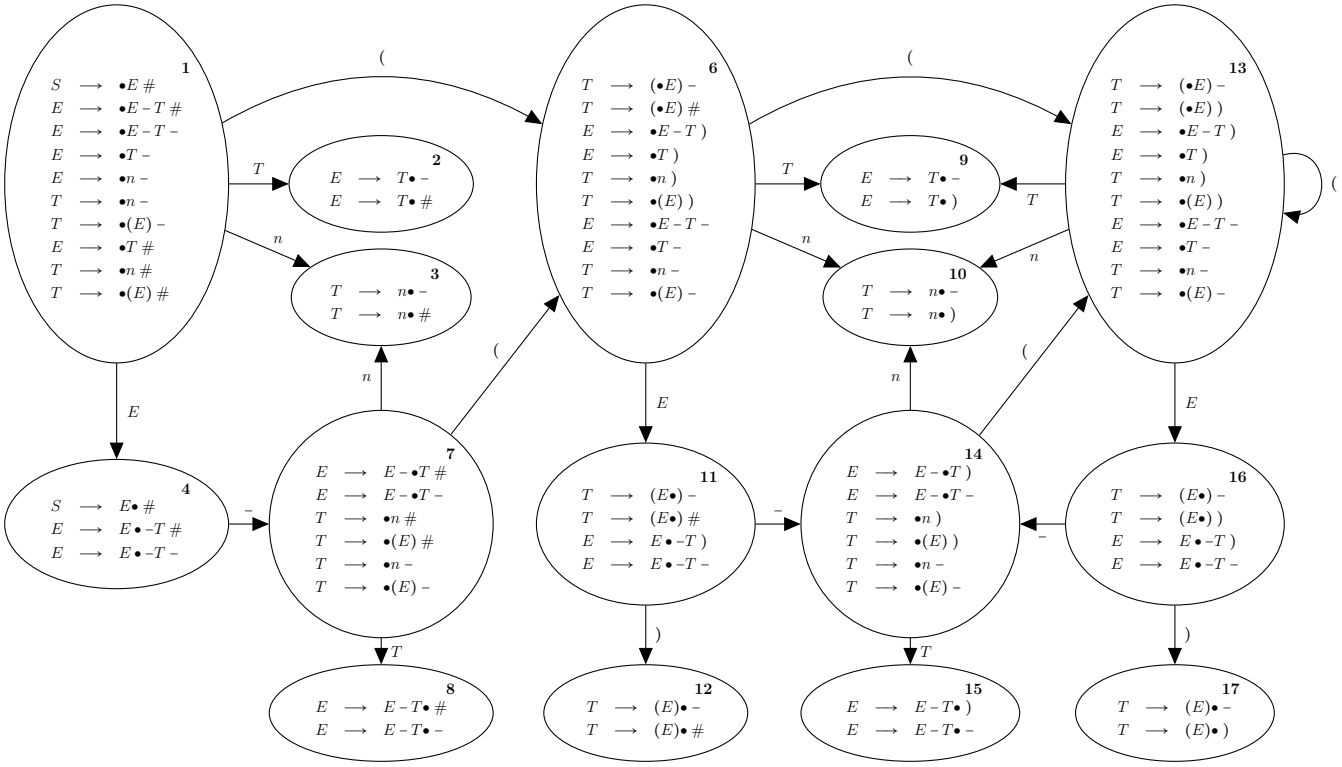


Figura 3.21: AFD $LR(1)$ de la gramática 3.19.

Se puede observar cómo el conflicto en el estado 4 ha sido resuelto: cuando el anali-

zador tenga el estado 4 en el tope de la pila y el átomo de adelanto sea "$\#$", se realiza un reduce con base en elemento $[S \longrightarrow E\bullet, \#]$; en otro caso, si el átomo de adelanto es "$-$", entonces se realiza un shift con base en el elemento $[E \longrightarrow E\bullet - T, -]$; cualquier otro caso se trata de un error.

Al AFD que acabamos de presentar se le denomina AFD $LR(1)$. A un analizador que utiliza un AFD $LR(1)$ se le denomina analizador $LR(1)$ pues lee la entrada de izquierda a derecha y encuentra la derivación por la derecha utilizando un átomo de adelanto. Es fácil notar que el número de estados del AFD $LR(1)$ como era de esperar, es mayor que el AFD $LR(0)$ pues contiene mayor información. En general, en la práctica se tiene que el número de estados es en promedio 10 veces mayor en un AFD $LR(1)$ en comparación con un AFD $LR(0)$, por lo que las tablas ACTION y GOTO correspondientes a un analizador $LR(1)$ ocupan demasiado espacio en memoria; este problema era aún más grave en años anteriores en los que el tamaño de la memoria de una computadora era reducido, lo que dejaba fuera de la práctica utilizar analizadores $LR(1)$.

Observemos ahora que si quitamos la información del átomo de adelanto en el AFD $LR(1)$ de la figura 3.21 varios de sus estados se unifican y se obtiene como resultado el AFD $LR(0)$ de la figura 3.20 como era de esperarse. Por ejemplo fijémonos en los estados 3 y 10 que tienen los elementos $[T \longrightarrow \bullet, -]$, $[T \longrightarrow n\bullet, \#]$ y $[T \longrightarrow n\bullet, -]$, $[T \longrightarrow n\bullet,)]$ respectivamente; si hacemos caso omiso del átomo de adelanto, tenemos los mismos elementos en cada uno de los estados, es decir, si el primer componente de sus respectivos elementos coincide decimos que tales elementos tienen el mismo *núcleo* (*core*); de esta manera, al quitar la información referente al átomo de adelanto a los estados 3 y 10 vemos que sus elementos tienen el mismo núcleo y en consecuencia se unen para formar el estado 3 del AFD $LR(0)$ de la figura 3.20; de esta manera; los estados cuyos elementos comparten el mismo núcleo se unifican para formar un único estado en el AFD $LR(0)$.

La idea ahora es unificar los estados que tienen elementos con el mismo núcleo pero manteniendo la información del átomo de adelanto, de la siguiente manera: tomemos como ejemplo los estados 2 y 9 que contienen los elementos $[E \longrightarrow T\bullet, -]$, $[E \longrightarrow T\bullet, \#]$ y $[E \longrightarrow T, -]$, $[E \longrightarrow T\bullet,)]$. El núcleo que comparten el primer par de elementos es $E \longrightarrow T\bullet$ y el que comparten el segundo par de elementos es $E \longrightarrow T\bullet$. Estos estados se unifican para formar el estado 2 del AFD $LR(0)$. En su lugar, bajo nuestra nueva estrategia, mantenemos la información almacenando los elementos $[E \longrightarrow T\bullet, \#]$, $[E \longrightarrow T\bullet, -]$ y $[E \longrightarrow T\bullet,)]$ y por simplicidad simplemente escribiremos $[E \longrightarrow T\bullet, [\#-)]]$. Esto siempre es posible pues, como vimos, todos estos elementos comparten el mismo núcleo, por lo que basta con escribir un único elemento con el núcleo compartido, seguido de todos los posibles átomos de adelanto. El AFD que se construye utilizando la estrategia anterior se muestra a continuación.

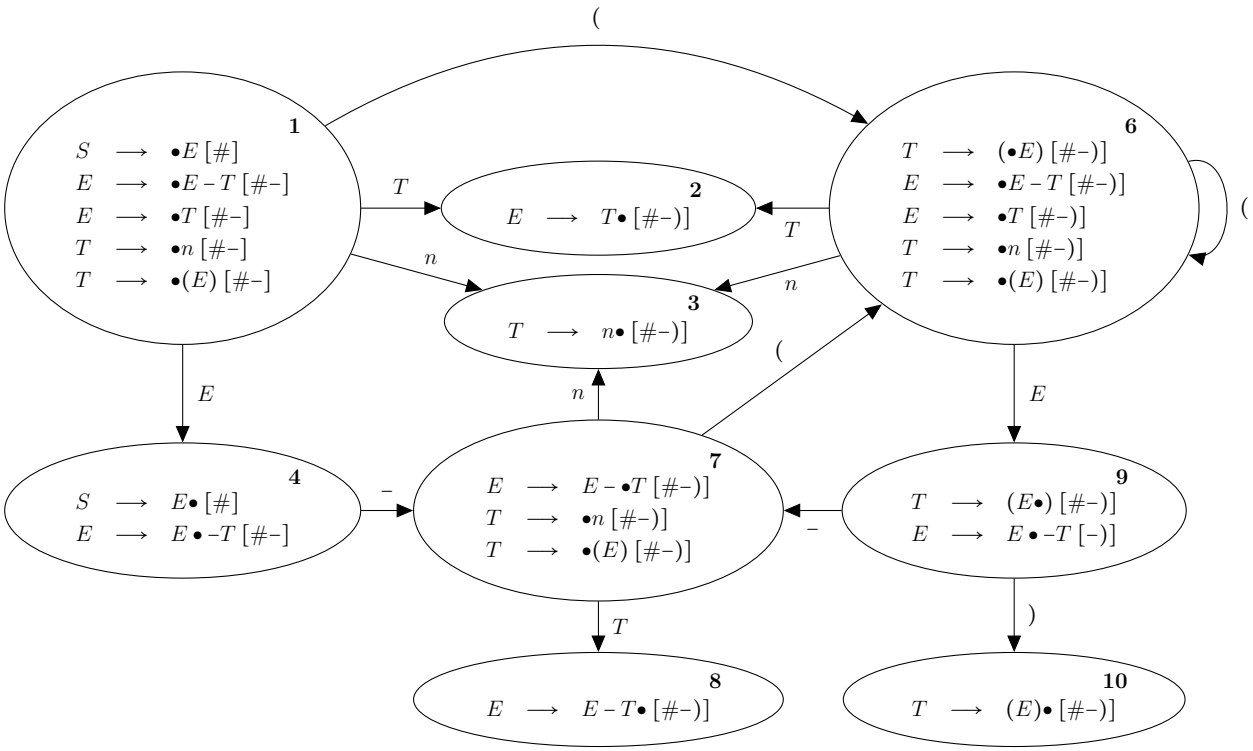


Figura 3.22: AFD LALR(0) correspondiente a la gramática 3.19.

Al AFD anterior se le denomina AFD $LALR(1)$, que indica que hace uso de un átomo de adelanto con base en un AFD $LR(0)$ (*Look Ahead* $LR(0)$). A un analizador que utiliza un AFD $LALR(1)$ se le denomina *analizador* $LALR(1)$.

Es fácil notar que el conflicto en el estado 4 del AFD $LR(0)$ está resuelto en el AFD $LALR(1)$. En efecto, en un AFD $LALR(1)$ se tiene el mismo número de estados que un AFD $LR(0)$ (aunque se ocupa un poco más de memoria debido a que se debe almacenar la información del átomo de adelanto), mientras que se mantiene casi la misma información que en un AFD $LR(1)$.

Es claro que podemos construir un AFD $LALR(1)$ construyendo primero un AFD $LR(1)$ y a partir de este último unificar los estados hasta obtener el AFD $LALR(1)$, como describimos anteriormente. Aunque este método es ilustrativo, es muy ineficiente, por lo que se han desarrollado diferentes algoritmos para realizar la construcción de un AFD $LALR(1)$; uno de ellos es el descrito en [Aho+07] y es el que utiliza yacc, un generador de analizadores sintácticos, mientras que un algoritmo más eficiente es el que se describe en [DP82] y es el que se utiliza en bison.

Aun cuando en un AFD $LR(1)$ se pueden resolver conflictos presentes en un AFD $LR(0)$, puede darse el caso en el que existan conflictos en un AFD $LR(1)$; de la misma

manera puede darse el caso que un AFD $LALR(1)$ presente conflictos. Informalmente, cuando el AFD $LR(1)$ de una gramática G no tiene conflictos se dice que G es una gramática $LR(1)$; de la misma manera, cuando un AFD $LALR(1)$ correspondiente a una gramática G no tiene conflictos se dice que G es una gramática $LALR(1)$.

3.4.5. Recapitulación

Como era nuestro objetivo, hemos desarrollado diferentes tipos de analizadores que encuentran un análisis en tiempo $O(n)$ con respecto al tamaño de la entrada, siguiendo dos estrategias generales, la primera de arriba hacia abajo (descendente) y la segunda de abajo hacia arriba (ascendente).

Los analizadores deterministas descendentes que estudiamos nos permiten trabajar con gramáticas $LL(1)$ pero, como vimos, puede ser que existan conflictos en la tabla correspondiente al analizador $LL(1)$, (en cuyo caso por supuesto la gramática no sería $LL(1)$). En este tipo de analizadores utilizamos un átomo de adelanto. Podemos generalizar esta idea y en lugar de utilizar un solo átomo de adelanto, utilizar k átomos de adelanto con la intención de que si se presenta conflictos en un analizador que haga uso de k átomos de adelanto, entonces podemos probar si en un analizador que utilice $k+1$ átomos de adelanto estos conflictos se resuelven, lo que da origen a las gramáticas $LL(k)$ con $k \geq 0$, que informalmente son aquellas en las que su correspondiente tabla, que utiliza k átomos de adelanto, no presenta conflictos.

Por otra parte, los analizadores ascendentes que estudiamos nos permiten trabajar con gramáticas $LR(0)$ y $LR(1)$. De la misma forma que en el párrafo anterior, podemos generalizar la idea de utilizar un átomo de adelanto para en su lugar utilizar k átomos de adelanto, con la intención de resolver conflictos, lo que da lugar a las gramáticas $LR(k)$ con $k \geq 0$. Por otro lado, los analizadores ascendentes que vimos nos permiten trabajar con gramáticas $LALR(1)$, que de igual forma podemos generalizar obteniendo gramáticas $LALR(k)$ con $k \geq 0$.

Pasamos a puntualizar estos conceptos.

Definición 3.10. Se dice que una gramática es LL si es $LL(k)$ para algún $k \geq 0$.

Definición 3.11. Se dice que una gramática es LR si es $LR(k)$ para algún $k \geq 0$.

Definición 3.12. Se dice que una gramática es $LALR$ si es $LALR(k)$ para algún $k \geq 0$.

En la práctica los analizadores más utilizados son los analizadores $LL(1)$ fuertes (que como dijimos, usualmente se les suele denominar simplemente analizadores $LL(1)$) y los $LALR(1)$, pues utilizar más átomos de adelanto resulta impráctico debido al tamaño de sus tablas. Sin embargo, hay autores que defienden que se debe utilizar más de un

átomo de adelanto [PQ96], lo que permite al diseñador de la gramática mayor libertad; en particular, la herramienta antlr permite generar analizadores $LL(k)$.

Podemos en este momento preguntarnos qué propiedad común tienen las gramáticas $LL(k)$, $LR(k)$ y $LALR(k)$ y qué relación existe entre ellas. Presentemos a continuación algunos de los resultados más importantes en este sentido, cuyas demostraciones se pueden encontrar en [AU72], [SS90] y [RS69] respectivamente.

3.3. Es indecidible determinar si existe un número natural k tal que una gramática libre del contexto G sea $LL(k)$.

3.4. Dadas dos gramáticas $LL(k)$ G_1 y G_2, es decidible determinar si $L(G_1) = L(G_2)$.

3.5. Para todo natural k, la clase de gramáticas $LL(k)$ está contenida propiamente en la clase de gramáticas $LL(k+1)$.

3.6. Para toda $k \geq 0$ existen lenguajes que son $LL(k+1)$ pero no $LL(k)$.

3.7. Para toda $k \geq 0$, la clase de las gramáticas $LR(k)$ está contenida propiamente en la clase de las gramáticas $LR(k+1)$.

3.8. Para toda gramática $LR(k)$ con $k \geq 1$ existe una gramática $LR(1)$ equivalente.

3.9. Cualquier lenguaje $LR(k)$, con $k \geq 0$ es un lenguaje $LR(1)$.

3.10. Todo lenguaje libre del contexto determinista tiene una gramática $LR(1)$ que lo genera.

3.11. Para cualquier alfabeto Σ, la familia de los lenguajes deterministas sobre Σ coincide con la familia de los lenguajes $LR(1)$ sobre Σ.

3.12. Si el lenguaje generado por una gramática $LR(k)$ es libre de prefijo, entonces podemos encontrar una gramática equivalente que sea $LR(0)$.

3.13. Un lenguaje L tiene una gramática $LR(0)$ si y sólo si L es un lenguaje libre del contexto determinista con la propiedad de prefijo.

3.14. Es indecidible determinar cuándo una gramática libre del contexto es $LR(k)$ para algún natural k.

3.15. Para toda $k \geq 0$, la clase de las gramáticas $LL(k)$ está contenida propiamente en la clase de las gramáticas $LR(k)$.

3.16. Para toda $k \geq 0$, la familia de los lenguajes $LL(k)$ está contenida propiamente en la familia de los lenguajes $LR(1)$.

3.17. Dada una gramática $LR(k)$ con k conocido, es decidible si existe un k' tal que la gramática sea $LL(k')$.

3.18. La clase de las gramáticas $LALR(0)$ coincide con la clase de las gramáticas $LR(0)$. Para $k \geq 1$ la clase de las gramáticas $LALR(k)$ está contenida propiamente en la clase de las gramáticas $LR(k)$.

3.19. Sea k un número natural. Cualquier gramática G puede ser transformada en una gramática estructuralmente equivalente que sea $LALR(k)$ si y sólo si G es $LR(k)$.

3.20. Para cualquier natural $k \geq 0$, la familia de los lenguajes $LR(k)$ y la familia de los lenguajes $LALR(k)$ son iguales.

3.21. Para $k \geq 1$, la clase de las gramáticas $LL(k)$ es incomparable con la clase de las gramáticas $LALR(k)$.

3.22. Toda gramática G tal que G es LL no es ambigua.

3.23. Toda gramática G tal que G es LR no es ambigua .

3.24. Toda gramática G tal que G es $LALR$ no es ambigua.

Analicemos algunas de las afirmaciones anteriores comenzando por las gramáticas $LL(k)$; en este tipo de gramáticas tenemos que se forma una jerarquía tanto de gramáticas como de lenguajes siempre que se incrementa k. Por ejemplo, si tenemos una gramática G que es $LL(2)$ y que en consecuencia no es $LL(1)$ (suponiendo que su respectiva tabla $LL(1)$ presenta conflictos) entonces el lenguaje $L(G)$ generado por G es $LL(2)$ y podemos afirmar que no existe ninguna gramática $LL(1)$ G' tal que $L(G') = L(G)$, lo que representa un ejemplo de que la clase de las gramáticas $LL(1)$ está contenida en la clase de las gramáticas $LL(2)$ y de que la familia de los lenguajes $LL(1)$ está contenida propiamente en la de los lenguajes $LL(2)$. Por una parte, es claro que si la tabla $LL(1)$ de una gramática cualquiera no presenta conflictos, entonces tampoco presentará conflictos en su correspondiente tabla $LL(2)$, con lo que todas las gramáticas $LL(1)$ están contenidas en las $LL(2)$; pero además, existen gramáticas, como la de nuestro ejemplo, que son $LL(2)$ y no son $LL(1)$, debido a que no tienen conflictos en su respectiva tabla $LL(2)$ pero sí los tienen en su tabla $LL(1)$, con lo que las gramáticas $LL(1)$ están contenidas propiamente en las gramáticas $LL(2)$. Por otra parte, al no existir ninguna gramática $LL(1)$ G' tal que genere el mismo lenguaje generado por una gramática $LL(2)$ G cualquiera, es claro que los lenguajes $LL(1)$ están contenidos propiamente en los lenguajes $LL(2)$.

En cuanto a las gramáticas $LR(k)$ se tiene que se forma una jerarquía en este tipo de gramáticas siempre que se incrementa k, pero, a diferencia de lo que sucedía con las $LL(k)$, aquí no se forma una jerarquía respecto a los lenguajes, debido a que siempre

que se tiene una gramática $LR(k+1)$ G con $k \geq 1$ se puede transformar en una gramática G' $LR(k)$ tal que $L(G') = L(G)$ (aunque el autómata correspondiente presentara conflictos). El que la clase de gramáticas $LR(k)$ esté contenida propiamente en la clase de las gramáticas $LR(k+1)$ es debido a que si se construye un AFD $LR(k)$ correspondiente a una gramática $LR(k)$ G cualquiera, no presentará conflictos (por definición) y es claro que si se construye el AFD $LR(k+1)$ tampoco presentará conflictos, con lo que todas las gramáticas $LR(k)$ son $LR(k+1)$; en cambio, si tomamos una gramática $LR(k+1)$ G', su correspondiente AFD $LR(k+1)$ no tendrá conflictos (por definición), pero si construimos el AFD $LR(k)$ correspondiente a G' tendrá conflictos (pues en otro caso G' sería $LR(k)$ y no $LR(k+1)$). Por lo tanto las gramáticas $LR(k)$ están contenidas propiamente en las gramáticas $LR(k+1)$; en cambio, la familia de los lenguajes $LR(k)$ con $k \geq 0$ coincide con la familia de los lenguajes $LR(1)$ (aunque la familia de lenguajes $LR(0)$ está contenida propiamente en la familia de lenguajes $LR(1)$), pues siempre que tengamos un lenguaje L que sea $LR(k+1)$ con $k \geq 1$, éste tuvo que ser generado por una gramática mínima $LR(k+1)$ G; entonces podemos transformar a G en una gramática equivalente $LR(k)$ G' tal que $L(G') = L$, con lo que es claro que la familia de lenguajes $LR(k+1)$ y $LR(k)$ coincide; podemos aplicar iterativamente la misma transformación hasta obtener una gramática $LR(1)$ que genere al lenguaje L y que por lo tanto será un lenguaje $LR(1)$; de aquí se sigue que la familia de los lenguajes $LR(k)$ es la misma que la familia de los lenguajes $LR(1)$. El lector podrá preguntarse por qué no realizar la misma transformación una iteración más para obtener una gramática $LR(0)$, pero esto es imposible excepto en el caso que el lenguaje L que sea $LR(1)$, generado por una gramática $LR(1)$ G cualquiera sea libre de prefijo, en cuyo caso la transformación es posible y podemos obtener una gramática $LR(0)$ que genere al lenguaje L, lo que se establece en la siguiente afirmación, cuya demostración se puede ver en [**pp.260**; HU79] y [**pp.636**; Knu65].

Afirmación 3.25. Un lenguaje L tiene una gramática $LR(0)$ si y sólo si L es $LR(k)$ con $k \geq 1$ con la propiedad de prefijo.

Nótese que pedir que L sea libre de prefijo no es una restricción fuerte, pues siempre que tengamos que L sobre un alfabeto Σ no es libre de prefijo, podemos tomar $L' = L\$$ con $\$ \notin \Sigma$ y de esta manera podemos obtener una gramática $LR(0)$ G que genere L', con lo que L' será un lenguaje $LR(0)$.

Podemos ahora estudiar la relación entre las gramáticas $LL(k)$ y $LR(k)$. Se tiene que $\forall k \geq 0$, la clase de las gramáticas $LL(k)$ está contenida propiamente en la clase de las gramáticas $LR(k)$ –[SS90] y [AU73]–; con este resultado, y lo que vimos respecto a las gramáticas $LR(k)$, tenemos que $\forall k \geq 0$ la familia de los lenguajes $LL(k)$ está contenida propiamente en la familia de los lenguajes $LR(1)$.

Todo señalaría que deberíamos utilizar analizadores $LR(1)$ pero, como vimos, este tipo de analizadores están lejos de utilizarse en la práctica, no así los analizadores

$LALR(1)$; nos preguntamos ahora la relación entre las gramáticas $LALR(k)$ y las gramáticas $LR(k)$ y entre las gramáticas $LALR(k)$ y las $LL(k)$.

Las gramáticas $LALR(k)$ con $k \geq 1$ están contenidas propiamente en las gramáticas $LR(k)$, pero a diferencia de lo que sucedía con las gramáticas $LL(k)$, la familia de los lenguajes $LALR(k)$ coincide con la familia de los lenguajes $LR(k)$ (que a su vez es la misma que la familia de los lenguajes $LR(1)$), debido a que si G es una gramática $LR(k)$ entonces podemos transformarla en una gramática estructuralmente equivalente $LALR(k)$ G', tal que $L(G') = L(G)$. En cuanto a las gramáticas $LL(k)$, las gramáticas $LALR(k)$ y las gramáticas $LL(k)$ son incomparables, pues existe por ejemplo al menos una gramática que es $LL(1)$ y que no es $LALR(k)$ para cualquier k y al menos una que es $LALR(1)$ y que no es $LL(k)$ para cualquier k [SS90].

Sin embargo, de lo anterior podemos concluir que si tenemos una gramática G $LL(k)$ para un k fijo, G es $LR(k)$ y tenemos dos casos: ya sea que G es $LALR(k)$ o bien es $LR(k)$ pero no $LALR(k)$, en cuyo caso podemos transformar G en una gramática G' equivalente que sea $LALR(k)$. Este resultado es particularmente importante para nosotros, pues la gramática de Python es una gramática $LL(1)$ y nosotros utilizaremos bison que trabaja con gramáticas $LALR(1)$, así que los dos casos posibles son que la gramática de Python sea ya en efecto una gramática $LALR(1)$ o en otro caso tenemos la certeza que podemos transformarla en una de ellas.

Con lo anterior queda claro que las gramáticas más expresivas son las $LR(k)$ con $k \geq 1$ y por tanto son las que definen los lenguajes libres del contexto deterministas. En nuestra búsqueda de un tipo de gramáticas que defina la sintaxis de un lenguaje de programación podemos decir, en principio, que se pueden utilizar gramáticas $LR(k)$; sin embargo lo deseable es que se utilicen gramáticas $LALR(1)$ o gramáticas $LL(1)$, pues, como hemos mencionado, los analizadores que se utilizan en la práctica son analizadores $LALR(1)$ y $LL(1)$, y aunque se pueden obtener analizadores $LALR(1)$ a partir de gramáticas $LR(k)$ (mediante transformaciones de la gramática), lo ideal es que se utilicen gramáticas $LALR(1)$ o $LL(1)$ en la especificación para hacer más fácil la implementación del lenguaje, y que la semántica de las categorías sintácticas del diseñador del lenguaje sean más claras, ya que no será necesario realizar ninguna transformación en la gramática original.

Por último sólo nos queda por señalar que los analizadores de descenso recursivo que estudiamos en la sección 3.4.1, son muy utilizados en la práctica debido a su facilidad de implementación, además de que es fácil comprender su funcionamiento y que en principio trabajan con cualquier gramática libre del contexto; sin embargo, como señalamos anteriormente, presentan complejidad que puede llegar a ser exponencial y si la gramática es recursiva izquierda o no es libre de prefijo no presentan la propiedad de terminación.

3.5. Bison

A lo largo de la sección 3.4 vimos cómo construir diferentes analizadores sintácticos a partir de una gramática y observamos que los diferentes métodos de construcción correspondientes a los diferentes tipos de analizadores se llevan a cabo mediante algoritmos, por lo que es natural pensar que esta construcción se puede automatizar e implementarse en un programa. Es precisamente a este tipo de programas a los que se les denomina generadores de analizadores sintácticos y a los que también se les ha denominado en el pasado compilador de compiladores (*compiler compiler*). Un generador de analizadores sintácticos toma como entrada la especificación de una gramática G, junto con las acciones semánticas asociadas a cada una de sus producciones, y genera como salida un analizador sintáctico capaz de reconocer los enunciados del lenguaje que genera la gramática G, realizando, durante el análisis las acciones semánticas asociadas con las producciones.

Se pueden construir generadores de analizadores sintácticos para los diferentes tipos de analizadores que estudiamos; así, podemos tener generadores sintácticos que construyan analizadores $LL(k)$, $LR(k)$, $LALR(k)$ e incluso analizadores de descenso recursivo con retroceso. Como hemos mencionado, dado que los analizadores que más se utilizan en la práctica son analizadores $LL(1)$ y $LALR(1)$, la mayoría de los generadores de analizadores sintácticos construyen analizadores $LL(1)$, $LALR(1)$ o ambos, aunque también existen generadores que construyen los otros tipos de analizadores; en particular el generador de analizadores sintácticos antlr es capaz de construir analizadores $LL(k)$ para $k \geq 1$.

Quizás el generador de analizadores sintácticos más conocido es yacc (*yet another compiler compiler*) que fue desarrollado a principios de los años setentas por S.C. Johnson y descrito en [Joh75]. Del nombre de este generador de analizadores sintácticos se puede notar lo popularidad de este tipo de generadores desde aquellos tiempos. yacc genera analizadores $LALR(1)$.

Un generador de generadores sintácticos más reciente de código libre inspirado en yacc es bison. bison es compatible con yacc, aunque tiene diversas extensiones que no están presente en este último; en particular, bison por omisión genera analizadores sintácticos $LALR(1)$, al igual que yacc, aunque tiene la capacidad de generar analizadores $GLR(1)$, que a grandes rasgos se comportan exactamente igual que un analizador $LALR(1)$, excepto cuando durante el análisis se tienen por explorar dos o más caminos debido a la presencia de un conflicto $LALR(1)$, en cuyo caso se sigue la estrategia general en amplitud que estudiamos en la sección 3.4.4. Esto permite que se trabaje con gramáticas que no son estrictamente $LALR(1)$; es claro que seguir esta estrategia puede llevar a que el tiempo de análisis de un enunciado sea exponencial, pero en la práctica se utilizan los analizadores $GLR(1)$ sólo cuando se presentan algunos cuantos conflictos en la gramática que evitan que sea $LALR(1)$ y el tiempo que toma un análisis en la mayoría de estos casos no llega a ser exponencial.

Aunque bison y yacc, en principio, ambos generan analizadores $LALR(1)$,[9] el algoritmo que utilizan internamente para generarlos es diferente; el algoritmo que utiliza yacc es el descrito en la página 270 de [Aho+07], mientras que el que utiliza bison se describe en [DP82], y este último es más eficiente.[10]

A continuación describiremos cómo funciona bison.

bison toma como entrada un archivo de texto con la extensión .y y genera como salida un archivo con extensión .c, que contiene el código fuente escrito en lenguaje C, correspondiente al analizador sintáctico generado a partir del archivo de entrada. También es capaz de producir código en C++, en cuyo caso el archivo de entrada debe tener extensión .yy y bison generará como salida un archivo con extensión .cc. Adicionalmente, en sus versiones recientes, bison es capaz de producir analizadores sintácticos escritos en Java.

A continuación se presenta el esqueleto de un archivo de entrada para bison.

```
%{

Prólogo

%}

Declaraciones de bison

%%

Producciones

%%

Epílogo
```

Código 3.2: Esqueleto de un archivo de entrada válido para bison

Como podemos observar, un archivo de entrada válido de bison está compuesto de 3 secciones: declaraciones de bison, producciones y epílogo, y una de ellas (la sección de declaraciones) contiene la subsección del prólogo. Enseguida describiremos con mayor detalle cada una de estas secciones. Como elemento adicional podemos decir que tenemos la posibilidad de utilizar comentarios largos delimitados por "/*" y "*/" al estilo C en cualquiera de ellas.

[9] Si bien, como acabamos de ver, bison también es capaz de generar analizadores $GLR(1)$, característica no presente en yacc.

[10] Desde la invención de las gramáticas $LALR(1)$ por Frank DeRemer en 1969 ha habido interés entre los investigadores en encontrar un método eficiente que permita construir analizadores $LALR(1)$, por lo que a partir de entonces se han desarrollado diferentes algoritmos con este fin.

3.5.1. Declaraciones de bison

Prólogo

El prólogo es una subsección de la sección de declaraciones de bison que está delimitada por "%{" y "%}" y cuyo contenido se pone sin modificación al inicio del archivo de salida, por lo que se utiliza para que el programador tenga la posibilidad de establecer las declaraciones, definiciones y directivas de preprocesador que le sean necesarias y que es común utilizar al inicio de un archivo de C o C++.

Por otra parte, la sección de declaraciones de bison permite al programador definir los símbolos terminales de la gramática del lenguaje a implementar, es decir, los átomos que el analizador léxico debe ir reportando en el proceso de análisis. Por convención, los átomos que están formados únicamente por un sólo carácter no se deben definir explícitamente, pues el propósito de definir los átomos en esta sección es (como ya se ha mencionado anteriormente) que bison internamente represente estos átomos como un entero, por lo que debe asignar un entero único diferente a cada uno de ellos; en el caso de los átomos formados por un sólo carácter, el entero que se utilizará es el código ASCII correspondiente a dicho carácter y por tal motivo no es necesario definir explícitamente estos átomos. Es precisamente en parte por lo anterior, como se lleva a cabo la cooperación entre flex y bison cuando se utilizan de manera conjunta, pues bison es el encargado de realizar esta asociación entre enteros y átomos y flex de utilizarla (por lo que el programador debe incluir el archivo de cabecera generado por bison en el archivo de entrada de flex). Por ello la función principal que genera flex, yylex, regresa un entero.

Cuando un programador regresa un átomo en alguna de las acciones en flex, por ejemplo mediante el enunciado return IF;, lo que sucede es que se está utilizando la asociación generada por bison y el valor que se está regresando en realidad es un entero. Por supuesto, en nuestro ejemplo el átomo IF debió haber sido declarado en el archivo de entrada de bison, precisamente en la sección que estamos describiendo. Este valor es entregado por el analizador léxico a la función principal que genera bison, yyparse, que es la encargada de realizar el análisis sintáctico. Pero yyparse depende de una función que lleve a cabo el análisis léxico y que tenga como nombre yylex, que es, precisamente como acabamos de describir, el nombre que le da flex a la función principal que genera.

Si bison no se utiliza junto con flex, el programador debe encargarse de escribir una función con el mismo nombre y que realice el análisis léxico. Esta situación se da cuando el programador utiliza bison pero no utiliza flex, y en su lugar escribe un analizador léxico a pie. Como es de imaginarse a estas alturas, la función yyparse invoca a la función yylex siempre que necesita consumir el siguiente átomo durante el proceso de análisis. De esta manera podemos ver cómo, durante el proceso de análisis sintáctico de un archivo fuente, la primera vez que yyparse invoca a yylex, esta función se encarga de entregar el primer átomo del flujo de entrada (regresar el primer entero); paulatinamente yyparse

requerirá el siguiente átomo e invocará por segunda vez a yylex, quien entregará el segundo átomo (regresará el segundo entero) presente en el flujo y así sucesivamente; este proceso continuará hasta que se agoten los átomos del flujo de entrada o, en su defecto, que en alguno de los pasos yyparse determine que el átomo que recibió no es válido de acuerdo a las producciones de la gramática, lo que representa un error y abortará en tal situación.

Para definir un átomo se utiliza la siguiente sintaxis:

```
%token ATOM
```

donde ATOM es el identificador del átomo que se está definiendo. Por convención todos los identificadores de átomos se escriben exclusivamente con mayúsculas y pueden estar formados por letras, dígitos (aunque no pueden comenzar con un dígito), guion bajo y punto.

Para definir más de un átomo se pueden utilizar varios enunciados como el anterior o también es posible definir más de un átomo en el mismo enunciado, escribiendo cada uno de los identificadores correspondientes a cada uno de los átomos, dejando al menos un espacio en blanco entre cada uno de ellos de la siguiente manera:

```
%token ATOM0 ATOM1 ... ATOMN
```

Como hemos mencionado desde el capítulo del análisis léxico, dependiendo del lenguaje fuente existen ciertos átomos que tienen información adicional importante para la siguientes etapas del compilador; un ejemplo clásico es que el lenguaje trabaje con enteros de precisión finita como el caso de C, en cuyo caso el átomo que representa todos los posibles enteros válidos en C se denomina INTEGER; para el analizador sintáctico (que debe en principio verificar únicamente la sintaxis del lenguaje) basta que cuando pida el siguiente átomo al analizador léxico,[11] este último simplemente regrese el átomo INTEGER (en términos de bison y flex, que flex regrese el entero que bison asignó al átomo INTEGER); desde el punto de vista del analizador sintáctico basta con verificar que INTEGER sea un átomo que se esperaba de acuerdo a las producciones de la gramática, es decir, se está verificando que el programa de entrada sea sintácticamente válido de acuerdo a la gramática del lenguaje fuente (tarea del analizador sintáctico). Con lo anterior puede notarse cómo, desde el punto de vista del analizador sintáctico, no es necesaria más información, en otras palabras basta con que se le entregue el tipo del átomo y nada más. Sin embargo, lo anterior no es cierto desde el punto de vista del generador de código, pues éste necesita el valor del entero, no nada más asegurar que se trata de un entero. Suponiendo que ese valor en el código de entrada participa en una suma, es

[11] En este párrafo estamos suponiendo en todo momento que tanto el analizador léxico como el sintáctico se implementan haciendo uso de flex y bison respectivamente. Nótese sin embargo que otras implementaciones podrían utilizar diferentes formas de comunicación entre cada una de estas fases del compilador.

decir [12] que es parte del enunciado `int x = 2+5;`, el generador de código debe generar código de máquina[13] que realice la suma $2+5$ y el resultado asignárselo a la dirección de memoria o el registro que el generador de código haya seleccionado para representar la variable x. Como es fácil notar, el que el generador de código cuente con el valor 5 es de suma importancia para que, en tiempo de ejecución, se lleven a cabo las evaluaciones correctas. En terminología de bison se dice que un átomo puede tener un valor semántico[14], aunque es claro que no todos los átomos tienen que tener un valor semántico, como es el caso del átomo IF por ejemplo. bison permite definir el tipo del valor semántico correspondiente a un átomo particular. Para ilustrar lo anterior, supongamos que estamos desarrollando un intérprete en C que permite sumar números enteros y flotantes, por lo que nuestro analizador léxico generará los átomos INTEGER (que representa a los enteros) y FLOAT (que representa a los flotantes); desde luego el valor semántico de cada uno de los átomos es relevante, pero ¿cuál es el tipo que debe tener cada uno de estos valores? Como veremos más adelante, bison permite acceder al valor semántico de los átomos, así que si estamos pensando en realizar un intérprete, que en este caso se valdrá de las características ofrecidas en el lenguaje de implementación para realizar la evaluación, entonces es natural seleccionar el tipo `int` de C para el valor semántico del átomo INTEGER y el tipo `float` de C para el valor semántico del átomo FLOAT. Como podemos ver, el tipo del valor semántico de cada uno de los átomos de un lenguaje fuente puede ser diferente. Así, cada átomo puede tener un valor semántico con tipo diferente o en su caso dos o más valores semánticos pueden tener el mismo tipo. bison permite declarar cada uno de los tipos correspondientes a cada uno los valores semánticos, de la manera en que lo ilustra el siguiente ejemplo:

```
%union{
  int integert;
  float floatt;
}
```

Con lo anterior hemos declarado que los posibles tipos que pueden tener los valores semánticos de cada uno de los átomos son `int` y `float`; desde luego que se pueden utilizar como tipos cualquier tipo nativo de C o cualquier tipo válido definido por el usuario (ya sea mediante una estructura , una unión o cualquier otro posible camino válido de definir un tipo en C; si estamos utilizando C++ además podemos utilizar un tipo definido mediante una clase o cualquier otro camino válido de definir un tipo en C++). Para indicar qué tipo particular tiene el valor semántico de un átomo utilizamos el enunciado para declarar el átomo, con una modificación, como se muestra a continuación:

[12] Utilizamos un ejemplo trivial simplemente para ilustrar la importancia de lo que estamos señalando.

[13] Estamos suponiendo que este compilador generará código de máquina nativo para una arquitectura específica.

[14] Cuando vimos el analizador léxico definimos a los átomos como parejas $(tipo, valor)$; esta pareja coincide con el valor semántico de bison.

```
%token<integert> INTEGER!
```

Con lo anterior, como ya sabíamos, estamos declarando el átomo INTEGER pero además estamos indicando que el tipo del valor semántico del átomo INTEGER es int, como se describió al escribir union arriba. De manera análoga en nuestro ejemplo debemos tener la siguiente definición:

```
%token<floatt> FLOAT
```

Nótese que lo que se escribe entre < y > es el nombre de la variable y no el tipo. Como el lector tal vez ya lo haya percibido, lo que hace bison es definir en su archivo de salida en C (o C++) una unión conformada internamente por todas las variables declaradas dentro del enunciado compuesto %union{ ... }. A esta unión bison le asignará el nombre de YYSTYPE y utilizará un enunciado typedef para ponerle un alias con nombre YYSTYPE. De esta manera bison hace posible que se declare una variable con tipo YYSTYPE (en su archivo de salida) y que se refiere a la unión anterior[15]; es precisamente con este tipo que se declara la variable que almacena el valor semántico de un determinado átomo. Al hacer, por ejemplo, la declaración %token<integert> INTEGER, bison sabrá que la variable de la unión en la que debe almacenar el valor es integert si es que se trata del átomo INTEGER; en cambio, si se trata de el átomo FLOAT, entonces sabe que debe utilizar la variable floatt.

Lo anterior es un breve descripción de algunas de las cosas que realiza bison internamente y que en principio no deben preocupar al programador, aunque es valioso entenderlo.

Como es de esperar, si más un átomo comparte el mismo tipo de valor semántico, puede utilizarse el mismo enunciado para definir más de un átomo con el mismo tipo de valor semántico, de la siguiente forma:

```
%token<varm> ATOM0 ATOM1 ... ATOMN
```

con lo que se definen n átomos, los cuales todos tienen como tipo de sus respectivos valores semánticos el tipo tm, suponiendo que se tiene antes la declaración:

```
%union{
  t0 var0;
  t1 var1;
    ⋮
  tm varm;
}
```

[15]Cuando se utiliza bison en su versión de C++, a la unión se le da el nombre de semantic_type y se declara dentro de la clase que bison genera; en este caso no se le asigna un alias.

Ahora que sabemos cómo definir el tipo de valor semántico para un determinado átomo, debemos prestar atención a cómo almacenar el valor semántico. Retomando nuestro ejemplo, nuestro intérprete debe ser capaz de reconocer un enunciado como el que mencionamos (5+2); es claro que el analizador léxico, al ver el 5, generará el átomo INTEGER, el cual establecimos que almacenará un valor semántico de tipo int; pero lo que nos interesa es almacenar el valor 5. En este punto es el analizador léxico quien cuenta con la información que deseamos almacenar pues, una vez establecido el flujo de átomos, el primer lexema que encontró el analizador léxico fue 5, el cual determinó que pertenecía al conjunto de cadenas representado por el átomo INTEGER; en este punto el lexema que tiene actualmente el analizador léxico es la información que deseamos almacenar. Lo primero que debemos fijar es dónde deseamos guardar el valor semántico y aquí existen diversas alternativas: una de ellas es comunicar el valor al analizador sintáctico y que sea él quien decida qué hacer con dicho valor (por ejemplo, guardarlo en una estructura de datos o utilizarlo directamente); otra posibilidad es guardarlo directamente en una estructura de datos como un tabla de símbolos, que sea manipulada por las posteriores etapas del compilador. En el caso de nuestro intérprete requerimos que este valor sea comunicado al analizador sintáctico, quien lo utilizará directamente para evaluar las sumas. Es aquí donde estudiamos cómo se lleva a cabo está comunicación de información.

Vimos que cuando flex se encuentra leyendo el flujo de entrada y determina la presencia de un lexema que pertenece al conjunto representado por un átomo, se realizan las acciones asociadas con tal átomo y es en este momento cuando el analizador léxico cuenta con el lexema y aún no se ha reportado el átomo; lo que necesita hacer el programador en las acciones es guardar el lexema en donde se haya determinado y posteriormente reportar el átomo. Por otra parte, recordemos que bison genera analizadores LALR(1), es decir, utiliza un símbolo de adelanto, el cual almacena en una variable de nombre yylval y, como el lector puede haber ya discernido, el tipo de esta variable es YYSTYPE, por lo que la comunicación directa entre flex y bison se da a través de esta variable, la que como vimos, cuando la función yyparse requiera el siguiente átomo de adelanto llamará a la función yylex y será en ese momento cuando el analizador léxico esté en posibilidad de comunicar el valor semántico de manera directa. En nuestro ejemplo, debemos poner lo siguiente en la acciones correspondientes al átomo INTEGER en el archivo de entrada de flex:

```
{integer} { yylval.integert = atoi(yytext); return INTEGER; }
```

yytext es la variable de tipo char* (es decir una cadena al estilo C) donde flex almacena el lexema actual, que es precisamente lo que nos interesa guardar; pero, como establecimos, el valor semántico que debemos almacenar para un átomo INTEGER tiene que ser de tipo int, por lo que debemos realizar un conversión de tipos; de manera similar, para el átomo FLOAT tendríamos

```
{float} { yylval.floatt = atof(yytext); return FLOAT;}
```

De esta manera, cuando utilicemos el valor semántico de los átomos INTEGER y FLOAT, como veremos en la próxima sección, éstos estarán listos pues el analizador léxico los estableció de manera adecuada. Tal vez pudiera parecernos extraño que si bien establecimos el valor semántico del átomo de adelanto, cómo es que éste no se pierde cuando dicho átomo no es más el átomo de adelanto; como hemos mencionado, nuestro intérprete debe ser capaz de realizar sumas de números enteros y flotantes, por lo que utilizaremos la siguiente gramática no ambigua para establecer el lenguaje fuente que acepta nuestro intérprete:

$$E \longrightarrow E + N$$
$$E \longrightarrow N$$
$$N \longrightarrow \textbf{\textit{integer}}$$
$$N \longrightarrow \textbf{\textit{float}}$$

Figura 3.23

Con la gramática establecida, podemos analizar el funcionamiento del analizador sintáctico de nuestro intérprete con el enunciado de entrada $5 + 2.3$. En primer lugar, la función yyparse llama por primera vez a yylex, quien determina el primer átomo INTEGER y establece su valor semántico en 5; en este momento el analizador sintáctico realiza un paso de desplazamiento, pues es su única opción; es aquí cuando el átomo deja de ser un átomo de adelanto y bison lo guarda en la pila de análisis junto con su respectivo valor semántico. Por ello aunque deja de ser el átomo de adelanto, su valor semántico no se pierde. Después bison debe realizar un paso de reducción, donde debe reducir el átomo INTEGER a number; nótese que number es un símbolo no terminal y lo que nosotros deseamos es que el valor semántico de nuestro átomo ahora sea adoptado por number, pues nuestro átomo INTEGER no existirá más debido al paso de reducción. Con esto nos debe quedar claro que los símbolos no terminales, al igual que los átomos, deben tener lo posibilidad de almacenar un valor semántico, lo que desde luego bison permite definir de la manera como lo ilustra el siguiente ejemplo:

```
%type<integert> number
```

Aquí hemos declarado que el símbolo no terminal number tendrá el tipo de la variable integert que, como declaramos previamente, es int. Como se observa, el enunciado que permite establecer el tipo del valor semántico de un símbolo no terminal es similar al enunciado que permite definir el tipo del valor semántico de un átomo, solo que en lugar de utilizar la palabra token se usa la palabra type; en este paso resulta claro para nosotros que el valor semántico del símbolo no terminal number debe ser establecido como el valor semántico del átomo integer, pues se está haciendo un paso de reducción con

base en la producción $N \longrightarrow$ ***integer***; sin embargo, hay casos en los que una producción puede tener más de un símbolo del lado derecho de la producción y entonces es labor del programador determinar qué valor será el que almacenará el símbolo no terminal del lado izquierdo de la producción, lo que debe ser establecido explícitamente en las acciones asociadas a esa producción, como se mostrará en la próxima sección; si no se establecen acciones explícitas asociadas con una producción, la acción por omisión que realiza bison es establecer el valor semántico del lado izquierdo de la producción como el valor semántico del primer símbolo que aparece del lado derecho de la producción, que es precisamente lo que nosotros queremos para esta producción. Aun cuando ésta es la acción por omisión que realizaría bison, es mejor escribirla explícitamente, pues da la certeza de qué es lo que el programador desea. Una vez que se completó este paso, se realiza un paso de reducción del símbolo no terminal number al símbolo no terminal exp, por lo que tendríamos que hacer la definición

```
%type<integert> exp
```

Una vez más, como es de esperarse, bison permite que se defina el tipo del valor semántico de más de un símbolo no terminal en un mismo enunciado, siempre que aquéllos compartan el mismo tipo, de las misma forma que sucedía con los átomos, por lo que nosotros en lugar de escribir las dos definiciones anterior escribiríamos:

```
%type<integert> number exp
```

y pediríamos en las acciones de la producción $E \longrightarrow N$ que el valor semántico del símbolo no terminal exp adoptara el valor semántico del símbolo no terminal number. A continuación yyparse invoca por segunda ocasión a yylex, quien detecta en el flujo de entrada el átomo + y lo reporta; el analizador sintáctico realiza un desplazamiento y se invoca por tercera ocasión a yylex; esta vez yylex regresa el átomo FLOAT, estableciendo su valor semántico como 2.3; el analizador sintáctico realiza un desplazamiento y posteriormente un paso de reducción del átomo FLOAT al símbolo no terminal number; es aquí donde estamos en problemas, pues habíamos establecido que el tipo del valor semántico del símbolo no terminal number es int, pues cuando se reduce un átomo INTEGER a number, deseamos que number almacene un valor entero; pero por otra parte, en este momento se va a reducir un átomo FLOAT a number, por lo que queremos que el tipo del valor semántico de number sea float; desde luego podemos establecer que el tipo del valor semántico de number sea float, y cuando se reduzca un átomo INTEGER convertir el valor semántico de INTEGER a tipo float, lo que implicaría que la suma se realizara entre números flotantes, aunque esto no fuese necesario en el que caso en que ambos operandos fueran de tipo entero. Quisiéramos que cuando se sumen dos enteros se realice directamente una adición entera. Una posible solución es que declaremos el tipo del valor semántico tanto de los átomos INTEGER y FLOAT, como de los símbolos no terminales number y exp, como una estructura que contenga en su interior una unión

conformada por una variable de tipo int y otra de tipo float, junto con una enumeración que nos indique si hemos guardado un entero o un flotante, por lo que tendríamos las siguientes definiciones:

```
  %union{
   struct envol{
    enum {INTEGERT,FLOATT} kind;
    union{
      int integert;
      float floatt;
    } uin;
  } sin;
}

%token<sin> INTEGER FLOAT
%type<sin> number exp
```

Así, cuando se reduzca un átomo INTEGER a number como dijimos, queremos que number almacene un entero, con lo que siguiendo la estrategia anterior, number almacenará una estructura que contiene una unión, donde se puede almacenar ya sea un valor entero o un flotante, y una etiqueta que indique qué tipo de valor se guardó, por lo que es tarea del programador guardar el entero dentro de la unión y establecer en la enumeración que se guardó un valor de tipo entero; similarmente para un valor de tipo flotante.

De esta manera hemos librado el conflicto de tipos en los valores semánticos que teníamos, pues number debía guardar tanto un valor entero debido a un átomo INTEGER, como un valor flotante debido a un átomo FLOAT, y lo que hemos hecho es envolver ambos tipos en una estructura, que es deber del programador manipular internamente de manera correcta. Con lo anterior, el tipo de la variable yylval definido como YYSTYPE como mencionamos antes, es una unión que contiene en su interior una estructura de tipo struct envol, la cual a su vez contiene en su interior una enumeración y una unión. Así, cuando el analizador léxico encuentre un átomo INTEGER se deben realizar las siguientes acciones:

```
{integer} { yylval.sin.kind=INTEGERT;
            yylval.sin.uin.integert=atoi(yytext); return INTEGER;}
```

mientras que cuando encuentre un átomo FLOAT las siguientes:

```
{float}     { yylval.sin.kind=FLOATT;
              yylval.sin.uin.floatt = atof(yytext); return FLOAT;}
```

Como podemos ver, el tipo de la variable yylval en este ejemplo es una unión que contiene un su interior una estructura, donde en este caso la unión resulta irrelevante

ya que el único elemento que contiene y que vamos a utilizar de ella es la estructura; lo que quisiéramos es que YYSTYPE fuera directamente la estructura. El que YYSTYPE sea una unión es debido a la declaración de bison:

```
%union{
  t0 var0;
     ⋮
  tn varn;
}
```

que en principio, como mencionamos, sirve para poder establecer los tipos de los valores semánticos tanto de los símbolos terminales como los no terminales; en el caso de nuestro intérprete esto no fue de mucha ayuda, ya que si bien nos permitía establecer un tipo diferente para lo átomos INTEGER y FLOAT, cuando cada uno de ellos se tenía que reducir a number teníamos un conflicto de tipos, ya que el símbolo no terminal number debía guardar tanto un entero como un flotante. En nuestra estrategia estamos diciendo que tanto los símbolos terminales INTEGER y FLOAT como los símbolos no terminales number y exp tendrán el mismo tipo mediante las definiciones:

```
%token<sin> INTEGER FLOAT
%type<sin> number exp
```

que corresponde al tipo de la variable sin, que es structenvol, lo que hace innecesario la unión; lo que queremos es que YYSTYPE sea un alias del tipo struct envol, por lo que podemos declarar directamente la estructura y después utilizar la directiva del preprocesador #define para definir YYSTYPE como un alias del tipo struct envol; así, yylval (y todos los valores semánticos de los símbolos de la gramática) tendrán el tipo struct envol, con lo cual ya no es necesario hacer uso del enunciado %union{...} ni especificar explícitamente el tipo de los valores semánticos de los símbolos de la gramática. Lo anterior se declara de la siguiente manera:

```
%code requires{
#define YYSTYPE struct envol
struct envol{
  enum {INTEGERT,FLOATT} kind;
  union{
    int integert;
    float floatt;
  } uin;
};

}

%token INTEGER FLOAT
```

y ahora las reglas de los átomos INTEGER y FLOAT en el analizador léxico deben ser las siguientes:

```
{integer} { yylval.kind=INTEGERT; yylval.uin.integert = atoi(yytext);
            return INTEGER;}
{float}   { yylval.kind = FLOATT; yylval.uin.floatt = atof(yytext);
            return FLOAT;}
```

Como resumen podemos resaltar que podemos hacer uso del enunciado %union{...} junto con los enunciados %token y %type para definir el tipo de los valores semánticos, tanto de los símbolos terminales como de los no terminales; o en otro caso definir directamente el tipo de YYSTYPE, que será el tipo de los valores semántico de todos los símbolos y el programador encargarse explícitamente de manejarlo correctamente.

Por último decimos que debemos colocar al final de esta sección, en una nueva línea, los caracteres %% que denotan al fin de la sección.

3.5.2. Producciones

La presente sección permite al programador definir las producciones de la gramática y asociar con cada una de ellas un conjunto de acciones que deben realizarse siempre que durante un paso del análisis sintáctico de un enunciado de entrada intervenga la producción.

Para escribir una producción se debe comenzar escribiendo el lado izquierdo de la producción al inicio de una línea seguido de ":" (que juega el papel de "$\longrightarrow$" en bison); después se debe dejar al menos un espacio en blanco y escribir los símbolos que forman el lado derecho de la producción, dejando al menos un espacio en blanco entre cada uno de ellos. Una vez que se ha escrito por completo el cuerpo de la producción, se debe dejar al menos un espacio en blanco y escribir las acciones semánticas correspondientes a dicha producción, contenidas en un bloque delimitado por "{" y "}", donde las acciones semánticas son enunciados escritos en C o en C++, por ejemplo. Por último, debe colocarse un ";" al final de la producción (usualmente se coloca al principio de la siguiente línea), lo que denota que ha terminado la definición de la producción. Por ejemplo, la producción $N \longrightarrow$ ***integer*** de la gramática de nuestro intérprete, se escribe en bison de la siguiente manera:

```
num : INTEGER {$$$=$1;}
;
```

Cuando dos o más producciones comparten el mismo lado izquierdo podemos utilizar el carácter " | " para escribir las diferentes producciones con el mismo lado izquierdo. Por ejemplo, para escribir las producciones $N \longrightarrow$ ***integer*** y $N \longrightarrow$ ***float*** de nuestra gramática lo hacemos de la siguiente forma:

```
num: INTEGER {$$$=$1;}
        | FLOAT     {$$$=$1;}
;
```

Nótese que el ";" debe escribirse sólo una vez al final después de haber escrito todas las producciones que comparten el símbolo del lado izquierdo.

Como podemos observar en nuestro ejemplo, en las acciones podemos acceder a los valores semánticos de cada uno de los símbolos que forman parte de la producción. Para acceder al valor semántico del lado izquierdo de la producción utilizamos $$ y para acceder al valor semánticos de los símbolos del lado derecho de la producción utilizamos $$n donde n denota el n-ésimo lugar que ocupa el símbolo en el lado derecho de la producción, así para acceder el valor semántico del primer símbolo que está del lado derecho de la producción utilizamos $$1.

En nuestro ejemplo, en la primer producción estamos estableciendo en las reglas que el valor semántico del átomo INTEGER, que es el primer símbolo del lado derecho de la producción, sea almacenado como el valor semántico del símbolo no terminal num; de forma análoga en la segunda producción estamos pidiendo que el valor semántico del átomo FLOAT se almacene como valor semántico del símbolo num. La definición de las otras producciones de la gramática de nuestro intérprete son las siguientes:

```
exp: exp '+' num  { if( $1.kind == INTEGERT && $3.kind == INTEGERT ){
                      printf("Se realizo una adicion entera\n");
                      int r = $1.uin.integert + $3.uin.integert;
                      printf("El resultado es: %d\n",r);
                                  }else{
                       float r;
                       if( $1.kind == INTEGERT && $3.kind == FLOATT )
                          {
                           r = $1.uin.integert + $3.uin.floatt;
                       } else if( $1.kind == FLOATT && $3.kind ==
                          INTEGERT){
                          r = $1.uin.floatt + $3.uin.integert;
                       } else {
                         r = $1.uin.floatt + $3.uin.floatt;
                       }
                       printf("Se realizo una adicion punto flotante\
                          n");
                       printf("El resultado es: %f\n",r);
                    }
                  }
| num {$$$=$1;}
;
```

Al igual que con la sección de declaraciones, el fin de la sección de producciones se establece colocando "%%" al final de la sección al inicio de una nueva línea. No es necesario que este fin de sección se coloque si no hay sección de epílogo.

3.5.3. Epílogo

El epílogo es una sección opcional. Cuando está presente se debe escribir "%%" al inicio de una nueva línea, al final de la sección de reglas de la gramática, en cuyo caso denota el final de ésta y el comienzo del epílogo. Todo lo que se escriba en esta sección se escribirá sin modificación al final del archivo de salida producido por bison. Aquí podemos escribir las definiciones de funciones auxiliares que sean útiles para el analizador sintáctico (por ejemplo yyerror) , funciones que sean de ayuda para el programador en las acciones semánticas de las producciones, o bien funciones que hagan uso de la función yyparse que genera bison.

La función principal que genera bison, como ya mencionamos, es yyparse, que tiene el encabezado:

```
int yyparse(void)
```

La comunicación entre flex y bison, como vimos, por omisión se realiza mediante la variable global yylval para pasar el valor semántico del átomo; adicionalmente se usa la variable global yylloc para pasar la ubicación del átomo en la entrada. Utilizar variables globales para la comunicación ciertamente no es recomendable, porque entre otros aspectos son variables que no están protegidas. Para remediar esto en bison contamos con la opción %define api.pure, que debemos colocar al inicio del archivo de entrada de bison y que causa que las variables yylval y yylloc sean variables locales de la función yyparse, que pasa como argumentos a la función yylex un apuntador a cada una de ellas; de esta manera se realiza la comunicación entre ellas. Para más información acerca del funcionamiento de flex y bison puede consultarse [PM07] y [DS09] respectivamente.

3.6. Analizador sintáctico de Python

Debemos enfocar ahora nuestros esfuerzos en realizar la implementación del analizador sintáctico de nuestro subconjunto de Python.

Nuestro objetivo es construir una representación intermedia de la entrada, a partir de la cual posteriormente podamos realizar el análisis semántico y la generación de código, aunque en su lugar podríamos realizar lo que resta del proceso de compilación, es decir, el análisis sintáctico, semántico y generación de código en una sola pasada, en un solo módulo al estilo de un compilador dirigido por la sintaxis. Esto tiene varias desventajas como hacer el compilador más difícil de entender, mantener y desarrollar,

además de que perdemos la posibilidad de realizar diferentes optimizaciones; asimismo dificulta de manera importante la implementación de las restantes etapas en el proceso de compilación, como lo habíamos discutido ya en el capítulo 1.

Requerimos de los árboles de derivación, a los que también se les denomina árboles de análisis (*parse trees*) o árboles de sintaxis concreta (*concrete syntax trees*), que reflejan (de manera gráfica) la derivación de un enunciado de entrada con base en la gramática. En ellos, cada uno de sus nodos interiores refleja el uso de un símbolo no terminal en el proceso de derivación, mientras que sus nodos hoja representan cada uno de los símbolos terminales presentes en el enunciado de entrada.

Esta información es excesiva para nuestros fines, pues en nuestra gramática puede haber símbolos no terminales que simplemente sean superfluos y que sólo sean símbolos auxiliares; o símbolos terminales como un nodo hoja correspondiente a "**;**", que usualmente denota el fin de un enunciado en un lenguaje de programación, pero que no es relevante en las siguientes etapas del compilador. Lo que a nosotros nos interesa es tener nodos que correspondan a estructuras del lenguaje como son: un bloque **if** o una asignación de una variable. La representación intermedia que nos interesa construir es un *árbol de sintaxis abstracta* (*Abstract Syntax Tree*, *AST*). En un árbol de sintaxis abstracta cualquier estructura del lenguaje puede modelarse mediante un nodo, que tenga como hijos otros nodos que contengan información semántica relevante, y que formen parte de esa estructura; por ejemplo, para una estructura **if** podemos tener un nodo que tenga tres nodos como hijos: el primero que represente la guardia del **if**, el segundo que represente el cuerpo del **if**, es decir, el bloque que se debe ejecutar en caso de que el valor de la guardia sea verdadero, y por último un nodo que represente el cuerpo del else, es decir el bloque que se debe ejecutar en caso de que la guardia se evalúe como falso. Nótese que en la gramática puede haber más símbolos no terminales que ayuden a construir una estructura **if**, pero que no formarán parte del AST.

En otras palabras, un AST retiene sólo la información relevante que es necesaria para las siguientes fases en el proceso de compilación. Por ello debemos construir un AST.

Si bien nuestro principal objetivo es construir un AST en memoria, puede ser que también sea conveniente construir otro tipo de representaciones del AST, por ejemplo una representación gráfica o una representación basada en texto, ambas con fines de depuración; o puede ser que en un futuro sea conveniente construir una representación distinta que sea de utilidad en nuestro compilador. Tomando en cuenta lo anterior, una vez más haremos uso del patrón Builder como lo hicimos en la implementación de nuestro analizador léxico.

Como vimos en el capítulo anterior, el patrón Builder nos permite construir de manera transparente diferentes representaciones del objeto que deseamos crear, que en este caso es un AST; pero además necesitamos un diseño que nos permita manipular de manera uniforme, es decir, mediante una misma interfaz, cada uno de los diferentes tipos de nodos del AST, no importando de manera particular de qué tipo de nodo se trate –ya

sea un nodo que represente una estructura **if** o un nodo hoja del AST que represente un literal de cadena–, lo que facilita en gran manera la construcción de un árbol a partir de uno o más (sub)árboles de manera recursiva y la manipulación del AST. Por ello utilizaremos el patrón Compuesto (*Composite*) con este fin. El patrón Composite tiene los siguiente participantes:

- Componente (*Component*).
  - Declara la interfaz para los objetos que forman parte de la composición (del AST en nuestro caso).
  - Implementa el comportamiento por omisión de la interfaz.
  - Declara una interfaz para acceder y modificar los componentes hijo.
- Hoja (*Leaf*). Representa una hoja dentro de la composición, además de definir el comportamiento de ellas.
- Compuesto (*Composite*)
  - Define el comportamiento de aquellos componentes que tienen hijos.
  - Almacena componentes hijo.
  - Implementa los métodos para manipular hijos presentes en la interfaz establecida por Componente.
- Cliente (*Client*). Manipula objetos que forman parte de la composición a través de la interfaz establecida en Componente.

Además, una vez que se haya construido el AST, dependiendo de la organización y arquitectura del compilador, se realizan uno o más recorridos (pasadas) sobre el árbol que lleven a cabo las siguientes etapas del proceso de compilación como el análisis semántico y, posteriormente, el generador de código. Esto implica que debemos ser capaces de realizar operaciones específicas particulares para cada tipo de nodo diferente en el AST. Si bien el patrón *Composite* nos permite manipular de manera uniforme los diferentes tipos de nodo del AST, cuando se realiza una operación sobre todo el árbol –por ejemplo la verificación de tipos–, necesitamos que ésta se comporte de manera diferente dependiendo del tipo del nodo del AST; así, si se está realizando la verificación de tipos de un nodo IF queremos que el tipo de la guardia sea booleano, mientras que para el cuerpo del IF debemos realizar recursivamente la verificación de tipos sobre cada uno de sus enunciados, lo mismo para el cuerpo del ELSE. En cambio, si estamos realizando la verificación de tipos para un nodo PLUS, debemos verificar que tanto su hijo izquierdo, que representa el primer operando, como su hijo derecho, que representa el segundo operando, tengan tipos compatibles, para que se pueda llevar a cabo la suma entre ellos. Por tanto, un camino es, a medida que se va recorriendo recursivamente el árbol, hacer un análisis de casos para ver de qué tipo concreto es el nodo actual sobre el que estamos

trabajando, y con base en este tipo realizar las operaciones específicas correspondientes, lo cual podemos implementar mediante un **switch**. Otra opción es implementar la operación dentro de las diferentes clases concretas que representan los diferentes tipos de nodo, poniendo en cada una ellas un método con el nombre de la operación que se realizará; por ejemplo, typecheck que lleve a cabo de manera recursiva la verificación de tipos; aquí el método typecheck es diferente para cada una de las clases y debe contener código específico que realice la verificación de tipos, dependiendo de qué clase concreta se trate, es decir, del tipo del nodo del AST; ésta, aunque parece una mejor opción, tiene el inconveniente que por cada operación diferente que se realice sobre el árbol tenemos que modificar cada una de las clases que representan un tipo de nodo diferente del AST, añadiendo a cada una de ellas el método que realizará tal operación. Lo que nosotros buscamos es un diseño que nos permita realizar operaciones específicas para los diferentes tipos de nodos del AST sin modificar cada una de las clases, y que siempre que se requiera realizar un nueva operación sobre el árbol ésta pueda ser añadida de manera independiente, lo que se logra mediante el uso del patrón Visitante (*Visitor*). Los participantes del patrón *Visitor* se describen a continuación:

- Visitante (*Visitor*)
  - Declara una operación Visita para cada clase de ElementoConcreto (*ConcretElement*) de la estructura conformada por objetos; en este caso, para cada clase concreta que representa un nodo concreto distinto dentro del AST.
- VisitanteConcreto (*ConcreteVisitor*)
  - Implementa cada una de las operaciones declaradas por Visitante. Cada operación implementa un fragmento del algoritmo definido para una clase específica, cuyos objetos forman parte de la estructura. VisitanteConcreto provee el contexto para el algoritmo y guarda su estado local. Este estado frecuentemente acumula resultados durante el recorrido de la estructura.
- Elemento (*Element*)
  - Define una operación Aceptar (*Accept*) que toma un visitante como argumento.
- ElementoConcreto (*ConcretElement*)
  - Implementa una operación Aceptar que toma un visitante como argumento.
- EstructuraObjetos (*ObjectStructure*)
  - Puede enumerar sus elementos.
  - Puede proveer una interfaz de alto nivel que permita al visitante visitar sus elementos.
  - Puede ser un Compuesto (como en nuestro caso) o una colección, tal como una lista o un conjunto.

A continuación describiremos el diseño de nuestro analizador sintáctico, haciendo uso de los patrones: Builder, Composite y Visitor.

La estrategia general es hacer uso del patrón Builder para construir una estructura del patrón Composite, que en nuestro caso será un AST, y después hacer uso del patrón Visitor para realizar cada una de las operaciones que deban realizar las siguientes etapas del compilador sobre el AST.

Primero describiremos las clases que implementan el patrón Builder.

La clase ASTBuilder juega el papel de Constructor y provee la interfaz, es decir, las declaraciones de los métodos que permiten crear las diferentes partes, en este caso los diferentes nodos del AST. Algunos de estos métodos son los siguientes:

```
class ASTBuilder{
public:
 virtual void makeSiblings(Node* n,Node* ns)=0;
 virtual Node* bPlusNode()=0;
 virtual Node* bIFNode()=0;
};
```

El primero de ellos hace que dos nodos diferentes se conviertan en nodos hermano; el segundo permite construir un nodo que representa una adición, mientras que el tercero permite construir un nodo IF. La clase MASTBuilder juega el papel de ConstructorConcreto y se encarga de construir y ensamblar los diferentes nodos para construir un AST en memoria principal, implementando la interfaz establecida por ASTBuilder. La implementación ofrecida por MASTBuilder de los métodos anteriores es la siguiente:

```
void MASTBuilder::makeSiblings(Node* n,Node* ns){
  n->makeSiblings(ns);
}

Node* MASTBuilder::bPlusNode(){
  return new PlusNode;
}

Node* MASTBuilder::bIFNode(){
  return new IFNode;
}
```

En caso de que se desee construir una representación diferente del AST, como puede ser una representación en texto en lugar de memoria principal, debemos crear una clase, por ejemplo TASTBuilder, que implemente la interfaz provista por ASTBuilder y que construya el AST codificado en texto. La clase Parser, que es la que va generar bison, juega el papel de Director, pues construye un AST utilizando la interfaz provista por

ASTBuilder; es esta clase que al utilizar la interfaz de ASTBuilder puede trabajar con diferentes constructores concretos de manera transparente; por ejemplo, si se crea un objeto de la clase MASTBuilder y se establece como el ConstructorConcreto de Parser, entonces se construirá un AST en memoria; en cambio, si se construye un objeto de TASTBuilder y se establece como el ConstructorConcreto de Parser, entonces se construirá un AST en texto de manera completamente transparente; es decir, el código de Parser no tiene que cambiar en lo absoluto.

A continuación mostramos una de las producciones de nuestro analizador sintáctico en bison:

```
atom:  NUMBER   {  $$ = astb->bIntNode($$1); }
;
```

Aquí podemos notar cómo estamos haciendo el objeto astb, que es un objeto de una clase ConstructorConcreto (ya sea MASTBuilder o TASTBuilder), que mediante la interfaz provista por el Constructor (ASTBuilder) nos permite construir un nodo INT del AST. La clase Node juega el papel de Producto, ya que el producto que está construyendo es un AST que se representa mediante un objeto de alguna de las clases concretas, que implementan la interfaz Node, que sea la raíz del AST.

El patrón Composite

Describiremos ahora las clases que implementan el patrón Composite. La clase Node juega el papel de Componente, pues declara la interfaz para los objetos en la composición, que en nuestro caso constituyen los diferentes tipos de nodo del AST; además, implementa la interfaz (es decir, los métodos) con el comportamiento por omisión que deben tener los objetos (los diferentes tipos de nodo) en la composición (en el AST), y permite que, ya sea un Compuesto (un tipo de nodo que tenga hijos) o una Hoja (un tipo de nodo sin hijos), los implemente (los sobrescriba) para modificar este comportamiento por omisión por uno específico del Compuesto o de la Hoja; declara, además, una interfaz para acceder y manipular los nodos hijo, que los tipos de nodos que tengan hijos (los compuestos) deben implementar. El papel Hoja lo juegan todas aquellas clases que representan nodos hoja en el AST, como son IntNode, StNode, BreakNode y PassNode, por mencionar algunos; además, hemos creado una jerarquía de clases que nos da una mayor claridad en el diseño y nos permite factorizar aquellas operaciones que son comunes a un grupo de clases que comparten ciertas características; por ejemplo, hemos declarado una clase LeafNode que hereda de Node para cumplir con su interfaz, pero además todas aquellas clases que representen una Hoja, heredarán de LeafNode; de esta manera, si en algún momento se requiere que los nodos Hoja tenga un comportamiento común por omisión, exclusivo de los nodos Hoja, basta con implementar el correspondiente método en la clase LeafNode; si una de estas clases requiere un comportamiento

específico, aún es libre de implementar la interfaz (el o los métodos requeridos). También, como dijimos, esto da claridad en el diseño y en el código, ya que al ver que estas clases heredan de LeafNode rápidamente se tiene la idea de que son nodos Hoja.

El papel de Compuesto lo juegan todas aquellas clases que representan los diferentes tipos de nodos del AST que tienen hijos. Similarmente al caso de la clase Hoja, hemos definido una jerarquía de clases para que todas las clases que tienen nodos hijo hereden (son hijos o descendientes) de la clase INode; para ello esta clase declara como un atributo privado una lista de nodos hijo y define, entre otros métodos, los métodos addFChild y addLChild, que agregan el nodo que se les pasa como argumento, como primer o último hijo (dentro de su lista de hijos) respectivamente; de esta clase hereda directamente, por ejemplo, la clase BlockNode, que representa un bloque de código, como puede ser el cuerpo de una función; sus hijos son los nodos que representan los enunciados que conforman el bloque. Esta clase, por supuesto, cuenta con la implementación de los métodos addFChild y addLChild, que heredó de INode y que no es necesario que los sobrescriba. En cambio, en la clase BinNode, que representa aquellos nodos que deben tener exactamente dos hijos, también hereda de la clase INode y está pensada que de ella hereden las clases que tendrán exactamente dos hijos. Por ello aquí se deben sobrescribir los métodos addFChild y addLChild respectivamente, que no tienen sentido en esta clase ya que ésta sólo debe permitir que se almacenen dos hijos y no un número arbitrario de hijos, para que lancen excepciones y en su lugar definir los métodos setFChild y setSChild que permitan establecer tanto el primer como el segundo hijo.

Entre las clases que extienden a BinNode podemos mencionar FDParamNode, que representa un argumento por omisión en una función y que está compuesto tanto por el nombre del argumento como por su valor por omisión, por lo que tiene exactamente dos hijos.

Desde luego, para mantener la uniformidad que establece el patrón Composite, debemos definir setFChild y setSChild en la clase Node y que en dicha clase estos métodos lancen excepciones, porque en ese nivel de la jerarquía no tienen sentido. En cambio, al nivel de la clase BinNode si lo tienen. Por supuesto que podríamos seguir utilizando los métodos addFChild y addLChild y confiar que el programador tenga el cuidado de establecer únicamente dos hijos y en el orden correcto, pero esto es una mala práctica de programación y de ingeniería de software, ya que la interfaz debe imponer las restricciones para que una clase se utilice únicamente de manera correcta: se sigue el principio de que si algo es incorrecto es mejor que se detecte lo antes posible y no al revés; en este caso, con nuestro diseño, un mal uso de la clase se detectará en tiempo de compilación, mientras que si no imponemos las restricciones será un error que puede ser reportado por el sistema en tiempo de ejecución o, aún peor, que no sea detectado por éste y que el programador tenga que hacer una depuración minuciosa para detectar el error debido a resultados inesperados en la ejecución del programa.

Entonces, las clases que juegan el rol de Compuesto deben implementar la interfaz

que manipula los hijos a diferente niveles de la jerarquía, según el diseño que se establezca. Como vimos en los ejemplos anteriores también deben encargarse de guardar la lista de hijos (que corresponda con la especificidad de cada uno de ellos). De lo anterior, INode declara una lista de hijos como un atributo privado, pero también diferentes nodos interiores del AST pueden tener ya sea un número arbitrario de hijos, como es el caso de un BlockNode, o un número fijo de hijos como es el caso de FDParamNode (en este caso dos). Por lo tanto, quisiéramos que la representación interna de la lista de hijos en el caso de la clase BlockNode fuera una lista ligada, mientras que en el caso de FDParamNode –y todas las clases que heredan de BinNode– tuvieran como representación interna un vector de tamaño dos, por lo que con este fin hemos creado la clase NodeList y hemos declarado la lista de nodos hijo de la clase INode como NodeList* chen. La clase NodeList nos sirve para encapsular la implementación interna de la lista, permitiendo (aún) que se manipule de la misma forma por las clases externas, no importando que sea internamente una lista ligada, un vector u otra representación que en un futuro pueda ser de utilidad. La clase NodeList define las operaciones que se pueden realizar sobre la lista, pero permite que se crean objetos de ella; se debe crear un clase diferente por cada representación distinta que se requiera y cada una de esas clases debe implementar la interfaz establecida por NodeList; en particular, tenemos la clase LNodeList que representa internamente la lista como una lista ligada y la clase VNodeList que representa internamente la lista como un vector; como es la clase INode la que mantiene este atributo, es ésta quien ofrece constructores que pueden llamar a sus clases descendientes para poder inicializar este atributo con el tipo concreto de NodeList que elijan. Así la clase BinNode tiene el siguiente constructor:

```
BinNode(): INode(2){}
```

que llama al siguiente constructor de la clase INode:

```
INode(int nls): Node(), chen(new VNodeList(nls))
```

Este último creará un lista de nodos representada internamente por vector de tamaño dos, que es justo el comportamiento que esperábamos, mientras que para BlockNode tenemos:

```
BlockNode(): INode(){}
```

y

```
INode(): Node(), chen(new LNodeList) {}
```

respectivamente, con lo que se creará un lista de nodos representada internamente por una lista ligada.

El papel del Cliente lo juegan la clase MASTBuilder –pues manipula los diferentes tipos de nodo mediante la interfaz establecida por la clase Node– y la clase PrintVisitor .

También consideramos acá todas las clases que juegan el papel de un VisitanteConcreto en el patrón Visitor, que utilizan la interfaz provista por Node para manipular alguno de los nodos o todo el AST.

El patrón Visitor

Toca el turno a las clases que implementan el patrón Visitor. Como vimos, el patrón Visitor permite realizar de manera independiente diferentes operaciones sobre una estructura, que en este caso es el AST. Por ejemplo, una de estas operaciones puede ser la verificación de tipos, otra la generación de código. El patrón Visitor permite aislar estas operaciones de las clases que representan los elementos que conforman la estructura.

En nuestro caso, el patrón Visitor permite definir las operaciones que deseamos sin tener que modificar las clases que representan los diferentes tipos de nodos del AST. Siempre que queramos añadir una nueva operación basta con agregar un nuevo VisitanteConcreto. Cabe señalar que el patrón Visitor permite que una operación (por ejemplo la verificación de tipos) que se va a realizar sobre la estructura (el AST), sea una operación específica en el tipo del elemento (el nodo) sobre el que se está realizando, lo que nos evita tener que hacer un análisis de casos con base en el tipo de nodo actual sobre el cual estamos realizando la operación. En esta implementación específica, el analizador sintáctico construirá un AST en memoria y regresará como resultado un apuntador al nodo raíz del AST; dicho nodo sólo sabemos que es de tipo Node y no sabemos en específico de qué clase concreta de nodo se trata. Si en este momento queremos realizar una operación, como la verificación de tipos, un posible camino sería comenzar un análisis de casos para determinar el tipo de nodo concreto del que se trata y tener así la posibilidad de ejecutar la verificación de tipos correspondiente a ese nodo, e ir realizando recursivamente la verificación de tipos sobre los hijos haciendo un análisis de casos, en todo momento sobre el nodo actual.

La clase que juega el papel de Visitante es NodeVisitor, que declara un método visit para cada clase de ElementoConcreto, que en esta implementación corresponde a cada una de las clases que representan un nodo concreto del AST. El papel de VisitanteConcreto lo juegan, entre otras, la clase PrintVisitor , ya que, como mencionamos, puede ser que en algún momento requiramos construir una representación diferente del AST –como son una basada en texto o una gráfica con fines de depuración o didácticos–. Por eso hicimos uso del patrón Builder que nos permite ya sea construir un AST en memoria o en alguna otra representación, aunque por el momento sólo hemos implementado el constructor concreto que construye el AST en memoria. Podemos utilizar un VisitanteConcreto que, una vez construido el AST en memoria, lo imprima en pantalla, utilizando notación prefija al estilo LISP.

Es claro que también podríamos implementar un ConstructorConcreto que construya imprimiendo en pantalla, lo cual nos ahorraría recursos pues en lugar construir el AST en memoria y luego generar la salida en pantalla, éste simplemente generaría di-

rectamente la salida en pantalla; pero aquí, por el momento, hemos elegido la primera opción, porque, por un lado nos evita tener que implementar el ConstructorConcreto, y por el otro nos permite visualizar si el AST que se construyó en memoria es el que se esperaba. Asimismo nos permite ver, con un ejemplo sencillo, el uso y los beneficios del patrón Visitor. El papel de ElementoConcreto lo juegan cada una de las clases que representan los diferentes tipos nodos del AST y el papel de EstructuraObjetos lo juega la clase Node que, como dijimos, representa al AST. Así, si tenemos el siguiente programa de entrada válido de Python:

```
5+7
```

el AST que se construirá será un nodo Plus que tiene como hijo izquierdo un nodo Int que almacena internamente el valor 5; y tiene como hijo derecho un nodo Int que almacena internamente el valor 7, el cual, a su vez, será hijo único de un nodo Expr; éste representa una expresión, que a su vez será hijo único de un nodo SStmtList, que representa una lista de enunciados simples en la cual claro sólo hay un enunciado; a su vez, este nodo será hijo único de un nodo StmtList, que representa una lista de enunciados (ya sea simples o compuestos) y que será la raíz del árbol; por tanto, la salida generada por nuestro PrintVisitor será:

```
(StmtList(SStmtList(Expr(+(5)(7)))))
```

Como mencionamos, estamos utilizando notación prefija al estilo LISP para representar nuestro árbol, por lo que la parte más anidada (+(5)(7)) representa la parte del nodo Plus . Hemos elegido representar el nodo Plus con el carácter + y como estamos utilizando notación prefija, podemos ver el tipo de nodo como la operación, que es la que se escribe primero, seguido por sus operandos, que son, en este caso, nodos Int. Un nodo Int se representa por medio del entero que guarda internamente; de esta manera obtenemos (+(5)(7)), que a su vez se puede ver como el único operando de la operación Expr, es decir, como el único hijo del nodo Expr y así obtenemos (Expr(+(5)(7))) y lo restante se obtiene de manera similar.

A continuación se muestran las partes de las clases NodeVisitor y PrintVisitor que intervienen en nuestro ejemplo:

```
class NodeVisitor{
public:
   virtual void visit(IntNode*){}
   virtual void visit(PlusNode*){}
   virtual void visit(ExpressionNode*){}
   virtual void visit(SStmtListNode*){}
   virtual void visit(StmtListNode*){}
```

```
protected:
   NodeVisitor(){}
};

class PrintVisitor: public NodeVisitor{
public:
   virtual void visit(IntNode*);
   virtual void visit(PlusNode*);
   virtual void visit(ExpressionNode*);
   virtual void visit(SStmtListNode*);
   virtual void visit(StmtListNode*);
   PrintVisitor() {}
private:
  void printOp(std::string op,Node* n);
  void printLeaf(std::string s);
};

void PrintVisitor::visit(IntNode* n){
  cout << "(" << n->getNumber()->getInt() << ")";
}

void PrintVisitor::visit(PlusNode* n){
  printOp("+",n);
}

void PrintVisitor::visit(ExpressionNode* n){
  printOp("Expr",n);
}

void PrintVisitor::visit(SStmtListNode* n){
  printOp("SStmtList",n);
}

void PrintVisitor::visit(StmtListNode* n){
  printOp("StmtList",n);
}

void PrintVisitor::printOp(string op,Node* n){
  cout << "(" << op;
  NodeList* tmp = n->getCh();
  NodeList::iterator* it;
```

```
  for(it= tmp->begin(); (*it) !=*(tmp->end());++(*it)){
    (**it)->pfmPass(*this);
  }
  cout << ")";
  delete it;
}

void PrintVisitor::printLeaf(string s){
  cout << "(" << s << ")";
}
```

Comunicación entre flex y bison

Describiremos ahora como se da la comunicación entre el analizador léxico y el sintáctico en nuestro compilador.

Como mencionamos, la comunicación entre flex y bison por omisión se lleva a cabo mediante variables globales como yylval pero ésta no es la única variable con ese fin; también existe la variable yylloc que sirve para que el analizador léxico le transmita al sintáctico el lugar (dentro del código fuente) donde encontró un lexema dado. Como ya se sabe, hacer la comunicación mediante variables globales no es una buena idea, por lo que modificamos el comportamiento por omisión de bison y de flex para que la comunicación se lleve de manera distinta; concretamente, que la función yyparse invoque a la función yylex, pasándole como argumento un apuntador tanto a la variable yylval como a la variable yylloc para que, en su caso, yylex establezca los valores de éstas de manera adecuada. Como mencionamos en el capítulo 2, la clase Lexer es nuestra abstracción del analizador léxico, la que envuelve y protege a la función yylex que genera flex, por lo que nuestro analizador sintáctico se debe comunicar con el léxico a través de la interfaz provista Lexer. Desde el punto de vista de bison, debe llamar a la función yylex –por omisión no le pasa argumentos0–.

Sin embargo, si utilizamos la declaración define api pure entonces pedimos a bison que genere un analizador sintáctico reentrante puro, con lo que ahora se deberán pasar como argumentos un apuntador tanto de yylval como de yylloc; pero nosotros no utilizaremos esta opción, sino que utilizaremos la declaración skeleton "lalr1.c" que tiene bison, para que genere un analizador sintáctico en C++, siendo el tipo del valor semántico de los átomos yy::Parser::semantic_type, donde Parser es la clase que generará bison dentro del espacio de nombres yy y de la que nosotros podemos establecer explícitamente su nombre.

En este caso hemos utilizado la declaración define parser_class_name "Parser" con este fin, mientras que el tipo de la localidad de un átomo es: yy::Parser::location_type. Lo que a nosotros nos interesa es, entonces, que yylex tenga como parámetros un apunta-

dor yylval con tipo yy::Parser::semantic_type y un apuntador yylloc con tipo yy::Parser::location_type y regrese como resultado un valor de tipo yy::Parser::token_type, que es el tipo de la enumeración donde bison hace la asociación entre enteros y átomos; de esta forma es que se declara la función yylex en nuestra clase Lexer.

Pero ahora tenemos que siempre que yyparse necesite llamar a yylex lo que realmente haga es invocar la función yylex de nuestra clase Lexer, por lo que necesitamos un método para que nuestro analizador sintáctico tenga un apuntador a un objeto de la clase Lexer que se haya construido previamente, es decir, el analizado sintáctico necesita un parámetro donde le pasemos el apuntador al objeto de la clase Lexer, lo que logramos en bison por medio de la declaración parse-param{Lexer* lex}. Pero además, como dijimos que Parser jugará el papel de director del patrón Builder y necesita un apuntador a un ConstructorConcreto, también tenemos la siguiente declaración

```
parse-param{ASTBuilder* astb}
```

Ahora, siempre que yyparse invoque yylex, debido a que estamos utilizando el modo C++ de bison, éste le pasará como argumentos yylval y yylloc; pero como lo que queremos es que invoque a la función yylex de la clase Lexer –no que invoque directamente a yylex–, necesitamos una forma que, de manera transparente, cuando yyparse invoque a yylex lo que pase en realidad es que se invoque al método yylex de la clase Lexer. Para eso debemos añadir la declaración lex-param{Lexer* lex}, con lo que estamos pidiendo a bison que siempre que yyparse llame a yylex, además de pasar yylval y yylloc, pase como tercer argumento un apuntador a lex, que es un objeto de la clase Lexer y que es, en efecto, el apuntador que habíamos pasado previamente como parámetro al analizador sintáctico. Ahora tenemos que definir la función yylex dentro de nuestro analizador sintáctico de la siguiente manera:

```
token_type yylex(semantic_type* yylval,location_type* yylloc,
                 Lexer* lex){
    return lex->yylex(yylval,yylloc);
}
```

Con la anterior logramos que, desde el punto de vista del analizador sintáctico, siempre que yyparse llame a yylex, piense que lo está ejecutando directamente, mientras que lo que está pasando realmente es que se está ejecutando el método yylex de la clase Lexer, que a su vez es un envoltorio de la función yylex generada por flex. Además hemos agregado una nueva clase de nombre PParser dentro del espacio de nombres yy, que hereda de la clase Parser generada por bison y que será la interfaz de nuestro analizador léxico para las otras etapas del proceso de compilación. Esta clase cuenta con dos métodos, parse, que es el que realiza el análisis sintáctico, y getAst(), que regresa un apuntador a un objeto de la clase Node que será la raíz del AST.

Desde luego este método está pensado para que se invoque sólo después de que se haya invocado el método parse. Con lo anterior hemos logrado que ambos módulos, el

analizador léxico y el sintáctico, sean completamente modulares e independientes y que sus interfaces estén bien definidas y sean claras, la clase Lexer en el caso del analizador léxico y la clase PParser en el caso del analizador sintáctico.

3.6.1. Evaluación de nuestra implementación

Presentamos a continuación un conjunto de programas (el mismo que utilizamos para evaluar la implementación del analizador léxico y que repetimos aquí por comodidad) acompañados con la respectiva salida que genera la implementación de nuestro analizador sintáctico.

Hemos utilizado aquí nuestro VisitanteConcreto PrintVisitor, que recorre el árbol y lo va imprimiendo en notación prefija al estilo LISP, tal como lo describimos en la sección 3.6.

```
class Pila(object):
        def __init__(self, pila):
                self.pila = pila

        def push(self,elemento) :
                self.pila.append(elemento)

        def pop(self) :
                return self.pila.pop()

        def isEmpty(pila) :
                return (self.pila == [])

        def suma(pila) :
                elem1 = pila.pop()
                elem2 = pila.pop()
                suma = elem1 + elem2
                pila.push(suma)

        def swap(pila) :
                elem1 = pila.pop()
                elem2 = pila.pop()
                pila.push(elem1)
                pila.push(elem2)
```

Código 3.3: Primer programa de prueba (continúa en la siguiente página)

```
        def imprime(self) :
                self.pila.reverse()
                for i in self.pila:
                        if i==-1:
                                sys.stdout.write("+")
                        elif i == -2:
                                sys.stdout.write("s")
                        else:
                                print i
                self.pila.reverse()

entrada = '1'
p = Pila([])
while entrada != 'x' :
        sys.stdout.write("#")
        entrada = sys.stdin.readline()
        entrada = entrada[:-1]
        if entrada == '+':
                p.push(-1)
        elif entrada == 's':
                p.push(-2)
        elif entrada == 'e':
                elem = p.pop()
                if elem == -1:
                        p.suma()
                elif elem == -2:
                        p.swap()
                else:
                        p.push(elem)
                        sys.stdout.write("No se puede evaluar un
                           entero")
        elif entrada == 'd':
                p.imprime()
        elif entrada == 'x':
                break;
        else:
                p.push(int(entrada));
```

Código 3.3: Primer programa de prueba

```
(StmtList(Class(CName(Pila))(CBList(object))(CBody(Block(FuncDef(FName
(__init__))(FParamList(FSParamList(FSParam(self))(FSParam(pila)))
(Null)(Null))(FBody(Block(SStmtList(Expr(Assign(TargetList(AttribRef
(self)(pila)))(AValue(pila)))))))(Null))(FuncDef(FName(push))
(FParamList(FSParamList(FSParam(self))(FSParam(elemento)))(Null)
(Null))(FBody(Block(SStmtList(Expr(Call(AttribRef(AttribRef(self)
(pila))(append))(CArgList(CSArgList(CSArg(elemento)))(Null)
(Null)))))))(Null))(FuncDef(FName(pop))(FParamList(FSParamList
(FSParam(self)))(Null)(Null))(FBody(Block(SStmtList(Return(RetValue
(Call(AttribRef(AttribRef(self)(pila))(pop))(Null)))))))(Null))
(FuncDef(FName(isEmpty))(FParamList(FSParamList(FSParam
(pila)))(Null)(Null))(FBody(Block(SStmtList(Return(RetValue
(Compare(AttribRef(self)(pila))(CompList(CPair(EqualC)(List)))))))))
(Null))(FuncDef(FName(suma))(FParamList(FSParamList(FSParam
(pila)))(Null)(Null))(FBody(Block(SStmtList(Expr(Assign(TargetList
(elem1))(AValue(Call(AttribRef(pila)(pop))(Null)))))(SStmtList(Expr
(Assign(TargetList(elem2))(AValue(Call(AttribRef(pila)(pop))
(Null)))))(SStmtList(Expr(Assign(TargetList(suma))(AValue(+
(elem1)(elem2)))))(SStmtList(Expr(Call(AttribRef(pila)(push))
(CArgList(CSArgList(CSArg(suma)))(Null)(Null)))))))(Null))(FuncDef
(FName(swap))(FParamList(FSParamList(FSParam(pila)))(Null)(Null))
(FBody(Block(SStmtList(Expr(Assign(TargetList(elem1))(AValue(Call
(AttribRef(pila)(pop))(Null)))))(SStmtList(Expr(Assign(TargetList
(elem2))(AValue(Call(AttribRef(pila)(pop))(Null)))))(SStmtList(Expr
(Call(AttribRef(pila)(push))(CArgList(CSArgList(CSArg(elem1)))(Null)
(Null)))))(SStmtList(Expr(Call(AttribRef(pila)(push))(CArgList
(CSArgList(CSArg(elem2)))(Null)(Null)))))))(Null))(FuncDef(FName
(imprime))(FParamList(FSParamList(FSParam(self)))(Null)(Null))
(FBody(Block(SStmtList(Expr(Call(AttribRef(AttribRef(self)(pila))
(reverse))(Null))))(For(FTarget(i))(FIter(AttribRef(self)(pila)))
(FBody(Block(IF(IFGuard(Compare(i)(CompList(CPair(EqualC)
(-(1))))))(IFBody(Block(SStmtList(Expr(Call(AttribRef(AttribRef(sys)
(stdout))(write))(CArgList(CSArgList(CSArg(+)))(Null)(Null)))))))
(Else(IF(IFGuard(Compare(i)(CompList(CPair(EqualC)(-(2))))))(IFBody
(Block(SStmtList(Expr(Call(AttribRef(AttribRef(sys)(stdout))(write))
(CArgList(CSArgList(CSArg(s)))(Null)(Null)))))))(Else(Block(SStmtList
(PrintNl(PrintOutput(Null))(PrintList(i)))))))))(Null))(SStmtList
(Expr(Call(AttribRef(AttribRef(self)(pila))(reverse))(Null))))))
(Null)))))(SStmtList(Expr(Assign(TargetList(entrada))(AValue(1))))))
```

Salida del analizador sintáctico correspondiente al Código 3.3 (continúa en la siguiente página)

```
(SStmtList(Expr(Assign(TargetList(p))(AValue(Call(Pila)(CArgList
(CSArgList(CSArg(List)))(Null)(Null)))))))(While(WGuard(Compare
(entrada)(CompList(CPair(NEqual)(x)))))(WBody(Block(SStmtList(Expr
(Call(AttribRef(AttribRef(sys)(stdout))(write))(CArgList(CSArgList
(CSArg(#)))(Null)(Null)))))(SStmtList(Expr(Assign(TargetList(entrada))
(AValue(Call(AttribRef(AttribRef(sys)(stdin))(readline))(Null))))))
(SStmtList(Expr(Assign(TargetList(entrada))(AValue(SubscSlice(entrada)
(Slice(Null)(-(1))))))))(IF(IFGuard(Compare(entrada)(CompList(CPair
(EqualC)(+)))))(IFBody(Block(SStmtList(Expr(Call(AttribRef(p)(push))
(CArgList(CSArgList(CSArg(-(1))))(Null)(Null)))))))(Else(IF(IFGuard
(Compare(entrada)(CompList(CPair(EqualC)(s)))))(IFBody(Block
(SStmtList(Expr(Call(AttribRef(p)(push))(CArgList(CSArgList
(CSArg(-(2))))(Null)(Null)))))))(Else(IF(IFGuard(Compare(entrada)
(CompList(CPair(EqualC)(e)))))(IFBody(Block(SStmtList(Expr(Assign
(TargetList(elem))(AValue(Call(AttribRef(p)(pop))(Null))))))(IF
(IFGuard(Compare(elem)(CompList(CPair(EqualC)(-(1))))))(IFBody(Block
(SStmtList(Expr(Call(AttribRef(p)(suma))(Null))))))(Else(IF(IFGuard
(Compare(elem)(CompList(CPair(EqualC)(-(2))))))(IFBody(Block(SStmtList
(Expr(Call(AttribRef(p)(swap))(Null))))))(Else(Block(SStmtList(Expr
(Call(AttribRef(p)(push))(CArgList(CSArgList(CSArg(elem)))(Null)
(Null)))))(SStmtList(Expr(Call(AttribRef(AttribRef(sys)(stdout))
(write))(CArgList(CSArgList(CSArg(No se puede evaluar un entero)))
(Null)(Null))))))))))))))(Else(IF(IFGuard(Compare(entrada)(CompList
(CPair(EqualC)(d)))))(IFBody(Block(SStmtList(Expr(Call(AttribRef(p)
(imprime))(Null))))))(Else(IF(IFGuard(Compare(entrada)(CompList(CPair
(EqualC)(x)))))(IFBody(Block(SStmtList)))(Else(Block(SStmtList(Expr
(Call(AttribRef(p)(push))(CArgList(CSArgList(CSArg(Call(int)(CArgList
(CSArgList(CSArg(entrada)))(Null)(Null)))))(Null)
(Null))))))))))))))))))))(Null)))
```

Salida del analizador sintáctico correspondiente al Código 3.3

```
'''
Created on 15/02/2011

Clase que implementa la funcionalidad basica
de un interprete de pila

@author: Ricchy Alain Perez Chevanier
'''
```

Código 3.4: Segundo programa de prueba (continúa en la siguiente página)

```
class Interprete:

    '''
    Constructor
    '''
    def __init__(self):
        self.stack = []

    '''
    Inserta una cadena en el stack que representa al inter-
    prete o realiza acciones segun la cadena proporcionada
    al interprete. Regresa un boolean que indica si el inter
    prete debe de continuar recibiendo cadenas.
    '''
    def insertWord(self, word):
        if word=="x" :
            return False
        elif word=="d" :
            for x in self.stack:
                print x
        elif word=="e" :
            op = self.stack.pop(0)
            if op== "s" :
                op1 = self.stack.pop(0)
                op2 = self.stack.pop(0)

                self.stack = [op2, op1] + self.stack
            elif op=="+" :
                op1 = int(self.stack.pop(0))
                op2 = int(self.stack.pop(0))
                res = op1 + op2
                self.stack = [res] + self.stack
            else :
                self.stack = [op] + self.stack
        else :
            self.stack = [word] + self.stack
        return True
```

Código 3.4: Segundo programa de prueba

```
(StmtList(SStmtList(Expr(
Created on 15/02/2011

Clase que implementa la funcionalidad basica
de un interprete de pila

@author: Ricchy Alain Perez Chevanier
)))(Class(CName(Interprete))(Null)(CBody(Block(SStmtList(Expr(
    Constructor
    )))(FuncDef(FName(__init__))(FParamList(FSParamList(FSParam
(self)))(Null)(Null))(FBody(Block(SStmtList(Expr(Assign(TargetList(
   AttribRef(self)(stack)))(AValue(List)))))))(Null))(SStmtList(Expr(
    Inserta una cadena en el stack que representa al inter-
    prete o realiza acciones segun la cadena proporcionada
    al interprete. Regresa un boolean que indica si el inter
    prete debe de continuar recibiendo cadenas.
    )))(FuncDef(FName(insertWord))(FParamList(FSParamList(FSParam
(self))(FSParam(word)))(Null)(Null))(FBody(Block(IF(IFGuard(Compare
(word)(CompList(CPair(EqualC)(x)))))(IFBody(Block(SStmtList(Return
(RetValue(False))))))(Else(IF(IFGuard(Compare(word)(CompList(CPair
(EqualC)(d)))))(IFBody(Block(For(FTarget(x))(FIter(AttribRef(self)
(stack)))(FBody(Block(SStmtList(PrintNl(PrintOutput(Null))
(PrintList(x))))))(Null))))(Else(IF(IFGuard(Compare(word)(CompList
(CPair(EqualC)(e)))))(IFBody(Block(SStmtList(Expr(Assign(TargetList
(op))(AValue(Call(AttribRef(AttribRef(self)(stack))(pop))(CArgList
(CSArgList(CSArg(0)))(Null)(Null)))))))(IF(IFGuard(Compare(op)
(CompList(CPair(EqualC)(s)))))(IFBody(Block(SStmtList(Expr(Assign
(TargetList(op1))(AValue(Call(AttribRef(AttribRef(self)(stack))(pop))
(CArgList(CSArgList(CSArg(0)))(Null)(Null)))))))(SStmtList(Expr(Assign
(TargetList(op2))(AValue(Call(AttribRef(AttribRef(self)(stack))(pop))
(CArgList(CSArgList(CSArg(0)))(Null)(Null)))))))(SStmtList(Expr(Assign
(TargetList(AttribRef(self)(stack)))(AValue(+(List(op2)(op1))
(AttribRef(self)(stack)))))))))(Else(IF(IFGuard(Compare(op)(CompList
(CPair(EqualC)(+)))))(IFBody(Block(SStmtList(Expr(Assign(TargetList
(op1))(AValue(Call(int)(CArgList(CSArgList(CSArg(Call(AttribRef
(AttribRef(self)(stack))(pop))(CArgList(CSArgList(CSArg(0)))(Null)
(Null)))))(Null)(Null)))))))(SStmtList(Expr(Assign(TargetList(op2))
(AValue(Call(int)(CArgList(CSArgList(CSArg(Call(AttribRef(AttribRef
(self)(stack))(pop))(CArgList(CSArgList(CSArg(0)))(Null)(Null)))))
```

Salida del analizador sintáctico correspondiente al Código 3.4 (countinúa en la siguiente página)

```
(Null)(Null)))))))(SStmtList(Expr(Assign(TargetList(res))(AValue
(+(op1)(op2))))))(SStmtList(Expr(Assign(TargetList(AttribRef(self)
(stack)))(AValue(+(List(res))(AttribRef(self)(stack)))))))))(Else
(Block(SStmtList(Expr(Assign(TargetList(AttribRef(self)(stack)))
(AValue(+(List(op))(AttribRef(self)(stack))))))))))))))(Else(Block
(SStmtList(Expr(Assign(TargetList(AttribRef(self)(stack)))(AValue
(+(List(word))(AttribRef(self)(stack))))))))))))))(SStmtList
(Return(RetValue(True))))))(Null))))))
```

Salida del analizador sintáctico correspondiente al Código 3.4

```
class Token:
    """Esta clase representa un elemento de StackMachine"""

    def __init__(self, val):
        self.value = val

    def __str__(self):
        return str(self.value)

class Operator(Token):
    """Token que puede servir como funcion"""
    def __init__(self, val, representation):
        Token.__init__(self, val)
        self.representation = representation

    def __call__(self,stack):
        self.value(stack)

    def __str__(self):
        return self.representation

class StackMachine:
    """Esta clase representa una calculadora de pila"""
    def __init__(self):
        self.stack = []

    def push(self, token):
        """Agrega un elemento a la parte superior de la pila"""
        self.stack.append(token)
```

Código 3.5: Tercer programa de prueba (continúa en la siguiente página)

```
    def pop(self):
        """Quita y regresa el elemento superior de la pila"""
        return self.stack.pop()

    def execute(self):
        """Ejecuta el elemento superior"""
        if len(self.stack) == 0:
            return

        top = self.pop()
        if callable(top):
            top(self)
        else:
            self.push(top)

    def __str__(self):
        concatenator = lambda accum, x: str(x) + '\n' + accum
        return reduce(concatenator, self.stack, '')


class CommandLineInterpreter:
    """
    Clase que representa una linea de comandos que maneja un
    StackMachine.

    """
    def __init__(self):
        """Crea una linea de comandos con un stack vacio"""
        self.calc = StackMachine()
        self.directives = tables.directives
        self.operators = tables.operators

    def start(self):
        """Entra en un loop a la espera de input de usuario"""
        while True:
            try:
                input_string = raw_input('# ')
                if input_string in self.directives:
                    self.directives[input_string](self.calc)
```

Código 3.5: Tercer programa de prueba (continúa en la siguiente página)

```
                elif input_string in self.operators:
                    token = Operator(self.operators[input_string],
                        input_string)
                    self.calc.push(token)
                else:
                    try:
                        self.calc.push(Token(int(input_string)))
                    except ValueError:
                        print 'Entrada invalida'
            except EOFError:
                # Si el cacacter es EOF (^D en la consola) imprimimos
                # un blanco y salimos del ciclo
                print
                break

if __name__ == '__main__':
    cli = CommandLineInterpreter()
    cli.start()
```

Código 3.5: Tercer programa de prueba

```
(StmtList(Class(CName(Token))(Null)(CBody(Block(SStmtList(Expr
(Esta clase representa un elemento de StackMachine)))(FuncDef
(FName(__init__))(FParamList(FSParamList(FSParam(self))(FSParam(val)))
(Null)(Null))(FBody(Block(SStmtList(Expr(Assign(TargetList(AttribRef
(self)(value)))(AValue(val)))))))(Null))(FuncDef(FName(__str__))
(FParamList(FSParamList(FSParam(self)))(Null)(Null))(FBody(Block
(SStmtList(Return(RetValue(Call(str)(CArgList(CSArgList(CSArg
(AttribRef(self)(value))))(Null)(Null)))))))))(Null)))))(Class(CName
(Operator))(CBList(Token))(CBody(Block(SStmtList(Expr
(Token que puede servir como funcion)))(FuncDef(FName(__init__))
(FParamList(FSParamList(FSParam(self))(FSParam(val))(FSParam
(representation)))(Null)(Null))(FBody(Block(SStmtList(Expr(Call
(AttribRef(Token)(__init__))(CArgList(CSArgList(CSArg(self))(CSArg
(val)))(Null)(Null)))))(SStmtList(Expr(Assign(TargetList(AttribRef
(self)(representation)))(AValue(representation)))))))(Null))(FuncDef
(FName(__call__))(FParamList(FSParamList(FSParam(self))(FSParam
(stack)))(Null)(Null))(FBody(Block(SStmtList(Expr(Call(AttribRef(self)
(value))(CArgList(CSArgList(CSArg(stack)))(Null)(Null)))))))(Null))
(FuncDef(FName(__str__))(FParamList(FSParamList(FSParam(self)))
```

Salida del analizador sintáctico correspondiente al Código 3.5 (continúa en la siguiente página)

```
(Null)(Null))(FBody(Block(SStmtList(Return(RetValue(AttribRef(self)
(representation)))))))(Null)))))(Class(CName(StackMachine))(Null)
(CBody(Block(SStmtList(Expr
(Esta clase representa una calculadora de pila)))(FuncDef(FName
(__init__))(FParamList(FSParamList(FSParam(self)))(Null)(Null))
(FBody(Block(SStmtList(Expr(Assign(TargetList(AttribRef(self)
(stack)))(AValue(List)))))))(Null))(FuncDef(FName(push))(FParamList
(FSParamList(FSParam(self))(FSParam(token)))(Null)(Null))(FBody(Block
(SStmtList(Expr(Agrega un elemento a la parte superior de la pila)))
(SStmtList(Expr(Call(AttribRef(AttribRef(self)(stack))(append))
(CArgList(CSArgList(CSArg(token)))(Null)(Null)))))))(Null))(FuncDef
(FName(pop))(FParamList(FSParamList(FSParam(self)))(Null)(Null))
(FBody(Block(SStmtList(Expr
(Quita y regresa el elemento superior de la pila)))(SStmtList(Return
(RetValue(Call(AttribRef(AttribRef(self)(stack))(pop))(Null))))))
(Null))(FuncDef(FName(execute))(FParamList(FSParamList(FSParam
(self)))(Null)(Null))(FBody(Block(SStmtList(Expr
(Ejecuta el elemento superior)))(IF(IFGuard(Compare(Call(len)(CArgList
(CSArgList(CSArg(AttribRef(self)(stack))))(Null)(Null)))(CompList
(CPair(EqualC)(0)))))(IFBody(Block(SStmtList(Return(Null)))))(Null))
(SStmtList(Expr(Assign(TargetList(top))(AValue(Call(AttribRef(self)
(pop))(Null))))))(IF(IFGuard(Call(callable)(CArgList(CSArgList(CSArg
(top)))(Null)(Null))))(IFBody(Block(SStmtList(Expr(Call(top)(CArgList
(CSArgList(CSArg(self)))(Null)(Null)))))))(Else(Block(SStmtList(Expr
(Call(AttribRef(self)(push))(CArgList(CSArgList(CSArg(top)))(Null)
(Null)))))))))(Null))(FuncDef(FName(__str__))(FParamList(FSParamList
(FSParam(self)))(Null)(Null))(FBody(Block(SStmtList(Expr(Assign
(TargetList(concatenator))(AValue(Lambda(FParamList(FSParamList
(FSParam(accum))(FSParam(x)))(Null)(Null))(LBody(+(+(Call(str)
(CArgList(CSArgList(CSArg(x)))(Null)(Null)))(
))(accum)))))))))(SStmtList(Return(RetValue(Call(reduce)(CArgList
(CSArgList(CSArg(concatenator))(CSArg(AttribRef(self)(stack)))
(CSArg()))(Null)(Null)))))))))(Null)))))(Class(CName
(CommandLineInterpreter))(Null)(CBody(Block(SStmtList(Expr(
    Clase que representa una linea de comandos que maneja un
    StackMachine.

    )))(FuncDef(FName(__init__))(FParamList(FSParamList(FSParam
(self)))(Null)(Null))(FBody(Block(SStmtList(Expr
```

Salida del analizador sintáctico correspondiente al Código 3.5 (continúa en la siguiente página)

```
(Crea una linea de comandos con un stack vacio)))(SStmtList(Expr
(Assign(TargetList(AttribRef(self)(calc)))(AValue(Call(StackMachine)
(Null))))))(SStmtList(Expr(Assign(TargetList(AttribRef(self)
(directives)))(AValue(AttribRef(tables)(directives))))))(SStmtList
(Expr(Assign(TargetList(AttribRef(self)(operators)))(AValue
(AttribRef(tables)(operators)))))))(Null))(FuncDef(FName(start))
(FParamList(FSParamList(FSParam(self)))(Null)(Null))(FBody(Block
(SStmtList(Expr(Entra en un loop a la espera de input de usuario)))
(While(WGuard(True))(WBody(Block(TryExcept(TEBody(Block(SStmtList
(Expr(Assign(TargetList(input_string))(AValue(Call(raw_input)(CArgList
(CSArgList(CSArg(# )))(Null)(Null)))))))(IF(IFGuard(Compare
(input_string)(CompList(CPair(In)(AttribRef(self)(directives))))))
(IFBody(Block(SStmtList(Expr(Call(SubscIndex(AttribRef(self)
(directives))(Index(input_string)))(CArgList(CSArgList(CSArg(AttribRef
(self)(calc))))(Null)(Null)))))))(Else(IF(IFGuard(Compare
(input_string)(CompList(CPair(In)(AttribRef(self)(operators))))))
(IFBody(Block(SStmtList(Expr(Assign(TargetList(token))(AValue(Call
(Operator)(CArgList(CSArgList(CSArg(SubscIndex(AttribRef(self)
(operators))(Index(input_string))))(CSArg(input_string)))(Null)
(Null)))))))(SStmtList(Expr(Call(AttribRef(AttribRef(self)(calc))
(push))
(CArgList(CSArgList(CSArg(token)))(Null)(Null)))))))(Else(Block
(TryExcept(TEBody(Block(SStmtList(Expr(Call(AttribRef(AttribRef(self)
(calc))(push))(CArgList(CSArgList(CSArg(Call(Token)(CArgList(CSArgList
(CSArg(Call(int)(CArgList(CSArgList(CSArg(input_string)))(Null)
(Null)))))(Null)(Null)))))(Null)(Null)))))))(TEExceptList(TEExcept
(TEEType(ValueError))(Null)(TEEBody(Block(SStmtList(PrintNl
(PrintOutput(Null))(PrintList(Entrada invalida)))))))(Null)))))))))
(TEExceptList(TEExcept(TEEType(EOFError))(Null)(TEEBody(Block
(SStmtList(PrintNl(PrintOutput(Null))(Null)))(SStmtList)))))(Null))))
(Null))))(Null)))))(IF(IFGuard(Compare(__name__)(CompList(CPair
(EqualC)(__main__)))))(IFBody(Block(SStmtList(Expr(Assign(TargetList
(cli))(AValue(Call(CommandLineInterpreter)(Null))))))(SStmtList(Expr
(Call(AttribRef(cli)(start))(Null))))))(Null)))
```

Salida del analizador sintáctico correspondiente al Código 3.5

Una vez más podemos notar que contar con una representación basada en texto en las etapas del compilador hace que la depuración de éstas sea más simple.

En este caso para cada uno de los programas que se alimentaron como entrada se obtuvo la salida correcta y que es desde luego la que se esperaba que generara nuestra implementación del analizador sintáctico.

Con esto damos por concluido el capítulo del análisis sintáctico.

CAPÍTULO 4

Análisis semántico

Conocimientos supuestos

Para la etapa del análisis semántico estamos suponiendo que el lector conoce lo siguiente:

1. Definiciones de:
   - Estructura de bloques de un programa.
   - Alcance léxico de nombres de variables.
   - Alcance dinámico de nombres de variables.
   - Gráficas, gráficas dirigidas.
2. Conocimientos de:
   - Complejidad de algoritmos.
   - Función de Ackerman.
3. Estructuras de datos:
   - Funciones de dispersión (*hash*).
   - Árboles.
   - Patrones de diseño.

Estos temas se incluyen en cursos de programación y estructuras de datos.

4.1. Introducción

Conceptualmente el proceso de compilación se realiza en dos etapas:

- Análisis, que divide el código fuente en piezas e impone una estructura gramatical sobre ellas.

- Síntesis, que determina el significado del código fuente y construye un programa objetivo con el mismo significado.

Mencionamos en el capítulo anterior que la definición de un lenguaje de programación debe incluir la especificación tanto de su sintaxis (estructura) como de su semántica (significado). La especificación sintáctica, como ya vimos, la haremos usando gramáticas libres del contexto, por lo que pasamos a uno de los temas de este capítulo, la especificación de la semántica.

La semántica de un lenguaje de programación comúnmente se divide en:

- Semántica estática (o en tiempo de compilación).
- Semántica dinámica (o en tiempo de ejecución).

Cabe aclarar que, en este contexto, el concepto estático es todo aquello que se puede realizar sin necesidad de la ejecución del programa, es decir, previo a su ejecución en el sentido más estricto de este último término, pues bajo esta última caracterización nos estamos refiriendo la mayor parte de las veces al tiempo de compilación, pero no está restringido únicamente a él, ya que, por ejemplo, un intérprete puede realizar análisis estático antes de la estricta ejecución del programa. El concepto dinámico, en cambio, concuerda con precisión con el momento de la ejecución del programa.

La semántica estática de un lenguaje provee un conjunto de reglas (las cuales se pueden verificar estáticamente) que especifican cuáles de los programas procesados por el analizador sintáctico son realmente válidos. Ejemplos típicos de estas reglas son que se declare un identificador antes de que se utilice, que operadores y operandos tengan tipos compatibles y que una función se llame con el número correcto de argumentos. En realidad varias de las reglas impuestas por la semántica estática pertenecen al campo del análisis sintáctico. Por ejemplo, el pedir que un identificador se declare antes de utilizarse de manera abstracta implica analizar sintácticamente el lenguaje $L = \{wxw \mid w, x \in (a|b)^*\}$, que es un lenguaje dependiente del contexto. El problema reside precisamente en que, como vimos, el analizador sintáctico trabaja, en el mejor de los casos,[1] con gramáticas libres del contexto que no incluyen la especificación de este tipo de reglas. Pedir que estas reglas sean verificadas por el analizador sintáctico implicaría que éste trabajara con gramáticas dependientes del contexto, donde, como mencionamos en el capítulo anterior, todos los algoritmos de análisis que se conocen para este tipo de gramáticas toman tiempo exponencial. Es debido a esto que en la práctica se libera al analizador sintáctico de la verificación de estas reglas dependientes del contexto y se tienen que verificar como semántica estática en el analizador semántico. Por ello autores como Cooper en [TC11]

[1] Aquí estamos utilizando la expresión "el mejor de los casos" refiriéndonos que desde este punto de vista mientras más generales sean las gramáticas con las que trabaje el analizador sintáctico es mejor, pero no se debe confundir con el objetivo que el analizador sea eficiente.

lo denominan "análisis dependiente del contexto", aunque clásicamente en la literatura se conoce como "análisis semántico".

Siendo precisos, el análisis dependiente del contexto, entendido como la verficación de características sintácticas dependientes del contexto, es un subconjunto del análisis semántico; esto es más claro en lenguajes implícita y estáticamente tipados como ML, ya que si bien ML cuenta con características sintácticas dependientes del contexto que podemos decir verifica su *analizador dependiente del contexto*, al analizador semántico no le basta sólo con verificar estas características, sino que además debe realizar un algoritmo de inferencia de tipos que permita determinar los tipos de cada una de las expresiones del programa, y posteriormente verificar que sean consistentes, con lo que podemos observar claramente que el conjunto de reglas del análisis dependiente del contexto es un un subconjunto del conjunto de reglas del análisis semántico. En realidad, al implementar un compilador se sigue la convención de las especificaciones de lenguajes de programación y no se suele hacer explícita la distinción entre análisis sintáctico dependiente del contexto y aquella parte del análisis semántico que no lo es, sino simplemente se engloba en un conjunto todo el análisis semántico. Lo que sucede es que en el caso particular de los lenguajes explícita y estáticamente tipados, el análisis dependiente del contexto coincide con el análisis semántico, ya que el uso de los tipos es parte de la sintaxis; son este tipo de lenguajes como Fortran, C o Java los que se suelen utilizar como ejemplo en la mayoría de los libros de texto de compiladores.

La semántica estática se puede especificar formal o informalmente. Lo que se ha estudiado más ampliamente para especificar de manera formal la semántica estática son las gramáticas con atributos.

La semántica dinámica se utiliza para especificar qué debe calcular un determinado programa; ésta, al igual que la semántica estática, se puede especificar de manera formal o informal. Los principales métodos para definir formalmente la semántica dinámica son los siguientes:

- Semántica operacional
- Semántica denotacional
- Semántica axiomática

El analizador semántico de un compilador para un lenguaje estática y explícitamente tipado generalmente realiza, entre otras, las siguientes tareas:

- Verifica que una variable se haya declarado antes de ser utilizada.
- Verifica que cuando se llame una función o método éste haya sido definido y que se le pase el número correcto de argumentos.
- Verifica que se cumplan las reglas de tipos.

Todo lo anterior lo hace el analizador semántico con la ayuda de una tabla de símbolos que construye y consulta según sea conveniente. En otras palabras, el analizador semántico de este tipo de lenguajes verifica que se cumpla con la semántica estática del lenguaje.

De esta forma los analizadores léxico, sintáctico y semántico realizan el *análisis* y las siguientes etapas serán las encargadas de realizar la *síntesis*.

Pongamos un ejemplo para ilustrar la diferencia entre semántica estática y dinámica. Supongamos que en Java tenemos un tipo polimórfico, es decir, contamos con dos clases una de nombre Padre y la otra de nombre Hijo que hereda de Padre; supongamos que ambas tienen un método saluda con el mismo encabezado; ahora declaramos una variable p de tipo Padre, creamos un objeto de la clase Hijo y se la asignamos a p; cuando invoquemos al método saluda sobre p ¿qué método se ejecutará? ¿el de la clase Padre o el de la clase Hijo? Lo anterior se muestra en el siguiente código:

```
class Padre{
    public void saluda(){
        System.out.println("Hola soy el Padre");
    }
}

class Hijo extends Padre{
    public void saluda(){
        System.out.println("Hola soy el hijo");
    }

    public void otro(){
        System.out.println("Otro");
    }

    public static void main(String[] args){
        Padre p = new Hijo();
        p.saluda();
        //p.otro(); //Error la clase Padre no tiene un metodo con
        //nombre otro
    }
}
```

Código 4.1: Ejemplo que ilustra la diferencia entre semántica estática y dinámica

Según la definición de Java se ejecutará el método de la clase Hijo, ¿pero esta regla corresponde a la semántica estática o a la dinámica? La respuesta es a la dinámica y veamos por qué. Con base en el código fuente, lo que el analizador semántico puede verificar estáticamente es que cuando p invoca al método saluda, la clase Padre (que es

el tipo con el que se declaró p) cuente con un método de nombre saluda que no tenga argumentos, lo que resulta verdadero, y es parte de la semántica estática. En este punto el analizador semántico puede asegurar que se puede ejecutar el método saluda de la clase Padre. Sin embargo, en principio no se puede determinar, con base en el código fuente, si el objeto asignado a la variable p es un objeto de la clase Padre o de alguno de sus descendientes, ya que esto únicamente lo sabrá el sistema en tiempo de ejecución cuando se ejecute el programa. Si la definición de Java hubiera dictado que se ejecute el método de clase con la que se declaró el tipo de la variable p (Padre) entonces la regla correspondería a la semántica estática, pues estáticamente el analizador semántico está en la posibilidad de dictar que se ejecute el método de la clase Padre, pero como no es así, Java, en su semántica dinámica, pide que se ejecute el método de la clase que realmente es el tipo concreto del objeto en tiempo de ejecución. Por esto es que se dice que Java tiene *despacho dinámico*. A diferencia de Java, C++ por omisión tiene *despacho estático*, es decir, en C++ se ejecutará el método de la clase con el que se declaró el tipo de la variable, aunque también cuenta con la opción de realizar despacho dinámico, siempre que se agregue la palabra reservada virtual al encabezado del método especificado.

Desde el punto de vista del escritor de compiladores siempre se quisiera tener la mayor información posible en tiempo de compilación. Por ejemplo, en el caso del despacho dinámico de Java, el sistema en tiempo de ejecución tiene que determinar, durante la ejecución del programa, qué método se va a invocar en realidad, lo que hace que se degrade el desempeño del programa, aumentando su tiempo de ejecución. Por ello los compiladores optimizadores siempre buscan una manera de tener más información en tiempo de compilación sobre la ejecución del programa, por lo que se han desarrollado con este fin una amplia gama de técnicas de análisis de flujo de control y de flujo de datos para, con base en estos análisis, poder determinar en tiempo de compilación, en algunos casos, qué método se debe ejecutar en Java, con lo que ganamos un mejor desempeño en la ejecución del programa.

4.2. Sistemas de tipos

La tarea principal del analizador semántico es realizar verificación de tipos,[2] por lo que pareciera que estamos dando por concedido que todo lenguaje cuenta con un sistema de tipos. Pero ¿para qué nos sirve que un lenguaje cuente con un sistema de tipos? El propósito fundamental de un sistema de tipos, tal como se indica en [Car04], es prevenir la existencia de errores de ejecución durante la ejecución de un programa.

Sin embargo la adopción de este propósito como tarea principal es relativamente nueva. Desde el punto de vista del escritor de compiladores, la principal ventaja de que

[2]Siempre que éste cuente con la información de tipos necesaria.

un lenguaje de programación cuente con sistema de tipos[3] es que permite al compilador obtener la información necesaria para producir código eficiente. Fue este objetivo lo que motivó que los primeros lenguajes de programación, como es el caso de Fortran, contaran con un sistema de tipos, pues tal como se indica en [Pie02] esto permitía distinguir entre expresiones aritméticas enteras y expresiones aritméticas reales y así mejorar la eficiencia de cálculos numéricos.

Al contar con información de tipos en tiempo de compilación, un compilador puede calcular la cantidad de memoria precisa que utilizarán cada uno de los datos de un programa y seleccionar el código específico más conveniente para manipularlos, lo que da como resultado un código objeto más eficiente. Por ejemplo, cuando se realiza una suma y se conoce (debido al sistema de tipos) que ambos operandos son enteros, se puede determinar la cantidad de memoria que ocuparán ambos operandos y sabemos que se debe emitir el código correspondiente a la suma de enteros (y no a la suma de reales), es decir, se generaría una instrucción suma de aritmética entera, que lleva menos tiempo que la instrucción suma de aritmética punto flotante.

Desafortunadamente en lenguajes con tipado dinámico tal información no está disponible en tiempo de compilación.

A continuación damos algunas definiciones formales respecto al sistema de tipos de un lenguaje de programación.

Idealmente un sistema de tipos formal debe ser parte de la definición de todos los lenguajes de programación tipados.

Definición 4.1. Un *tipo* es un conjunto de valores que pueden tomar las variables de un programa, y las operaciones permitidas en ese tipo. Algunas veces se especifica también su representación interna.

Definición 4.2. A los lenguajes en los que se puede asignar tipos (no triviales) a sus variables se denominan *lenguajes tipados*.

Definición 4.3. Un lenguaje tipado donde los tipos forman parte de la sintaxis se denomina *explícitamente tipado*.

Definición 4.4. Un lenguaje tipado donde los tipos no forman parte de la sintaxis se denomina *implícitamente tipado*.

Definición 4.5. Un *sistema de tipos* es un componente de un lenguaje tipado que lleva el seguimiento de los tipos de las variables y en general de los tipos de todas las expresiones en un programa.

Definición 4.6. A los lenguajes que no restringen el rango de las variables se denominan *lenguajes no tipados*. Estos lenguajes no tienen tipos o, equivalentemente, tienen un único tipo universal que contiene todos los valores.

[3]Suponiendo que la información de tipos esté disponible en tiempo de compilación.

Definición 4.7. Se dice que un fragmento de un programa tiene *buen comportamiento* si no permite que sucedan errores que no pudieron ser detectados en la semántica estática.

Definición 4.8. Un lenguaje donde todos los programas legales tienen buen comportamiento se llama *fuertemente verificado*.

Definición 4.9. Un lenguaje *débilmente verificado* es aquel en el que se detectan sólo algunas operaciones inseguras estáticamente mientras que otras no son detectadas.

Definición 4.10. Un lenguaje en el cual a toda expresión se le puede asignar un tipo no ambiguo se le llama *fuertemente tipado*.

Definición 4.11. Un lenguaje en el cual a algunas de sus expresiones se les puede asignar un tipo no ambiguo se le llama *débilmente tipado*.

Definición 4.12. Un lenguaje tipado puede imponer buen comportamiento mediante la realización de verificaciones estáticas. A un lenguaje que así lo hace se le llama *estáticamente verificado*.

Definición 4.13. Un lenguaje que impone buen comportamiento mediante la realización de verificaciones dinámicas se le llama *dinámicamente verificado*.

Definición 4.14. Un lenguaje en el cual el tipo de toda expresión puede ser determinado estáticamente se dice que es *estáticamente tipado*.

Definición 4.15. Un lenguaje en el que el tipo de algunas expresiones puede ser determinado únicamente en tiempo de ejecución se le denomina *dinámicamente tipado*.

Cabe señalar que los lenguajes no tipados pueden imponer seguridad realizando verificaciones en tiempo de ejecución, mientras que los lenguajes tipados pueden imponer seguridad rechazando estáticamente todos los programas que son potencialmente inseguros o, alternativamente, pueden realizar una mezcla de verificaciones estáticas y dinámicas.

Existen lenguajes que son fuertemente verificados incluso cuando no son estáticamente verificados y no cuentan con un sistema de tipos, ya que realizan verificaciones en tiempo de ejecución, como es el caso de LISP.

También debemos prestar atención a que el hecho que un lenguaje sea explícita y estáticamente tipado no implica que sea fuertemente tipado como es el caso de C que, como se menciona explícitamente en la página 3 de [KR88], no es un lenguaje fuertemente tipado, sino que es un lenguaje débilmente tipado ya que cuenta con un sistema de tipos y se puede determinar el tipo no ambiguo de algunas de sus expresiones pero no de todas, como cuando se suma un entero a un apuntador, en la que el tipo del resultado puede ser visto como un entero o un apuntador.

De igual manera debemos fijarnos que un lenguaje que es implícitamente tipado no necesariamente implica que sea un lenguaje dinámicamente tipado. Por ejemplo, ML es un lenguaje implícita, estática y fuertemente tipado, de igual forma que Haskell.[4] Lo anterior lo logran debido a que cuentan con un algoritmo de inferencia de tipos que se lleva a cabo estáticamente, lo que permite que se realice la verificación de tipos de forma estática.

Desde luego en un lenguaje estáticamente tipado la verificación de tipos pertenece a su semántica estática, mientras que para uno dinámicamente tipado (o uno no tipado en el que se realice verificación de tipos) a su semántica dinámica.

En el caso de lenguaje tipados que permiten que se realice una parte de la verificación de tipos estáticamente y otra dinámicamente, la primera pertenece a su semántica estática y la segunda a su semántica dinámica. En este caso podemos situar a lenguajes como C++, que permite que la mayor parte de su verificación de tipos se realice estáticamente. Además, provee al programador el operador `dynamic_cast` que permite realizar una conversión de tipos en tiempo de ejecución; si la conversión es correcta regresa un apuntador válido y un apuntador nulo si es incorrecta; es decir, realiza ímplicitamente una verificación de tipos en tiempo de ejecución. C++ también cuenta con el operador `static_cast` que, cuando se utiliza, realiza la conversión de tipos estáticamente, es decir, no se verifica que sea correcta en tiempo de ejecución, con lo cual desde luego se pueden permitir conversiones inválidas que conducirían a un error durante la ejecución del programa. Como C++ permite que la mayor parte de la verificación de tipos se realice estáticamente, mientras que sólo algunas verificaciones específicas explícitas por parte del programador se llevan a cabo en tiempo de ejecución, se suele catalogar a C++ como un lenguaje estáticamente tipado.

Una propiedad con la que generalmente cuentan los lenguajes de programación es con la propiedad de ser Turing completos.

Si un lenguaje de programación es Turing completo entonces saber si un programa tendrá un error es indecidible.

Como resultado, cualquier sistema de tipos consistente y decidible es incompleto, es decir, rechazará algunos programas válidos.

En general, la labor del analizador semántico de un compilador consiste en implementar la semántica estática de un lenguaje, mientras que al sistema en tiempo de ejecución le corresponde implementar la semántica dinámica.

En lenguajes dinámicamente tipados, en principio le corresponde al sistema en tiempo de ejecución realizar la verificación de tipos. Esto, como vimos, lo puede realizar un intérprete, en donde él mismo juega el papel del sistema en tiempo de ejecución, o lo puede propiciar un compilador generando código que realice directamente verificación de tipos o código que llame a una biblioteca en tiempo de ejecución.

[4] Aunque en el caso de Haskell su tipado no es puramente implícito, ya que el programador debe, si así lo requiere el compilador o lo desea el programador, escribir algunos de los tipos explícitamente.

En ambos casos, como se mencionó en el capítulo 1, realizar la verificación de tipos en tiempo de ejecución es desafortunado en lo que a la eficiencia de un programa concierne, debido a que en promedio su rendimiento se degrada de un 10 a un 25 % a causa de estas verificaciones de tipos en tiempo de ejecución.

En nuestro caso, Python es un lenguaje implícita y dinámicamente tipado, por lo que en principio tenemos que realizar la verificación de tipos en tiempo de ejecución. Sin embargo podemos ser más ambiciosos y tratar de inferir información acerca del comportamiento de un programa en ejecución, en tiempo de compilación, siguiendo el mismo principio de nuestro ejemplo del despacho dinámico de Java, para poder de esta forma mover algunas de las verificaciones de tipos de tiempo de ejecución a tiempo de compilación.

4.3. Analizador semántico

Tabla de símbolos

La mayoría de los lenguajes de programación permite la declaración, definición y uso de símbolos para representar constantes, variables, métodos, tipos y objetos, por lo que el compilador debe encargarse de verificar que estos símbolos se usen correctamente con base en la definición del lenguaje.

Una característica con la que cuentan una gran cantidad de lenguajes es tener estructura de bloques (un concepto introducido por Algol) y por lo general estos lenguajes también pertenecen a la familia de lenguajes que cuentan con alcance léxico.

A diferencia de los lenguajes que tienen alcance léxico, existen lenguajes que en su lugar cuentan con alcance dinámico. Los lenguajes de hoy en día en su gran mayoría cuentan con alcance léxico y no dinámico.

En los lenguajes con estructura de bloques y alcance léxico, una declaración de una variable en un bloque interno puede esconder la declaración de un variable con el mismo nombre que se encuentre en un bloque más externo. Desde luego el compilador debe ser capaz de seguir el rastro de cada uno de los símbolos del programa y saber a qué declaración se está refiriendo cada uso de una variable particular, es decir, el compilador debe saber en todo momento cuál es el alcance del bloque actual que se está compilando y todos los alcances de los bloques exteriores que están activos, para poder de esta manera determinar si hay una declaración viva o varias de la variable que se está utilizando y, de ser así, determinar la declaración (más interna) que es a la que se está refiriendo tal uso.

Cada símbolo en un programa tiene asociado a él una serie de atributos, que se derivan de la sintaxis y la semántica del lenguaje fuente y de la declaración y uso del símbolo en un programa particular. Algunos de estos atributos más relevantes son: el nombre del símbolo, su tipo, alcance y tamaño en memoria.

Esta información adicional, en forma de atributos de cada uno de los símbolos del programa, es indispensable para que el compilador sea capaz de realizar algunas de sus tareas más importantes, como es la verificación de tipos y la generación de código (ya que sin ella no sería posible realizar estas tareas).

La tabla de símbolos es una estructura de datos que almacena los símbolos y sus respectivos atributos. Además, una tabla de símbolos de un compilador con estructura de bloques y alcance léxico, debe permitir almacenar un símbolo dentro de su correspondiente alcance, es decir, la tabla de símbolos debe auxiliar al compilador ofreciéndole una interfaz que le permita, entre otras, cosas abrir un nuevo alcance siempre que sea necesario y cerrarlo cuando ya no esté activo.

De esta forma los principales métodos con los que debe contar la interfaz de una tabla de símbolos para este tipo de lenguajes son:

- lookUp(name) regresa la estructura que contiene los atributos de la declaración válida actual de name. Si no existe una declaración válida activa de nombre regresa un valor que lo indique, generalmente un apuntador nulo.
- insert(name, record) ingresa nombre en la tabla de símbolos bajo el alcance activo. record contiene los atributos del símbolo, obtenidos a partir de su declaración.
- openScope() abre un nuevo alcance en la tabla de símbolos. Los símbolos nuevos se ingresarán en el alcance resultante.
- closeScope() cierra el alcance más recientemente abierto en la tabla de símbolos. Las subsecuentes referencias a símbolos se revierten hacia los alcances exteriores.
- declaredLocally(name) verifica si name está presente en el alcance actual (más interno) y de ser así regresa **true**. Si name está en un alcance exterior, o no está en la tabla de símbolos, regresa **false**.

El análisis semántico se lleva a cabo después de haberse realizado el análisis sintáctico, por lo que debe realizarse sobre la representación intermedia que genera el analizador sintáctico, siendo ésta la mayor parte de las veces una representación de alto nivel como un AST.

En nuestro caso, contamos en este momento con el AST que construyó el analizador sintáctico que estudiamos en el capítulo anterior. Así que para llevar a cabo el análisis semántico debemos realizar un recorrido sobre el AST.

Podemos decir entonces, como se indica en [**pp.103**; App04], que el análisis semántico de un compilador (de un lenguaje explícitamente y estáticamente tipado) se encarga de conectar definiciones de variables a sus usos[5] y verifica que cada expresión tenga un tipo correcto.

[5]Generalmente haciendo uso de una tabla de símbolos.

Por lo que al manejo de la tabla de símbolos concierne, debemos recorrer el AST con dos objetivos:

1. Procesar declaraciones de símbolos y
2. conectar cada uso de un símbolo con su respectiva declaración.

Durante el recorrido del AST, si se encuentra un nodo que indique que comienza un nuevo bloque se debe abrir un nuevo alcance con ayuda de la tabla de símbolos; cuando termina el bloque se debe cerrar este alcance. Si se trata de un nodo que represente la declaración de un símbolo, entonces debe verificarse que no haya una declaración previa activa incompatible con la actual; si la hay, se manda un mensaje de error; en caso contrario se almacena la declaración en la tabla de símbolos. En el caso de un nodo que represente un uso de un símbolo (una presencia del símbolo que no es su declaración) debe verificarse si hay al menos una declaración activa del símbolo y de ser así, con base en el alcance, elegir la más interna y enriquecer al nodo con una referencia a la entrada del símbolo en la tabla de símbolos. Si dicha conexión no se puede realizar, entonces el uso del símbolo es incorrecto debido a que no se declaró previamente, o su uso es inconsistente con su declaración; en ambos casos el compilador debe generar un mensaje de error. Si no hubo error, las fases subsecuentes pueden utilizar la referencia para obtener información del símbolo, como su tipo y la cantidad de espacio que ocupará en memoria.

Aunque el uso de una tabla de símbolos no es estrictamente necesario, debido a que siempre que debamos consultar la información de un símbolo podemos remitirnos a su declaración, es claro que su uso es de gran ayuda, ya que por una parte facilita tanto el diseño como la implementación del compilador (actuando como un repositorio de información que se puede compartir entre la diferentes etapas del compilador) y por otra mejora la eficiencia ya que, como se puede notar, el consultar la declaración de un símbolo, en principio, implicaría hacer un recorrido sobre el AST en busca de ella, lo que nos lleva a que cada vez que se requiriera consultar la información de un símbolo se tendría que hacer este recorrido, lo que es desde luego una estrategia sumamente ineficiente.

Una manera intuitiva y comúnmente utilizada para realizar el análisis semántico en lenguajes explícitamente tipados, como es el caso de C o Java, es realizando un recorrido sobre el AST, en donde la principal tarea a realizar es la verificación de tipos; por ejemplo, en el subárbol del AST que representa la expresión `x+3` (suponiendo que el lenguaje sólo admite la suma sobre números enteros) debe verificar que tanto la expresión representada por su hijo izquierdo como la representada por su hijo derecho tenga tipo entero.

Cuando el patrón Visitor es parte del diseño del AST, se puede utilizar un visitante concreto que se encargue de realizar el recorrido sobre el AST llevando a cabo el análisis semántico.

Es claro que nosotros no estamos en posibilidad de realizar el análisis semántico como en el caso anterior, debido a que Python es un lenguaje implícita y dinámicamente tipado, lo que se ve reflejado en el AST (que construimos en el capítulo 3), y por ello no cuenta con nodos que representen declaraciones explícitas de variables que incluyan sus tipos. Este resultado es fundamental para comprender por qué usualmente se suele implementar a los lenguajes implícita y dinámicamente tipados por medio de intérpretes, ya que en tiempo de compilación **no** se cuenta con la información necesaria para poder realizar la verificación de tipos.

Una posible salida (rápida) al problema es no realizar verificación de tipos, es decir, que el analizador semántico del compilador no realice ningún tipo de verificación de tipos, pero además que en tiempo de ejecución tampoco realice ninguna de estas verificaciones. Y en efecto, existen compiladores, como es el caso de Chez Scheme, que mediante una opción permite al usuario indicarle que debe generar código que no realice verificación de tipos, aunque su comportamiento por omisión es generar código que sí realice este tipo de verificaciones. La ventaja de no realizar verificación de tipos en lo absoluto es el tiempo que se gana en tiempo de compilación, pero sobre todo en tiempo de ejecución, aunque desde luego su principal desventaja es clara, ya que no se pueden detectar los errores de tipos que el programador pueda cometer, con lo que se pierde el sentido que un lenguaje cuente con sistema de tipos ya que no se verificará que se cumplan sus reglas.

Una segunda opción es realizar inferencia de tipos, la cuál sería una solución ideal y deseable, y es la opción que utilizan lenguajes como ML y Haskell, que como se mencionó, aunque ambos son implícitamente tipados, también son estáticamente tipados debido a que es posible determinar los tipos en tiempo de compilación, mediante un algoritmo de inferencia de tipos.

Nosotros quisiéramos poder realizar inferencia de tipos con el fin de llevar a cabo las verificaciones de tipos estáticamente, lo cual en nuestro caso lamentablemente no es posible. El sistema de tipos de ML es un sistema bien diseñado y definido formalmente, lo que dio lugar a que se cuente con un algoritmo de inferencia de tipos en este lenguaje. En cambio, lenguajes como Python que en sus objetivos al diseñarlo no figuraba el tener un sistema de tipos elegante y que contara con un algoritmo de inferencia de tipos, sino más bien hacer la vida del programador "más fácil" en las tareas cotidianas, nos dejan en una situación incómoda en este sentido. Aun para este tipos de lenguajes, que no se diseñaron pensando en contar con este tipo de característica, se han investigado diferentes algoritmos que nos ayuden a recabar información acerca de los tipos en tiempo de compilación, incluso con todas las dificultades que presentan en esta parte de su diseño (en su sistema de tipos).

Algunos de estos algoritmos tratan de remediar esta situación realizando aproximaciones conservadoras de los tipos que puedan tener cada uno de los enunciados de un

programa, intuitivamente siguiendo el posible flujo de control de un programa, por lo que debe ser claro que estos algoritmos requieren realizar análisis de flujo.

El análisis de flujo es algo indispensable para realizar optimizaciones en un compilador, pero no sólo sirve a este fin. Por ello, a continuación presentaremos una sección dedicada a las optimizaciones y otra más dedicada al análisis de flujo, pero antes presentaremos el código de tres direcciones, una representación intermedia muy utilizada al realizar análisis de flujo.

En nuestra sección de optimizaciones prestaremos particular atención a una representación intermedia de gran utilidad para llevarlas a cabo, conocida como forma SSA. Cabe mencionar que esta forma hace que el algoritmo que utilizaremos para realizar la inferencia de tipos sea más preciso.

La forma SSA puede ser vista como una representación de la información contenida en cadenas clásicas uso-definición y definición-uso. Las cadenas uso-definición y definición-uso representan información del flujo de datos en las variables de un programa. Una cadena definición-uso de una variable conecta la definición de esa variable a todos sus usos posteriores, mientras que una cadena uso-definición conecta un uso a todas la definiciones posteriores.

Hemos decidido estudiar la forma SSA no sólo porque hace que el algoritmo de inferencia de tipos que utilizaremos sea más preciso, sino porque es una representación que contiene la información necesaria para poder realizar un gran número de optimizaciones. Tradicionalmente las optimizaciones que se han utilizado dentro de un compilador requieren que se realice cierto análisis particular del flujo de control y del flujo de datos de un programa, que permitan obtener información que es útil para realizar una optimización específica. También usualmente se requiere que se construyan ciertas estructuras de datos particulares a cierta optimización, así que la forma SSA, al permitir capturar la información útil necesaria no sólo para una determinada optimización sino para varias, representa una gran ventaja, pues de esta manera no se tienen que estar haciendo cálculos específicos para cada optimización distinta que se realice.

4.4. Código de tres direcciones

Una representación intermedia lineal consiste de una secuencia de instrucciones que se ejecutan en el orden en que aparecen. Las representaciones intermedias lineales que se utilizan en los compiladores son parecidas al código ensamblador de una máquina abstracta.

El código de tres direcciones es una representación intermedia lineal.

El término "código de tres direcciones" (*three-address code*) proviene de las instrucciones que tienen como forma general $x = y\ op\ z$ y que cuentan con tres direcciones: dos correspondientes a los operandos, y y z, y una correspondiente al resultado, x.

El nivel de abstracción del código de tres direcciones puede ser desde alto hasta bajo, dependiendo de la elección de los operadores (aunque suele utilizarse como una representación intermedia de bajo nivel). Por ejemplo en el caso de las expresiones, las diferencias entre un AST y su correspondiente código de tres direcciones son superficiales. En el caso de las estructuras de control, como son los ciclos, la situación es diferente, ya que en este caso un AST representa los componentes del ciclo, mientras que el código de tres direcciones contiene etiquetas e instrucciones de salto para representar explícitamente el flujo de control, de la misma forma que en el código de máquina.

En el código de tres direcciones hay a lo más un operador en el lado derecho de una instrucción. Por ejemplo, si la expresión $x + y + z$ está presente en el código fuente, su correspondiente código de tres direcciones sería la siguiente secuencia de instrucciones (de tres direcciones):

$$t_1 = x * y$$
$$t_2 = x + t_1$$

donde t_1 y t_2 son variables temporales introducidas por el compilador. Esta forma intermedia es ideal para aquellas optimizaciones en las que se requiere reordenar las instrucciones, ya que al haberse introducido nuevas variables para los cálculos intermedios, cambiar el orden de las instrucciones es mucho más fácil que en otras representaciones intermedias como un AST.

4.4.1. Cuádruplas

Una instrucción de tres direcciones se puede implementar de diferentes maneras. Tres de las más utilizadas son: cuádruplas (*quadruples*), tripletas (*triples*) y tripletas indirectas (*indirect triples*). De estas tres, nosotros solo utilizaremos cuádruplas, que estudiaremos en la sección 4.6.

4.5. Optimizaciones

Una etapa opcional, que se ha convertido en indispensable en los compiladores de nuestros días, es la etapa de optimización. Más aún, los compiladores optimizadores modernos de ahora, como vimos en el capítulo 1, no suelen implementar una sino varias etapas de optimización durante el proceso de compilación, realizando cada una de ellas sobre diferentes representaciones intermedias del código fuente o, en algunos compiladores, llevando a cabo diferentes etapas de optimización sobre la misma representación intermedia. Este es el caso de compiladores que realizan una optimización sobre una representación r_1, después la transforman a r_2 sobre la que realizan otra etapa de optimización y, posteriormente, transforman r_2 a r_1 y realizan una segunda etapa de

optimización sobre r_1. La razón de estos posibles escenarios es que cada una de las optimizaciones requiere de información específica del código fuente y, como hemos visto, diferentes representaciones intermedias contienen diferente información al respecto. De esta manera hay optimizaciones que se realizan sobre representaciones de alto nivel como el AST, otras sobre representaciones de nivel intermedio como código tres direcciones y otras más que se llevan a cabo sobre representaciones de bajo nivel como árboles de bajo nivel o código ensamblador. Algunas de ellas pueden realizarse sobre representaciones de diferentes niveles pero, además, existen dependencias entre ellas. Hay optimizaciones que se deben realizar únicamente después de haberse realizado otras y pueden darse dependencias circulares, en cuyo caso podemos iterar realizando las optimizaciones contenidas en una dependencia dada hasta que no haya más oportunidades de optimización en el código, lo que nos lleva a casos como el de nuestro ejemplo, en el que algunas de las optimizaciones que se realizan sobre r_1 deben realizarse después de las que se realicen sobre r_2 y, a la inversa, algunas de las optimizaciones que se realicen sobre r_2 deben realizarse después de haber realizado algunas de las optimizaciones sobre r_1.

Para poder realizar optimizaciones es necesario comenzar realizando análisis de flujo del programa.

4.6. Análisis de flujo

Definición 4.16. El *análisis de flujo* está compuesto por:

- Análisis de flujo de control.
- Análisis de flujo de datos.

El análisis del flujo de control es, en general (aunque no necesariamente), prerrequisito para el análisis del flujo de datos.

El análisis de flujo se realiza previo a la ejecución de un programa, es decir, de manera estática.

Definición 4.17. *Análisis de flujo de control* es la codificación del posible flujo de control o estructura de flujo de control pertinente de un programa para un consiguiente análisis de flujo de datos.

Definición 4.18. *Análisis de flujo de datos* es el proceso estático de averiguar y recabar información acerca de la posible modificación, preservación y uso en tiempo de ejecución de cantidades en un programa.

Usualmente se seleccionan como las cantidades bajo escrutinio a los valores posibles de variables, porque proveen información fundamental.

La información obtenida por medio del análisis de flujo tiene diferentes aplicaciones; entre las más destacadas están:

1. Permitir al programador razonar acerca del comportamiento del programa.
2. Mejorar la eficiencia de la ejecución de un programa.

En cuanto a la primera de estas aplicaciones, es fácil notar que se pueden desarrollar diferentes aplicaciones de depuración que sirvan de ayuda al programador; mientras que en la segunda de estas aplicaciones el análisis de flujo permite que se realicen una gran cantidad de optimizaciones en un programa.

Análisis de flujo de control

El análisis de flujo de control se puede realizar sobre representaciones de nivel alto, intermedio o bajo de un programa.

En algunas ocasiones, durante el análisis de flujo de control, se aplican transformaciones equivalentes que preservan el flujo de control de un programa para simplificar el consiguiente análisis de flujo de datos. Por ejemplo, separando nodos (*node splitting*) y expandiendo algunas funciones existentes en línea (*in-line*) son algunas posibilidades.

Para realizar el análisis de control de flujo se supone ya sea que

- el control de flujo de un programa es inmediato a nivel de código fuente (quizás después de reemplazar enunciados cuyo significado involucre flujo de control por enunciados donde el flujo de control sea más explícito) o bien,
- el programa ha sido transformado por el front end del compilador en una representación intermedia, en la cual el flujo de control es explícito o muy fácilmente derivable.

Una representación intermedia donde el flujo de control es explícito, y que se utiliza para realizar análisis de flujo por la gran mayoría de los compiladores, es toda aquella representación que está basada en cuádruplas.

Una *cuádrupla* tiene cuatro campos. La forma típica de una cuádrupla es:

$$(< operador >, < operando_1 >, < operando_2 >, < resultado >)$$

que usualmente se interpreta diciendo que el $< resultado >$ se obtiene aplicando el $< operador >$ a los dos argumentos $< operando_1 >$ y $< operando_2 >$. En caso de que el operador se trate de un operador unario, el segundo operando es vacío; además se suelen introducir variables temporales en los cálculos intermedios.

Por ejemplo, las cuádruplas correspondientes al enunciado $x = y + z$ son:

1. $(+, y, z, t_1)$
2. $(=, t_1, , x)$

donde t_1 es una variable temporal que se introdujo.

Las cuádruplas normalmente aparecen en el orden en que se ejecutarán. Usualmente se asocian banderas con los operandos y el resultado, que indican el tipo e información adicional .

Definición 4.19. Un *bloque básico* (*basic block*) es una secuencia lineal de longitud máxima de enunciados que tiene un único punto de entrada (el primer enunciado) y un único punto de salida (el último enunciado).

Una vez que el primer enunciado se ejecuta, todos los enunciados restantes del bloque se ejecutan secuencialmente. En otras palabras en un bloque básico no existen transferencias de flujo excepto en su último enunciado.

En un bloque básico cada enunciado que represente una llamada a función se trata como si fuera un enunciado de una asignación, es decir, no se toma en cuenta el flujo de control de éstos.

Definición 4.20. Una *gráfica de flujo de control* (*Control Flow Graph, CFG*) es una tripleta $G = (N, E, n_0)$, donde:

1. N es un conjunto finito de nodos y $\forall n \in N$, n es un bloque básico.
2. $E \subset N \times N$ es un conjunto de aristas. La arista (x, y) *ingresa* al nodo y y *deja* al nodo x. Se dice que x es un *predecesor* de y y que y es un *sucesor* de x.
3. $n_0 \in N$ es el *nodo inicial*. $\forall n \in N$ existe una trayectoria de n_o a n.

En una gráfica de flujo de control cada nodo corresponde a un bloque básico y cada arista (n_i, n_j) corresponde a una posible transferencia de control del bloque n_i al bloque n_j.

Podemos suponer que cada gráfica de control de flujo tiene un único nodo de entrada n_0 y único nodo de salida n_f. En una gráfica de control de flujo , n_0 corresponde al punto de entrada de la función.

Si la función tiene múltiples punto de entrada, se puede insertar un nuevo nodo n_0 único y añadir aristas de n_0 a cada nodo que represente un punto de entrada. Si la función tiene múltiples puntos de salida (más común que el que una función tenga múltiples puntos de entrada), se agrega un nuevo nodo n_f y se añaden aristas de cada bloque que representa un punto de salida a n_f.

Análisis del flujo de datos

Podemos distinguir diferentes niveles de análisis, que son:

- Enunciado o cuádrupla;

- bloque básico (*intrablock*);
- función (*intraprocedural*) y
- programa (*interprocedural*).

Algunos autores describen el análisis a nivel de enunciado y de bloque básico (*intrablock*) con el término *local*, mientras que el análisis a nivel de función (*intraprocedural*) y de programa (*interprocedural*) con el término *global*.

Nosotros trabajaremos a nivel de función (*intraprocedural*) a menos que se indique lo contrario. Existen diferentes algoritmos de análisis de flujo de datos, de los cuáles dos de los más destacados son:

1. El algoritmo iterativo de Kildall y Ullman.
2. Análisis de intevalos de Allen y Cocke.

Estos dos algoritmos han servido de base para la formulación de nuevos algoritmos y modificaciones de los mismos. Sin duda el más utilizado y conocido de estos dos es el algoritmo iterativo.

4.6.1. Un algoritmo rápido para encontrar dominadores

Definición 4.21. Una *gráfica de flujo* es una tripleta $G = (N, A; s)$, donde (N, A) es una gráfica (finita) dirigida y $\forall n \in N$ existe una trayectoria del nodo inicial $s \in N$ a n.

Una gráfica de flujo es un modelo gráfico abstracto de una gráfica de flujo de control con enunciados, instrucciones o cuádruplas dentro de cada bloque básico.

Definición 4.22. Sea $G = (V, E; r)$ una gráfica de flujo. Si x e y son dos nodos (no necesariamente distintos) en V, entonces x *domina* y si y sólo si toda trayectoria del nodo inicial r a y contiene a x; lo denotamos como $x \gtrdot y$.

Denotamos como $DOM(y) = \{x \mid x \gtrdot y\}$ al conjunto de dominadores del nodo $y, \forall y \in V$.

Definición 4.23. Sea $G = (V, E, r)$ una gráfica de flujo y sean $x, y \in V$, se dice que x *domina propiamente* a y si y sólo si $x \neq y$ y x domina y y lo denotamos como $x \gg y$.[6] Si x no domina propiamente a y escribimos $x \not\gg y$.

Definición 4.24. Decimos que un nodo x es el *dominador inmediato* de un nodo y y se denota $x = idom(y)$ si y sólo si

1. x domina propiamente a y y
2. si un nodo z domina propiamente a y y $z \neq x$, entonces z domina propiamente a x.

[6]En lo siguiente se supone, cuando hablemos de dominadores, que nos referimos a dominadores propios a menos que se indique lo contrario.

Propiedad 4.1. Todo vértice de una gráfica de flujo $G = (V, E, r)$, excepto r, tiene un único dominador inmediato.

Definición 4.25. Sea $G = (V, E; r)$ una gráfica de flujo. El conjunto de aristas

$$\{(idom(w), w) \mid w \in V - \{r\}\}$$

forman un árbol dirigido con raíz r, al que se le denomina *árbol dominador* de G.

En el árbol dominador se cumple que un nodo v domina a un nodo w si y sólo si v es un ancestro propio de w.

Definición 4.26. Si $G = (V, E)$ es una gráfica y $T = (V', E'; r)$ es un árbol tal que (V', E') es una subgráfica de G y $V = V'$, entonces T es un *árbol generador* (*spanning tree*) de G.

Nosotros buscamos construir el árbol dominador de un gráfica de flujo G arbitraria. El algoritmo que describiremos a continuación para calcular el árbol dominador consiste de tres partes. Primero se realiza un recorrido en profundidad sobre la gráfica de flujo de entrada $G = (V, E; r)$, comenzando en la raíz r y numerando los nodos de V de 1 hasta n en el orden que se hayan visitado durante el recorrido. El recorrido genera un árbol generador T con raíz r con nodos numerados en preorden. Por conveniencia supondremos que los nodos se identifican con su número.

Como segunda parte se calcula el *semidominador* de cada nodo $w \neq r$, denotado por $sdom(w)$, y que se define como $sdom(w) = min\{v \mid$ existe una trayectoria $v = v_0, v_1, \ldots, v_k = w$ tal que $v_i \gg w$ para $1 \leq i \leq k-1\}$.

Como tercera y última parte, se utilizan los semidominadores para calcular los dominadores inmediatos de todos los nodos.

Propiedad 4.2. Sea $w \neq r$ y sea u un vértice para el cual $sdom(u)$ es mínimo entre los vértices u que satisfacen que $sdom(w)$ es un ancestro propio de u y u es un ancestro de w. Entonces:

$$idom(w) = \begin{cases} sdom(w) & \text{si } sdom(w) = sdom(u), \\ idom(u) & \text{en otro caso.} \end{cases}$$

La siguiente propiedad provee una forma de calcular semidominadores.

Propiedad 4.3. Para cualquier vértice $w \neq r$,

$$sdom(w) = min\Big(\{v \mid (v, w) \in E \text{ y } u < w\} \bigcup \{sdom(u) \mid u > w \text{ y } \exists (u, w) \in E \text{ tal que } u \text{ es un ancestro de } v\}\Big)$$

El algoritmo consiste en los siguientes cuatro pasos:

1. Realizar un recorrido en profundidad sobre la gráfica de flujo de control de entrada. Numerar los vértices de 1 hasta n conforme se vayan visitando en el recorrido. Inicializar las variables que se utilizarán en los siguientes pasos.
2. Calcular los semidominadores de todos los nodos aplicando la propiedad 4.3, realizando el cálculo nodo por nodo, en orden decreciente, según la numeración obtenida en el paso anterior.
3. Implícitamente definir el dominador inmediato de cada nodo aplicando la propiedad 4.2.
4. Explícitamente definir el dominador inmediato de cada nodo, realizando el cálculo nodo por nodo en orden creciente.

A continuación se presenta el pseudocódigo del algoritmo:

```
procedure DOMINATORS(integer set array succ(1::n); integer r, n;
integer array dom(1::n));
   begin
      integer array parent, ancestor, [child,] vertex(1 :: n);
      integer array label, semi [,size](0 :: n);
      integer set array pred, bucket(1 :: n);
      integer u,v, x;

      procedure DFS(integer v);
         begin
            semi(v) := n := n+1;
            vertex(n) = label(v) := v;
            ancestor(v) := [child(v):=] 0;
            [size(v):=1;]
            for each w ∈ succ(v) do
               if semi(w)=0 then parent(w) := v; DFS(w) fi;
               add v to pred(w) od
       end DFS;
       procedure COMPRESS(integer v);
          if ancestor(ancestor(v)) ≠ 0 then
             COMPRESS(ancestor(v));
             if semi(label(ancestor(v)) < semi(label(v)) then
                label(v) := label(ancestor(v)) fi;
             ancestor(v) := ancestor(ancestor(v)) fi;
      integer procedure EVAL(integer v);
         if(ancestor(v) = 0 then EVAL := v
            else COMPRESS(v); EVAL := label(v) fi;
```

Algoritmo 4.1: Un algoritmo rápido para encontrar dominadores (continúa)

```
      procedure LINK(integer v, w);
         ancestor(w) := v;
      step1: for v := 1 until n do
                     pred(v) := bucket(v) := ∅; semi(v) := 0
                     od;
                  n :=0;
                  DFS(r);
                  [size(0) := label(0) := semi(0) := 0;]
                  for i := n by −1 until 2 do
                     w := vertex(i);
      step2:     for each v ∈ pred(w) do
                        u := EVAL(v);
                           if semi(u) < semi(w) then semi(w) := semi(u)
                           fi od;
                        add w to bucket(vertex(semi(w)));
                        LINK(parent(w),w);
      step3:         for each v ∈ bucket(parent(w)) do
                        delete v form bucket(parent(w));
                        u := EVAL(v);
                        dom(v) := if semi(u) < semi(v) then u
                                             else parent(w) fi od od;
      step4:      i:= 2 until n do
                        w := vertex(i);
                        if dom(w) ≠ vertex(semi(w))
                           then dom(w) := dom(dom(w)) fi od;
                     dom(r) := 0
end DOMINATORS
```

Algoritmo 4.1: Un algoritmo rápido para encontrar dominadores

La versión simple del algoritmo consiste en el pseudocódigo anterior, eliminando todo lo que se encuentra entre corchetes.[7] La versión sofisticada del algoritmo consiste en el pseudocódigo anterior incluyendo todo lo que se encuentra entre corchetes y substituyendo los funciones EVAL y LINK con las siguientes versiones:

```
integer procedure EVAL(integer v);
   if ancestor(v)=0 then EVAL := label(v)
      else COMPRESS(v);
         EVAL := if semi(label(ancestor(v))) ≥ semi(label(v))
                     then label(v) else label(ancestor(v)) fi fi;
```

Versión sofisticada de la función EVAL.

[7]Hemos utilizado los corchetes inspirándonos en el significado que tienen en las expresiones regulares en donde denotan la ausencia o exactamente una presencia de la expresión regular contenida entre ellos.

```
procedure LINK(integer v,w);
   begin integer s;
      s := w;
      while semi(label(w)) < semi(label(child(s))) do
         if size(s) + size(child(child(s))) ≥ 2·size(child(s))
            then ancestor(child(s)) := s;
                 child(s) := child(child(s))
         else size(child(s)) := size(s);
            s := ancestor(s) := child(s) fi od;
      label(s) := label(w);
      size(v) := size(v) + size(w);
      if size(v) < 2·size(w) then s, child(v) := child(v), s fi;
      while s ≠ 0 do ancestor(s) := v; s := child(s) od
end LINK;
```

Versión sofisticada de la función LINK.

La versión simple del algoritmo tiene complejidad $O(m \log n)$ en tiempo, donde m es el número de aristas y n es el número de nodos, mientras que la versión sofisticada tiene complejidad $O(m\alpha(m,n))$ donde $\alpha(m,n)$ es el inverso funcional de la función de Ackerman.

En el pseudocódigo del algoritmo se utiliza la siguiente estructura como entrada,

- succ(v): El conjunto de vértices w tal que (v,w) es una arista de la gráfica.

Las estructuras que siguen se calculan durante la ejecución del algoritmo.

- parent(w): El vértice que es el padre del vértice w en el árbol generador construido por el recorrido.
- pred(w): El conjunto de vértices v tal que (v,w) es una arista de la gráfica.
- semi(w): Un número que se define de la siguiente manera:
  1. Antes que el vértice w se numere, semi(v)=0.
  2. Después que w se numere pero antes de que $sdom$ se calcule, semi(w) es el número de w.
  3. Después de que $sdom(w)$ se calcule, semi(w) es el número de $sdom(w)$.
- vertex(i): El vértice cuyo número es i.
- bucket(w): $\{x \in V \mid sdom(x) = w\}$.
- dom(w): Un vértice que se define de la siguiente manera:

1. Después del paso 3, si el semidominador de w es su dominador inmediato, entonces dom(w) es el dominador inmediato de w. En otro caso dom(w) es un vértice v cuyo número es menor que el de w y cuyo dominador inmediato es también el dominador inmediato de w.
2. Después del paso 4, dom(w) es el dominador inmediato de w.

En lugar de convertir los nombres de los vértices a números durante el paso 1 y convertir los números de regreso a nombres al final del cálculo, nos referimos a los vértices tanto como sea posible por su nombre. Los arreglos semi y vertex incorporan todo lo que se necesita saber de los números de los vértices. El arreglo semi tiene un propósito dual, pues representa (aunque no simultáneamente) tanto al número de un vértice como al número de su semidominador. Mientras que ahorra espacio, esta estructura nos permite simplificar el cálculo de semidominadores, combinando los dos casos de la propiedad 4.3 en uno.

4.7. La forma SSA

4.7.1. Construcción de la forma SSA

Definición 4.27. Un programa está en forma SSA si cada variable es objetivo de exactamente un enunciado de asignación en la sección ejecutable del programa.

La conversión a forma SSA reemplaza al programa original por un nuevo programa con la misma gráfica de flujo de control. Para cada variable original V, las siguientes condiciones se deben cumplir en el nuevo programa:

1. Si dos trayectorias no triviales $X \overset{+}{\to} Z$ y $Y \overset{+}{\to} Z$ convergen en un nodo Z y los nodos X e Y contienen asignaciones a V (en el programa original), entonces una función ϕ trivial $V \leftarrow \phi(V_1, \ldots, V_k)$ se insertó en Z (en el nuevo programa).
2. Cada presencia de V en el programa original o en una función ϕ insertada ha sido reemplazada por una presencia de una nueva variable V_i, dejando al nuevo programa en forma SSA.
3. Junto con cualquier trayectoria de flujo de control, considerar cualquier uso de una variable V (en el programa original) y el correspondiente uso de V_i (en el programa nuevo). Entonces V y V_i tienen el mismo valor.

La conversión a forma SSA mínima se realiza en tres pasos:

1. Se calcula la frontera de dominación de cada uno de los nodos de la gráfica de flujo de control.

2. Utilizando las fronteras de dominación, se determinan los lugares de las funciones ϕ para cada variable en el programa original.

3. Las variable se renombran reemplazando cada presencia de una variable original V por una presencia correcta de una nueva variable V_i.

Definición 4.28. Sea $G = (V, E, r)$ una gráfica de flujo de control; la *frontera de dominación* $DF(X)$ de un nodo $X \in V$ es el conjunto de todos los nodos $Y \in V$ tal que X domina un predecesor de Y pero no domina estrictamente a Y:

$$DF(X) = \{Y \mid (\exists P \in Pred(Y))(X \gtrdot P \text{ y } X \not> Y)\}.$$

Calcular $DF(X)$ directamente a partir de esta definición tomaría tiempo cuadrático; para calcular la frontera de dominación en tiempo lineal se definen dos conjuntos (que sirven como cálculos intermedios) DF_{local} y DF_{up} para cada nodo, tal que se cumple la siguiente ecuación:

$$DF(X) = DF_{local}(X) \bigcup \Big(\bigcup_{Z \in Children(X)} DF_{up}(Z) \Big)$$

donde

$$DF_{local}(X) \stackrel{\text{def}}{=} \{Y \in Succ(X) \mid X \not> Y\}$$

y

$$DF_{up}(Z) \stackrel{\text{def}}{=} \{Y \in DF(Z) \mid idom(Z) \not> Y\}.$$

Los conjuntos que sirven como resultados intermedios se pueden calcular con una simple verificación de igualdad de la siguiente manera[8]:

Propiedad 4.4. Para cualquier nodo X,

$$DF_{local}(X) = \{Y \in Succ(X) \mid idom(Y) \neq X\}.$$

Propiedad 4.5. Para cualquier nodo X y cualquier hijo Z de X en el árbol dominador,

$$DF_{up}(Z) = \{Y \in DF(Z) \mid idom(Y) \neq X\}.$$

A continuación se muestra el pseudocódigo de un algoritmo que calcula $DF(X)$ para cada nodo X de una gráfica de flujo de control.

[8]La demostración de estas propiedades se puede consultar en [Cyt+91].

```
for each X en un recorrido ascendente del arbol dominador do
    DF(X) ← ∅
    for each Y ∈ Succ(X) do
        if idom(Y) ≠ X then DF(X) ← DF(X) ∪ {Y}
    end
    for each Z ∈ Children(X) do
        for each Y ∈ DF(Z) do
            if idom(Y) ≠ X then DF(X) ← DF(X) ∪ {Y}
        end
    end
end
```

Algoritmo 4.2: Cálculo de la frontera de dominación para cada nodo X

La complejidad del peor caso del algoritmo anterior es $O(E + N^2)$, aunque en la práctica tiene un comportamiento lineal.

Definición 4.29. Dado un conjunto φ de nodos de una gráfica del flujo de control, el conjunto $J(\varphi)$ de nodos de encuentro se define como el conjunto de todos los nodos Z tal que existen dos trayectorias no triviales que comienzan en dos nodos distintos en φ y convergen en Z.

Definición 4.30. El conjunto iterado de encuentro $J^+(\varphi)$ es el límite de la secuencia creciente de conjuntos de nodos

$$J_1 = J(\varphi),$$
$$J_{i+1} = J(\varphi \cup J_i).$$

En particular, si φ es el conjunto de nodos de asignación de una variable V, entonces $J^+(\varphi)$ es el conjunto de funciones ϕ de V.

Definición 4.31. Extendemos la frontera de dominación de nodos a conjunto de nodos:

$$DF(\varphi) = \bigcup_{X \in \varphi} DF(X).$$

Definición 4.32. La frontera de dominación iterada $DF^+(\varphi)$ es el límite de la secuencia creciente de conjuntos de nodos

$$DF_1 = DF(\varphi),$$
$$DF_{i+1} = F(\varphi \cup DF_i).$$

De las definiciones 4.30 y 4.32 es clara la relación entre el conjunto iterado de encuentro y la frontera de dominación iterada. Y se cumple la siguiente propiedad:

Propiedad 4.6. Si φ es el conjunto de nodos de asignación de una variable V, entonces

$$J^+(\varphi) = DF^+(\varphi)$$

.

Construcción de la forma SSA mínima

El algoritmo 4.3 inserta funciones ϕ triviales. El ciclo externo se realiza una vez para cada variable V en el programa.

```
IterCount ← 0
for each node X do
    HasAlready(X) ← 0
    Work(X) ← 0
W ← ∅
for each variable V do
    IterCount ← IterCount + 1
    for each X ∈ A(V) do
        Work(X) ← IterCount
        W ← W ∪ {X}
    while W ≠ ∅ do
        take X from W
        for each Y ∈ DF(X) do
            if HasAlready(Y) < IterCount then do
                place ⟨V ← φ(V, ..., V)⟩ at Y
                HasAlready(Y) ← IterCount
                if Work(Y) < IterCount then do
                    Work(Y) ← IterCount
                    W ← W ∪ {Y}
```

Algoritmo 4.3: Inserción de funciones ϕ

El algoritmo utiliza las siguientes estructuras de datos:

- W es la lista de trabajo de los que se están procesando. En cada iteración, W se inicializa como el conjunto $A(V)$ de nodos que contienen asignaciones a V. Cada nodo X en la lista de trabajo garantiza que cada nodo Y en $DF(X)$ reciba una función ϕ. Cada iteración termina cuando la lista de trabajo se queda sin elementos, es decir, vacía.
- $Work(*)$ es un arreglo de banderas, con una bandera por cada nodo, donde $Work(X)$ indica si X ha sido agregado a W durante la iteración actual del ciclo externo.
- $HasAlready(*)$ es un arreglo de banderas, una por cada nodo, donde $HasAlready(X)$ indica si una función ϕ de V ya ha sido insertada en X.

```
for each variable V do
    C(V) ← 0
    S(V) ← EmptyStack
end

call SEARCH(Entry)

SEARCH(X):
    for each enunciado A in X do
        if A es una asignacion ordinaria
            then
                for each variable V utilizada in RHS(A) do
                    reemplazar el uso de V por el uso de V_i
                                                donde i = Top(S(V))
                end
        for each V in LHS(A) do
            i ← C(V)
            reemplazar V por el nuevo V_i in LHS(A)
            push i en S(V)
            C(V) ← i + 1
        end
    end
    for each Y ∈ Succ(X) do
        j ← WichPred(Y, X)
        for each funcion φ F in Y do
            reemplazar el j-esimo operando V in RHS(F) por V_i
                                            donde i = Top(S(V))
        end
    end
    for each Y ∈ Children(X) do
        call SEARCH(Y)
    end
    for each asignacion A in X do
        for each V in oldLHS(A) do
            pop S(V)
        end
    end
end SEARCH
```

Algoritmo 4.4: Renombramiento de presencias de variables

El algoritmo 4.4 renombra todas las presencias de variables. Para cada variable V se generan variables nuevas que se denotan como V_i, donde i es un entero. Algunas de las estructuras que se utilizan en el algoritmo son:

- $S(*)$ es un arreglo de pilas, una pila por cada variable V. Las pilas pueden almacenar enteros. El entero i en el tope de $S(V)$ se utiliza para construir la variable V_i que debe reemplazar un uso de V.
- $C(*)$ es un arreglo de enteros, uno por cada variable V. El valor del contador $C(V)$ indica cuántas asignaciones a V han sido procesadas.
- $WichPred(X, Y)$ es un entero que indica que el predecesor de Y en la gráfica de flujo de control es X. El j–ésimo operando de una función ϕ en Y corresponde al j-ésimo predecesor de Y de la lista de aristas de entrada de Y.

Algoritmo 4.1 (Construcción de la forma SSA mínima)**.** Para construir la forma SSA mínima se deben seguir los siguientes tres pasos:

1. Calcular la frontera de dominación DF para cada nodo de la gráfica de flujo de control utilizando el algoritmo 4.2.
2. Colocar funciones ϕ triviales para cada variable en los nodos de la gráfica de flujo control utilizando el algoritmo 4.3.
3. Renombrar las variables utilizando el algoritmo 4.4.

Definición 4.33. Sea G una gráfica de flujo de control. Definimos $R := max\{|N|, |E|, |A|, |M|\}$ como el tamaño del programa original donde

- $|N|$ es el número de nodos de G,
- $|E|$ es el número de aristas de G,
- $|A_{orig}|$ el número de asignaciones originales a variables y
- $|M_{orig}|$ el número de presencias originales de variables.

El peor caso toma tiempo $O(R^2)$ calcular las fronteras de dominación y $O(R^3)$ construir la forma SSA mínima. Sin embargo, en la práctica, la conversión completa a la forma SSA mínima, incluyendo el cálculo de las fronteras de dominación, es lineal, es decir, toma tiempo $O(R)$.

Como acabamos de observar, en el algoritmo 4.1 se deben calcular previamente la fronteras de dominación para a partir de éstas calcular las funciones ϕ. En algunos casos este cálculo de las fronteras de dominación toma tiempo cuadrático en el número de nodos de la gráfica de flujo de control.

En comparación, existe un algoritmo simple desarrollado por Sreedhar y descrito en [SG95] que calcula las funciones ϕ en tiempo lineal.

4.7.2. Variantes de la forma SSA

Forma SSA podada

Choi en [CCF91] propone una variante de la forma SSA llamada SSA podada (*pruned SSA*). En esta forma, una vez que las fronteras de dominación han sido calculadas, se realiza un análisis de variables vivas para encontrar el conjunto de variables que están vivas en la entrada de un bloque básico. En la forma SSA podada se insertará una función ϕ para una variable V en un bloque básico b únicamente si V está viva al entrar a b.

Aunque es claro que la forma SSA podada puede insertar menos funciones ϕ que la forma SSA mínima, también lo es que consume más recursos construirla debido a que se debe realizar un análisis de flujo de datos global.

Forma SSA semipodada

Briggs en [Bri+98] presenta una variante de nombre SSA semipodada (*semi-pruned SSA*) basada en la observación de que muchas variables temporales tienen tiempos de vida muy cortos, esto es, se definen y usan solamente dentro de un único bloque básico. Estas variables no requerirán funcione ϕ en los puntos de encuentro. Durante la construcción de la forma SSA semipodada se realizan los cálculos precisamente sobre ese conjunto de variable que están vivas el entrar a algún bloque básico. A estas variables se les denomina *no locales*, esto es, aquellas que tienen definiciones fuera del bloque básico actual. Al construir la forma SSA semipodada se calcula de frontera de dominación iterada únicamente para las definiciones *no locales*. De aquí que el número de funciones ϕ insertadas será no mayor que en la forma SSA mínima y no menor que en la forma SSA podada. Es claro que el costo de calcular *no locales* es menor que realizar un análisis de variables vivas completo, ya que no se realizan iteraciones ni eliminaciones.

En [Bri+98] también se menciona que diferentes optimizaciones basadas en la forma SSA pueden sacar ventaja de una variante de SSA particular. Por ejemplo, las funciones ϕ adicionales de la forma SSA mínima pueden ser de provecho en la optimización conocida como numeración de valores (*value numbering*), ya que pueden ayudar a descubrir coincidencias de valores que sin ellas no se encontrarían.

Si la forma SSA se utiliza para encontrar rangos de vida (*live ranges)* durante el alojamiento de registros, las forma SSA podada y semipodada son de mayor utilidad, ya que las funciones ϕ adicionales pueden provocar que el tiempo de vida de las variables que tienen como argumentos se extienda innecesariamente.

Por otro lado, existen optimizaciones en las que se obtiene el mismo resultado independientemente de la variante de la forma SSA que se utilice, como es el caso de la propagación de constantes (*constant propagation*).

4.7.3. Destrucción de la forma SSA

Se pueden realizar diferentes análisis y optimizaciones sobre la forma SSA. Sin embargo, nuestro objetivo principal es que el código correspondiente a un programa fuente se ejecute en la máquina objetivo. Las arquitecturas de las máquinas reales generalmente no cuentan con instrucciones que implementen la semántica de las funciones ϕ nativamente, por lo que es claro que debemos traducir la forma SSA de regreso a código en el que no haya funciones ϕ, reemplazando éstas por instrucciones comúnmente implementadas en las arquitecturas.

Ingenuamente, una función ϕ con k entradas puede ser reemplazada en la entrada de un nodo X por k asignaciones ordinarias, una al final de cada predecesor (en el flujo de control) de X.

Dicha traducción puede insertar un gran número de copias, por lo que generalmente se confía en la etapa de fusión de copias (*copy coalescing phase*) del alojador de registros para eliminar tantas de éstas como sea posible.

Este algoritmo ingenuo dado por Cytron en [Cyt+91] funciona bien para la forma SSA que se produce a partir de transformaciones que no cambian radicalmente el espacio de nombres, como en las optimizaciones de propagación de constantes y eliminación de código muerto (*dead code eliminataion*). Sin embargo, las transformaciones que alteran radicalmente el espacio de nombres pueden causar que el algoritmo de inserción ingenua de copias produzca código incorrecto. Por ejemplo, la numeración de valores agresiva (*aggresive value numbering*) se beneficia trabajando sobre la forma SSA pero puede crear circunstancias que causen que el algoritmo ingenuo falle.

Briggs en [Bri+98] encontró dos tipos de problemas que se pueden producir debido a la aplicación del algoritmo ingenuo, a los que denominó el problema de la copia perdida (*the lost copy problem*) y el problema del intercambio (*the swap problem*), y da una solución a la destrucción de la forma SSA que explota las propiedades estructurales de la gráfica de flujo de control y de la gráfica SSA. Esta solución detecta patrones particulares y utiliza información de rangos de vida para guiar inserciones de copias que eliminan las funciones ϕ; cualquier copia redundante introducida durante la eliminación de las funciones ϕ se elimina en la etapa de fusión de copias.

Sreedhar en [Sre+99] da un nuevo marco de trabajo en el que presenta y evalúa tres métodos para realizar la destrucción de la forma SSA sin que se produzcan los problemas de la copia perdida y del intercambio. A diferencia de la solución presentada por Briggs en [Bri+98], en estos métodos no se hace uso de las propiedades estructurales de la gráfica de flujo de control o de la gráfica SSA para asegurar que las copias se han colocado correctamente.

Existe una comparación empírica de ambas soluciones (la de Briggs y el tercer método de Sreedhar) en [SIK09], en la que entre otros resultados se concluye lo siguiente:

- En el método de Briggs se inserta un gran número de copias que no se pueden

fusionar. Éstas afectan el rendimiento del código objeto más de lo que incrementan el uso de un mismo registro, que es una preocupación en el método de Sreedhar.

- Cuando hay relativamente pocos registros alojables, el método de Sreedhar que realiza unión (*uniting*) sobre las funciones ϕ, es superior debido al costo dinámico reducido de los derrames (*spills*) y el número reducido de copias ejecutadas. Su tiempo de ejecución es mejor que el método de Briggs por un bajo porcentaje en general y hasta 28 % como máximo.
- Cuando hay relativamente muchos registros alojables, el método de Sreedhar es favorable en la mayoría de los casos, porque el número de copias que se ejecutan es bajo. Su tiempo de ejecución es mejor que el método de Briggs por bajo porcentaje en la mayoría de los casos.

Recientemente, Boissinot en [Boi+09] propone un nuevo enfoque para la destrucción de la forma SSA, basado en fusión de copias y una visión precisa de interferencias, en el que correctud y optimizaciones están separadas. Se menciona que es probablemente correcto, genérico, más fácil de implementar, puede beneficiarse de técnicas presentes en los alojadores de registros, sin parches o casos particulares como en las soluciones previas, mientras que reduce el número de copias generadas. En particular toma en cuenta reglas de renombrado de registros (*register renaming constraints*), esto es, registros dedicados, convenciones de llamadas a funciones, etc. Los experimentos que se han hecho muestran que es dos veces más rápido y consume diez veces menos memoria que el tercer método de Sreedhar, lo que lo hace ideal para uso en compiladores justo a tiempo (*just-in-time*). También se presentan casos especiales (nunca antes presentados en la literatura) en los que la solución dada por Seerdhar falla, por lo que se menciona que la destrucción de la forma SSA debe ser analizada con cuidado. Por todo lo anterior pensamos que la solución que se presenta en [Boi+09], se dice es "probablemente" correcta debido a que sus autores no se quedaron con la certeza de que en el futuro se puedan observar casos particulares en los que las soluciones dadas hasta ahora no fallen. Queda como problema abierto investigar y desarrollar un algoritmo de destrucción de la forma SSA en el que sea posible demostrar formalmente su correctud.

4.8. El algoritmo del producto cartesiano

Antes de describir este algoritmo necesitamos dar dos definiciones.

Definición 4.34. Un *tipo abstracto* describe propiedades o invariantes de objetos.

Definición 4.35. Un *tipo concreto* es un elemento de un conjunto de tipos que implementan a algún tipo abstracto.

Por ejemplo, una expresión que regresa un objeto de la clase ListStack tiene tipo concreto $\{ListStack\}$ y una expresión que regresa un objeto que pudiera ser de la clase ArrayStack o de la clase ListStack tiene como posibles tipos concretos $\{ArrayStack, ListStack\}$. Los tipos concretos proveen información detallada ya que ellos sirven para distinguir incluso diferentes implementaciones del mismo tipo abstracto.

A diferencia de los tipos concretos, los tipos abstractos capturan las propiedades abstractas de los objetos. Los tipos abstractos caracterizan el comportamiento externo observable de los objetos y no distinguen entre las diferentes implementaciones del mismo comportamiento. En nuestro ejemplo, la pila implementada con un lista y la implementada con un arreglo pueden tener una interfaz como la siguiente:

Stack_IF = [push:elmType −> **void**; pop−>elmType].

Stack_IF especifica que los objetos tienen una operación push y una pop, pero no especifica cómo se implementan estas operaciones. Cualquier objeto que defina una operación push y una pop, sin importar cómo éstas están definidas, tiene el tipo de dato abstracto Stack_IF.

Los tipos de datos concretos y abstractos son extremos en el contexto de los sistemas de tipos. Los tipos basados en la definición de clases, como los presentes en C++, no son ni completamente abstractos (es imposible expresar que el tipo de cualquier objeto tenga una operación push y una pop), ni completamente concretos (declarar un objeto de la clase Stack no revela la subclase específica de la cual es un ejemplar).

4.8.1. Polimorfismo

Los lenguajes orientados a objetos obtienen una expresividad significativa a partir del *polimorfismo*, la habilidad para utilizar un objeto de cualquier tipo concreto, siempre que éste satisfaga el tipo abstracto requerido por el contexto. Por ejemplo, los ejemplares de ListStack y ArrayStack se pueden utilizar intercambiablemente en contextos que esperen pilas, sin importar si tal código fue escrito con una u otra implementación en mente.

Haremos una distinción entre dos tipo de polimorfismo:

- *Polimorfismo paramétrico*, que es la habilidad de las funciones de ser invocadas con argumentos de varios tipos.
- *Polimorfismo de datos*, que es la habilidad de una variable de almacenar objetos de diferentes tipos.

Una función *tamaño* que trabaja sobre una lista ligada y una doblemente ligada exhibe polimorfismo paramétrico, mientras que si se utilizan objetos Link para construir listas de Integers, Floats y Strings, su variable contents exhibe polimorfismo de datos debido a que puede almacenar objetos de varios tipos.

Las declaraciones explícitas de los tipos concretos (hechas por el programador) son indeseables debido a que limitan el polimorfismo y la reutilización de código.

Irónicamente, mientras el deseo de maximizar el polimorfismo demanda inferencia de tipos, el polimorfismo es el reto más difícil para los algoritmos de inferencia de tipos.

4.8.2. El algoritmo básico

Palsberg y Schwartzbach presentan el algoritmo básico de inferencia de tipos en [PS91] como un problema de solución de restricciones (*constraint-solving*):

- Derivar un conjunto de restricciones del programa que se está analizando.
- Resolver las restricciones utilizando un algoritmo de punto fijo.
- Llevar la solución de regreso al programa para obtener la información de tipos deseada.

El algoritmo tiene deficiencias cuando analiza código polimórfico, pero constituye el núcleo de los algoritmos mejorados. Tomaremos una visión operacional, presentando la inferencia de tipos como una combinación de análisis de flujo de control y datos sobre el dominio abstracto de tipos. Esta perspectiva de interpretación abstracta permite una correspondencia directa entre analizar un programa y ejecutarlo, haciendo el análisis de los algoritmos más fácil de entender. Refiriéndonos al programa que está siendo analizado como el *programa objetivo*, el algoritmo básico se compone de tres pasos:

1. **Asignar variables de tipo.** El primer paso es asociar una *variable de tipo* con toda variable y expresión en el programa objetivo. Una variable de tipo es simplemente una variable cuyos valores posibles son tipos, es decir, conjuntos. Inicialmente todas las variables de tipo están vacías pero los siguientes dos pasos añaden suficientes tipos de objetos a éstas para hacerlas almacenar tipos válidos para las variables y expresiones con las que están asociadas. Nunca se elimina nada de una variable de tipo. Esta propiedad de monotonicidad es una parte integral del análisis de algoritmos.

2. **Alimentar variables de tipo.** El segundo paso añade un único tipo de objeto a ciertas variables de tipo. El objetivo es capturar el *estado inicial* del programa objetivo, es decir, justo antes de la ejecución. El estado inicial se captura inicializando variables de tipo que corresponden a variables o expresiones en el programa donde inicialmente se encuentran objetos: las variables de tipo se *alimentan*. Por ejemplo, a la variable de tipo de la variable x← null se le asigna el valor {null}. De forma similar, a la variable de tipo de un objeto literal tal como "bycycle" se le agrega **String**. Al finalizar este paso, algunas de las variables de tipo tendrán un único miembro y el resto aún estarán vacías.

3. **Establecer restricciones y propagar.** El tercer paso construye una gráfica dirigida cuyos nodos son las variables de tipo. Las aristas, que se agregan una por una, representan *restricciones*. Una restricción es el tiempo en inferencia de tipos equivalente al tiempo en ejecución del flujo de datos. Por ejemplo, si el programa objetivo ejecuta la asignación x=exp, hay un flujo de datos de exp a x. El flujo de datos indica que cualquier objeto que pueda resultar de evaluar exp también puede estar en la variable x. Para asegurar la validez de los tipos inferidos, cualquier tipo de objeto en el tipo de exp debe también estar en el tipo de x. Cuando el algoritmo encuentra este flujo de datos, agrega una arista de la variable de tipo de exp a la variable de tipo de x, reflejando que $\mathsf{type(exp)} \subseteq \mathsf{type(x)}$.

   Siempre que una restricción (es decir, una arista) se agrega a la gráfica, los tipos de objetos se *propagan* a través de ésta. Como cada vez más y más restricciones se agregan a la gráfica, los tipos de objetos que fueron originalmente únicos en las variables de tipo alimentadas pueden fluir más y más lejos. La propagación pronta de tipos de objetos a través de las aristas hace que las relaciones de subconjuntos tales como $\mathsf{type(x)} \subseteq \mathsf{type(y)}$ siempre se mantengan: si un tipo de objeto se agrega a type(x), éste inmediatamente se propaga a type(y), restableciendo la relación de subconjunto.

   Agregar restricciones hace necesaria más propagación. Lo contrario también se cumple: cuando la propagación hace que crezca el tipo receptor de un envío, el despacho dinámico indica que el envío puede invocar nuevos métodos, por lo que pueden ser necesarias más restricciones para capturar esas invocaciones. Por ello el paso 3 consiste en repetidamente establecer restricciones y propagar, hasta que no se pueda hacer más.

Para garantizar la validez de los tipos inferidos, cada flujo de datos posible en el programa objetivo debe generar una restricción. Diferentes lenguajes de programación cuentan con distintos enunciados que producen flujos de datos, pero algunos ejemplos generales son los siguientes:

- Una asignación genera un flujo de datos del valor de la nueva expresión a la variable asignada.
- El uso de variables (como valores) genera flujo de datos de las variables accedidas a las expresiones que las acceden.
- Los envíos de mensajes generan un flujo de datos de las expresiones que son los argumentos reales (incluyendo al receptor) a los parámetros formales de los métodos invocados. Mas aún, flujos de datos regresan los resultados de los métodos invocados a los envíos de los mensajes.
- Los tipos de datos primitivos como enteros y flotantes y sus operaciones, incluyendo estructuras de control, también generan flujos de datos.

4.8.3. Plantillas

Tal como los métodos suben el nivel de abstracción encapsulando enunciados, las plantillas suben el nivel de abstracción encapsulando restricciones.

Comenzando con la gráfica de restricciones sin estructura producida por el algoritmo básico, podemos dividirla en varias subgráficas, donde cada una corresponde a un método en el programa objetivo. Estas subgráficas son plantillas. Más precisamente, la *plantilla* para un método M es la subgráfica que consiste de:

- Los nodos (variables de tipo) que corresponden a expresiones, variables locales, parámetros formales y el valor de retorno.
- Las aristas (restricciones) que se originan de esos nodos.

Los nodos correspondientes a variables de ejemplares no son parte de ninguna plantilla, ya que las variables de ejemplares no pertenecen a ningún método particular (aunque desde luego tienen un tipo).

4.8.4. Cómo funciona el algoritmo del producto cartesiano

El algoritmo del producto cartesiano (*Cartesian Product Algorithm*, CPA) convierte el análisis de *cada* envío en un análisis de casos. Dado un envío a analizar, CPA calcula el producto cartesiano de los tipos de los argumento reales. Cada tupla en el producto cartesiano se analiza como un caso independiente. Esto hace que la información exacta de tipos esté disponible inmediatamente para cada caso. La información de tipos se utiliza para asegurar tanto precisión (evitando que los tipos se mezclen) como eficiencia (compartiendo casos para evitar análisis redundantes).

La idea detrás del CPA se comprende mejor retomando la analogía entre la ejecución del programa y el análisis del programa. Durante la ejecución del programa, los registros de activación siempre se crean "monomórficamente", simplemente debido a que cada variable contiene un único objeto. Considérese, por ejemplo, un envío polimórfico que invoca un método `max` con un receptor entero o uno flotante. Esto significa que algunas veces el envío invoca `max` con un receptor entero y otras veces con un receptor flotante. En cualquier invocación particular el receptor es un entero o un flotante, pero no puede ser ambos. Resumimos esta observación como:

No existe tal cosa como una llamada polimórfica, solamente sitios de llamadas polimórficas.

CPA explota esta observación. Todos los parámetros formales en las plantillas que el CPA crea, tienen tipos monomórficos.

Veamos un envío como:

```
rcvrExp.max(argExp)!
```

Sea R =type(rcvr) y A =type(**arg**) y supongamos que (de alguna forma) se sabe que

$$R = \{r_1, r_2, \ldots, r_s\} \quad \text{y} \quad A = \{a_1, a_2, \ldots, a_t\}$$

Para analizar este envío, el CPA calcula el producto cartesiano del tipo del receptor y todos los tipos de los argumentos. En el presente caso sólo hay un argumento, por lo que el producto cartesiano es un conjunto de pares. En el caso general es un conjunto de tuplas con $k + 1$ entradas, donde k es el número de argumentos (sin tomar en cuenta al receptor).

$$R \times A = \{(r_1, a_1), \ldots, (r_1, a_t), \ldots, (r_i, a_j), \ldots, (r_s, a_1), \ldots, (r_s, a_t)\}$$

Enseguida, el CPA propaga cada $(r_i, a_j) \in R \times A$ en una plantilla que corresponde a max . Si la plantilla para max ya existe para un par dado (r_i, a_j), éste se reutiliza; si dicha plantilla no existe, se crea una nueva y está disponible para éste y pares futuros (r_i, a_j) en otros envíos. Finalmente, el tipo del envío se obtiene como la unión de los tipos del resultado de las plantillas a las que el envío estaba conectado.

Para obtener un algoritmo eficiente, es necesario mantener repositorios de plantillas por método en lugar de por envío, para asegurar que envíos diferentes, cuyos productos cartesianos tengan tuplas en común, puedan compartir plantillas. Vale la pena enfatizar que los repositorios de plantillas no se llenan "por adelantado". En lugar de eso, las plantillas se van agregando a los repositorios gradualmente conforme nuevas combinaciones de argumentos (tuplas) se vayan encontrando durante el análisis de todos los envíos en el programa objetivo. En efecto, es imposible conocer por adelantado qué plantillas se van a necesitar. Esto sólo va quedando claro gradualmente, conforme el análisis vaya progresando.

El algoritmo del producto cartesiano posee las siguientes dos características:

- Es *preciso* en el sentido que puede analizar cadenas de llamadas polimórficas de profundidad arbitraria sin pérdida de precisión.
- Es *eficiente* porque evita análisis redundante.

Para una descripción más amplia y detallada del algoritmo y algunos de los pormenores relacionados con él, consultar [Age95] y [Age96a].

4.9. Nuestro analizador semántico

Ya que debido a las razones que hemos presentado, no podemos realizar el análisis semántico en la forma en que se haría en un compilador de un lenguaje estáticamente tipado, tenemos que conformarnos con utilizar un algoritmo que realice inferencia de tipos aproximados, el algoritmo del producto cartesiano. Para que éste sea más efectivo

debemos alimentarle el código en forma SSA. Por esto, lo primero que debemos construir es la forma SSA y, posteriormente, aplicar el algoritmo del producto cartesiano.

Generalmente la forma SSA se construye sobre una representación intermedia como el código de tres direcciones, pero exclusivamente después de haber realizado el análisis semántico; se construye con la finalidad de realizar una gran variedad de optimizaciones. En nuestro caso esto no es posible, ya que todavía no hemos realizado el análisis semántico y necesitamos la forma SSA. Por esta razón los compiladores que generalmente se implementan en los libros de texto o durante cursos de compiladores son para lenguajes estáticamente tipados, para no entrar en estas dificultades.

Lo que podemos hacer es implementar una etapa en el compilador, de manera que sea transparente, que tome como entrada el AST que generó el análisis sintáctico y entregue como salida un AST con aproximaciones de tipos. Llamaremos a esta etapa *inferencia de tipos*. Así, cuando en un curso de compiladores no se esté interesado en revisar inferencia de tipos, simplemente se da la implementación de esta etapa y entonces el alumno sólo se encargará de implementar el análisis semántico de la manera tradicional, como si se tratase de un lenguaje estáticamente tipado.

Revisando internamente la etapa de inferencia de tipos, ésta toma como entrada el AST que generó el analizador sintáctico y debe construir la forma SSA, por lo que cabe señalar que no se está construyendo la forma SSA después de haber realizado el análisis semántico y por lo tanto no cumple con una de las condiciones establecidas en [Bri+98]. Sin embargo, en este contexto dicha condición no es significativa y la podemos pasar por alto. También es por esta razón que hemos decidido que esta construcción de la forma SSA no se debe utilizar para realizar la implementación de optimizaciones, pues no cuenta con tipos, sino que en todo caso se puede realizar una nueva construcción de la forma SSA una vez que se haya realizador el análisis semántico, que tenga como objetivo implementar todo tipo de optimizaciones posibles. Por ello, a la primera construcción de la forma SSA la llamaremos *construcción SSA para inferencia de tipos* y a la segunda *construcción SSA para optimizaciones*; de esta forma la inferencia de tipos consistirá en:

1. Generar código de tres direcciones a partir del AST.
2. Construir una gráfica de flujo de control (CFG) a partir del código de tres direcciones.
3. Construir la forma SSA a partir de la CFG.
4. Realizar al algoritmo del producto cartesiano sobre la forma SSA.
5. Destruir la forma SSA y obtener una CFG.
6. A partir de la CFG construir un AST (con las respectivas anotaciones de tipos).

Una vez que hemos realizado lo anterior estamos en posibilidad de realizar el análisis semántico, casi de igual forma que en un lenguaje con tipado estático. La diferencia es que habrá algunos nodos del AST sobre los cuales, dada la imprecisión de los tipos, no se

podrá verificar con certeza su tipo; en dichos nodos se puede realizar una anotación para que una etapa posterior genere el código correspondiente para realizar la verificación de tipos en tiempo de ejecución o, incluso, directamente en ese momento agregar nodos al AST que correspondan a la instrucción de verificación de tipos en tiempo de ejecución. Al final de esta etapa se contará con un AST decorado.

Una vez que se haya realizado el análisis semántico de manera tradicional, podemos evaluar el construir la forma SSA para optimizaciones. En un primer curso de compiladores esta etapa no se implementará a menos que llegue a dar tiempo, pero desde luego será indispensable en un segundo curso de compiladores, por lo que una vez más se puede realizar de forma transparente pero será opcional.

En concreto, la etapa de optimización consistirá en:

1. Tomar el AST decorado.
2. Generar código de tres direcciones.
3. Construir una CFG a partir del código de tres direcciones.
4. Construir la forma SSA a partir de la CFG.
5. Realizar todo tipo de optimizaciones sobre la forma SSA.
6. Destruir la forma SSA y obtener una CFG.
7. Construir un AST (decorado) a partir de la CFG.

Sea o no que se implemente la etapa de optimizaciones, en este punto contaremos con un AST decorado y hemos terminado con el análisis semántico.

Cabe mencionar que en nuestro diseño anterior se requieren realizar diferentes tareas, entre las cuales están: generar código de tres direcciones, construir un gráfica de flujo de control y construir un AST a partir de una gráfica de flujo de control. A continuación presentamos las respectivas secciones que describen cómo realizar cada una de estas tareas.

4.9.1. Generación de código de tres direcciones

Como mencionamos en la sección 4.4, el nivel de abstracción del código de tres direcciones puede oscilar desde alto hasta bajo. En nuestro caso, lo que nos interesa en este punto es trabajar con una representación intermedia donde el flujo de control sea explícito, pero no nos interesa generar código de máquina aún, por lo que queremos que tenga, en la medida de lo posible, un nivel abstracción cercano al de un AST, con excepción, claro, del caso de las estructuras de control que desde luego deben tener un nivel de abstracción menor. Es decir, deseamos que el código de tres direcciones en esta etapa tenga un nivel medio de abstracción. Vale la pena señalar que en este punto las llamadas a funciones se mantienen en alto nivel, es decir, en términos de la expresión que representa el nombre de la función y la expresión que representa los argumentos.

Con nuestro objetivo presente, la correspondencia entre un AST y código de tres direcciones se ilustra con el siguiente ejemplo:

$$(Assign(TargetList(x))(AValue(+(+(5)(7))(3))))$$

(a) AST en notación prefija

$$\begin{aligned} t_1 &= 5 + 7 \\ t_2 &= t_1 + 3 \\ x &= t_2 \end{aligned}$$

(b) Código de tres direcciones

Figura 4.1: AST de la expresión $x = 5 + 7 + 3$ y su correspondiente código de tres direcciones.

Como se puede observar en nuestro ejemplo, en general podemos linealizar un nodo del AST realizando las siguientes tareas:

- Asignar una variable[9] al nodo, –a partir del propio nodo o creando una nueva variable temporal–, o tomar su valor.
- La operación que representa el nodo será el operador de la instrucción en su correspondiente instrucción de tres direcciones. El operando izquierdo será la variable que represente a su hijo izquierdo y el operando derecho será la variable que represente a su hijo derecho, variables que se obtienen realizando recursivamente un recorrido en postorden. Finalmente la variable objetivo (*target*) será la variable asignada en el punto anterior.

En nuestro ejemplo, el subárbol $(+(5)(7))$, al nodo + se le asigna una nueva variable temporal t_1, se realiza un recorrido en postorden y se visita el nodo 5, al cual, por ser un literal entero, se toma directo, es decir, no se le asigna una nueva variable y se toma como un valor inmediato correspondiente al operando izquierdo en la instrucción; luego se visita el nodo derecho 7 y análogamente, al ser un literal entero, su valor se toma como el valor inmediato del operando derecho de la instrucción.

Alternativamente, al visitar el nodo 5 se le puede asignar una nueva variable temporal t_{11}, por lo que se tendría que crear una nueva instrucción $t_{11} = 5$; análogamente, al visitar el nodo 7 se introduciría una nueva variable temporal t_{12} y su correspondiente instrucción $t_{12} = 7$. En este caso, en lugar de la instrucción $t_1 = 5 + 7$ tendríamos la instrucción $t_1 = t_{11} + t_{12}$. La elección de qué alternativa utilizar es decisión del escritor de compiladores, pero se suele utilizar esta última, ya que permite obtener más

[9]En este contexto hablamos de variables en el código de tres direcciones para hacer notar que estamos trabajando, en el código de tres direcciones, con un nivel de abstracción cercano al de un AST (que tiene un nivel de abstracción alto). Sin embargo, como hemos mencionado, es común utilizar el código de tres direcciones como una representación intermedia de bajo nivel, en el que se dice que una instrucción tiene como objetivo registros y no variables. No obstante nosotros consideramos más adecuado utilizar el término variable en este contexto, aunque esta diferencia es sutil y se pueden utilizar ambos términos indistintamente.

oportunidades para optimizar el código, aunque a priori podría parecer que introduce innecesariamente variables temporales.

4.9.2. Construcción de un AST a partir de código de tres direcciones

Para realizar la conversión contraria, es decir, a partir del código de tres direcciones construir un AST, como es necesario para nosotros según nuestro diseño anterior, debe ser claro que podemos seguir el procedimiento inverso.

Ya sea que se hayan realizado optimizaciones o no, cuando sea necesario obtener un AST a partir del código de tres direcciones habrá asignaciones a variables temporales que se utilizan una única vez, por lo que será necesario encontrarlas y convertirlas de regreso en el nodo que originalmente ocupaban en el AST, desapareciendo de esta forma la variable temporal que se había introducido al convertir a código de tres direcciones. Vale la pena resaltar que el procedimiento anterior sólo debe aplicarse si es que la variable se utiliza una única vez; si se utiliza más de una vez entonces es mejor construir el subárbol para dicha instrucción donde la variable almacena el resultado que se utilizará en más de un cálculo, lo que permite utilizar la variable en esos otros cálculos y de esta forma evitar recalcular el valor en cada expresión diferente donde se utilice.

4.9.3. Construcción de la gráfica de flujo de control

La gráfica de flujo de control se construye por medio de los siguientes pasos:

1. Partir el código de tres direcciones en bloques básicos, que son secuencias máximas consecutivas de instrucciones de tres direcciones con las siguientes propiedades:
   a) El flujo de control sólo puede entrar al bloque básico por medio de la primer instrucción en el bloque. Esto es, no hay saltos en medio del bloque.
   b) El flujo de control saldrá del bloque sin parar o saltar, excepto posiblemente en la última instrucción del bloque.
2. Los bloque básicos son los nodos del gráfica de flujo de control, cuyas aristas indican qué bloques pueden seguir a continuación de otros bloques.

Se puede realizar el primer paso mediante el siguiente algoritmo.

Algoritmo 4.2. Partir instrucciones de tres direcciones en bloques básicos.
Entrada: Una secuencia de instrucciones de tres direcciones.
Salida: Una lista de bloques básicos para tal secuencia, en la cual cada instrucción se asigna a exactamente un bloque básico.

Método: Primero se determinan aquellas instrucciones en el código de tres direcciones que son líderes (*leaders*), estos es, la primer instrucción de cada uno los bloque básicos. La instrucción que sea la siguiente al fin del programa en el código de tres direcciones no se incluye como líder. Las reglas para encontrar los líderes son:

1. La primer instrucción en el código de tres direcciones es un líder.
2. Cualquier instrucción que es el objetivo de un salto condicional o incondicional es un líder.
3. Cualquier instrucción que está inmediatamente después de un salto condicional o incondicional es un líder.

Luego, para cada líder, su correspondiente bloque básico consiste del propio líder y todas las instrucciones hasta el siguiente líder exclusive o, en su defecto, hasta el final del programa en el código intermedio.

Una vez que el código de tres direcciones se parte en bloques básicos podemos representar el flujo de control entre ellos mediante un gráfica de flujo de control. Existe un arista del bloque B al bloque C si y sólo si es posible para la primer instrucción del bloque C ser alcanzada inmediatamente después de la ultima instrucción del bloque B. Existen dos formas en que dicha arista puede surgir:

- Existe un salto condicional o incondicional del final de B al inicio de C.
- C está inmediatamente después de B en el orden original de las instrucciones de tres direcciones y B no termina con un salto incondicional.

En este caso B es un predecesor de C y C es un sucesor de B.

Además, usualmente se agregan dos nodos llamados *entrada*(*entry*) y *salida* (*exit*), que no corresponden a instrucciones intermedias ejecutables. Existe una arista de la entrada al primer node ejecutable de la gráfica de flujo de control, esto es, al bloque básico que se forma a partir de la primer instrucción del programa. Si la instrucción final del programa no es un salto incondicional, entonces el bloque que contiene la instrucción del programa es un predecesor de salida, pero también lo es cualquier bloque básico que contiene un salto a código que no es parte del programa[10].

4.10. Analizador semántico de Python

Como lo hemos discutido ya, algunas de las características particulares del diseño del lenguaje Python hacen que la implementación de éste en la forma de un compilador

[10]Según nuestro diseño existe un paso en el que debemos obtener un AST a partir de la gráfica de flujo de control; en su lugar, en la sección 4.9.2 hemos descrito como obtener un AST a partir del código de tres direcciones. Para obtener un AST directamente a partir de la gráfica de flujo de control, se puede realizar exactamente el mismo proceso e incluso se puede sacar provecho de la información explícita del control de flujo que existe en la gráfica o simplemente ignorarlo según sea conveniente.

sea un reto, en particular la implementación del análisis semántico.

Para realizar la implementación de nuestro analizador semántico nos basaremos en el diseño que describimos en la sección anterior.

De esta forma la primer tarea que debemos realizar es generar código de tres direcciones a partir del AST que generó el analizador sintáctico, para posteriormente construir la gráfica de flujo de control. Para poder realizar estas tareas se pueden seguir los pasos descritos en las secciones 4.9.1 y 4.9.3 respectivamente, es decir, primero generar el código de tres direcciones con base en lo descrito en la sección 4.9.1, y una vez obtenido el código de tres direcciones construir la gráfica de flujo de según lo descrito en la sección 4.9.3.

Nosotros, en lugar de seguir esta estrategia, lo que haremos será generar directamente la gráfica de flujo de control a partir del AST, es decir, debemos realizar un recorrido sobre el AST e ir linealizando los nodos del AST en código de tres direcciones, al tiempo que vamos construyendo los bloques básicos junto con sus respectivos sucesores y predecesores, que representan el flujo de control entre cada uno de ellos. De esta forma, en lugar de hacer cada una de estas tareas por separado en dos pasadas independientes, se realizarán al mismo tiempo en una única pasada. Para lo anterior utilizaremos la estrategia descrita en el capítulo 4 de [Mor98] adaptada a las estructuras de flujo de control de Python.

Es importante notar que, en términos generales, se contruye una gráfica de flujo de control por cada función que exista en un programa.

En el caso de Python (que no es un lenguaje imperativo puro al estilo de C), un fragmento de código dado puede pertenecer a una y sólo una de las siguientes estructuras:

1. funciones,
2. clases,
3. métodos y
4. espacio global.

Por lo anterior no sólo debemos tener una gráfica de flujo de control por cada función, sino también por cada clase (ya que puede haber código dentro de una clase que no pertenezca a ninguno de sus métodos); asimismo por cada método (ya que son funciones miembro de una clase) y para aquel código que esté en el espacio global, es decir, que no pertenezca a alguna de las estructuras anteriores.

Vale la pena recordar que Python permite el uso de funciones, métodos y clases anidadas, es decir, que se definan funciones dentro de otras funciones, métodos dentro de otros métodos y clases dentro de otras clases, sin restricción en el nivel de anidamiento. Por ello, para realizar lo anterior, utilizaremos una jerarquía de clases para distinguir la estructura que va a representar una gráfica de flujo de control: de esta forma tenemos una clase `CFG` y las clases `FCFG`, `CCFG`, `MCFG` y `GCFG`, que heredan de `CFG` y que representan a una gráfica de flujo de control de una función, de una clase, de un método y del espacio global respectivamente.

Es claro que debemos almacenar todas las gráficas de flujo de control asociadas con un programa, por lo que podemos, en principio, pensar en almacenar en una lista cada una de éstas. Sin embargo una organización con una mejor estructura y que nos permite reflejar las estructuras anidadas, es utilizar listas de pares donde el primer elemento sea una gráfica de flujo de control mientras que el segundo sea una lista de gráficas de flujo de control; de esta manera, si se tienen estructuras dentro de otras estructuras, por ejemplo dos funciones f_{11} y f_{12} definidas dentro de la función f_1, nuestra lista principal tendrá una entrada con un par cuyo primer elemento sea la gráfica de flujo de control correspondiente al espacio global y la segunda sea una lista que tendrá como elemento un par donde el primer elemento de este par sea la gráfica de flujo de control correspondiente a la función f_1 y la segunda entrada será una lista que almacene un par cuyo primer elemento será la gráfica de flujo de control de la función f_{11}, mientras que el segundo será NULL (que refleja el hecho que en la función f_{11} no hay funciones anidadas dentro de ella), y otro par donde el primer elemento es la gráfica de flujo de control de la función f_{12} y, de igual manera que en el caso anterior, su segunda entrada será NULL. Con esta lista, que almacena pares de gráficas de flujo de control y listas con los mismos elementos recursivamente, podemos reflejar de manera natural las estructuras anidadas de un programa.

Para ir construyendo nuestra gráficas de flujo de control a partir del AST que obtuvimos gracias al analizador sintáctico, utilizaremos una combinación de los patrones Visitor y Builder, nuestros viejos aliados en los capítulos anteriores.

Nuestra estrategia general es utilizar el patrón Builder para poder construir de forma independiente diferentes representaciones de una gráfica de flujo de control, ya sea una representación en memoria, en texto o cualquier otra que nos pueda ser de utilidad en un futuro. Utilizamos un VisitanteConcreto (en nuestro patrón Visitor) que al mismo tiempo funja como Director en nuestro patrón Builder y realice un recorrido sobre el AST. El VisitanteConcreto emite las órdenes necesarias (utilizando la interfaz provista por el Constructor del patrón Builder), para que, con base en el tipo de nodo actual (el nodo que se está visitando), un ConstructorConcreto genere y agregue al bloque básico actual (el bloque básico que esté bajo construcción en ese momento) las instrucciones de tres direcciones correspondientes a ese nodo específico.

Cabe señalar que, como mencionamos, habrá varias gráficas de flujo de control por programa, mientras que sólo hay un AST, por lo que al realizar el recorrido sobre el AST no sólo debemos generar un única CFG, sino una por cada subárbol que represente, ya sea una función, clase, método o el espacio global. Por ello, es natural pensar que su pueden construir de manera concurrente las diferentes gráficas de flujo de control del programa, pues se puede trabajar al mismo tiempo sobre los diferentes subárboles. Podemos utilizar diferentes hilos de control para conseguir lo anterior, teniendo un hilo de control maestro que realice un recorrido sobre el AST y cada vez que encuentre la raíz de un (sub)árbol que represente a una función, método, clase o el espacio global, cree

un nuevo hilo de control que se encargue de construir la gráfica de flujo de control correspondiente a dicho (sub)árbol; de esta manera nuestro hilo maestro sólo se encargará de recorrer el árbol e ir creando nuevos hilos que se encarguen de construir cada una de las gráficas de flujo de control del programa. Desde luego, cada vez que esté lista una de éstas, se debe agregar a nuestra lista de pares <gráfica de flujo de control, lista> que describimos anteriormente.

En nuestro caso, el lenguaje de implementación, C++, no cuenta con soporte nativo de hilos y tampoco dentro de su biblioteca estándar, por lo que es una práctica común, cuando se trabaja dentro del entorno UNIX, utilizar una biblioteca estándar de hilos de este entorno, conocida como *pthreads*, que es la que nosotros podemos usar. Sin embargo, finalmente en el nuevo estándar de C++, conocido como *C++11* (debido a que se liberó en el año 2011), se agregó soporte nativo de hilos por lo que sería recomendable utilizar el nuevo soporte nativo, ya que se gana la portabilidad perdida al trabajar con *pthreads*.

Un bloque básico se implementa por medio de una lista de instrucciones pertenecientes al bloque básico, una lista de bloques básicos que son sus sucesores y una lista de bloques básicos que son su predecesores, además de una etiqueta del bloque. La clase que modela los bloques básicos es BBlock y la que modela las etiquetas de los bloques es Label.

A continuación se mencionan las clases junto con el papel que juegan en el patrón Builder, patrón que utilizamos para construir las gráficas de flujo de control.

La clase CFGBuilder juega el rol Constructor, la clase MCFGBuilder juega el rol ConstructorConcreto y se encarga de construir una representación en memoria de una gráfica de flujo de control; aquí también se debe agregar una nueva clase, es decir, un nuevo ConstructorConcreto por cada nueva representación de un CFG que se quiera construir; la clase CBVisitor juega el rol de Director y finalmente la clase CFG juega el rol Producto.

Para el caso del patrón Visitor los roles permanecen igual que en el capítulo anterior, sólo que aquí se agregó un nuevo VistanteConcreto, rol que juega la clase CBVisitor. Cabe señalar que esta clase juega dos roles al mismo tiempo, uno dentro del patrón Builder y otro dentro del patrón Visitor.

Estudiemos ahora cómo implementar las instrucciones de tres direcciones.

Recordemos que según nuestro diseño de la sección anterior debemos construir dos veces una gráfica de flujo de control, la primera para la etapa de inferencia de tipos y la segunda para la etapa de optimizaciones. Lo anterior significa que debemos generar instrucciones de tres direcciones en dos ocasiones, la primera cuando a partir del AST que generó el analizador sintáctico, se hace un recorrido sobre él para ir generando las instrucciones de tres direcciones, como parte del proceso para construir la gráfica de flujo de control, y la segunda una vez que se tiene el AST decorado sobre el cual ya se ha realizado verificación de tipos y se desea realizar optimizaciones sobre el código. Recordemos también que al irse generando las instrucciones de tres direcciones éstas se

van agregando a su bloque básico correspondiente, que en nuestro caso las va almacenando en una lista. Pensemos ahora en qué estructura debe tener una instrucción bien formada. Como mencionamos, el código de tres direcciones es muy similar al código de una arquitectura RISC, donde las instrucciones son uniformes. En una arquitectura RISC una instrucción está formada por los siguientes elementos:

- Código de operación.
- Primer operando.
- Segundo operando.
- Destino.

El código de operación es un código único que identifica la operación a realizar; el primer y el segundo operando, como su nombre lo indica, son los operandos de la operación, mientras que el destino es donde se almacenará el resultado de la operación. Dependiendo del tipo de instrucción de la que se trate, ésta contará con alguna combinación del primer y segundo operando, y del destino; por ejemplo, hay instrucciones que sólo cuentan con el primer operando, como un instrucción jump que realiza un salto incondicional a la dirección del primer operando.

Es claro que podemos realizar nuestra implementación de las instrucciones con base en la arquitectura anterior, es decir, nuestras instrucciones deben contar con un código de operación, un primer operando, un segundo operando y un destino; pero ¿con eso nos basta? o ¿debemos almacenar otro tipo de información en las instrucciones? Depende... Recordemos que en la primer etapa que se generan las instrucciones, éstas están pensadas para tener un nivel de abstracción lo más cercano como sea posible al AST, pero lo más importante es que en ese momento no contamos con información de tipos, por eso es precisamente que se están generando estas instrucciones de tres direcciones para poder inferir sobre ellas los tipos de sus operandos y de su destino. Por ello debe ser claro que no nos basta sólo con almacenar ambos operandos y su destino, sino que, además, por cada uno de ellos debemos almacenar una variable de tipo, es decir, una variable por cada uno de ellos que almacene su tipo durante la inferencia de tipos y que se inicializa con base en nuestro algoritmo de inferencia de tipos. Por otra parte, la segunda vez (durante la etapa de optimizaciones) que se deben generar instrucciones de tres direcciones, éstas no tienen por qué almacenar en lo absoluto variables de tipo, pues se está suponiendo que en ese momento ya se ha realizado la verificación de tipos y se conoce con certeza el tipo de cada expresión con la que se está trabajando. Esto nos lleva a que, por una parte, en la primer etapa en que se generan instrucciones éstas deben almacenar variables de tipo y en la segunda no, por lo que en principio nuestra implementación de las instrucciones podría almacenar variables de tipo y cuando se generen instrucciones por segunda vez simplemente ignorarlas.

Por otro lado, con base en el argumento anterior, la primera vez que se generan instrucciones, estas instrucciones no pueden ser específicas, pues no se cuenta con información de tipos, es decir, no se puede generar instrucción que imponga de alguna forma el tipo que deben tener sus operandos. Por ejemplo, una instrucción `addi` que realiza un suma con operandos exclusivamente enteros, en su lugar lo único que podemos hacer es generar una instrucción genérica `add` que indique se realizará una suma, pero que no impone qué tipo deben tener sus operandos; esto empata con nuestro diseño en que el nivel de abstracción del código de tres direcciones es lo más cercano posible al del AST. En comparación, cuando se generan instrucciones para la etapa de optimizaciones, esta vez sí ya se pueden generar instrucciones específicas que impongan restricciones sobre los tipos de sus operandos, pues estamos suponiendo que ya se cuenta con el tipo de cada expresión en el programa (que está anotada en el AST).

Además sería deseable que nuestras instrucciones tuvieran alguna organización a nivel lógico que se refleje directamente en la implementación, para que en el futuro, de ser necesario, se puedan hacer modificaciones con base en esta organización. Por esto hemos decidido clasificarlas de acuerdo a los siguientes criterios:

- Número de operandos.
- Tipo de operación.

De acuerdo al número de operandos una instrucción puede ser unaria, binaria o terciaria, mientras que de acuerdo al tipo de operación una instrucción puede ser aritmética, de memoria, de brinco, entre otras.

Además, los operandos y destino de una instrucción sólo pueden ser ya sea un valor inmediato, un registro o una dirección de memoria, que en esta etapa representamos con una etiqueta (ya que en este momento no contamos con la información necesaria para calcular las direcciones de memoria reales en las que se cargará el programa). Bajo el criterio anterior también podemos clasificar las instrucciones, pues por ejemplo, hay desde instrucciones donde tanto sus dos operandos como su destino son necesariamente registros y otras en las que sólo se utiliza un operando y es una etiqueta como el caso de salto incondicional. Cabe señalar que si conocemos qué operandos debe tener una instrucción dada y las restricciones que deben cumplir éstos de acuerdo al criterio anterior, entonces podemos verificar que se cumplan estas reglas y no permitir que se genere una instrucción mal formada.

Por otra parte podemos pensar en cómo formar una instrucción y, siguiendo este enfoque constructivo, podemos decir que una instrucción se forma a partir de un primer operando, un segundo operando y un destino, donde en algunas instrucciones uno o más de ellos no son parte de ella. Así podemos decir que el primer operando, el segundo operando y el destino son las partes elementales a partir de las cuales se construye una instrucción y donde no necesariamente una instrucción está conformada por las tres, sino que puede formarse con una, dos o tres de ellas.

Debemos también tener presente que conforme vamos generando las instrucciones de tres direcciones, éstas se van ir guardando en la lista de instrucciones que son parte del bloque básico actual, por lo que como una primera aproximación podemos pensar en verificar que se cumplan las restricciones mencionadas dos párrafos arriba al construir una instrucción. Podemos entonces analizar el tipo de elementos que debe almacenar la lista de instrucciones de un bloque básico: ¿debe almacenar elementos cuyo tipo represente una instrucción con una interfaz genérica? o ¿debe almacenar elementos cuyo tipo represente a una instrucción concreta? La respuesta es simple: debido a que se almacenarán instrucciones concretas diferentes, la única posibilidad es que almacene elementos cuyo tipo represente una instrucción con interfaz genérica. Tenemos, entonces, dos opciones al crear una instrucción: una es crearla por medio de la interfaz genérica y la otra es crearla por medio de, en su caso, su interfaz concreta. ¿Cuál de las dos debemos elegir? Si nos fijamos que al final de construir nuestra gráfica de flujo de control tendremos que manipular los operandos y destino de las instrucciones, sabemos que debemos contar con una interfaz genérica uniforme que nos permita manipularlos, por lo que si debemos de contar de cualquier forma con dicha interfaz pudiéramos pensar en utilizarla también para crear las instrucciones, utilizando por ejemplo el patrón constructor virtual (*virtual constructor*); sin embargo, recordemos que unos párrafos arriba teníamos como primera aproximación poder verificar ciertas restricciones al construir nuestras instrucciones, lo cual, utilizando esta interfaz, resulta imposible, pues al ser una interfaz común a todas las instrucciones no podemos imponer restricciones particulares a instrucciones concretas diferentes. En cambio, en nuestra segunda opción, al crear una instrucción a través de su interfaz concreta sí podemos verificar las restricciones deseadas, pero no contamos con una interfaz genérica para poder manipular los operandos y el destino cuando sea necesario. En conclusión, con nuestra primer opción obtenemos uniformidad pero perdemos seguridad, mientras que en la segunda contamos con seguridad pero no con uniformidad. Esto nos lleva a pensar en una solución que integre tanto uniformidad como seguridad, sin que el uso de una implique la ausencia de la otra y viceversa.

Hemos desarrollado una implementación que tome en cuenta todos los criterios anteriores. Desde luego, como esta solución es a nivel de implementación, es deseable que ésta saque todo el provecho posible del lenguaje de implementación y es claro que si el lenguaje de implementación nos ofrece características avanzadas que permitan que nuestro diseño se implemente de forma directa y limpia, entonces debemos utilizarlas.

En nuestro caso el lenguaje de implementación es C++; tomemos como punto de partida el enfoque constructivo en el que una instrucción se construye a partir de un primer y segundo operando, y un destino, y no necesariamente debe contar con los tres sino se puede construir con uno, dos o tres de ellos, dependiendo de la instrucción. Para este punto, C++ cuenta con una característica avanzada que nos permite realizar la implementación de nuestro diseño de forma directa y simple, la herencia múltiple, por lo que

podemos tener una clase que represente al primer operando, otra al segundo y una más al destino y así, cuando se requiera construir una instrucción, la clase que represente a la instrucción que se desea generar simplemente debe heredar de aquellas clases que representen los elementos que la conforman. Por ejemplo, si es una instrucción de salto incondicional sólo debe heredar de la clase que represente al primer operando, mientras que si es una instrucción que realiza una suma entonces su respectiva clase tiene que heredar de las tres clases que representan, respectivamente, al primer operando, al segundo y al destino.

Por otra parte, con base en el criterio de que dependiendo del tipo de instrucción ésta tiene restricciones en cuanto a que sus elementos pueden ser un valor inmediato, un registro, o una dirección, lo que se puede hacer es que exista una clase por cada restricción de cada uno de los elementos. Así habrá, por ejemplo, una clase que represente que el primer operando sólo puede ser un valor inmediato. Con base en esto debemos modificar lo descrito en el párrafo anterior, ya que esta clase debe heredar de la clase que represente al primer operando y, posteriormente, si la instrucción debe estar compuesta entre otros elementos de un primer operando que sea un inmediato, entonces la clase que represente a dicha instrucción tiene que heredar de la clase que representa al primer operando con valor inmediato. Se debe seguir la misma lógica para cada uno de los operandos y el destino y con cada uno de los posibles valores: inmediatos, registros y etiquetas.

En cuanto a la clasificación a nivel lógico hay una clase que representa la clasificación de acuerdo al número de operandos y otra que representa la clasificación según el tipo de operación que realiza una instrucción.

Para la clasificación a nivel lógico por número de operandos se debe contar con una clase por cada posible tipo de instrucción. Así, debe haber una clase que represente las instrucciones unarias, una que represente las binarias y una más a las terciarias. Todas estas deben heredar de la clase que representa la clasificación según el número de operandos. Además debe ser claro que una instrucción unaria se forma a partir de un primer operando, por lo que la clase que representa a una instrucción unaria adicionalmente debe heredar de la clase que representa al primer operando y se debe hacer lo análogo para las clases que representan a las instrucciones binarias y terciarias; también debemos tomar en cuenta que una instrucción, ya sea unaria, binaria o terciaria, puede contar o no con un destino, por lo que en un nivel inmediatamente abajo en la jerarquía de herencia debe haber una clase que represente a las instrucciones unarias que no cuenten con un destino y que herede de la clase que representa a las instrucciones unarias; y una clase que represente a las instrucciones unarias que cuentan con un destino y que herede de la misma clase y de la clase que representa un destino; es claro que se debe hacer lo análogo para las instrucciones binarias y terciarias.

Por otra parte, en lo que se refiere a nivel lógico, con base en el tipo de operación que realiza una instrucción debe haber una clase que represente cada uno de los dife-

rentes tipos de operación en los que se puede clasificar una instrucción. Así, habrá una clase que represente a las instrucciones que realicen operaciones aritméticas, otra que represente a las que realicen operaciones de memoria y, análogamente, por cada tipo de operación diferente. Desde luego cada una de estas clases deben heredar de la clase que representa a la clasificación de acuerdo al tipo de operación de las instrucciones.

Recordemos además que se deben generar instrucciones genéricas durante la primera etapa e instrucciones específicas durante la segunda. Respecto a este punto debe existir una clase por cada instrucción, ya sea ésta genérica o específica. La diferencia radica en que en las clases que representan instrucciones genéricas, sus métodos y constructores no deben imponer restricciones en cuanto al tipo específico que deben tener los operandos y destino de las instrucción, mientras que en las clases que representan a las instrucciones específicas sí se deben imponer estas restricciones.

Además es importante señalar que, como mencionamos anteriormente, deseamos poder construir diferentes representaciones de una gráfica de flujo de control de forma transparente, por lo que debemos, en particular, ser capaces de construir de manera transparente diferentes representaciones de una instrucción. Respecto a este punto, siguiendo la idea del patrón Builder, debe haber una clase que represente la interfaz de la instrucción y debe haber una clase por cada representación diferente de la instrucción que se quiera construir y que, desde luego, debe heredar de la clase que representa la interfaz. Se puede pensar en construir una instrucción utilizando la interfaz mientras que realmente se está utilizando una clase concreta para construir la representación que se desee. En este punto podemos utilizar un constructor virtual.

Recordemos además que algo muy importante es que la primera vez que se generan las instrucciones, éstas, además de los operandos y destino correspondientes (dependiendo del tipo de instrucción), deben almacenar las variables de tipo que se utilizarán en el algoritmo de inferencia de tipos, por lo que unos párrafos arriba mencionamos que, en principio, las instrucciones deben almacenar variables de tipo y en la segunda ocasión que se generen las instrucciones simplemente ignorarlas. Esto es una solución indeseable, pues lo que quisiéramos es que si no se van a utilizar las variables de tipo éstas no formen parte de las clases; pero, por otra parte, son necesarias la primera vez que se generan la instrucciones, es decir, lo que deseamos es que la primera vez que se generan las instrucciones, las clases respectivas que representan a las instrucciones cuenten además con variables de tipo, mientras que en la segunda ocasión estas mismas clases no cuenten con ello.

Una primera solución sería hacer un conjunto de clases con variables de tipo que se utilicen la primera ocasión y otro conjunto sin variables de tipo que se utilice en la segunda, lo cual no es satisfactorio aún, ya que sería duplicar el código, excepto que en una ocasión incluyendo variables de tipos y la otra sin estas variables. Como segunda aproximación lo que quisiéramos es poder modificar una clase para que la primera vez que se utilice ésta cuente con variables de tipo, pero la segunda modificarla para que las

variables de tipo ya no formen parte de ella. Este tipo de modificaciones a la estructura interna de una clase es posible en tiempo de ejecución en lenguajes como Python, *Ruby* y en particular en Java a través del API *Reflection*. La desventaja de utilizar este tipo de características en estos lenguajes es que el tiempo de ejecución se degrada, es decir, la modificación de una clase en tiempo de ejecución es "cara". Es aquí donde explotaremos una de las características avanzadas de C++ que es difícil encontrar en otros lenguajes, la especialización de plantillas (*template specialization*). Mediante el uso de especialización de plantillas un programador puede establecer diferentes definiciones de una misma clase y decidir cuál de ellas utilizar según se requiera en un momento específico, lo que nos permite realizar exactamente lo que nosotros queremos, pues podemos para una clase que represente una instrucción dar una definición que contenga variables de tipos y otra que no contenga variables de tipos en lo absoluto. De esta manera, la primera vez que se generen instrucciones se utiliza la definición con variables de tipos y la segunda se utiliza la definición sin variables de tipos, la cual es la solución ideal a nuestro problema. La gran ventaja de la especialización de plantillas de C++ sobre la posibilidad de modificar las clases en tiempo de ejecución, presente en otros lenguajes, es que la elección de qué definición de la clase utilizar se hace en tiempo de compilación y por lo tanto no se tiene una penalización en el tiempo de ejecución en lo absoluto; la desventaja es que requiere las diferentes definiciones de la clase necesariamente antes de la ejecución del programa, lo que es más restrictivo y por tanto menos flexible que modificar la estructura de la clase en tiempo de ejecución.

Recordemos también que deseamos que cuando se construya una nueva instrucción se realicen verificaciones de las restricciones que se deben cumplir para que dicha instrucción esté bien formada, ya sea en el caso de las instrucciones genéricas –que se verifiquen en su caso sus operandos y destino y que cumplan que con ser un inmediato, un registro o una etiqueta según sea el caso– y en el caso de las instrucciones específicas; además de lo anterior, que tengan en su caso el tipo que se espera;[11] pero por otra parte necesitamos una interfaz uniforme para poder manipular las instrucciones una vez que se ha finalizado la construcción de la gráfica de flujo de control.

Lo que haremos es que siempre que se requiera construir una instrucción, construirla por medio de su interfaz particular, es decir, por medio de la clase que representa la interfaz de una instrucción concreta, lo que nos permite tanto realizar la verificación de las restricciones necesarias y construir diferentes representaciones de la instrucción, como utilizar la interfaz genérica uniforme cuando se requiera manipular las instrucciones;

[11] En principio imponer el tipo de los operandos y destino de la instrucción no es necesario en este punto, porque estamos suponiendo que en este momento ya se realizó la verificación de tipos, pero es una forma de asegurar que, en efecto, la instrucción se está formando con los operandos y destino con el tipo que espera la instrucción específica, para de esta forma poder verificar que es consistente lo que se está generando a partir del AST, con la forma correcta en la que se debe construir una instrucción. Lo anterior es para asegurar que el programador no cometa errores al escribir el código que construya las instrucciones de tres direcciones a partir del AST.

de esta forma, contaremos con la dos interfaces, una para construir las instrucciones con seguridad y otra para manipularlas con uniformidad, y de esta forma contaremos tanto con seguridad como con uniformidad.

Tomando en cuenta todos los criterios anteriores, se ha desarrollado como resultado una jerarquía de clases autoreconfigurable con base en el tipo de características con las que se desea que cuenten las instrucciones. Por ejemplo, cuando es necesario generar instrucciones para realizar la inferencia de tipos, se debe seleccionar que las instrucciones cuenten con variables de tipo, que estén clasificadas de acuerdo a su número de operandos pero no de acuerdo con el tipo de operación que realizan y que no cuenten con una interfaz uniforme para manipular los operandos (porque en principio no se tienen que manipular los operandos durante la inferencia de tipos, necesidad que surge durante la etapa de optimizaciones). Lo anterior se realiza seleccionando la constante correcta –de una enumeración conformada por un conjunto de constantes que representan cada una de las diferentes combinaciones de características válidas para las instrucciones– que representa la combinación de características que deseamos. Al pasar como argumento a la plantilla de la clase que representa la instrucción a construir, esta plantilla, a su vez, propaga la constante a sus ascendientes por medio de su clase padre (que como sabemos forma parte de la declaración de la clase) y, de esta manera, se termina obteniendo una jerarquía de clases que refleja exactamente las características deseadas.[12] Desde luego, cuando se requiera generar instrucciones con otras características lo único que se tiene que hacer es seleccionar la constante correcta que represente las características que se necesiten. Con lo anterior, además, obtenemos flexibilidad porque, por ejemplo, si en un futuro se decide realizar optimizaciones durante la inferencia de tipos, se pueden agregar las características necesarias para llevarlas a cabo durante la inferencia de tipos; en particular, se puede agregar uniformidad simplemente seleccionando una constante diferente al momento de construir una instrucción, con lo que hemos obtenido un diseño limpio, elegante, flexible y que permite escalar, sin tener una penalización en tiempo de ejecución utilizando características avanzadas de nuestro lenguaje de implementación.

La declaración de la jerarquía de clases anterior se encuentra en el archivo `inst.h`.

Nótese que como resultado de utilizar herencia múltiple se tiene la posibilidad de tener más de una vez una misma clase como clase ancestro. Por ejemplo, si se tiene una clase A y dos clases B y C que heredan de A y finalmente una clase D que hereda de ambas B y C, entonces D tendrá como clase ancestro a A dos veces; en este caso, en C++ D heredará dos veces de A, por lo que heredará dos veces los métodos de A; si en lugar de esto lo que se quiere es que se unifique la herencia y que D sólo herede una única

[12] En realidad cada vez que se pasa como argumento a la plantilla una constante diferente, que representa la combinación de características con las que se desea que cuenten las instrucciones, se obtiene una jerarquía de clases diferente que refleja exactamente las instrucciones con la combinación de características seleccionada.

vez los métodos de A, entonces A debe ser una clase base virtual (*virtual base class*); el soporte de clases bases virtuales (que se deriva de contar con herencia múltiple) es otra de las características presentes en C++ que es difícil encontrar en otros lenguajes de programación.

Construcción de la forma SSA

Toca el turno a la construcción de la forma SSA. Como primer paso tenemos que construir el árbol dominador de cada una de las gráficas de flujo de control, para lo cual existen varios algoritmos. Es por esto que hemos decidido utilizar el patrón Estrategia (*Strategy*) en este punto, pues en general este patrón nos permite intercambiar de forma transparente el algoritmo que realiza una cierta tarea. En este caso, nos permite realizar la implementación de diferentes algoritmos donde cada uno de éstos construye el árbol dominador de una gráfica de control de flujo y poder elegir cuál es el que queremos utilizar en un momento determinado.

Nosotros sólo hemos implementado el algoritmo 4.1, pero el uso del patrón Strategy nos permite que si en un futuro se desea probar, y en su caso comparar, uno o más algoritmos que calculen el árbol dominador de una gráfica de flujo de control, sea fácil, independiente y transparente agregar la implementación de cada uno de éstos y utilizar la que se elija para construir el árbol dominador.

Una vez más utilizaremos el patrón Builder para poder construir de forma transparente diferentes representaciones del árbol dominador.

Ya que se ha realizado lo anterior, continuamos con la construcción de la forma SSA utilizando nuevamente una combinación del patrón Strategy y el patrón Builder, donde Strategy nos permitirá, como ya mencionamos, utilizar diferentes algoritmos para construir la forma SSA, mientras que Builder nos permite construir diferentes representaciones de la forma SSA.

Tanto para el árbol dominador como para la gráfica en forma SSA se utilizan representaciones diferentes e independientes entre ellas y la gráfica de flujo de control. De esta forma, una vez construida la forma SSA no es necesario mantener la gráfica de flujo de control ni el árbol dominador.

Una vez construida la forma SSA, es tiempo de iniciar la inferencia de tipos. En esta parte una vez más utilizaremos el patrón Strategy para poder utilizar diferentes algoritmos de inferencia de tipos. En este momento sólo se implementará el algoritmo del producto cartesiano que presentamos en la sección 4.8. Para esto se construirá una representación diferente de la forma SSA que será la que genere el algoritmo del producto cartesiano, ya que de esta forma será totalmente independiente la implementación del algoritmo del producto cartesiano de la representación de la gráfica en forma SSA que construya un algoritmo de construcción de la forma SSA.

En la implementación del algoritmo del producto cartesiano utilizaremos el patrón Builder para construir diferentes representaciones de la forma SSA junto con los tipos

inferidos. También se utilizará el patrón Observador (*Observer*) –cuyo propósito es definir una dependencia de uno a muchos entre objetos, tal que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente durante la propagación de los valores de las variables de tipo–. Así, cada vez que una variable de tipo reciba un nuevo valor, es decir, cada vez que cambie su estado, se notificará, con base en las restricciones, a todas las otras variables de tipo que deben recibir ese valor y se actualizarán con dicho valor automáticamente, es decir, el valor se propagará.

A partir de la forma SSA con tipos inferidos construiremos un AST, siguiendo la estrategia que presentamos en la sección 4.9, para lo cual, lo primero que haremos es destruir la forma SSA y así obtener las diferentes gráficas de flujo de control que conforman el programa. En este paso haremos uso del patrón Strategy, para que de esta forma tengamos con la flexibilidad de poder contar con diferentes algoritmos de destrucción de la forma SSA. Por el momento nosotros únicamente implementaremos el algoritmo ingenuo que mencionamos en la sección 4.7.3, pero en un futuro se pueden implementar los otros algoritmos mencionados en la sección o incluso los nuevos algoritmos que pudieran llegar a surgir; desde luego este algoritmo hará uso del patrón Builder que mencionamos anteriormente para construir las correspondientes gráficas de flujo de control. Una vez que hemos obtenido las gráficas de flujo de control, toca el turno a construir el AST a partir de éstas, utilizando la estrategia de la sección 4.9.2. Para esto utilizaremos el patrón Visitor aplicado a las instrucciones, para que cada vez que se visite una instrucción diferente se conozca el nodo del AST correspondiente que se debe generar. Además, en este momento se anotará este nodo con la información de tipos obtenida durante la inferencia de tipos. Desde luego, tal como describimos en la sección 3.6, también contamos con el patrón Builder para construir de forma transparente diferentes representaciones del AST. Una vez concluido lo anterior estamos listos para realizar el análisis semántico como se realizaría en un compilador de un lenguaje explícita y estáticamente tipado.

Para llevar a cabo las tareas que comúnmente se realizan durante el análisis semántico (entre ellas la verificación de tipos) utilizaremos un VisitanteConcreto que realice cada una de ellas sobre el AST. En particular utilizaremos un VisitanteConcreto para implementar la verificación de tipos.

Mientras que la implementación de la tabla de símbolos (que entre otras tareas se utiliza durante la verificación de tipos) está basada en lo descrito en [GJR79], una vez más hemos utilizado el patrón Builder para construir diferentes representaciones de la tabla de símbolos. También hemos separado la interfaz de la implementación para que en un futuro, si se requiere tener diferentes implementaciones de la tabla de símbolos, lo único que se tiene que hacer es agregar una nueva clase que implemente dicha interfaz por cada implementación diferente que se requiera.

Con esto damos por concluido el presente capítulo.

CAPÍTULO 5

Generación de código

Conocimientos supuestos

Para la etapa de generación de código estamos suponiendo que el lector conoce lo siguiente:

1. Definiciones de:
   - Número cromático.
   - Gráfica perfecta, cordal y clanes (*cliques*).
2. Conocimientos de:
   - Arquitecturas de computadoras.
   - Coloración de gráficas.
   - Lenguaje ensamblador.
3. Estructuras de datos:
   - Pilas de ejecución.
   - Heap (*montículo*).

Estos temas se incluyen en cursos de lenguajes de programación y arquitectura de computadoras y estructuras de datos.

5.1. Introducción

En algún momento el compilador debe generar código de máquina que se ejecute en la arquitectura objetivo. Hasta el momento todos las etapas que se han realizado han sido independientes de la arquitectura.[1] Dependiendo de los objetivos específicos de un

[1] En realidad queremos decir independientes de la plataforma.

compilador, éste debe ser capaz de generar código de máquina para una única arquitectura o para varias de ellas, aunque hoy día lo más común es que genere código de máquina para varias arquitecturas. Si un compilador debe ser capaz de generar código para diferentes arquitecturas es importante que cuente con un diseño que permita de forma sencilla que sea *reobjetivizable*, es decir, que permita de forma modular añadir al compilador la sección de código necesaria para que sea capaz de generar código de una arquitectura específica. De esta forma, cuando se desee soportar una nueva arquitectura únicamente se debe agregar una nueva sección de código que permita que el compilador genere código de máquina para la nueva arquitectura a soportar.

Para que un programa sea capaz de ejecutarse requiere de un ambiente que dé soporte para que se realice esta ejecución. Entre algunas de las tareas que deben realizarse están: asignar memoria para el programa y llevar el seguimiento de cuál es la instrucción actual a ejecutar, entre otras. Al sistema que ofrece todo este soporte para que se realice la ejecución de un determinado programa se le conoce como *sistema en tiempo de ejecución* (*run-time system*).

Por otra parte es claro que el compilador debe ser capaz de generar código de máquina, es decir, debe ser capaz de generar instrucciones de máquina que la máquina objetivo sea capaz de ejecutar; como bien se sabe, una arquitectura cuenta con un número limitado de registros, por lo que las instrucciones que genere el compilador claramente deben utilizar únicamente estos registros. Además, la gran mayoría de las arquitecturas hoy en día son arquitecturas superescalares (*superscalar*) o con una palabra de instrucción muy grande (*Very Long Instruction Word*, *VLIW*), en donde en ambos tipos es fundamental[2] el orden en que se ejecutan las instrucciones para sacar el mayor provecho posible a la arquitectura[3] y, como consecuencia, el programa se ejecute con una mayor eficiencia en tiempo de ejecución.

Con base en lo anterior la etapa de generación de código del compilador se compone de las siguientes tres partes:

1. Selección de instrucciones (*instruction selection*).
2. Alojamiento en registros (*register allocation*).
3. Planificación de instrucciones (*instruction scheduling*).

En la etapa de selección de instrucciones, como su nombre lo indica, el generador de código selecciona las instrucciones de máquina que se deben generar a partir de una representación intermedia del programa. Como mencionamos, una arquitectura cuenta con un número limitado de registros, por lo que en principio dichas instrucciones seleccionadas deben utilizar únicamente estos registros, es decir, en principio le correspondería al seleccionador de instrucciones emitir instrucciones que hagan uso únicamente de

[2] Aunque más en las arquitecturas VLIW

[3] En el caso de las arquitecturas VLIW el compilador juega un papel fundamental, ya que es el encargado de construir la palabra de instrucción grande de la mejor manera.

tales registros, lo que haría que el seleccionador de instrucciones se volviera mucho más complejo, pues además de seleccionar las instrucciones adecuadas tendría que determinar qué valores y en qué momento es necesario mantener en los registros y, con base en esto, emitir instrucciones de derrame.[4] Es por esto que dicha labor suele dividirse en dos etapas completamente independientes, la primera en la que el seleccionador de instrucciones emite instrucciones suponiendo un número ilimitado de registros –conocida simplemente como *asignación de registros*– y la segunda en la que se determina qué valores deben residir en registros en cada punto del programa, para con base en esto se emitan las instrucciones de derrame necesarias para que se logre tal objetivo; a esta etapa se le denomina *alojamiento en registros*.

Además, como mencionamos, es fundamental que las instrucciones generadas estén ordenadas de la mejor manera posible para sacar el mayor provecho de la arquitectura en la que se ejecutará el programa. En este punto es importante resaltar que durante el alojamiento en registros se generan instrucciones de derrame, por lo que debemos preguntarnos ¿en qué punto debe realizarse la etapa de planificación de instrucciones? ¿después de la selección de instrucciones?, ¿antes o después del alojamiento en registros? Es claro que si se realiza antes del alojamiento en registros entonces no se tomarán en cuenta las instrucciones de derrame que se generan durante el alojamiento en registros. Por otra parte, si se realiza después del alojamiento en registros entonces se tomarán en cuenta las instrucciones de derrame que se generan durante el alojamiento en registros, pero se pueden perder oportunidades de optimización que se tenían antes de que se realizara el alojamiento en registros. Es por esto que la tendencia en la actualidad es que se realice dos veces la etapa de planificación de instrucciones, una antes de que se realice el alojamiento en registros y una después de éste, pues se ha observado en la práctica que al hacerlo de esta manera se obtienen una mejor planificación de instrucciones y un mejor alojamiento en registros. Aunque desde luego se puede realizar una única vez la planificación de instrucciones, ya sea antes o después del alojamiento en registros.

5.2. Sistemas en tiempo de ejecución

Un sistema en tiempo de ejecución es un sistema que provee los mecanismos necesarios que un programa requiere durante su ejecución.

Para que un programa se ejecute es necesario que el sistema operativo lo cargue en memoria y que previamente le haya asignado un espacio de memoria, tanto para su código ejecutable como para los datos del programa.

[4]Cuando decimos instrucciones de derrame nos referimos a las instrucciones que se deben generar cuando no hay registros disponibles debido a que todos se encuentran ocupados y se debe cargar un nuevo valor en alguno de ellos, por lo que será necesario almacenar en memoria el valor de algún registro mediante una instrucción “guardar” y una vez que requiera utilizar de nuevo el valor que se guardó debe cargarse en algún registro mediante una instrucción “cargar”.

Además, es necesario contar con un mecanismo para dar soporte a las llamadas a funciones y que en éste se puedan reflejar las diferentes estrategias de evaluación ofrecidas por el lenguaje fuente[5]. Generalmente las arquitecturas cuentan con soporte en hardware para este propósito, es decir, cuentan con una pila de ejecución y con las instrucciones necesarias para poder realizar una llamada a función, aunque desde luego es labor del compilador saber cómo implementar la semántica del lenguaje fuente utilizando de la mejor manera posible el soporte ofrecido en hardware. No siempre el hardware cuenta con el soporte necesario para implementar en él de forma directa algunas de las características presentes en la semántica del lenguaje; en esos casos el compilador debe realizar un mayor trabajo para implementar estas características utilizando únicamente el soporte ofrecido en hardware.

Sabemos que para poder realizar una llamada a función es necesario pasarle a ésta los argumentos que espera.

Como dijimos, las arquitecturas cuentan con una pila para implementar las llamadas a funciones, por lo que tenemos dos opciones para pasar los argumentos, la primera es pasar los argumentos en los registros y la segunda es en la pila. Ambas soluciones tienen ventajas y desventajas. Cuando se pasan argumentos en los registros se obtiene una mayor eficiencia, pero se tiene la desventaja que existe la posibilidad de que una función tenga más parámetros que registros disponibles o que los registros existentes se utilicen para otros propósitos; en este caso es posible pasar tantos argumentos como registros haya disponibles en los registros, y pasar el resto de los argumentos en la pila; la ventaja de pasar los argumentos en la pila es que, en principio, no tenemos límite en cuanto al número de éstos; la desventaja es que toma un mayor tiempo de ejecución.

Además, cuando una función f hace a su vez una llamada a una función g, en el momento que f llama a g el contenido de los registros son los valores que almacenó ahí la función f y con los que se encuentra trabajando. Pero es claro que al realizar la llamada a g los valores de esos registros se pueden modificar durante la ejecución de g, por lo que cuando termine la llamada a g y f continúe su ejecución, f podría ver valores distintos a los que ella dejó en los registros. Para evitar este problema, en principio f puede tomar una actitud conservadora y siempre antes de llamar a función guardar el contenido actual de los registros en la pila, y siempre que termine la llamada recargar el contenido de los registros; en otro caso g puede ser educada y al iniciar su ejecución guardar en la pila el contenido de los registros tal como los encontró y antes de regresar restablecer ese contenido en los registros respectivamente; alternativamente se puede realizar una estrategia mixta en la que f guarde un cierto conjunto de registros seleccionado previamente y de igual forma g guarde otro conjunto de registros también seleccionado previamente. En general a la función a la que llama se le denomina *invocodora* (*caller*), mientras que a la función llamada se le denomina *invocada* (*callee*). Con

[5]De nuevo cabe la posibilidad que haya un único lenguaje fuente o varios de ellos, por lo que al decir "lenguaje fuente" estamos contemplando ambas posibilidades.

base en lo anterior, si un registro r debe ser guardado por la función invocadora se le denomina *registro guardado por la función invocadora* (*caller-saved register*), mientras que si r debe ser guardado por la función invocada se le denomina *registro guardado por la función invocada* (*callee-saved register*).

Asimismo, en principio una función debe regresar el valor que calculó, que también se puede regresar en un registro o en la pila.

Como podemos observar a partir de lo anterior, hay diferentes estrategias que se pueden utilizar en la práctica, pero sería deseable que todos los compiladores utilizaran la misma, ya que si en algún momento una función f se compila en un compilador c y desea utilizar una función g de una biblioteca que se compiló con un compilador d, si no se utilizó la misma estrategia en cuanto a los situaciones anteriores, entonces el resultado claramente será incorrecto, pues por ejemplo g puede estar esperando que se le pasen los argumentos en la pila y f pasarlos en los registros y para hacer el ejemplo aun más caótico g podría regresar el resultado en un registro, mientras que f lo espera en la pila.

Para dar solución a este tipo de problemas (y algunos otros) que podrían llegar a presentarse, se ha definido un estándar para cada una de las diferentes plataformas[6] conocido como *interfaz binaria de aplicación* (*Application Binary Interface*, *ABI*), en el que se establecen las convenciones a seguir en cada una de las situaciones planteadas anteriormente (y otras).

Cada invocación de una función cuenta con sus propios datos locales que son de utilidad únicamente durante la ejecución de esa invocación de la función. Por otra parte, sabemos que se pueden almacenar datos en la memoria de la máquina. A partir de lo anterior, cuando se ejecute una invocación de una función debemos ser capaces de almacenar sus datos locales en memoria, y una vez que la invocación ha terminado, simplemente descartar estos datos pues ya no son de utilidad. En principio, tenemos la libertad de decidir en dónde y cómo almacenar estos datos locales, pero debemos siempre cumplir con la semántica que acabamos de delinear. Una estructura de datos que es particularmente útil para este propósito es una pila, ya que cuenta con las operaciones *push* y *pop* por lo que siempre que se requiera almacenar un dato local en memoria simplemente se debe hacer un push de ese dato; y siempre que se requiera descartar un dato local simplemente se debe hacer un pop. Por otra parte, la memoria principal de una máquina se puede pensar como un arreglo (muy largo) con n entradas, donde n es la cantidad de memoria con la que cuenta la máquina y que, debido al mecanismo de memoria virtual ofrecido por el sistema operativo, se puede pensar como si se contara con una cantidad de memoria ilimitada. Cada entrada del arreglo ocupa un byte y cada una de estas entradas tiene un dirección de memoria diferente. Cuando se almacena un dato que ocupa más de un byte, simplemente ocupará más de una entrada. Con base

[6] Al decir plataforma generalmente nos referimos a la combinación de un sistema operativo y una arquitectura específica.

en lo anterior podemos delimitar una sección del arreglo que representa a la memoria principal para utilizarla como una pila; una vez hecho lo anterior, debe ser claro que la pila puede crecer y decrecer según se agreguen o eliminen datos en ella, por lo que es necesario saber qué localidad de memoria es, en un momento determinado, el tope actual de la pila; para esto se asigna un registro encargado únicamente de esto; a este registro se le conoce como *apuntador de la pila* (*stack pointer*). Generalmente, el inicio de la pila se encuentra en una dirección d asignada por el sistema operativo y la pila crece hacia abajo,[7] es decir, hacia las direcciones de memoria menores, por lo que cuando se almacena un dato local por medio de un push el apuntador de pila decrece, lo que podría resultar extraño a primera vista; por la misma razón cuando se realiza un pop el apuntador de pila aumenta.[8]

Hemos dicho que cada invocación de una función cuenta con sus propios datos locales, por lo que si hace uso de la pila para guardarlos, es necesario delimitar y asignar una sección de la pila a cada una de las diferentes invocaciones a función. A la sección de la pila asignada a la invocación de una función se le conoce como *marco de pila*[9] (*stack frame*). De esta forma se creará un marco de pila por cada invocación a función durante la ejecución de un programa. Veamos cómo es que se crean estos marcos de pila.

Supongamos que se encuentra en ejecución una función f que va a invocar a una función g. Dependiendo de dónde deben pasarse los argumentos, f debe calcular previamente los argumentos que va a pasar a g y después colocarlos ya sea en registros o en pila. Si suponemos que los argumentos deben pasarse en la pila, el apuntador de pila en ese momento contiene la dirección actual del tope de la pila, por lo que se debe hacer un push con cada uno de los argumentos que se van a pasar a g. Desde luego el apuntador a la pila cambiará de acuerdo a lo anterior. Una vez realizado esto se debe almacenar en algún lugar la dirección de la siguiente instrucción que se debe ejecutar una vez terminada la llamada, a la que se le conoce como *dirección de regreso* (*return address*). Esta dirección se puede guardar en la pila (como en el caso de la arquitectura x86) o en un registro especial asignado para esto (como en el caso de la arquitectura SPARC). Cabe señalar que guardar la dirección de regreso es algo fundamental, pues si no se hiciera así, sería imposible determinar dónde continuar la ejecución del programa una vez terminada la llamada a la función.

Una vez guardada la dirección de regreso, se debe realizar un salto a la primer instrucción de la función g. En este momento g toma el control de la ejecución, –usualmente las arquitecturas cuentan con una instrucción *call* que realiza estos dos pasos, es decir, guarda la dirección de regreso y realiza un salto a la primer instrucción de la función

[7] Lo cual es dependiente de la plataforma, por lo que podría ser que la pila creciera hacia arriba.

[8] Mencionamos aquí estos aspectos, que son claramente dependientes de la plataforma, porque es necesario tomarlos en cuenta en la etapa de selección de instrucciones que estudiaremos en la siguiente sección.

[9] Algunos textos se refieren a esta estructura como *registro de activación*.

invocada–. Debemos tener claro que la dirección de regreso es el primer dato local de la función invocada, es decir, los argumentos que colocó f en la pila son datos que pertenecen a f no a g, mientras que la dirección de regreso sí es un dato que pertenece a g. Con esto debe quedar claro que el marco de pila de la función invocada comienza, en este caso, después del último argumento colocado por la función invocadora en la pila, justamente antes de donde se coloca la dirección de retorno (si es que ésta se almacena en la pila). En el caso en el que todos los argumentos se pasen en los registros, entonces el marco de pila de la función invocada comienza en donde se encontraba el tope de la pila de la función invocadora antes de transferir el control a la función invocada.

Una vez que g toma el control debe ser capaz de acceder a los argumentos que f puso en la pila para poder trabajar con ellos. Para lograr esto, g debe conocer la dirección de memoria en la que se encuentran cada uno de los argumentos que f colocó en la pila, lo que consigue con base en el apuntador de pila, que en este momento se encuentra justo después de la dirección de regreso. Es necesario conocer cuánto espacio ocupa la dirección de regreso y cada uno de los argumentos, para poder calcular la dirección de cada uno de ellos (nótese que lo anterior es dependiente de la arquitectura). Por ejemplo, en la arquitectura x86 la dirección de retorno ocupa 4 bytes (32 bits), por lo que la dirección del primer argumento se encuentra en la dirección $apuntador\ de\ pila + 4$, pues en $apuntador\ de\ pila$ se encuentra la dirección de retorno. Con base en el espacio que ocupa cada argumento se podrá determinar la dirección del siguiente argumento.

Aquí el apuntador de pila juega un papel fundamental para poder determinar la dirección de los argumentos, pero como cuando se guarda un dato local en la pila (o se reserva espacio para guardar futuros datos locales), el apuntador de pila cambia, se requiere registrar el valor de este apuntador antes de iniciar la ejecución de g. Para ello se hace uso de un registro especial conocido como *apuntador de marco* (*frame pointer*): una vez que la función invocada ha tomado el control, lo primero que se hace es guardar el valor del apuntador de marco en la pila e inmediatamente después establecer el valor actual del apuntador de pila como el nuevo valor del apuntador de marco. De esta forma ahora podemos determinar las direcciones de memoria con base en el apuntador de marco; tenemos que tomar en cuenta que hemos guardado el valor antiguo del apuntador de marco en la pila, que en la arquitectura x86 ocupa 4 bytes de espacio, por lo que, en este caso, el primer argumento se encuentra en la dirección $apuntador\ de\ marco + 8$, mientras que el segundo en $apuntador\ de\ marco + 8 + n$. La gran ventaja que tenemos ahora es que el valor del apuntador de marco no cambiará durante la ejecución de la función invocada, mientras que el valor del apuntador de pila puede cambiar libremente sin que afecte el cálculo para determinar las direcciones de los argumentos.

A este código que se encarga de guardar el apuntador de marco de la función invocada y establecer su nuevo valor como el valor del apuntador de pila actual al inicio de la ejecución de la función invocada se le conoce como *prólogo*[10].

[10]Como puede verse, el uso de un apuntador de marco no es estrictamente necesario. En lugar de eso

Dependiendo del ABI de la plataforma, en el prólogo puede ser que se deban realizar tareas adicionales como reservar espacio en la pila para posibles datos locales y guardar los registros guardados por la función invocada en la pila. Una vez realizado lo anterior, la función invocada puede comenzar su ejecución. Cuando la función invocada termine de ejecutarse, antes de regresar el control a la función invocadora obedeciendo al ABI, debe colocar el resultado en el lugar apropiado –en el caso de la arquitectura x86 en el registro *eax*–, restaurar los registros guardados por la función invocada –los valores correspondientes están respaldados en la pila– y eliminar de la misma todos los datos locales[11], restaurar el valor antiguo del apuntador de pila y eliminarlo de ésta[12] y, finalmente, hacerle un pop a la dirección de regreso y saltar a ella. En el caso de la arquitectura x86 esto último se puede realizar con una única instrucción *ret*; al código que representa a todas estas tareas que se deben realizar antes de devolver el control a la función invocadora se les conoce como *epílogo*.

En general, cuando se realiza una llamada a función se llevan a cabo los siguientes pasos:

1. La llamada a función ensambla los argumentos que se van pasar a la función y transfiere el control a la función.
   - Cada argumento se evalúa y se pone en el registro o localidad de la pila adecuada.
   - La dirección del código de la función se determina (o para la mayoría de los lenguajes fue determinada en tiempo de compilación o tiempo de ligado).
   - Los registros que están en uso y que deben ser respaldados por la función invocadora se guardan en memoria.
   - La dirección de retorno se salva en un registro o en la pila y se ejecuta un salto al código de la función (usualmente esto lo realiza una única instrucción, la instrucción call).
2. Al entrar, el prólogo de la función establece el entorno apropiado de direccionamiento y además puede realizar otras tareas, tales como salvar los registros que la función utilice para sus propios cálculos.
   - El apuntador de pila de la función invocadora se salva, el antiguo valor del apuntador de pila se estable como el valor actual del apuntador de marco y, posteriormente, se calcula el nuevo apuntador de pila.
   - Los registros que utilizará la función invocada, y que sean registros que debe proteger, se guardan en memoria.

puede utilizarse como un registro de propósito general y utilizar el apuntador de pila para determinar las direcciones necesarias.

[11] En realidad basta con incrementar el valor del tope de la pila y no es estrictamente necesario poner el valor de 0 en los lugares que éstos ocupaban en la pila.

[12] Aplica la nota anterior

3. La función realiza su trabajo (posiblemente llama a otras funciones).
4. Al terminar el epílogo de la función, restablece los valores de los registros y entorno de direccionamiento de la función invocadora, ensambla el valor que se va a regresar y regresa el control a la función invocadora.
   - Los registros guardados por la función invocada se restablecen con sus respectivos valores guardados en memoria.
   - El valor (si es que hay uno) a ser regresado se pone en el lugar apropiado.
   - Los anteriores apuntadores de pila y marco respectivamente se restablecen.
   - Se ejecuta un salto a la dirección de regreso.
5. Finalmente, el código de la función invocadora, que se encuentra después de la llamada a la función invocada, termina restableciendo su entorno de ejecución y recibe el valor de retorno que la función invocada calculó.
   - Los registros guardados por la función invocadora se restablecen con los valores respectivos guardados en memoria.
   - Se utiliza el valor de regreso.

Con el mecanismo descrito anteriormente debe ser claro que, por ejemplo, en el caso de una función recursiva f se crearán tanto marcos de pila como invocaciones recursivas haya; cada uno de ellos contendrá los datos locales de cada una de las diferentes invocaciones a f; el último marco que se creará hasta abajo en la pila será el marco correspondiente a la invocación de f que sea el caso base; a partir de ese momento se pasará el resultado del caso base de la función f hacia el marco de pila que está inmediatamente arriba, con lo que su correspondiente invocación a f podrá continuar ejecutándose hasta calcular su resultado correspondiente, mismo que pasará ahora el marco que está inmediatamente arriba, y así sucesivamente se irá resolviendo la recursión, pasando los resultado de las invocaciones hacia los marcos de arriba (y dejando de existir los de abajo), hasta que finalmente se llegue al marco correspondiente a la primera invocación de la función f y ésta calcule el resultado final.

Para ilustrar todo lo anterior, a continuación presentaremos tres implementaciones diferentes de la función de Ackerman, una escrita en ensamblador de x86, otra escrita en ensamblador de x86-64 y finalmente una escrita en ensamblador de SPARC. Cada una de éstas cumple con la respectiva ABI de la plataforma para la cual se escribieron.

```
#Programa en ensamblador de 32 bits x86
#que calcula la función de Ackerman
#Autor: Angel Francisco Zúñiga Chávez
#version: 0.9
#26 de Febrero de 2009
#version: 1.1
```

Código 5.1: Programa en ensamblador de x86 que calcula la función de Ackerman

(continúa en la siguiente página)

```
#Febrero de 2010

        .data
salida: .asciz "El resultado es: "
line:   .byte '\n'
number: .skip 32
        .text
        .globl _start
        .globl _ackerman
_start: #Iniciamos calcularemos A(3,4)
        pushl   $4      #metemos m a la pila
        pushl   $1      #metemos n a la pila
        call    ackerman
        movl    %eax, %esi
        addl    $8, %esp
        movl    $4,%eax
        movl    $1,%ebx
        leal    salida, %ecx
        movl    $17, %edx
        int     $0x80
        pushl   %esi
        call    printi
        addl    $4, %esp
        movl    $4, %eax
        movl    $1, %ebx
        leal    line, %ecx
        movl    $1, %edx
        int     $0x80
        movl    $1, %eax
        int     $0x80
#.type ackerman,@function

ackerman:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %ebx  #movemos m al registro ebx
        movl    8(%ebp), %eax   #movemos n al registro eax
        addl    $1,%eax         #regresamos n+1
        cmpl    $0, %ebx        #comparamos si m es cero
        je      end_ackerman    #fin del primer caso
```

Código 5.1: Programa en ensamblador de x86 que calcula la función de Ackerman

(continúa en la siguiente página)

```
    movl        8(%ebp), %eax   #movemos n al registro eax
        cmpl    $0, %eax        #si n es diferente de cero
        jne     three   #brincamos al tercer caso
        decl    %ebx    #inicio segundo caso m=m-1
        pushl   %ebx    #pasamos m-1 como argumento
        pushl   $1      #pasamos n=1 como argumento
        call    ackerman
        jmp     end_ackerman    #fin del segundo caso

three: #inicio del tercer caso
        pushl   %ebx    #metemos m
        decl    %eax    #n=n-1
        pushl   %eax    #metemos n-1
        call    ackerman
        movl    12(%ebp), %ebx  #recargamos m
        decl    %ebx    #m=m-1
        pushl   %ebx    #metemos m-1
        pushl   %eax    #metemos ackerman(m,n-1)
        call    ackerman
                #fin del tercer caso
end_ackerman:
        movl    %ebp, %esp
        popl    %ebp
        ret

printi:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax   #entero a imprimir
        leal    number+31, %ebx
        movl    $10, %ecx
while:
        cmpl    $0, %eax        #comparamos si es mayor que 0
        je      end_while
        cltd
        idivl   %ecx
        addl    $48, %edx
        movb    %dl,(%ebx)
        decl    %ebx
        jmp while
end_while:
```

Código 5.1: Programa en ensamblador de x86 que calcula la función de Ackerman

(continúa en la siguiente página)

```
        movl $4, %eax
        movl $1, %ebx
        movl $number, %ecx
        movl $32, %edx
        int $0x80

        movl %ebp, %esp
        popl %ebp
        ret
```

Código 5.1: Programa en ensamblador de x86 que calcula la función de Ackerman

```
#Programa en ensamblador de 64 bits x86-64
#que calcula la función de Ackerman
#Autor: Angel Zúñiga
#version: 1.0
#22 de Febrero de 2010
        .data
salida:  .asciz "El resultado es: "
line:   .byte '\n'
number: .skip 64
        .text
        .globl _start
        .globl ackerman
        .globl printi

_start:
        movq    $4, %rdi        #m
        movq    $1, %rsi        #n
        call    ackerman
        movq    %rax, %rdi
        movq    $4, %rax
        movq    $1, %rbx
        leaq    salida, %rcx
        movq    $17, %rdx
        int     $0x80
        call    printi
        movq    $4, %rax
        movq    $1, %rbx
        leaq    line, %rcx
        movq    $1, %rdx
        int     $0x80
```

Código 5.2: Programa en ensamblador de x86-64 que calcula la función de Ackerman

(continúa en la siguiente página)

```
        movq    $1, %rax
        int     $0x80

ackerman:
        movq    %rsi, %rax
        addq    $1, %rax        #regresamos n+1
        cmpq    $0, %rdi
        je end_ackerman

        cmpq    $0, %rsi        #si n es diferente de cero
        jne     three   #brincamos al tercer caso
#       pushq   %rdi
        decq    %rdi    #inicio segundo caso m=m-1
#       pushq   %rsi
        movq    $1, %rsi
#aquí debemos guardar rdi y rsi porque no se preservan
        call    ackerman
#aqui debemos traer rdi y rsi pero en realidad no nos importan
        jmp end_ackerman        #fin del segundo caso


three:          #inicio tercer caso
                #aquí se debe guardar rdi y rsi si nos interesan despu
                   és
        pushq   %rdi
        decq    %rsi
        call ackerman
        movq    (%rsp), %rdi    #recargamos m
        decq    %rdi
        movq    %rax, %rsi
        call ackerman
        addq    $8, %rsp        #quitamos la variable local m que
           habiamos guardado

end_ackerman:
        ret

printi:
        movq    %rdi, %rax
        leaq    number+63, %rbx
        movq    $10, %rcx
```

Código 5.2: Programa en ensamblador de x86-64 que calcula la función de Ackerman

(continúa en la siguiente página)

```
while:
        cmpq    $0, %rax
        je      end_while
        cltd
        idivq   %rcx
        addq    $48, %rdx
        movb    %dl,(%rbx)
        decq    %rbx
        jmp while

end_while:
        movq    $4, %rax
        movq    $1, %rbx
        movq    $number, %rcx
        movq    $64, %rdx
        int     $0x80
        ret
```

Código 5.2: Programa en ensamblador de x86-64 que calcula la función de Ackerman

```
/* Programa en ensamblador SPARC de 64 bits
 * que calcula la función de Ackerman
 * Autor: Angel Zúñiga
 * version: 1.0
 * Febrero de 2010
 */
        .Section ".data"
salida: .asciz  "El resultado es: "
line:   .byte '\n'
number: .skip 64

        .section ".text"
        .globl  _start
        .globl  ackerman
        .globl  printi

_start:
        set     3, %o0  !m
        set     4, %o1  !n
        call    ackerman
        nop
```

Código 5.3: Programa en ensamblador de SPARC que calcula la función de Ackerman

(continúa en la siguiente página)

```
        mov     %o0, %l0
        mov     4, %g1
        mov     1, %o0
        set     salida,%o1
        set     17,%o2
        ta      0x90
        mov     %l0, %o0
        call    printi
        nop
        mov     4,%g1
        mov     1, %o0
        set     line, %o1
        set     1, %o2
        ta      0x90
        mov     1, %g1
        ta      0x90

ackerman:       !tenemos que mejorar este caso
        save    %sp, -96, %sp
        !en i0 m y i1 n tenemos los argumentos reales
        !en i0 tenemos que regresar el resultado
        cmp     %i0, 0
        bne second
        nop
        !entramos al primer caso
        mov     %i1, %i0        !aqui estamos destruyendo el valor de
        inc     %i0             !m que en principio de me importa
        ret
        restore

second:         !inicio segundo caso
        cmp     %i1, 0
        bne     three
        nop
        !entramos al segundo caso
        mov     %i0, %o0
        dec     %o0
        mov     1, %o1
        call ackerman
        nop
        mov     %o0, %i0
        ret
        restore
```

Código 5.3: Programa en ensamblador de SPARC que calcula la función de Ackerman

(continúa en la siguiente página)

```
three:
        mov     %i0, %o0
        mov     %i1, %o1
        dec     %o1
        call ackerman
        nop
        !el resultado debe estar en o0
        mov     %o0, %o1
        dec     %i0
        mov     %i0, %o0
        call ackerman
        nop
        mov     %o0, %i0
        ret
        restore

printi:
        save    %sp, -96, %sp
        set     number+63, %l0
while:
        cmp     %i0, 0
        be      end_while
        nop
        mov     %i0, %l1
        sdiv    %i0, 10, %i0
        mov     %i0, %l2
        smul    %l2, 10, %l2
        sub     %l1, %l2, %l2
        add     %l2, 48, %l2
        stb     %l2,[%l0]
        dec     %l0
        ba while
        nop

end_while:
        mov 4, %g1
        mov 1, %o0
        set number, %o1
        set 64, %o2
        ta 0x90
        nop
        ret
        restore
```

Código 5.3: Programa en ensamblador de SPARC que calcula la función de Ackerman

El ABI de las arquitecturas x86, x86-64 y SPARC se encuentra en [TA97], [Mat+12] y [Int99] respectivamente.

Para finalizar esta sección cabe decir que hay datos que queremos preservar más allá de la invocación de una función. Estos datos no deben ser datos que se almacenen en la pila, ya que cuando termine la invocación a la función desaparecerán. Para aquellos datos que se deben preservar, aun cuando en general una función se haya procesado, se deben almacenar en otro lugar en la memoria, generalmente en una sección de la memoria conocida como el *montículo* (*heap*).

Es un error común de los programadores que programan en Java y comienzan a programar en C++, pensar que los objetos que crean dentro de una función f seguirán vivos al terminarse de ejecutar f. Esto se debe a que en Java los objetos que se crean siempre se colocan en el heap, aun cuando se crean dentro de una función f. Lo que almacena la variable local es una referencia al objeto que se creó en el heap y esa referencia sí dejará de existir cuando la invocación a f termine (por lo que generalmente se regresa una referencia hacia él como resultado de la función), pero el objeto continuará vivo en el heap hasta que ya no haya ninguna referencia hacia él y el recolector de basura lo elimine. En cambio en C++ se tiene la libertad de elegir dónde se debe crear el objeto, si en la pila o en el heap; el error común es que un programador experimentado en Java cree un objeto en la pila (y regrese como resultado de la función la dirección de ese objeto, que será una dirección en la pila) y una vez que concluye la invocación a f piense que el objeto seguirá vivo.

El heap puede ser especialmente útil en la implementación de cerraduras (*closures*), generalmente presentes en lenguajes funcionales, en las que se requiere que ciertos datos sigan vivos aún después de que se ha procesado una función. En este caso el soporte en hardware no nos basta para realizar la implementación de funciones y el compilador debe realizar una tarea más compleja para poder implementarlas.

5.3. Selección de instrucciones

Una representación intermedia basada en *árboles de bajo nivel*[13] (*low-level trees*) expresa solamente un único operador en cada nodo de un árbol: traer de memoria (*memory fetch*), guardar (*store*), suma, resta, salto condicional o cualquier otro operador presente en una instrucción. Una instrucción de una máquina real en muchos casos es capaz de realizar varias de esas operaciones primitivas. Por ejemplo, casi cualquier máquina puede realizar una suma y un traer de memoria en la misma instrucción, que puede implementar el árbol en la figura 5.1:

El trabajo de la fase de *selección de instrucciones* de un compilador consiste en encontrar las instrucciones de máquina para implementar una representación intermedia dada basada en árboles de bajo nivel.

[13] También llamados *árboles de expresiones*.

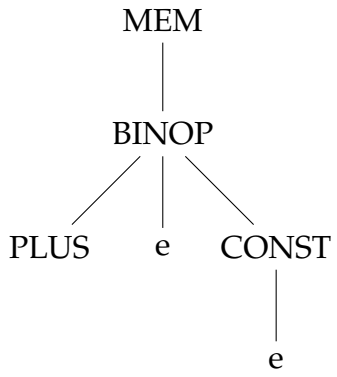


Figura 5.1

Podemos expresar las instrucciones de máquina como un fragmento de una representación intermedia basada en árboles de bajo nivel, al que llamaremos un *patrón en forma de árbol* (*tree pattern*). Es decir, por cada instrucción de máquina tendremos el o los patrones en forma de árbol que dicha instrucción puede implementar (las operaciones que dicha instrucción puede ejecutar). De esta forma, la selección de instrucciones se convierte en la tarea de enlosetar el árbol con un conjunto mínimo de patrones en forma de árbol.

Como mencionamos, algunas instrucciones pueden implementar más de un patrón. Por ejemplo, las instrucciones que realizan operaciones aritméticas, como la suma y la multiplicación, desde luego pueden implementar tanto un patrón con el operador respectivo y un orden fijo en sus operandos, como el patrón que representa la misma operación pero con los operandos conmutados.

La idea fundamental de la selección de instrucciones que trabaja sobre una representación intermedia basada en árboles de bajo nivel es enlosetar el árbol de la representación intermedia. Las losetas son el conjunto de patrones en forma de árbol que corresponden a instrucciones de máquina legales y el objetivo es cubrir el árbol sin que se encimen las losetas.

El mejor enlosetado de un árbol corresponde a una secuencia de instrucciones de costo mínimo, que puede considerarse la secuencia de instrucciones de menor longitud o, si las instrucciones toman diferentes tiempos de ejecución, la secuencia que tiene el tiempo total de ejecución más bajo.

Supongamos que se asigna un costo a cada tipo de instrucción. Entonces se puede definir un *enlosetado óptimo* como aquel cuyas losetas suman el valor más bajo posible.

En este punto podemos utilizar una gramática libre del contexto para describir las losetas, es decir, los patrones en forma de árbol; de esta forma, encontrar un enlosetado del árbol que representa un programa es equivalente a encontrar un análisis de dicho árbol con base en una gramática en la que se describen los patrones que pueden implementar cada una de las instrucciones de máquina. Debe quedar claro que dicha

gramática muy probablemente será altamente ambigua, es decir, habrá muchos análisis diferentes para un mismo árbol de entrada, como es de esperarse, pues habrá muchas secuencias de instrucciones diferentes que implementen la expresión representada por dicho árbol; para eliminar esta ambigüedad y seleccionar la secuencia de instrucciones óptima se añade un costo a cada uno de los patrones. Con base en lo anterior es posible seleccionar el análisis (enlosetado) de costo mínimo; adicionalmente se pueden agregar acciones a cada uno de los patrones, para que en ellas se indiquen las instrucciones de máquina que se deben generar con base en el patrón respectivo.

Entonces podemos decir que la etapa de selección de instrucciones de un compilador determina qué instrucciones son las mejores para realizar los cálculos de un programa. Generalmente, las instrucciones se seleccionan para minimizar el tamaño del código generado o para minimizar el tiempo de ejecución. El conjunto de instrucciones de la mayoría de las arquitecturas modernas es redundante, es decir, existen cálculos que se pueden evaluar vía dos o más secuencias diferentes de instrucciones. El seleccionador de instrucciones debe elegir entre las diferentes opciones correctas para producir el código óptimo.

Las arquitecturas no son triviales y no es siempre obvio qué código evaluará de la forma más económica posible una expresión. Muchas arquitecturas CISC cuentan con varios modos de direccionamiento diferentes, cada uno de los cuales puede incluir algunas sumas y desplazamientos; simplemente reconocer dónde son aplicables éstos puede resultar difícil. Más aún, comparar todas las posibles combinaciones legales que calculan el resultado deseado parece, a primera vista, computacionalmente intensivo.

Afortunadamente, si restringimos nuestra atención a árboles de expresiones (en lugar de gráficas dirigidas acíclicas *Directed Acyclic Graph*, *DAG*), seleccionar las instrucciones óptimas es fácil. Para lograr esto, expresaremos el conjunto de instrucciones (*instruction set*) de una arquitectura como un conjunto de patrones en forma árbol. Si a los patrones en forma de árbol que describen diferentes instrucciones se les asignan costos, entonces se puede utilizar programación dinámica para seleccionar el conjunto de instrucciones óptimas para evaluar el árbol.

La programación dinámica es una operación cara ya que encuentra todas las subsoluciones óptimas antes de encontrar la solución para el árbol completo. Afortunadamente *BURS* (*Bottom-Up Rewrite System*) [Pel88] puede eliminar este costo dentro del compilador, ya que BURS preprocesa los patrones en forma de árbol y sus respectivos costos asociados para construir un autómata que puede dirigir al seleccionador de instrucciones de forma rápida.

Los generadores de código basados en BURS son rápidos por dos razones: utilizan empatado de patrones en forma de árbol de abajo hacia arriba (*buttom-up tree-pattern matching*) y hacen toda la programación dinámica en tiempo de compilación del compilador (*compile-compile time*), es decir, cuando los patrones se preprocesan para construir el generador de código. Llevando a cabo la programación dinámica en tiempo de

compilación del compilador, un generador de código basado en BURS puede anticipar todos los posibles árboles de entrada con información guardada en tablas. Cuando se utiliza esta estrategia es necesario realizar con anticipación una enorme cantidad de cálculos para llevar a cabo la programación dinámica de todos los posibles patrones en forma de árbol. Por ello es importante contar con un generador de autómatas basado en BURS que sea eficiente.

Con lo anterior debe ser claro que llevar a cabo la selección de instrucciones de manera eficiente puede resultar difícil, debido a que las arquitecturas CISC suelen contar con muchas elecciones legales diferentes para evaluar la misma expresión. En muchas ocasiones es necesario dar soporte a una nueva arquitectura dentro de un compilador y, desde luego, en esos casos es necesario implementar un nuevo generador de código, por lo que se ha realizado mucho trabajo para facilitar la tarea de reobjetivizar un compilador.

El trabajo de crear compiladores reobjetivizables se puede hacer más manejable si el trabajo necesario para reobjetivizar un compilador se aisla dentro del generador de código, y si se cuenta con herramientas automáticas de ayuda para la creación de generadores de código reobjetivizables. Tales herramientas se conocen como *generadores de generadores de código* (*code generator generators*).

Los generadores de generadores de código crean automáticamente un generador de código para una representación intermedia particular, a partir de una descripción del conjunto de instrucciones de un procesador dado y una descripción de la forma de la representación intermedia. Estas descripciones pueden ser desde patrones de bajo nivel, que cuando empatan contra la representación intermedia señalan qué instrucciones de máquina se van a emitir, hasta descripciones de alto nivel de la máquina y la semántica de la representación intermedia, a partir de la cual estos patrones se deducen. Un generador de generadores de código permite automatizar la mecánica de encontrar los empates entre la representación intermedia y las instrucciones de máquina, reduciendo el trabajo de crear un generador de código y los errores que se puedan cometer durante la creación de éste.

Posiblemente la manera más fácil de visualizar y entender las instrucciones complejas y los modos de direccionamiento de una arquitectura es ver a éstos como árboles de expresiones, en los cuales las hojas representan registros y localidades de memoria, mientras que los nodos internos representan operaciones sobre valores de los operandos. Describir incluso el modo de direccionamiento más complejo se simplifica cuando se utilizan este tipo de árboles.

Debido a su expresividad, los árboles de bajo de nivel también sirven como una representación intermedia que se puede generar después del análisis semántico. Si el mismo dominio de árboles de bajo nivel se utiliza para describir instrucciones de máquina y se utiliza como representación intermedia, la selección de instrucciones para una representación intermedia en forma de árbol de bajo nivel se convierte en la tarea de empatar

patrones de instrucciones contra la instrucción intermedia generada, de tal manera que la representación intermedia sea cubierta (analizada) con patrones adyacentes.

El empates de patrones de árboles combinado con programación dinámica se puede utilizar en los generadores de código para crear código óptimo local para árboles de expresiones. Los generadores de código basados en BURS son extremadamente rápidos ya que toda la programación dinámica se realiza cuando el autómata basado en BURS se construye. Sólo es necesario hacer dos recorridos sobre el árbol, en tiempo de compilación del compilador,uno de abajo hacia arriba para etiquetar cada nodo con un estado que codifica todos los empates óptimos, y un segundo recorrido de arriba hacia abajo, que utiliza esos estados para seleccionar y emitir código.

El autómata que etiqueta el árbol es una máquina simple de estados con transiciones. Se realiza un recorrido del árbol de abajo hacia arriba y la etiqueta para cualquier nodo dado se determina por medio de una búsqueda en la tabla, dado el operador en el nodo y los estados que etiquetan a cada uno de sus hijos. El autómata que emite el código tiene, de igual manera, un diseño simple. El código que se va emitir se determina por el estado que etiqueta un nodo y un símbolo no terminal al cual ese nodo debe ser reducido, lo que se realiza por medio de otra búsqueda en la tabla.

Cuando se crea un generador de código basado en BURS surgen dos dificultades: la primera de ellas consiste en generar de forma eficiente los estados y las tablas de transiciones. Ya que todas las decisiones potenciales de la programación dinámica se llevan a cabo en tiempo de generación de la tabla, esta tarea se debe realizar de forma eficiente. La segunda dificultad consiste en crear una codificación eficiente del autómata.

En [Pro92] se describe un algoritmo de generación de tablas simple y eficiente.

La entrada para un generador de generadores de código basado en BURS es un conjunto de producciones. Cada producción indica un patrón en forma de árbol, un costo, un símbolo no terminal y una acción. Desde luego el conjunto de todas las producciones es una gramática. En la figura 5.2 se da una pequeña gramática de ejemplo (sin acciones).

#	Producción			Costo
1)	$goal$	$\longrightarrow$	reg	(0)
2)	reg	$\longrightarrow$	Reg	(0)
3)	reg	$\longrightarrow$	Int	(1)
4)	reg	$\longrightarrow$	$Fetch(addr)$	(2)
5)	reg	$\longrightarrow$	$Plus(reg, reg)$	(2)
6)	$addr$	$\longrightarrow$	reg	(0)
7)	$addr$	$\longrightarrow$	Int	(0)
8)	$addr$	$\longrightarrow$	$Plus(reg, Int)$	(0)

Figura 5.2

El símbolo no terminal está a la izquierda de una producción mientras que el patrón que se deriva está a la derecha en forma de árbol linealizado. En el ejemplo, $goal$, reg y $addr$ son símbolos no terminales. Adicionalmente a los símbolos no terminales de la gramática, se tiene que los patrones tienen $operadores$ que pueden recibir diferente número de argumentos. En el ejemplo, Reg, Int, $Fetch$ y $Plus$ son operadores que reciben 0, 0, 1 y 2 argumentos respectivamente.

Un análisis de costo mínimo se puede encontrar utilizando programación dinámica. Intentando todos los posibles empates en cada uno de los nodos, es posible recordar las producciones que llevan a la derivación de costo mínimo desde cada símbolo no terminal posible. En la figura 5.3 se aplican las producciones de la figura 5.2 al árbol que representa a $Fetch(Fetch(Plus(Reg, Int))$. Cada nodo se etiqueta con la derivación de costo mínimo a partir de cada símbolo no terminal.

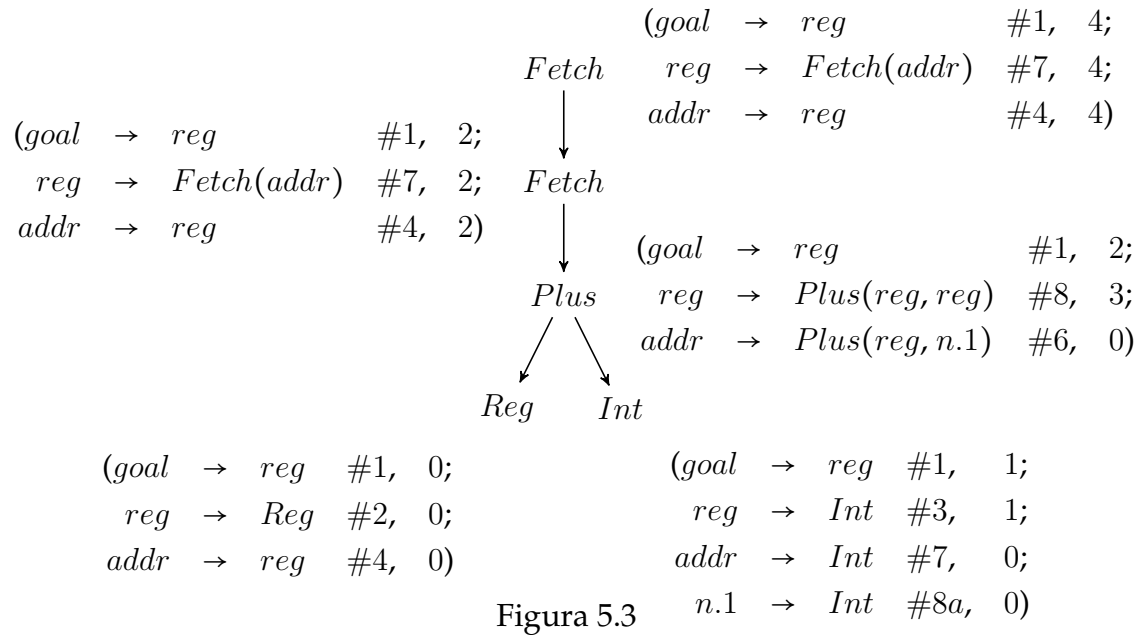


Figura 5.3

Un empatador de patrones basado en BURS encuentra el análisis de costo mínimo de un árbol objetivo para la gramática que reduce al símbolo no terminal $goal$. Cada nodo del árbol será etiquetado con un estado que codifica qué producción se utilizará cuando ese nodo sea reducido al símbolo no terminal correspondiente.

Estos estados codifican la información dada explícitamente en la figura 5.3. Por ejemplo, es posible derivar el nodo hoja Int, a partir de todos los símbolos no terminales. Int se puede derivar directamente de los símbolos no terminales reg, $addr$ y $n.1$, aplicando directamente las producciones $\#3$, $\#7$ y $\#8a$ respectivamente. El costo asociado con cada derivación es el costo de esa producción particular. La derivación a partir de $goal$ utiliza la producción, "$goal \rightarrow reg$", lo que exigirá que subsecuentemente Int se derive a partir de reg. Por lo tanto, mientras el costo asociado con la producción $\#1$ es 0, el costo de la derivación es 1, es decir, la suma de los costos de la derivación completa de Int a partir de $goal$.

Con todo lo dicho en esta sección debe ser claro cómo una vez que tenemos una representación intermedia basada en árboles de bajo nivel, podemos encontrar la secuencia óptima de instrucciones que implementa el programa representado por la representación intermedia.

Vale la pena señalar que la mayoría de las arquitecturas de diseño moderno son RISC. Generalmente cada instrucción RISC realiza únicamente un número pequeño de operaciones, por lo que sus losetas correspondientes son pequeñas y de costo uniforme; dado esto, las arquitecturas RISC se benefician menos de la estrategia que hemos descrito para la selección de instrucciones.

Como hemos mencionado, el seleccionador de instrucciones que hemos desarrolla-

do en la presente sección toma como entrada una representación intermedia en forma de árboles de bajo de nivel. Cuando se trabaja con una representación intermedia en forma de gráfica como entrada, la selección óptima de instrucciones es un problema NP-Completo. Fue esto lo que motivó a que se efectuara sobre árboles. Recientemente, con el surgimiento de la forma SSA que estudiamos en el capítulo anterior, se ha retomado el problema de encontrar la selección óptima de instrucciones, esta vez tomando como entrada la gráfica SSA, lo que a priori sería deseable pues no sería necesario realizar transformaciones a la representación intermedia (incluyendo la destrucción de la forma SSA) para obtener una representación intermedia en forma de árboles de bajo nivel. La solución de la selección de óptima de instrucciones directamente sobre la forma SSA con base en el problema PBQP se estudia, entre otros, en [BZ10] y en [Ebn+08].

5.4. Alojamiento en registros

El problema de alojamiento en registros puede abstraerse como un problema de coloración en gráficas. Chaitin [Cha+81] fue el primero en implementar esta solución, que aunque ya se había mencionado en la literatura previamente nunca antes se había llevado a la práctica. Los nodos en la gráfica representan *rangos de vida* (*live ranges*) o temporales que se utilizan en el programa. Una arista conecta dos temporales que están simultáneamente vivos en algún punto del programa si sus rangos de vida *interfieren*. El problema de coloración en gráficas consiste en asignar colores a nodos tal que a dos nodos que estén conectados por una arista no les asigna el mismo color. El número de colores disponible es igual al número de registros disponibles en la máquina. Colorear una gráfica general con k colores es un problema NP-Completo, por lo que se utiliza un algoritmo de aproximación que toma tiempo polinomial.

Hay cinco fases principales en un alojador de registros por coloración de gráficas del estilo Chaitin:

1. Construcción (*Build*): Construye la gráfica de interferencia. Se utiliza análisis de flujo para calcular el conjunto de registros que están vivos simultáneamente en un punto del programa y se agrega una arista a la gráfica por cada par de registros en el conjunto. Esto se repite para todos los puntos del programa.
2. Fusión (*Coalesce*): Elimina las instrucciones de movimiento (*move instructions*) innecesarias. Una instrucción de movimiento se puede eliminar del programa cuando la fuente y destino de la instrucción no tienen una arista en la gráfica de interferencia. En otras palabras, la fuente y destino pueden ser fusionados en un único nodo, que contiene las aristas combinadas de los nodos que se están reemplazando. Cuando todos los movimientos posibles han sido fusionados, reconstruir la gráfica de interferencia para el nuevo programa puede dar lugar a más oportuni-

dades de fusión. Por ello, las fases de construcción y fusión se repiten hasta que no haya más instrucciones por fusionar.

3. Simplificación (*Simplify*): Colorea la gráfica utilizando una heurística simple. Supongamos que la gráfica G contiene un nodo m con menos de k vecinos, donde k es el número de registros en la máquina. Sea G' la gráfica $G - \{m\}$ que se obtiene eliminando m de G. Si G' se puede colorear, en consecuencia G también se podrá colorear. Cuando se agregue m a la gráfica coloreada G', los vecinos de m tendrán a lo más $k-1$ colores entre ellos, por lo que siempre será posible encontrar un color libre para m. Esto nos conduce de forma natural a un algoritmo para colorear gráficas con base en una pila: iterativamente eliminar (y empujar en una pila) nodos de grado menor que k. Cada simplificación de este tipo hará que los grados de los otros nodos decrezcan, con lo que se obtienen más oportunidades para simplificar.
4. Derrame (*Spill*): Supongamos que en algún punto durante la simplificación la gráfica G solamente tiene nodos de *grado significativo*, esto es, nodos de grado mayor o igual que k. Entonces la heurística de simplificación fracasa y es necesario marcar un nodo para derrame. Esto es, se elije algún nodo de la gráfica (que corresponde a una variable temporal del programa) que se debe guardar en memoria y no en algún registro, durante la ejecución del programa. Una aproximación optimista al efecto de derramar es que el nodo derramado no interfiere con ninguno de los nodos restantes en la gráfica, por lo que se puede eliminar y continuar con el proceso de simplificación. En efecto, el nodo derramado debe ser traído de memoria justo antes de cada uso, por lo que éste tendrá varios rangos de vida pequeños, que interferirán con otros valores temporales en la gráfica. Si durante una pasada de simplificación uno o más nodos se marcan para derrame, el programa debe ser reescrito con instrucciones traer y guardar explícitas y se deben calcular nuevos rangos de vida utilizando análisis de flujo de datos. A continuación se repiten las pasadas de construcción y simplificación. Este proceso se itera hasta que la simplificación tiene éxito sin derrames; en la práctica, casi siempre basta con una o dos iteraciones.
5. Selección (*Select*): Asigna colores a nodos en la gráfica. La gráfica original se construye iterativamente agregando a la gráfica un nodo tomado del tope de la pila, comenzando con la gráfica vacía. Cuando se agrega un nodo a la gráfica, debe haber un color disponible para él, ya que como justificación de que ese nodo se eliminará en la fase de simplificación está que siempre es posible asignarle un color, es decir, dados los nodos restantes de la gráfica siempre se puede colorear con éxito.

Es fácil eliminar las instrucciones redundantes de movimiento con base en una gráfica de interferencia. Si en la gráfica de interferencia no existe una arista entre la fuente y el destino de una instrucción de movimiento, entonces la instrucción se puede eliminar.

Los nodos fuente y destino se fusionan en un nuevo nodo cuyas aristas son la unión de las aristas de los nodos que se están reemplazando.

En principio, cualquier par de nodos que no estén conectado por una arista de interferencia se pueden fusionar. Esta forma agresiva de propagación de copias es muy exitosa eliminando instrucciones de movimiento. Desafortunadamente, el nodo que se introduce tendrá más restricciones que aquellos que se eliminan, ya que contiene una unión de aristas. Entonces, es muy posible que una gráfica coloreable con k colores antes de fusionar, puede ser que ya no sea k-coloreable después de realizar sin cuidado (*reckless*) las fusiones, por lo que desearíamos fusionar solamente cuando es *seguro* hacerlo, esto es, cuando la fusión no haga que la gráfica sea vuelva incoloreable. Las siguientes dos estrategias son seguras:

1. Un par de nodos a y b pueden ser fusionados si el nodo resultante ab tendrá menos de k vecinos de grado significativo (es decir, que tenga k o más aristas). Esta fusión garantiza no convertir una gráfica k-coloreable en una gráfica no k-coloreable, ya que después de la fase de simplificación habrá eliminado todos los nodos con grado no significativo de la gráfica, el nodo fusionado será adyacente únicamente a aquellos vecinos que fueron de grado significativo. Como hay menos de k de éstos, la etapa de simplificación puede eliminar el nodo fusionado de la gráfica. Entonces, si la gráfica original era coloreable, esta estrategia de fusión conservadora no alterará la colorabilidad de la gráfica.

2. Un par de nodos a y b se pueden fusionar si para todo vecino t de a, t ya interfiere con b o t es de grado no significativo. Esta fusión es segura debido al siguiente razonamiento. Sea S el conjunto de vecinos con grado no significativo de a en la gráfica original. Si la fusión no se realizara, la fase de simplificación puede eliminar todos lo nodos en S, dejando una gráfica reducida G_1. Si la fusión se realiza, entonces la fase de simplificación puede eliminar todos los nodos en S, dejando una gráfica G_2. Pero G_2 es una subgráfica de G_1 (el nodo ab en G_2 corresponde al nodo b en G_1) y por esto debe ser al menos igual de fácil de colorear.

Estas estrategias son *conservadoras* ya que existen aún situaciones seguras en las cuales fallarán al fusionar. Esto significa que el programa puede realizar algunas instrucciones de movimiento innecesarias, pero es claro que es mejor que no hacer ninguna fusión y hacer solamente derrames.

Intercalando los pasos de simplificación con fusión conservadora se elimina la mayoría de las instrucciones de movimiento, mientras que se garantiza aún que no se introducen derrames. Las etapas de fusión, simplificación y derrame deben alternarse hasta que la gráfica esté vacía.

Lo anterior nos lleva a tener las siguientes etapas en el alojador de registros:

1. Construir: Construye la gráfica de interferencia y categoriza cada nodo ya sea como *relacionado con movimiento* (move-related) o *no relacionado con movimiento* (non-

move-related). Un nodo relacionado con movimiento es aquel que es ya sea fuente o destino de una instrucción de movimiento.

2. Simplificar: Elimina uno por uno los nodos no relacionados con movimiento de bajo grado (menor que k) de la gráfica.
3. Fusionar: Realiza fusiones conservadoras sobre la gráfica reducida obtenida en la fase de simplificación. Debido a que los grados de muchos nodos han sido reducidos por la etapa de simplificación, es muy probable que la estrategia conservadora encuentre mucho más movimientos para fusionar que los que se tenían en la gráfica de interferencia inicial. Después de que dos nodos han sido fusionados (y la instrucción de movimiento eliminada), si el nodo resultante ya no está relacionado con movimiento éste estará disponible para la siguiente ronda de simplificación. Las etapas de simplificar y fusionar se repiten hasta que únicamente restan nodos de grado significativo o relacionados con movimiento.
4. Congelar (*Freeze*): Si las etapas de simplificación y fusión no tienen mas trabajo por realizar, entonces se buscan nodos relacionados con movimiento de bajo grado. Se *congelan* los movimientos en los cuales estos nodos están involucrados: esto es, se concede la esperanza de fusionar estos movimientos. Esto causa que el nodo (y quizás otros nodos relacionados con los movimientos congelados) sea considerado como no relacionado con movimiento, lo que debe permitir más simplificación. En este momento, las etapas de simplificar y fusionar se reanudan.
5. Derramar: Si no hay nodos de grado bajo, se selecciona un nodo con grado significativo para un potencial derrame y se empuja en la pila.
6. Seleccionar: Se hace un pop a cada uno de los nodos de la pila y se les va asignando un color a cada uno de ellos.

Si es necesario hacer derrames, las etapas de construir y simplificar deben repetirse sobre todo el programa. La versión más simple del algoritmo descarta cualquier fusión encontrada si la etapa de construir debe repetirse. Entonces es fácil ver que fusionar no incrementa el número de derrames en cualquier ronda futura de la etapa de construir. Un algoritmo más eficiente debe preservar cualquier fusión hecha antes de que el primer derrame potencial haya sido descubierto, pero debe descartar (desfusionar) cualquier fusión hecha después de este punto.

En una máquina con muchos registros (más de veinte, como es el caso de la mayoría de las arquitecturas RISC), usualmente habrá pocos nodos derramados. Pero en una máquina con seis registros (tal como las basadas en la arquitectura x86, que hereda su diseño CISC), habrá muchos derrames. El front end pudo haber generado muchos temporales y transformaciones tales como la forma SSA los puede dividir en muchos más temporales. Si cada temporal derramado vive en su propia localidad del marco de pila, entonces el marco puede ser verdaderamente grande. Incluso peor, puede haber

muchas instrucciones de movimiento que involucren pares de nodos derramados. Pero para implementar $a \leftarrow b$ cuando a y b son ambos temporales derramados, se requiere una secuencia traer-guardar (*fetch-store*), $t \leftarrow M[a_{loc}];\ M[b_{loc}] \leftarrow t$. Esto es caro y también define un temporal t que puede a su vez causar que otros nodos se derramen.

Pero muchos de los pares derramados nunca están vivos simultáneamente, por lo que pueden ser coloreados en la gráfica fusionándolos. En efecto, debido a que no hay un límite fijo para el número de localidades en el marco de pila, se puede fusionar agresivamente, sin preocuparse de cuántos vecinos con alto grado tienen los nodos derramados, con lo que se obtiene el siguiente algoritmo:

1. Utilizar información de rangos de vida para construir la gráfica de interferencia de nodos derramados.
2. Mientras exista cualquier par de nodos derramados sin interferencia, conectados por una instrucción de movimiento, fusionarlos.
3. Utilizar las fases de simplificar y seleccionar para colorear la gráfica. No debe haber más derrames en esta coloración; en lugar de esto, la etapa de simplificar solamente toma el nodo con el grado más bajo, y la etapa de seleccionar toma el primer color disponible, sin ningún límite predeterminado en el número de colores.
4. Los colores corresponden a las localidades del registro de activación para las variables derramadas.

Esto se debe realizar antes de generar las instrucciones de derrame y regenerar la gráfica de interferencia de registros y temporales, para evitar crear secuencias traer-guardar para movimientos fusionados de nodos derramados.

Algunos temporales se *precolorean*, ya que representan registros de la máquina. El front end los genera cuando cumple con la convención estándar de llamadas a funciones. Para cada registro real que se utiliza para algún propósito específico, tal como el apuntador de marco, el registro estándar para el primer argumento, el registro estándar para el segundo argumento, etcétera, alguna parte del compilador debe utilizar el temporal particular que está permanentemente establecido a este registro. Para cualquier color dado (esto es, para cualquier registro dado de la máquina) debe haber solamente un nodo precoloreado con ese color.

Las etapas de seleccionar y fusionar pueden dar a un temporal ordinario el mismo color que el registro precoloreado siempre que éstos no interfieran, y de hecho es muy común que esto suceda. Entonces, un registro que se utiliza en la convención estándar de llamadas a funciones puede ser reutilizado dentro de una función como una variable temporal. Los nodos precoloreados pueden ser fusionados con otros nodos (no precoloreados) utilizando fusión conservadora.

Para una máquina con k registros, habrá k nodos precoloreados, donde cada uno de ellos interfiere con cada uno de los otros. Aquellos nodos precoloreados que no se

utilizan explícitamente (en una convención de paso de parámetros, por ejemplo) no interferirán con cualquier nodo ordinario (no precoloreado); pero un registro de máquina utilizado explícitamente tendrá un rango de vida que interferirá con cualquier otra variable que esté viva al mismo tiempo.

No se puede simplificar un nodo precoloreado, ya que esto significaría extraerlo de la gráfica con la esperanza de poderle asignar un color después, pero de hecho no tenemos libertad de qué color asignarle. Y no se deberían derramar a memoria los nodos precoloreados, ya que los registros de la máquina son, por definición, *registros*, por lo que se deben tratar como si tuviesen grado infinito.

El algoritmo de coloración trabaja llamando a las etapas de simplificación, fusión y derrame hasta que sólo restan los nodos precoloreados y la etapa de selección puede comenzar a agregar los otros nodos (y colorearlos).

Debido a que los nodos precoloreados no se derraman, el front end debe ser cuidadoso para mantener sus rangos de vida cortos. Esto lo puede lograr generando instrucciones de movimiento de valores hacia y desde nodos precoloreados. Por ejemplo, supongamos que r_5 es un registro respaldadoo por la función invocada, que se definió en la entrada de la función y se utiliza en la salida de la función. En lugar mantenerse en un registro precoloreado a lo largo de toda la función, éste se puede mover a un temporal t_1 y después moverlo de regreso. Si hay *presión por registros* (*register pressure*, una alta demanda de registros) en esta función, el temporal t_1 será derramado; si no es así, t_1 será fusionado con r_5 y la instrucción de movimiento se eliminará.

Las heurísticas más básicas de derrame pueden conseguir el efecto de alojar variables vivas a través de llamadas a funciones, en registros salvados por la función invocada. Una variable local o un temporal generado por el compilador que no está vivo a través de cualquier llamada a función, usualmente debería ser alojado en un registro salvado por la función invocadora, ya que en ese caso no será necesario en lo absoluto guardar y restaurar el registro. Por otro lado, cualquier variable que esté viva a través de varias llamadas a funciones debe ser mantenida en un registro salvado por la función invocada, ya que únicamente una instrucción guardar y una cargar serán necesarias (en la entrada y en la salida de la función que se está invocando).

El alojador de registros debe alojar variables en registros utilizando este criterio. Afortunadamente, un alojador por coloración de gráficas con derrames puede hacerlo muy fácilmente. Las instrucciones de llamadas pueden anotarse para *definir* (interferir con) todos los registros salvados por la función invocadora. Si una variable no está viva a través de una llamada a función, tenderá a ser alojada en un registro salvado por la función invocadora.

Si una variable x está viva a través de una llamada a función, entonces x interfiere con todos los registros (precoloreados) salvados por la función invocadora, e interfiere con todos los temporales nuevos (tales como t_1 de nuestro ejemplo anterior) creados para registros salvados por la función invocada. Entonces ocurrirá un derrame. Utilizando

las heurísticas derrame-costo comunes que derraman un nodo con grado alto pero con pocos usos, el nodo elegido para derramar no será x sino un temporal nuevo como t_1; ya que t_1 se derrama, r_5 estará disponible para colorear x (o alguna otra variable).

5.4.1. Alojamiento en registros en la forma SSA

El resultado central es que las gráficas de interferencias de los programas en forma SSA son cordales (*chordal*). Esto tiene un impacto significativo en la forma en la que se pueden construir los alojadores de registros:

- Las etapas de colorear y derramar pueden ser desacopladas completamente.
- Debido a que las gráficas cordales son *perfectas*, éstas heredan todas las propiedades de las gráficas perfectas, de las cuales la más importante es que el número cromático de la gráfica es igual al tamaño del clan (*clique*) más grande. Esta propiedad es aún más fuerte, ya que se cumple para cada subgráfica inducida de una gráfica perfecta. En otras palabras, la cordalidad, asegura que la presión por registros no solo es una cota inferior para la verdadera demanda de registros, sino una medida precisa. Determinando en el programa la instrucción donde la mayoría de las variables están vivas, se obtiene el número de registros necesarios para un alojamiento de registros válido del programa. A diferencia de los programas que no están en forma SSA, la estructura de la gráfica de flujo de control no puede causar demanda adicional de registros.
- Esto permite a la fase de derrame determinar exactamente las localidades en el programa donde las variables deben residir en memoria. Entonces, el mecanismo de derrame puede basarse en examinar las instrucciones en el programa en lugar de considerar los nodos en la gráfica de interferencia. Después de que la fase de derrame ha reducido la presión por registros a una cota dada, está garantizado que no se introducirán más derrames. Por ello la etapa de derrame sólo se tiene que realizar una única vez.
- Colorear una gráfica cordal se puede realizar en $O(|N|^2)$ donde $|N|$ es el número de nodos de la gráfica. Además, el orden en el cual los nodos de la gráfica de interferencia se colorean está relacionado con el orden de las instrucciones en el programa. Por esto, puede obtenerse una coloración a partir del programa sin tener que materializar la gráfica de interferencia en sí misma.
- La mayor fuente de instrucciones de movimiento en un programa son las instrucciones ϕ. Fusionar esas copias demasiado temprano puede resultar en una alta demanda innecesaria de registros. La etapa de fusión debe tener cuidado de que cuando se fusionen dos variables no se exceda el número de registros disponibles. En lugar de mezclar nodos en la gráfica de interferencia, se debe tratar de asignarles a estos dos nodos el mismo color. De esta forma la gráfica seguirá siendo

cordal y la etapa de fusión puede fácilmente mantener un registro del número cromático de la gráfica y rehusarse a fusionar dos variables si esto incrementara el número cromático más allá del número de registros disponibles. Sin embargo este problema de coloración de gráficas modificado es NP-Completo.

5.5. Generador de código de Python

Toca el turno de implementar el generador de código en nuestro compilador.

Como mencionamos en el capítulo 1, nuestro compilador está diseñado para ser reobjetivizable, esto es, debe ser capaz de generar código para diferentes arquitecturas y de soportar nuevas en un futuro de manera modular, simple y transparente.

Nuestro compilador inicialmente tendrá soporte para las arquitecturas x64 y SPARC.

Recordemos que el analizador semántico entrega un AST decorado y es éste el que debe ser la entrada para nuestro generador de código.

Nuestro primer objetivo es realizar la selección de instrucciones para cada una de las diferentes arquitecturas.

Debemos recordar que nuestra estrategia de la sección 5.3 para generar instrucciones toma como entrada árboles de bajo nivel, a partir de los cuales se realiza la selección óptima de instrucciones. Sin embargo, como acabamos de mencionar, nuestro analizador semántico entrega como salida un AST decorado, así que nuestra primer tarea consiste en realizar una transformación del AST decorado en árboles de expresiones[14]. Aquí debemos resaltar que los árboles de bajo nivel (entre otras cosas) exponen explícitamente las direcciones de variables y temporales, y recordemos, como se mencionó en la sección 5.2, que este direccionamiento dentro de los marcos de pila se suele hacer con ayuda ya sea del apuntador de pila o del apuntador de marco, por lo que estas características son claramente dependientes de la arquitectura. De esto, si queremos mantener nuestro compilador completamente reobjetivizable, debemos definir una interfaz que toma como entrada el AST decorado que produce el analizador semántico y entregue como salida una representación intermedia basada en árboles de expresiones. Asimismo requerimos que haya una implementación de esta interfaz por cada arquitectura diferente que el compilador soporte, para que de esta forma, si en un futuro se desea agregar una nueva arquitectura, únicamente se tendrá que realizar una nueva implementación de esta interfaz.

Para llevar a cabo esta labor se implementará un nuevo Visitante (del patrón Visitor) que realice el recorrido sobre el AST y que vaya construyendo los árboles de bajo nivel. Para esto último se utilizará el patrón Builder, en el que el Visitante que acabamos de describir fungirá como Director; se tendrá una clase que actúe como el Constructor, en la que se definan todas las operaciones para poder construir los árboles de bajo nivel.

[14]Utilizamos el término *árbol de expresión* de manera indistinta con el término *árbol de bajo nivel*.

Además, por cada arquitectura diferente que el compilador soporte, se implementará un ConstructorConcreto, el cual, con base en la arquitectura, sabrá las direcciones de memoria que debe generar en los árboles de bajo nivel. Dado este esquema, basta tener un único Visitante que recorra el AST decorado y que emita las instrucciones para construir los árboles de bajo nivel. Dependiendo del tipo de ConstructorConcreto que se esté utilizando, se construirán los árboles de bajo nivel adecuados al tipo de arquitectura. Además, como ya es costumbre, el patrón Builder nos servirá para construir diferentes representaciones de los árboles de expresiones, como pueden ser una basada en texto o una gráfica.

Una vez que hemos obtenido los árboles de expresiones correspondientes al tipo de arquitectura, utilizaremos la estrategia de la sección 5.3 para realizar la selección óptima de instrucciones, para lo que utilizaremos el generador de generadores de código `burg`.

La entrada de `burg` es una gramática que describe los patrones en forma de árbol que pueden implementar las instrucciones de una arquitectura específica junto con sus respectivas acciones semánticas, en las que se debe indicar que se generen las instrucciones que implementan el patrón correspondiente. Cabe señalar que durante esta etapa se puede suponer que la arquitectura cuenta con un número infinito de registros. `burg` emite como salida un archivo en C en el que implementa la estrategia de la sección 5.3 con base en la gramática que se le alimentó como entrada. De esta forma tenemos que elaborar un archivo de entrada de `burg` para la arquitectura x64, el cual contenga su gramática correspondiente junto con las acciones, en las que se coloca el código en C necesario para generar las correspondientes instrucciones de máquina, y hacer lo análogo para la arquitectura SPARC.

Cabe señalar que el seleccionador de instrucciones emitirá instrucciones de máquina, por lo que en este punto podemos elegir que emita código ensamblador o directamente código binario. Emitir código ensamblador tiene la ventaja que la depuración puede ser más sencilla, ya que se puede observar de manera más clara qué instrucciones está emitiendo el seleccionador de instrucciones; por otra parte tiene la desventaja que se necesitará realizar una etapa de ensamblado que finalmente genere el código de máquina; en cambio, generar código binario directamente nos evita tener que realizar una etapa más de ensamblado, pero es más difícil de depurar. De nuevo haremos uso del patrón Builder, para en las acciones semánticas de la gramática de entrada de `burg` utilizar las operaciones definidas en la clase que juega el papel de Constructor, y tener un ConstructorConcreto que genere código binario y otro ConstructorConcreto que genere código ensamblador.

En este punto, según lo descrito en 5.1, estaríamos listos para realizar, por primera ocasión, planificación de instrucciones. Sin embargo, por el momento no implementaremos esta etapa, aunque en un futuro se puede agregar. La planificación de instrucciones debe tomar como entrada el código que generó el seleccionador de instrucciones y debe entregar como salida código en el que la secuencia de instrucciones está ordenada de tal

forma que se saque un mayor provecho a la arquitectura.

Toca el turno de implementar el alojador de registros. Para hacer el alojador de registros modular y en el que sea fácil soportar una nueva arquitectura, hemos decidido definir un archivo en el que se describan las características relevantes de la arquitectura que le son útiles al alojador en registros. De esta forma, el alojador de registros puede leer dicha información antes de comenzar su funcionamiento y actuar en consecuencia, es decir, con base en la arquitectura para la cual se va hacer el alojamiento en registros.

La información relevante consiste de todas la convenciones y restricciones impuestas por la arquitectura, por ejemplo, cuáles son los registros de propósito general; qué registro es el apuntador de pila; si hay un apuntador de marco, qué registro cumple esta función; cuál es la convención de llamadas a funciones; si es que se pasan los argumentos en registros, en qué registros deben pasarse; si es que se debe regresa el valor que calcula una función, en qué registro debe regresarse. De esta forma tendremos una implementación del algoritmo que describimos en la sección 5.4 que sea capaz de alojar registros para cualquier arquitectura, siempre que se defina para ésta el archivo de entrada correspondiente que acabamos de describir.

Además, de forma general se utilizará el patrón Strategy para poder realizar en un futuroa las implementaciones de diferentes algoritmo de alojamiento de registros. En este momento sólo se contará con la implementación del algoritmo descrito en la sección 5.4, realizada según lo descrito en el párrafo anterior.

El alojador de registros tomará como entrada el código (en el que se está considerando que se cuenta con un número infinito de registros) que generó el seleccionador de instrucciones –código ensamblador o código binario–; con base en lo anterior, debe entregar como salida código ensamblador o código binario respectivamente, en el que las instrucciones únicamente utilicen los registros con los que cuenta la arquitectura para la que se está generando el código. Con este fin, una vez más haremos uso del patrón Builder, en el que la clase que se encargue de implementar el algoritmo descrito en la sección 5.4 será el Director y habrá un ConstructorConcreto que genere código binario y uno más que genere código ensamblador.

Con lo anterior concluimos el presente capítulo y con él el presente trabajo.

Epílogo

Durante el desarrollo del presente trabajo, y de nuestro compilador en particular, pudimos descubrir varios resultados y principios importantes, y diferentes líneas de investigación en desarrollo y por desarrollar; de las cuales a continuación describimos las que consideramos más importantes.

En general queda claro que en el ámbito de los compiladores existe una estrecha relación entre la teoría y la práctica, en la que ambas se retroalimentan, pues las herramientas que se utilizan para automatizar la creación de algunos de los componentes de un compilador descansan sobre bases teóricas sólidas y bien establecidas; su uso en el contexto de compiladores es sólo una aplicación particular de estas teorías. Por ello, la práctica se beneficia enormemente de la existencia de éstas, como es el caso de la teoría de autómatas finitos que se utiliza en los generadores de analizadores léxicos; la teoría de análisis en el caso de los generadores de anlizadores sintácticos; y BURS en el de los generadores de generadores de código. Por otra parte, mucho del desarrollo de estas teorías ha sido originado por una necesidad práctica, por lo que estos dos mundos convergen en la creación de compiladores. Sin embargo, como dijimos, muchas de las teorías abstractas en las que descansan las diferentes etapas de un compilador tienen muchas otras áreas de aplicación, como es el caso de análisis de flujo que se utiliza en el contexto de seguridad, sólo por mencionar un ejemplo. Por otra parte, muchos de los principios prácticos que se utilizan al desarrollar un compilador se pueden utilizar en muchos otros ámbitos al desarrollar software. Podemos concluir, entonces, que el conocimiento adquirido al desarrollar un compilador da al programador (y potencialmente futuro escritor o investigador de compiladores) una formación y visión amplia para saber qué conocimientos teóricos utilizar, en una solución concreta particular a los problemas que es común se presenten en las ciencias de la computación, así como los principios prácticos que se pueden utilizar para que esa solución sea limpia, clara y

eficiente.

Uno de los principios que se presentó más frecuentemente durante el desarrollo de nuestro compilador es el siguiente:

> *Dado un programa p, el cálculo de cierta información c que se obtiene con base en cierta información e, donde e únicamente está presente en tiempo de ejecución, si se mueve este cálculo a tiempo de compilación, el cálculo c se realizará una única vez en lugar de n, donde n es el número de veces que se ejecuta p.*

Cabe señalar que, generalmente, el número de cálculos necesarios para determinar c será mayor si se calcula en tiempo de compilación, pues no se cuenta con contexto único sino que se tienen que prever los posibles contextos.

Debemos también tener presente que en la mayoría de las situaciones que se nos presentan, uno de nuestros principales objetivos es minimizar el tiempo de ejecución.

Por ejemplo, mientras estudiábamos la etapa de análisis, se nos presentó la necesidad de calcular el símbolo ss que podía seguir a un símbolo s de la gramática, lo cual sólo era posible hacerlo con precisión durante la ejecución del programa. Al no poder conocer ss en tiempo de compilación, se tuvieron que tomar en cuenta todos los posibles símbolos siguientes de s que pudiesen llegar a presentarse, lo que nos llevó a hacer uso de los conjuntos $FOLLOW$. Debe resaltarse, sin embargo, que esto no era estrictamente necesario, pues en su lugar simplemente se pudo haber calculado ss durante la ejecución[15]. No obstante, lo que nos motivó a calcular todos los posibles símbolos que pudiesen seguir a s durante la ejecución, en tiempo de compilación, fue que si calculábamos ss en tiempo de ejecución, dicho cálculo tendría que realizarse cada una de las veces que se ejecutaran las secciones del programa donde s aparece; en cambio, si calculábamos todos los posibles símbolos siguientes en tiempo de compilación, aunque el número de cálculos necesarios para obtenerlos fuese mayor, éstos tendrían que ejecutarse una única vez durante la compilación del programa, con lo que obtendríamos como resultado que el tiempo de compilación del programa sería mayor, pero el tiempo de ejecución del programa sería menor. Es claro que el número de veces que se compila un programa es mucho menor que el número de veces que se ejecuta.

Un caso más en el que se nos presentó el principio anterior fue durante la verificación de tipos, en la que movimos, siempre que fue posible, una verificación de tipos de tiempo de ejecución a tiempo de compilación.

Además, el principio anterior es en el que descansa un gran número de optimizaciones, como la de nuestro ejemplo del capítulo 4, el determinar en tiempo de compilación el método que se debe llamar en los lenguajes que cuentan con despacho dinámico, conocida como análisis de apuntadores (*pointer analysis*), y donde la mayoría de estas

[15]Debe notarse que incluso el número de cálculos necesarios para calcular el símbolo siguiente es menor en ejecución que calcular todos los posibles símbolos siguientes en tiempo de compilación. Pero esto es suponiendo que dicho cálculo sólo se haga una vez.

optimizaciones hacen uso de teoría desarrollada para análisis de flujo que, como estudiamos, está compuesto por el análisis de flujo de control y el análisis de flujo de datos.

Este principio ha sido llevado al nivel del programador en lenguajes como C++, en donde, como vimos en el capítulo 4, mediante la especialización de plantillas el programador puede decidir estáticamente qué definición de clase utilizar, lo que posteriormente originó que se desarrollara la metaprogramación estática, cuando los programadores se dieron cuenta que mediante el uso de las plantillas de C++ podían mover cálculos de tiempo de ejecución a tiempo de compilación. Posteriormente (en tiempo reciente) se demostró que el sistema de plantillas de C++ es Turing completo y se ha desarrollado toda una teoría de metaprogramación estática que tiene su principal encarnación concreta en C++ y que ha influido enormemente para que este tipo de características se incluyan en otros lenguajes de programación, como es el caso de Haskell, tal como se describen en [Hud+07].

Este principio converge con el que es el principal objetivo de un sistema de tipos, que mencionamos en el capítulo 4 y que aquí enunciamos de la siguiente manera:

> *Si un programa tiene un error, es mejor detectarlo lo antes posible.*

El tiempo sensato, en el contexto de un programa, para "lo antes posible" es el tiempo de compilación. De esta forma, ambos principios convergen hacia y se encuentran en el tiempo de compilación. Incluso se pueden llevar a nivel de abstracción mayor y aplicarse en la vida real en situaciones que se presentan en la vida cotidiana.

Aunque a veces no se cuenta con la información necesaria en tiempo de compilación. En el caso de nuestro primer principio en ocasiones es imposible determinar todas las posibilidades[16] y en el caso del segundo que no se cuenta con la información de tipos, lo que nos motiva a buscar caminos que nos lleven a contar con esa información.

En el contexto de compiladores lo anterior se traduce en diseñar lenguajes que provean la mayor cantidad posible de información útil al compilador. En particular, para el segundo principio, esto se logra con un lenguaje de programación que permita determinar en tiempo de compilación los tipos de un programa[17]. Uno de los argumentos que se ha utilizado a favor de los lenguajes dinámicamente tipados es que éstos son, en su gran mayoría, implícitamente tipados y dan al programador la libertad de enfocarse en los pasos del algoritmo que se está implementando, sin preocuparse de definir con qué tipos de entidades se está trabajando, es decir, sin escribir explícitamente el tipo de éstas; en su contra está que en la mayoría de estos lenguajes es imposible determinar sus tipos estáticamente y, por lo tanto, se tiene que realizar la verificación de tipos en tiempo de ejecución. En favor de los lenguajes estáticamente e implícitamente tipados podemos mencionar que éstos ofrecen de la misma manera esta libertad, pero sin perder

[16] O el número de éstas puede ser de tal magnitud que no sea práctica.

[17] Lo anterior es un ejemplo de cómo las implementaciones de los lenguajes han influido en el diseño de éstos.

el segundo principio, es decir, en éstos si es posible determinar estáticamente sus tipos[18]. Con esto debe resaltarse y quedar claro que los programas de lenguajes estática e implícitamente tipados son, en general, más eficientes que sus correspondientes programas equivalentes de lenguajes dinámica e implícitamente tipados.

Uno de los lenguajes más representativos de los lenguajes estática e implícitamente tipados es ML. Uno de los paradigmas de programación con el que no cuenta ML, y que una gran parte de los programadores utiliza, es la orientación a objetos; por ello se desarrolló el lenguaje *OCaml* (*Objective Caml*), que es una extensión con orientación a objetos del lenguaje Caml un lenguaje de la familia ML.

Al poder determinarse los tipos de los programas en ML y OCaml da lugar a que sus correspondientes programas sean más eficientes; incluso existen comparaciones de estos lenguajes con lenguajes explícita y estáticamente tipados como es el caso de C++ (para ver los detalles de las comparaciones consultar [Conb] y [Cona]), en las que se muestra que para algunos programas incluso las programas en ML son más eficientes y los de Ocaml presentan una eficiencia muy similar, comparados con programas de C++.

Por lo anterior, como trabajo futuro el autor del presente trabajo pretende investigar la teoría de tipos y desarrollar un sistema de tipos para el cual exista un algoritmo de inferencia de tipos que permita la mayor libertad y expresividad posible, y una vez obtenido, crear un nuevo lenguaje de programación que incluya el sistema de tipos desarrollado, que cuente con las características necesarias para permitir al programador realizar metaprogramación estática y, finalmente, incluya un modelo de concurrencia basado en el Cálculo Π desarrollado por Milner, para dotar a los programadores de un lenguaje con diseño conciso, claro, limpio y elegante que cuente con las características antes mencionadas.[19]

Por otra parte, al utilizar las herramientas de desarrollo de compiladores descubrimos un tanto con sorpresa que la mayoría de ellas[20] están poco documentadas y que la documentación disponible es poco didáctica, además de que cuentan con un soporte pobre de orientación a objetos en el caso de flex y bison, y no cuentan en lo absoluto con soporte de orientación a objetos en el caso burg.

En el caso de flex y bison son herramientas que han sido (y son) ampliamente utilizadas durante varios años, por lo que entre las posibles tareas por desarrollar se encuentra la de ampliar la documentación de estas herramientas y la de añadir a cada una de ellas soporte de orientación a objetos.

burg en cambio no ha tenido un uso tan amplio como el de flex y bison, debido a que, en primer lugar, es una herramienta menos compleja y grande que las anterio-

[18] Por medio de un algoritmo de inferencia de tipos que se ejecuta en tiempo de compilación.

[19] Que han demostrado ser las características que permiten mayor expresividad y libertad al programador, mientras que al mismo tiempo permiten que se generen programas limpios, seguros, eficientes y que son propensos a cumplir propiedades, lo cual es algo sumamente deseable.

[20] En particular nos estamos refiriendo a todas las que nosotros utilizamos.

res, por lo que los grupos que desarrollan compiladores a veces implementan su propia herramienta para la selección de instrucciones. Por otra parte, existen herramientas propietarias con el mismo propósito que ofrecen un mayor soporte, la más conocida BEG, y finalmente muchos compiladores utilizan otros algoritmos (hechos a la medida) que toman como entrada representaciones intermedias distintas a la de los árboles de bajo nivel. En burg la programación dinámica se hace al generar el generador de código, por lo que los costos asociados que se utilizan tienen que ser constantes; existe una herramienta parecida a burg de nombre iburg, en la que la programación dinámica se hace en tiempo de compilación del programa[21] y, por lo tanto, los costos asociados no necesariamente tienen que ser constantes (se pueden calcular en tiempo de compilación del programa), lo que permite una mayor flexibilidad pero es menos eficiente. El uso de iburg ha sido mayormente con fines didácticos y de igual forma no es una herramienta con un uso tan amplio como flex y bison. Con base en lo anterior se invita al lector, de manera similar que en el caso de flex y bison, a que agregue una documentación más amplia y didáctica a burg e iburg[22] y se implemente soporte de orientación a objetos en ellos.

Además, en el caso de bison, como estudiamos en el capítulo 3, existen diversos algoritmos para crear analizadores LALR(1). En particular, como dijimos, bison implementa el algoritmo descrito en [DP82], que es más eficiente que el descrito en [Aho+07] que se utiliza en yacc. Sin embargo, existen varios algoritmos con el mismo propósito que se han desarrollado, y son más recientes que los mencionados (y de los que se tiene conocimiento son menos complejos, como el descrito en [BL89]) y que, sin embargo, no han sido adaptados por una herramienta en amplio uso como bison, por lo que se anima al lector a hacer una comparación de cada uno de ellos y determinar cuál es el mejor para que se utilice en la práctica; de ser éste diferente al que se utiliza en bison, agregárselo o, de ser necesario, crear un nuevo generador de analizadores sintácticos basado en él.

Por otra parte sería deseable contar con una herramienta de ayuda en la resolución de conflictos desplazamiento/reducción y reducción/reducción que surgen en la práctica durante el desarrollo de analizadores sintácticos utilizando bison, es decir, estudiar el problema de determinar si una gramática es LALR(1), y cuando ésta no lo sea, determinar si se puede realizar una transformación algorítmica de dicha gramática a una gramática LALR(1). En el caso que la transformación sea posible, conseguir que la herramienta la realice automáticamente; en caso contrario, que informe al programador qué estados están involucrados en el o los conflictos, dé la mayor información disponible y, de ser posible, sugerencias al programador sobre cómo resolver el o los conflictos.

[21] De donde el tiempo de compilación del programa será mayor que si los cálculos correspondientes a la programación dinámica se hicieran en tiempo de generación del generador de código.

[22] iburg, al igual que burg, no cuenta con soporte de orientación a objetos en lo absoluto.

Cabe señalar que en el mayoría de los lenguajes de programación se han desarrollado herramientas inspiradas en las anteriores (flex, bison y burg) en las que se suele contar con una documentación más amplia que en éstas y en algunos casos con un buen soporte de orientación a objetos.

En la etapa de generación de código, como mencionamos en el capítulo 5, la forma SSA recientemente ha capturado la atención de varios investigadores, por lo que se anima al lector que investigue nuevos algoritmos de selección de instrucciones que tomen como entrada un programa en forma SSA; en cuanto al alojamiento en registros sobre la forma SSA tal como se menciona en [Bri06], se presentan varios inconvenientes: En primer lugar la forma SSA no se preserva cuando se hace un derrame y, como consecuencia, es necesario reconstruir la forma SSA después de cada derrame, lo que puede ser muy costoso; en segundo lugar, la destrucción de la forma SSA después de la asignación de registros y localidades de memoria puede introducir un número significativo de instrucciones de derrame y puede incluso hacer introducir derrames adicionales, lo que hasta el momento ha frenado la adopción de los alojadores de registros basados en la forma SSA.

Adicionalmente, las restricciones impuestas en las convenciones de las plataformas acerca del uso de los registros, hace que el alojamiento y asignación de registros se vuelvan más complejos. En recientes publicaciones [Qui08] se ha atacado el alojamiento en registros basado en la forma SSA y tomando en cuenta estas restricciones impuestas en el uso de los registros. Por ello se invita al lector a seguir investigando ese problema.

Conclusiones

Hemos utilizado este material, aunque en una forma menos definida que la que se presenta en este trabajo, obteniendo de los estudiantes involucrados interés por conocer más del tema y satisfacción por el trabajo realizado.

Dada la modularidad del compilador y gracias al uso de patrones de diseño, hemos conseguido un compilador en el que, dado el caso, se puede sustituir el comportamiento de los módulos para ofrecer distintos algoritmos. También es importante mencionar que este diseño permite que, si se presenta que un estudiante no termine satisfactoriamente alguna de las etapas, puede usar la implementada en nuestro compilador de manera sencilla, para proseguir con las etapas posteriores. Por último, este compilador se puede tomar como base para explorar técnicas más avanzadas para la optimización global o local, para el flujo del control y de datos, y otros aspectos que son objeto de investigación o de cursos avanzados en el tema.

APÉNDICE A

C++ para programadores Java

A.1. Introducción

Desde la aparición de las computadoras digitales, surgió la inherente necesidad de programar aplicaciones que se ejecutaran en ellas. Esta programación se hizo en un principio mediante el lenguaje de máquina, con la aparición posterior del lenguaje ensamblador y después de éste los lenguajes de alto nivel que se conocen hoy en día. Una muestra de la evolución de estos lenguajes es la familia de lenguajes estructurados llamados "estilo *Algol*". Tal vez el más utilizado entre ellos es *C*, que es un lenguaje de propósito general que fue diseñado por Dennis Ritchie en los laboratorio *Bell* de *A&T*; originalmente su uso fue como un lenguaje de alto nivel para desarrollar sistemas operativos, en particular *UNIX* en ese momento, ya que permite acceso a las instrucciones de bajo nivel necesarias para poder desarrollar este tipo de sistemas. Anteriormente Ken Thompson usó el lenguaje ensamblador junto con el lenguaje *B* (el predecesor de C) para desarrollar las primeras versiones de UNIX. *C++* fue desarrollado por Bjarne Stroustrup a mediados de los ochentas en los laboratorios Bell, para dotar a C de nuevas características, siendo la orientación a objetos la principal de éstas; por ende C++ es una extensión de C (estaba implementado primero como un preprocesador) y en principio todo programa correcto en C es un programa correcto en C++. C++ ha sido utilizado desde su aparición como lenguaje en el desarrollo de grandes aplicaciones presentes en la industria y en aplicaciones de cómputo científico.

Java fue desarrollado por James Gosling en la compañía *Sun Microsystems*, diseñado como un lenguaje independiente de plataforma, con alta portabilidad para ser usado como lenguaje de desarrollo de aplicaciones en todo tipo de dispositivos electrónicos. En

general Java tiene una sintaxis parecida a C++, pero con un grado de abstracción mayor a este último. Para ganar su portabilidad Java corre sobre una máquina virtual, llamada máquina virtual Java (*Java Virtual Machine*, *JVM*) lo que permite que un programa ejecutable sea independiente tanto de la arquitectura como del sistema operativo. Java además, a diferencia de C y C++, es un lenguaje seguro ya que el uso de la JVM permite tener un control mucho más estricto de accesos a los datos o al código binario de un programa. Comparado con C++ hace más sencilla la vida al programador ya que libera a este último de hacer tareas de bajo nivel que éste tendría que hacer en C++, como el liberar la memoria de los datos que ya no se utilizan (Java cuenta con un recolector de basura) y trabajar con apuntadores.[1] El principal precio pagado por Java por estas características es el rendimiento, ya que al correr en una máquina virtual el código debe ser primero compilado a un lenguaje intermedio, que es el que la máquina virtual interpreta en lenguaje máquina (o compila a código binario nativo, dependiendo de la implementación del lenguaje) y esto degrada el tiempo de ejecución de la aplicación, razón por la cual las aplicaciones en las que se necesite un alto nivel de eficiencia en el tiempo de ejecución (como aplicaciones de tiempo real) no son candidatas a ser desarrolladas en Java pero si en C++; la máquina virtual y el propio compilador de Java distribuido por *Oracle* están desarrollados en C++.

El presente trabajo tiene como objetivo mostrar algunas diferencias sintácticas y semánticas de ambos lenguajes, para dar al programador la posibilidad de implementar la solución de un problema dado en ambos lenguajes o pasar sin demasiadas complicaciones de uno a otro.

A.2. La pila y el heap

Comenzaremos nuestra discusión hablando de dos estructuras que toda aplicación escrita en cualquiera de estos dos lenguajes tiene en tiempo de ejecución, a saber el *heap* y la *pila*. En el heap residen todos aquellos datos o estructuras que deben sobrevivir a las llamadas a funciones, es decir, que no son datos que deben estar vivos únicamente durante la ejecución de una función dada, –aunque en el caso de Java se colocan en el heap todos los objetos creados en cualquier momento de la ejecución del programa–. En particular, es común utilizar el heap para almacenar las estructuras de datos dinámicas, es decir, aquellas que su tamaño puede variar en tiempo de ejecución y no puede ser determinado estáticamente en tiempo de compilación. La pila es una estructura que permite de forma sencilla manejar las llamadas a las funciones y por tanto el alcance de las variables. Para comprender mejor el funcionamiento de estas estructuras veamos cómo es que estas trabajan en la arquitectura *x86*. Iniciemos con una breve introducción al lenguaje ensamblador de la arquitectura x86. La arquitectura x86 de *Intel* tiene los siguientes registros de propósito general:

[1] En la terminología de Java *referencias*.

- ax registro acumulador (*accumulator*).
- bx registro base (*base*).
- dx registro de datos (*data*).
- di registro de índice de destino (*target index*).
- si registro de índice de origen (*source index*).
- bp registro apuntador de base (*base pointer*).
- sp registro apuntador de pila (*stack pointer*).

Los dos últimos se usan para apuntar a datos que están en la pila y el registro apuntador de base es el registro que se utiliza como apuntador de marco en la arquitectura x86. También cuenta con los siguientes registros de segmento:

- cs segmento de código (*code segment*).
- ds segmento de datos (*data segment*).
- ss segmento de la pila (*stack segment*).
- es segmento adicional (*extra segment*).
- fs segmento adicional.
- gs segmento adicional.

Asimismo cuenta con el *apuntador de instrucción* (*instruction pointer*, *ip*) que se usa junto con el *cs* para apuntar y traer la siguiente instrucción a ejecutar.

Los registros de índice por lo general se usan como apuntadores en operaciones de cadenas de bajo nivel, pero también pueden ser utilizados como registro de propósito general. En un programa en ensamblador se tienen las siguientes secciones:

- .data datos inicializados.
- .bss datos sin inicializar.
- .text código.

Además cuenta con diferentes directivas para diversos propósitos; una de ellas, *.global*, permite declarar una sección de código que almacene datos globales, como pueden ser etiquetas de funciones, ofrece la posibilidad que una etiqueta que corresponde a una función sea vista desde otros módulos (archivos) por el ligador (*linker*). El registro *ss* se carga en tiempo de ejecución con la dirección de memoria del inicio del segmento de memoria que corresponde a la pila (usualmente los datos locales a la función también se guardan aquí) y el *sp* apunta al tope de la pila. Una operación *push* inserta un elemento en el tope de la pila y decrementa en n el *sp*, dejándolo apuntando al nuevo elemento. Una operación *pop* lee el elemento apuntado por el tope de la pila y luego le

suma n al *sp*, es decir, extrae el elemento de la pila y pone el valor en el registro indicado en el operando de pop. La pila es conveniente para guardar datos temporales; también se usa para llamar a funciones, pasar parámetros y almacenar variables locales.

Para hacer una llamada a función, la arquitectura x86 cuenta con la instrucción *call*, la cual guarda la dirección de la siguiente instrucción y hace un salto incondicional al código de la función (la dirección asignada en tiempo de ejecución a la etiqueta de la función).

La instrucción *ret* hace un pop a la pila y salta a la dirección que sacó (tenemos que asegurar que no haya otro dato ahí). La convención para llamar a una función es la siguiente:

El que llama tiene que poner los argumentos en la pila y hacer la llamada por medio de la instrucción call; al que se llamó tiene que guardar el valor actual del *bp* haciendo un `push bp`, después hacer `bp=sp` y `mov sp, bp`, realizar el código de la función, restaurar el valor anterior de *bp* haciendo `pop bp` y por último ejecutar `ret`. A continuación, el que llamó se tiene que encargar de remover los argumentos de la pila sumándole al *sp* con una instrucción `add` el número de bytes, que depende del número de argumentos, o haciendo tantas operaciones pop como argumentos se hayan pasado. Cuando tenemos variables locales les hacemos push o les reservamos espacio con una instrucción *sub* que tiene como argumento el número de bytes que ocupen las variables locales en la función llamada; cuando termina, tenemos que hacer operaciones pop o simplemente hacer `sp=bp`.

```
1 etiqueta_funcion:
2     push %ebp
3     mov  %esp, %ebp
4     sub  BYTES_LOCALES, %esp #espacio para variables locales
5     #código de la función
6     mov  %ebp, %esp #liberamos el espacio de las variables locales
7     pop  %ebp
8     ret
```

La pila, después de ingresar a la función llamada y hacer lo descrito anteriormente, contiene: los parámetros, la dirección de retorno, el valor original del *bp* y las variables locales; a esto se le llama *marco de pila* (*stack frame*). Toda invocación de una función crea un marco de pila dentro de la pila.

Hay una instrucción que nos simplifica manejar el *bp* y reservar espacio para las variables locales, a saber *enter*, que recibe como primer argumento el número de bytes que necesitan las variables locales y el segundo argumento siempre es cero; también existe una instrucción que nos simplifica liberar el espacio de las variables locales y restaurar el *bp*, y que es la instrucción *leave*

```
1 etiqueta_funcion:
2    enter BYTES_LOCALES, 0
3    #código de la función
4    leave
5    ret
```

Como podemos notar, la mayoría de las operaciones tienen uno o más operandos, que son los argumentos y el destino que almacenará el resultado de la operación; los argumentos pueden ser constantes o pueden ser leídos de los registros o de la memoria y el resultado puede ser guardado en registros o en memoria. Tenemos tres tipos de operando, a saber:

- inmediato, cuando se trata de una constante y se les antepone un `$`.
- registro, que denota el contenido de un registro y se antepone un `%` al nombre del registro.
- memoria, que se refiere, como su nombre lo indica, a una referencia a memoria, donde se accede a alguna localidad de memoria (que desde luego tiene una dirección).

Los modos de direccionamiento con los que cuenta la arquitectura x86 son los siguientes:

Tipo	Forma	Valor del operando	Nombre
Inmediato	$\$Imm$	Imm	Inmediato
Registro	$\%r_a$	r_a	Registro
Memoria	Imm	$M[Imm]$	Absoluto
Memoria	$(\%r_a)$	$M[r_a]$	Indirecto
Memoria	$Imm(\%r_b)$	$M[Imm + r_b]$	Base+desplazamiento
Memoria	$(\%r_b, \%r_i)$	$M[r_b + r_i]$	Indexado
Memoria	$Imm(\%r_b, \%r_i)$	$M[Imm + r_b + r_i]$	Indexado
Memoria	$(, \%r_i, s)$	$M[r_i \cdot s]$	Índice escalado
Memoria	$Imm(, \%r_i, s)$	$M[Imm + r_i \cdot s]$	Índice escalado
Memoria	$(\%r_b, \%r_i, s)$	$M[r_b + r_i \cdot s]$	Índice escalado
Memoria	$Imm(\%r_b, \%r_i, s)$	$M[Imm + r_b + r_i \cdot s]$	Índice escalado

Donde r_a se refiere al contenido del registro cuyo nombre es a, $M[x]$ se refiere al contenido de la memoria en la dirección x; s es un factor de escalado; r_a, r_b representan registros cualesquiera, mientras que r_i es uno de los registros de índice.

Analicemos el siguiente ejemplo que calcula la función recursiva de Ackerman. Se declara `_start` y `ackerman` como etiquetas con alcance global, que son etiquetas que corresponden a las funciones `_start` y `ackerman` respectivamente; la función `_start` es el punto de entrada de la ejecución del programa, que corresponde a la función `main` en C, C++ y al método `main` en Java. La función `_start` mete a la pila de forma directa los argumentos para los cuales se calculará la función de Ackerman, que en este caso son los argumentos 3 y 4; se llama a la función `ackerman`, remueve los argumentos de la función sumándole 8 al apuntador de pila *sp* –dado que fueron dos enteros y cada entero utiliza 4 bytes– y luego mueve el resultado de la función, que se encuentra en *eax* a *ebx*, que es de donde lo leeremos una vez ejecutado el programa. Por último pone un 1 en *eax* que es un argumento para la llamada al sistema `exit`, y realiza la llamada al sistema.

```
1  #Programa en ensamblador de 32 bits x86
2  #que calcula la función de Ackerman
3  #Autor: Angel Francisco Zúñiga Chávez
4  .section .data
5
6  .section  .text
7
8  .globl _start
9  .globl ackerman
10
11 _start:           #Iniciamos calcularemos A(3,4)
12         pushl $3  #metemos m a la pila
13         pushl $4  #metemos n a la pila
14         call ackerman
15         addl $8, %esp
16         movl %eax, %ebx
17         movl $1, %eax
18         int $0x80
19
20 #.type ackerman,@function
21 ackerman:
22         pushl %ebp
23         movl %esp, %ebp
24         movl 12(%ebp), %ebx #movemos m al registro ebx
25         movl 8(%ebp), %eax  #movemos n al registro eax
26         addl $1,%eax        #regresamos n+1
27         cmpl $0, %ebx       #comparamos si m es cero
28         je end_ackerman     #fin del primer caso
29
```

```
30          movl 8(%ebp), %eax  #movemos n al registro eax
31          cmpl $0, %eax       #si n es diferente de cero
32          jne three           #brincamos al tercer caso
```

```
33          decl %ebx           #inicio segundo caso m=m-1
34          pushl %ebx          #pasamos m-1 como argumento
35          pushl $1            #pasamos n=1 como argumento
36          call ackerman
37          jmp end_ackerman    #fin del segundo caso
38
39  three:                      #inicio del tercer caso
40          pushl %ebx          #metemos m
41          decl %eax           #n=n-1
42          pushl %eax          #metemos n-1
43          call ackerman
44          movl 12(%ebp), %ebx #recargamos m
45          decl %ebx           #m=m-1
46          pushl %ebx          #metemos m-1
47          pushl %eax          #metemos ackerman(m,n-1)
48          call ackerman
49                              #fin del tercer caso
50  end_ackerman:
51          movl %ebp, %esp
52          popl %ebp
53          ret
```

A grandes rasgos todo programa en ejecución tiene las siguientes áreas:

- Código. El código ejecutable de un programa se encuentra dentro de la sección delimitada por la directiva *.text* y comienza en una dirección fija (relocalizable).
- Datos. En la sección *.data* se encuentran todos los datos inicializados estáticamente del programa mientras que en la sección *.bss* se encuentran los datos sin inicializar. Los datos de ambas secciones, por omisión, tienen alcance de archivo, aunque si dichas secciones están precedidas por la directiva *.global* entonces tendrán alcance global, es decir, los datos podrán accederse desde otros archivos. Ambas áreas tienen un tamaño fijo determinado estáticamente, es decir, su tamaño no cambia en tiempo de ejecución.
- Heap. Inmediatamente después del área de código y datos se encuentra el área correspondiente al heap, que es una área dinámica que crece (hacia las direcciones más altas de memoria[2]) y decrece en tiempo de ejecución, como resultado de las

[2]Lo cual es algo dependiente de la plataforma y nosotros nos estamos refiriendo a la plataforma Linux x86.

llamadas a las funciones *malloc* y *free* en C y los operadores *new* y *delete* en C++. Además de los anteriores, se asigna y se libera espacio a la aplicación si en algún momento es necesario más espacio de lo que el sistema operativo le designó inicialmente en el heap, haciendo uso de la llamada al sistema *sbrk*[3] para hacer más grande este espacio –inicialmente el sistema operativo le concede un espacio por omisión–. En Java se asigna espacio en el heap a los objetos, que se crean mediante el operador *new* y se libera conforme lo dicta el recolector de basura.

- Pila. En el otro extremo de la memoria asignada a un proceso está la pila, que la mayoría de los compiladores utilizan para implementar las llamadas a funciones. Como en el caso del heap, la pila crece (en este caso hacia las direcciones más bajas de memoria) y decrece dinámicamente durante la ejecución del programa. En particular, cada vez que se llama a una función la pila crece y cada vez que termina una función la pila decrece.

Todos aquellos datos que se crean utilizando el operador *new* se encuentran en el heap, mientras que todos aquellos datos locales que se crean dentro de una función (sin el uso de *new*) se encuentran en la pila.[4] Un punto importante a notar, y que veremos más adelante, es que los datos locales que se crean dentro de una función dejan de existir cuando la función termina de ejecutarse.

A.3. C y C++

Cuando compilamos un programa escrito en C, utilizando el compilador `gcc`, se ejecutan las siguientes etapas:

1. Preproceso. Es la primera etapa que se lleva a cabo. Se llama al preprocesador de C –que es `cpp` en el caso del entorno UNIX– quien detecta las directivas del preprocesador que comienzan con **`#`** y las remplaza con el correspondiente texto, esto es, si tenemos el enunciado **`#include`** `<stdio.h>` en el archivo fuente, el preprocesador remplaza dicho enunciado con el contenido del archivo de sistema `stdio.h`.
2. Compilación. El compilador se encarga entonces de generar un archivo de texto que corresponde a la traducción del código fuente a lenguaje ensamblador.
3. Ensamblado. Es turno del ensamblador –`as` en el entorno UNIX– que traduce el código en lenguaje ensamblador a lenguaje de máquina relocalizable, también conocido como código objeto.

[3]Cabe señalar que las llamadas al sistema son desde luego dependientes del sistema operativo. Aquí estamos suponiendo que trabajamos con sistemas operativos tipo UNIX.

[4]En realidad, cuando se hace uso del operador *new* se crea en la pila un apuntador hacia un dato que se crea en el heap.

4. Enlazado. El enlazador –`ld` en el entorno UNIX– enlaza el código objeto con el código de las bibliotecas, para finalmente obtener un archivo ejecutable listo para ser cargado en memoria.

Veamos ahora un ejemplo sencillo de un programa en C

```
1 #include <stdio.h>
2
3 int main void(){
4    printf("hola mundo");
5 }
```

Se declaran primero las directivas para el preprocesador, **`#include`** `<stdio.h>`, para que estén disponibles las funciones de entrada/salida de la biblioteca de C contenidas en el archivo `stdio.h`. Cuando se desea incluir un archivo que no es de la biblioteca se usan `""` en lugar de `<>`, de la siguiente forma: **`#include`** `"miarchivo.h"`.

C++ hereda la biblioteca de C, pero cuenta con una nueva biblioteca estándar propia, donde está incluida la *Standard Template Library* (*STL*); se recomienda, siempre que sea posible, utilizar la nueva biblioteca, aunque como C++ es una extensión de C, varias de las aplicaciones que fueron escritas en C fueron integradas a aplicaciones en C++; por tanto es necesario en ocasiones trabajar con aplicaciones que utilizan la biblioteca de C aunque se este programando en C++. El programa anterior es válido en C++, aunque se adoptó como convención para los programas en C++ que hacen uso de la biblioteca de C el anteponer la letra "c" en el nombre del archivo de la biblioteca C que se está utilizando y no incluir la extensión, de la siguiente forma:

```
1 include<cstdio>
2
3 int main(void){
4    printf("Hola mundo\n");
5 }
```

Veamos ahora el programa análogo que hace uso de la biblioteca de C++:

```
1 #include <iostream>
2
3 int main(void){
4
5    std::cout << "hola mundo" << std::endl;
6
7 }
```

En general, la nueva biblioteca de C++ que hace uso de la orientación a objetos es más segura, sencilla, elegante y limpia, comparada con la biblioteca de C.

A.3.1. Tipos de datos

Los tipos de datos primitivos en C y C++ son **int**, **char**, **float** y **double**.

int es un entero que puede ser precedido por las palabras **short** o **long**, que denotan el tamaño del entero, y por **unsigned** o **signed**, en cuyo caso se usa o no, según sea el caso, un bit para el signo; **char** es un carácter que en general corresponde a un carácter del código *ASCII*; también se le pueden anteponer las palabras **signed** y **unsigned** para utilizar o no un bit de signo; **float** y **double** son números de punto flotante; **double** admite que se anteponga la palabra **long**. En general el tipo **double** es el que más se utiliza y es con el que trabajan las funciones de la biblioteca estándar; en Java su sintaxis es análoga, aunque en C y C++ no se especifica explícitamente qué tamaño deben tener todos los tipos anteriores y éste puede cambiar de una arquitectura a otra; no así en Java, donde en la especificación de la JVM se dice el espacio que ocupa cada uno de ellos: en el caso de los enteros, un **short** ocupa 16 bits, un **int** ocupa 32 bits y un **long** ocupa 64 bits; en C++ un entero **int** debe ser al menos del tamaño de un **short** y un **long** debe ser al menos tan largo como un **int**. Veamos el siguiente ejemplo

```
1  #include <iostream>
2
3  int main(){
4    int x;
5    double y;
6    std::cout << "Escribe un número entero: ";
7    std::cin >> x;
8    std::cout << "Escribe un número de punto flotante: ";
9    std::cin >> y;
10   double r = x*y;
11   std::cout << "El resultado de la multplicación es: "
12             << r << std::endl;
13
14 }
```

Como se observa, en el código anterior tenemos dos variables, una de tipo **int** y otra de tipo **double**; enseguida, haciendo uso de las funciones de la biblioteca estándar, se lee un número en cada una de ellas y se guarda el resultado de la multiplicación en una nueva variable de tipo **double**, que se imprime en pantalla. Cabe mencionar que en la operación aritmética que involucra variables de diferentes tipos, se realiza una coerción (*cast*) implícita al tipo de mayor precisión que, en este caso, hace que el tipo de la variable `y` se convierta a **double**.

Originalmente C no cuenta con un tipo booleano, en su lugar el valor `0` se evalúa a falso, mientras que todo valor distinto de `0` se evalúa a verdadero; por tanto una práctica común es incluir las siguientes directivas:

```
1 #define TRUE 1
2 #define FALSE 0
```

En C++ se incorporó el tipo **bool** para representar variables booleanas y puede tomar los valores **true** o **false**; sin embargo, internamente aún se siguen tratando como enteros donde 1 representa **true** y 0 representa false:

```
1 include <iostream>
2 using namespace std;
3
4 int main(){
5   bool b = true;
6   cout << "El valor es: " << b << endl;
7   b = false;
8   cout << "Ahora el valor es: " << b << endl;
9 }
```

El programa anterior mostrará primero 1 y luego 0. En Java se utiliza el tipo boolean:

```
1 public class Muestra{
2     public static void main(String[] args){
3         boolean b = true;
4         System.out.println("El valor es: "+b);
5         b = false;
6         System.out.println("Ahora el valor es: "+b);
7     }
8 }
```

En este caso se mostrará primero **true** y después **false**.

El tipo **void** se utiliza para denotar que una función o método no devuelven un valor; en C y C++ también se usa para denotar un tipo de apuntador que puede apuntar a una variable de cualquier tipo.

A.3.2. Constantes

En C originalmente, era común declarar una constante haciendo uso del preprocesador de la siguiente manera:

```
1 #include <stdio.h>
2 #include <math.h>
3 #define PI 3.14159
4
```

```
5  double area(double radio);
6
7  int main(void){
8    double a = area(5);
9    printf("El area del circulo es: %f\n",a);
10 }
11
12 double area(double radio){
13   return PI * pow(radio,2);
14 }
```

En el código anterior, en el enunciado **#define** PI 3.14159 se indica al preprocesador que siempre que encuentre la palabra PI la reemplace por 3.14159; aunque esto funciona, tiene algunas desventajas, pues al ser realizado por el preprocesador no cuenta con la verificación de tipos del compilador, no se puede obtener una dirección de la constante por lo que no se puede usar con apuntadores y, por último no hay control sobre el alcance.

C++ introdujo un nuevo modificador para definir una constate (que también está presente en el nuevo estándar de C), el modificador **const** que se utiliza de la siguiente forma:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  const double PI = 3.14159;
5
6  double area(double radio);
7
8  int main(void){
9    double a = area(5);
10   printf("El area del circulo es: %f\n",a);
11 }
12
13 double area(double radio){
14   const double PI = 3.1415926;
15   printf("La direccion de la constante PI es: %p\n", &PI);
16   return PI * pow(radio,2);
17 }
```

Aquí, el enunciado **const double** PI=3.14159 declara una constante global de tipo **double** de nombre PI; en este caso se cuenta con verificación de tipos, se puede obtener

la dirección de la constate `PI` y además se cuenta con alcance. Adicionalmente, la definición **`const double`** `PI = 3.1415926` define una constante local `PI` dentro del alcance de la función `area` y está oculta en dicha función la constante global del mismo nombre antes declarada.

En Java el modificador con el que contamos para declarar constantes es `final`; sin embargo, cuando trabajamos con objetos los modificadores `final` y **`const`** tienen semánticas diferentes ya que `final` en Java indica que la referencia asignada al objeto es constante y no se puede asignar otra distinta (no se puede cambiar de objeto, pero sí se puede modificar el estado del objeto), mientras **`const`** en C++ indica que es el objeto mismo el que es constante, esto es, que el objeto al que se está refiriendo no puede ser modificado. Esta situación en Java se muestra en el siguiente ejemplo.

```
1  public class Persona{
2      String nombre;
3      double altura;
4
5      public Persona(String nombre, double altura){
6          this.nombre = nombre;
7          this.altura = altura;
8      }
9
10     public void setNombre(String nombre){
11         this.nombre=nombre;
12     }
13
14     public void setAltura(double altura){
15         this.altura=altura;
16     }
17
18     public static void main(String[] args){
19         final Persona person = new Persona("Genaro",1.60);
20         person.setNombre("Genaro Alcala"); //permitido
21         person = new Persona("Ernesto",1.62); //error
22     }
23
24 }
```

Aquí, `person` se declara para ser constante, esto es, no se le puede asignar una referencia distinta a la que se le asignó en la línea 19, en su inicialización. En este caso dicha referencia es la dirección de memoria que se genera cuando se crea un nuevo objeto a través del enunciado en la línea 19. Sin embargo, dicho objeto puede ser modificado libremente, lo que se hace en la línea 20 en este caso. En cambio, cualquier intento de

asignarle una referencia distinta a `person`, como en el último enunciado de la función `main`, resultará en un error.

A.3.3. Alojamiento de variables

Las variables globales se definen afuera de cualquier función y están disponibles en cualquier parte del código (incluso en otros archivos,[5]) a menos que su valor se haya ocultado por la declaración de una variable con el mismo nombre dentro de una función (o una declaración de una variable con el mismo nombre en un bloque interno). Por ejemplo,

```
1 #include <stdio.h>
2
3 int x=5;
4
5 int main(){
6   printf("El valor de x es: %d\n",x);
7 }
```

imprimirá `5` correspondiente al valor de la variable global `x`; en cambio en el siguiente código

```
1 #include <stdio.h>
2
3 int x=5;
4
5 int main(){
6   int x=90;
7   printf("El valor de x es: %d\n",x);
8 }
```

se imprime `90`, correspondiente al valor de la variable local `x`, porque la variable local `x` de la función `main` oculta el valor de la variable global del mismo nombre. El modificador **`extern`** antepuesto a una variable global indica que dicha variable está declarada en otro archivo pero se usa en el archivo actual. Por ejemplo,

```
1 //global.c
2 #include <stdio.h>
3 void modifica();
4
```

[5]Se pueden usar en otros archivos sólo si no tienen el modificador **`static`**.

```
5  int x=5;
6
7  int main(){
8    modifica();
9    printf("El valor de x es: %d\n",x);
10 }
```

```
1  //globalaux.c
2  extern int x;
3
4  void modifica(){
5    x = 90;
6  }
```

En el archivo `globalaux.c` se modifica la variable global declarada en `global.c`, por lo que el programa imprimirá `El valor de x es: 90`. Nótese que sin el uso del modificador **`extern`** en la declaración de la variable `x` en el archivo `globalaux.c`, el compilador supone dos declaraciones diferentes de la misma variable, lo cual es un error. Las variables globales corresponden a las variables declaradas con la directiva `.global` en ensamblador.

El modificador **`extern`** también se usa para especificar que la definición de una variable se encuentra más adelante en el archivo. Por ejemplo,

```
1  #include <stdio.h>
2
3  extern void imprime(); //la definición de imprime se encuentra
4                         //más adelante
5
6  extern int x; //la definición de x se encuentra más adelante
7
8
9  int main(){
10   imprime();
11 }
12
13 int x=5;
14
15 void imprime(){
16   printf("El valor de x es: %d\n",x);
17 }
```

Las variables que se declaran dentro de una función están dentro del alcance del cuerpo de dicha función, esto es, automáticamente se crean cuando se entra a la función y se destruyen cuando se sale de ella (son variables locales a la función) y por eso se les llama variables *automáticas*. El modificador **auto** indica que una variable es automática; por omisión, las variables locales de una función tienen el modificador **auto** y no es necesario escribirlo.

```
1  #include <stdio.h>
2  int imprime();
3
4  int main(){
5    imprime();
6  }
7
8  int imprime(){
9    auto int x =90; //variable automática
10   int y = 100;  //varible automática sin escribir auto
11   int z = x+y;
12   printf("El resultado de x+y es: %d\n",z);
13 }
```

Las variables que se declaran con el modificador **register** sugieren que dicha variable sea alojada en un registro y así, en principio, sea más rápido el acceso a su valor; sin embargo, no hay garantía que se aloje en un registro y sólo es una sugerencia para el compilador; una variable con el modificador **register** sólo puede ser declarada dentro de un bloque y no puede ser global o estática.

```
1  #include <stdio.h>
2  //register int x = 80; no está permitido
3
4  int main(){
5    register int x = 90;
6    printf("El valor de x es: %d\n",x);
7  }
```

Hemos visto ya que las variables locales a una función se crean durante la ejecución de dicha función y desaparecen cuando ésta termina; el modificador **static** es una excepción a esta regla: si se antepone este modificador a una variable local, el efecto que se obtiene es que la primera vez que se accede a la función ésta se inicializa y se realizan los cambios que haga dicha función sobre ella, pero a partir de la segunda vez que se llama, en lugar de reinicializar dicha variable ésta conserva el valor que tenía al final de la ejecución de la última llamada a la función. Por ejemplo,

```
1  #include <stdio.h>
2  void aumenta();
3
4  int main(){
5    aumenta();
6    aumenta();
7    aumenta();
8    aumenta();
9    aumenta();
10 }
11
12 void aumenta(){
13   static int x=5; //sólo se realiza en la primera invocación
14   x+= 10; //se realiza en cada una de las invocaciones
15   printf("El valor de x es: %d\n",x);
16 }
```

Aquí, la variable `x` se inicializa con el valor `5` la primera vez que se invoca la función `aumenta`; en esta primera invocación se imprime en pantalla `El valor de x es: 15` y es éste el valor que contiene la variable `x` al iniciar la segunda llamada a la función `aumenta`. En la segunda invocación a `aumenta` se imprime `El valor de x es: 25` y en las sucesivas `35 45 ...`. Si bien podemos obtener el mismo efecto haciendo uso de una variable global, no es equivalente ya que una variable global puede ser modificada en cualquier otro lugar del código, lo cual en ocasiones no es deseable.

El otro uso del modificador **`static`** es cuando se antepone a una variable afuera de cualquier alcance (de funciones o clases), en cuyo caso indica que dicha variable tiene alcance de archivo y no se puede hacer uso de ella desde un archivo diferente.

```
1  #include <stdio.h>
2  void modifica();
3  static int x=5;
4
5  int main(){
6    modifica();
7    printf("El valor de x es: %d\n",x);
8  }
```

```
1  extern int x; //error no se puede usar x porque es static
2
3  void modifica(){
4    x = 90;
5  }
```

El código anterior da un error en tiempo de enlazado porque no se puede usar la variable `x` del archivo `alcance.c` en el archivo `alcanceaux.c`, porque ésta tiene el modificador **`static`** y por tanto su alcance es de archivo.

A.3.4. Estructuras y uniones

En muchas ocasiones es necesario para el programador definir estructuras de datos para resolver problemas. Por ejemplo, supongamos que se necesita hacer una base de datos de los pacientes de un consultorio, que contenga la siguiente información: nombre, edad, peso, y estatura; el programador de C, puede hacer uso de una estructura (*struct*) de la siguiente forma:

```
1  #include <stdio.h>
2  struct persona{
3    char* nombre;
4    int edad;
5    float peso;
6    float estatura;
7  };
8
9  int main(){
10   struct persona uno ={
11     "Juan",
12     23,
13     77,
14     1.69
15   };
16 }
```

Observamos que una estructura se inicializa escribiendo los valores de cada uno de las variables que conforman la estructura separadas por "," y encerrados entre { }. En C++ el código anterior es válido, aunque en C++ no es necesario anteponer la palabra **`struct`**, cuando declaramos una variable.

```
1  #include <stdio.h>
2  struct persona{
3    char* nombre;
4    int edad;
5    float peso;
6    float estatura;
7  };
8
```

```
9  int main(){
10   persona uno={
11     "Juan",
12     23,
13     77,
14     1.69
15   };
16 }
```

Para evitar al programador de C escribir la palabra **struct** se puede usar el enunciado **typedef**, el cual permite definir un nombre para un tipo de datos. En esta ocasión, en lugar de hacer uso de la inicialización anterior, utilizamos el operador "." que nos permite acceder a las variables de una estructura:

```
1  #include <stdio.h>
2  typedef short int sint;
3  typedef struct{
4    char* nombre;
5    sint edad;
6    float peso;
7    float estatura;
8  } persona;
9
10
11 int main(){
12   persona uno; //no escribimos struct
13   uno.nombre = "Juan";
14   uno.edad = 23;
15   uno.peso= 77;
16   uno.estatura = 1.67;
17 }
```

En el código anterior la estructura es anónima pero se le asigna un alias. Si bien podemos hacer uso de estructuras, en C++ contamos con clases, que corresponden a la evolución de las estructuras, con un mayor control de acceso sobre las variables que las conforman[6] mediante el uso de modificadores,[7] además de tener las funciones (métodos en la terminología orientada a objetos) asociados a ellas. El siguiente ejemplo muestra el programa anterior haciendo uso de la orientación a objetos en C++.

[6]También llamadas variables miembro o atributos.

[7]Hay tres tipos de modificadores **private**, **public** y **protected**.

```
1  #include <string>
2  #include <iostream>
3
4  using namespace std;
5
6  class Person{
7  public:
8
9    Person(string inombre, int iedad, float ipeso, float iestatura){
10     nombre = inombre;
11     edad = iedad;
12     peso = ipeso;
13     estatura = iestatura;
14   }
15
16   string getNombre(){
17     return nombre;
18   }
19
20 private:
21
22   string nombre;
23   int edad;
24   float peso;
25   float estatura;
26
27
28 };
29
30 int main(){
31   Person uno("Juan",23,77,1.67);
32   cout <<"El nombre es: " <<  uno.getNombre() << endl;
33 }
```

Las uniones son similares a las estructuras, sólo que únicamente almacenan el espacio correspondiente a una sola variable de las que conforman la estructura y por tanto su tamaño es igual al de la variable de mayor tamaño de las que integran la unión. De hecho todas las variables comparten un único espacio.

```
1  #include <stdio.h>
2  struct spersona{
3    char* nombre;
4    int edad;
5    float peso;
6    float estatura;
7  };
8
9  union upersona{
10   char* nombre;
11   int edad;
12   float peso;
13   float estatura;
14 };
15
16 int main(){
17   union upersona uno;
18   uno.nombre = "Juan";
19   uno.edad = 23;
20   uno.peso= 77;
21   uno.estatura = 1.69;
22   printf("El tamaño de la union es: %d\n",sizeof(uno));
23   printf("La estatura es: %f\n", uno.estatura);
24   //printf("El nombre es: %s",uno.nombre);
25   //error solo se puede guardar una variable a la vez
26
27   struct spersona dos;
28   dos.nombre = "Juan";
29   dos.edad = 23;
30   dos.peso= 77;
31   dos.estatura = 1.69;
32   printf("El tamaño de la estructura es: %d\n", sizeof(dos));
33   printf("El nombre es: %s\n",dos.nombre);
34   printf("La edad es: %d\n",dos.edad);
```

El operador *sizeof* devuelve el tamaño en bytes del argumento con el que se invoca.

El código anterior muestra que el tamaño de la unión es `8`, mientras que el tamaño de la estructura es `24`. Sin embargo, la desventaja de las uniones es que al almacenar sólo el valor de una variable a la vez, únicamente está disponible el valor de la última variable de la unión a la que se le asignó un valor, no así en las estructuras que están disponibles en todo momento los valores de cada una de las variables que la conforman.

Sin embargo, las uniones permiten que un dato se pueda interpretar de diferentes formas, esto es que se pueda ver como un dato que puede tomar diferentes tipos. La sintaxis de una unión es muy similar a la de una estructura, aunque su semántica es muy diferente: en las uniones sólo se almacena a la vez una única variable de las que conforman la unión (todas las variables comparten el mismo espacio de memoria). Las uniones se utilizan cuando se deben almacenar dos o más datos que son mutuamente excluyentes y sólo es necesario almacenar uno de ellos a la vez. Por ejemplo, supongamos que se necesita una forma de identificar a un alumno, entonces podemos identificarlo por medio de su número de cuenta o por medio de su curp; tenemos lo siguiente:

```
1  #include <stdio.h>
2
3  union alumno{
4    int nc;
5    char curp[18];
6  };
7
8
9  int main(void){
10   union alumno a;
11   a.nc = 39876567;
12   printf("El identificador del alumno a es: %d\n",a.nc);
13   union alumno b;
14   b.curp = "GACE230978HDFDDL03";
15   printf("El identificador del alumno b es: %s\n",b.curp);
16
17 }
```

Observamos que tenemos dos alumnos donde en uno almacenamos su identificador (que es su número de cuenta) como un entero y otro donde almacenamos su identificador (su curp) como un arreglo de caracteres. Aunque podríamos utilizar una estructura, esto implica un desperdicio de memoria innecesario ya que nunca estarán presentes ambos datos. En este ejemplo, la unión `alumno` se usa como un entero o como un arreglo de caracteres y por tanto el tamaño de la estructura es 20 bytes (que es el tamaño que ocupa un entero que son 2 bytes más el tamaño que ocupa el arreglo con 18 entradas que es 18 bytes), mientras que el tamaño de la unión es 18.[8]

A.3.5. Apuntadores y arreglos

Apuntadores

Un apuntador es un tipo de dato que toma como valores direcciones de memoria; éstas pueden ser ya sea de otras variables, constantes o incluso de funciones. El uso de

[8]El operador *sizeof* devuelve el tamaño en bytes del argumento con el que se invoca.

apuntadores permite gran flexibilidad al programador, aunque su uso debe ser cuidadoso para no tener resultados inesperados. Existen dos operadores que se utilizan cuando se trabaja con apuntadores:

- El operador * que tiene dos significados,[9] ya que se usa para declarar que una constante o variable es un apuntador y también para denotar la aplicación de la operación derreferencia; la operación derreferencia accede al valor que almacena la dirección de memoria sobre la cual se aplica; esta dirección estará almacenada en una variable de tipo apuntador.
- El operador &, también conocido como "dirección de",[10] se usa para obtener la dirección de una variable o constante.

Veamos un ejemplo:

```
1 #include <stdio.h>
2
3 int main(void){
4   int a=5;
5   int* ptr;
6   ptr = &a;
7   printf("El valor de a obtenido por medio del apuntador es: %d\n"
       ,*ptr);
8 }
```

Aquí se declara una variable `a` de tipo entero y después se declara una variable `ptr` de tipo apuntador a entero; por tanto, `ptr` debe almacenar direcciones de memoria que almacenen valores de variables de tipo entero. En este caso, la línea 6 denota que `ptr` toma como valor la dirección de la variable entera `a`; por último observamos el uso de la operación derreferencia sobre al apuntador `ptr`, que accede al valor de variable `a`.

El tipo **`void`**`*` se usa para denotar el tipo apuntador a cualquier tipo y es útil cuando se usa un mismo apuntador para referirse a variables de diferentes tipos, aunque si no se usa con precaución se pueden cometer errores. Se puede hacer uso de apuntadores a apuntadores en dos o más niveles, según sea necesario. Por ejemplo,

```
1 #include <stdio.h>
2
3 int main(void){
4   int x = 5;
5   int* ptr = &x;
```

[9] En realidad son tres significados, pues también se usa para multiplicar como operador binario.

[10] También tiene otro significado en C++ para denotar un paso de parámetros por referencia, el cual veremos más adelante.

```
6    int** ptrtoptr = &ptr;
7    printf("El valor de la variable x es: %d\n",x);
8    printf("La dirección de x es: %x\n",&x);
9
10   printf("El valor del apuntador ptr es: %x\n",ptr);
11   printf("La dirección del apuntador ptr es: %x\n", &ptr);
12   printf("Valor que almacena la dirección apuntada por ptr: %d\n",
13          *ptr);
14   printf("El valor del apuntador a apuntador ptrtoptr es: %x\n",
15          ptrtoptr);
16   printf("Dirección del apuntador a apuntador ptrtoptr: %x\n",
17          &ptrtoptr);
18   printf("Valor que almacena dirección apuntada por ptrtoptr: %x\n",
19          *ptrtoptr);
20   printf("Valor de x por medio del apuntador a apuntador: %d\n",
21          **ptrtoptr);
22 }
```

Como se observa en este caso, tenemos un variable `x` que almacena un valor entero `5`, un apuntador a entero `ptr` que apunta a la dirección de `x` y un apuntador a un apuntador de enteros `ptrtoptr` que apunta a la dirección de `ptr`; vemos cómo es posible obtener el valor de la variable `x` por medio de `ptrtoptr`, aplicando dos veces consecutivas la operación derreferencia `**ptrtoptr`. Lo que se realiza primero es `*ptrtoptr`, lo cual da como resultado `&x` y entonces tenemos `*(&x)` que da por último el valor de `x`, que es `5`. Siguiendo el esquema anterior es fácil ver que podemos tener apuntadores a apuntadores a apuntadores ... . También podemos tener apuntadores a estructuras y uniones.

```
1  #include <stdio.h>
2
3  struct rectangulo{
4    double largo;
5    double ancho;
6  };
7
8  int main(void){
9    struct rectangulo r = { 10, 5};
10   struct rectangulo* ptr = &r;
11   printf("El rectangulo originalmente mide %fx%f\n", (*ptr).largo,
12                       (*ptr).ancho);
```

```
13    ptr->largo= 20;
14    ptr->ancho = 10;
15    printf("El rectangulo ahora mide %fx%f\n", ptr->largo,
16            ptr->ancho);
17  }
```

Aquí hacemos uso de una estructura `rectángulo r` y tenemos un apuntador a una estructura `rectángulo ptr` que apunta a `r`.[11] Vemos que para acceder a las variables de la estructura por medio del apuntador primero tenemos que hacer una derreferencia y luego acceder a ella por medio del ya conocido operador ".". Como podemos observar en las líneas 12 y 12, en el código `(*ptr).largo` los paréntesis son necesarios, ya que sin ellos el enunciado se interpreta como `*(ptr.largo)`, lo cual es un error. Una forma más sencilla de acceder a las variables de las estructuras y uniones por medio de apuntadores, es haciendo uso de la siguiente sintaxis, `ptr->largo`, como se muestra en algunos enunciados del ejemplo anterior; tiene exactamente el mismo comportamiento que `(*ptr).largo`.

Apuntadores y el modificador const

Cuando se declara un apuntador, se puede utilizar el modificador **`const`** con dos motivos: el primero es para indicar que el apuntador será constante, es decir, que una vez que se inicializa con una dirección no puede almacenar otra diferente; el segundo motivo es indicar que a la variable a la que apunta será inmutable. Esto se ilustra en el siguiente ejemplo:

```
1  int main(void){
2    int x=10;
3    int y=20;
4    const int* ptc = &x; //el valor de x no se puede cambiar
5                         //mediante el apuntador
6    int* const cp = &x;  //apuntador constante cp no puede apuntar
7                         //a otra variable
8    const int* const cptc = &x;
9    //distinta de x
10   //*ptc = 20; error: no se puede cambiar el contenido de x
11                       // por medio de ptc
12   x = 20;
13   *cp = 25;
14   //cp = &y; //error: cp cambiar de valor
```

[11] Léase `ptr` almacena la dirección de la variable `r`.

```
15    ptc = &y;
16
17    //*cptc = 20; error
18    //cptc = y; error
19  }
```

El enunciado en la línea 4 indica que no es posible que el apuntador `ptc`, que almacena la dirección de la variable `x`, modifique el valor de `x`; en cambio, el enunciado de la línea 6, **`int*`** **`const`** `cp = &x;`, indica que `cp` es un apuntador constante y por tanto no puede tomar un valor diferente a la dirección de `x` en este caso; como se observa, se puede hacer uso de ambos modificadores en un apuntador: la línea 8 indica que el apuntador `cptc` es constante (no puede almacenar una valor diferente al de la dirección de la variable `x`) y no puede ser utilizado para modificar el valor de la variable a la que apunta, en este caso `x`.

Arreglos

Un arreglo es una estructura de datos, que sirve para almacenar diferentes elementos del mismo tipo; en C, C++ y Java los arreglos tienen soporte nativo, esto es, son parte del lenguaje; en C y en C++ se almacenan como bloques contiguos de memoria y son a menudo utilizados en cualquier tipo de aplicación; un arreglo está conformado por elementos y el tamaño del arreglo es el número de elementos que contiene el arreglo. Para acceder a un elemento del arreglo se usa un índice, que indica la posición del elemento dentro del arreglo, que existe para todo elemento del arreglo. Veamos un ejemplo:

```
1  #include <stdio.h>
2
3  int main(void){
4    int a[] = {2,4,6,8};
5    char* b[4];
6    b[0] = "primer";
7    b[1] = "segundo";
8    b[2] = "tercer";
9    b[3] = "cuarto";
10   int i =0;
11   while(i < 4){
12     printf("El %s elemento del arreglo es: %d\n",b[i],a[i]);
13     i++;
14   }
15 }
```

Aquí se declara un arreglo de enteros `a` que tiene como elementos los enteros 2, 4, 6 y 8. Nótese el uso de `[]`, que indica que la variable `a` es un arreglo. En este caso no es necesario indicar entre los `[]` el número de elementos del arreglo porque esta implícito en la inicialización; los elementos iniciales del arreglo se declaran entre `{}` y separados por ",". En este ejemplo el tamaño del arreglo es `4` ya que cuenta con 4 elementos. También se declara un arreglo de apuntadores a caracteres (cadenas en C) que también tiene como tamaño `4`; nótese la diferencia en la declaración del arreglo `b`, pues en este caso se define explícitamente el tamaño del arreglo y los valores iniciales quedan indefinidos y pueden contener información basura; sin embargo, en los siguientes enunciados se les asigna un valor a cada uno de los elementos del arreglo, y se observa que para acceder a un elemento del arreglo se hace uso del operador *[]* con un entero como índice, que denota la posición del elemento dentro del arreglo; con el uso de un **`while`** recorremos ambos arreglos a la vez mostrando en pantalla el valor de cada uno de sus elementos; es importante notar que el índice comienza en la posición 0, que es donde se encuentra el primer elemento del arreglo, y lo recorre hasta llegar a `tamaño-1` (en este caso 3), que es la posición del último elemento del arreglo.

Veamos ahora la relación entre apuntadores y arreglos.

Cuando declaramos un arreglo, por ejemplo **`int`** `a[5]`, indicamos al compilador que reserve espacio para almacenar 5 enteros consecutivos y cada uno de ellos ocupa 4 bytes o 32 bits (en la arquitectura x86); entonces reserva 20 bytes de memoria. Si nosotros declaramos un apuntador **`int`**`* ptr` podemos apuntar a cualquier entero, en particular al primer elemento del arreglo `a` por medio de `ptr = &a[0]` y después usar el apuntador `ptr` para tener acceso a los elementos del arreglo `a` haciendo uso de operaciones aritméticas con apuntadores, como en el siguiente ejemplo:

```
1  #include <stdio.h>
2
3  int main(void){
4    int a[5];
5    a[0]=2;
6    a[1]=4;
7    a[2]=6;
8    a[3]=8;
9    a[4]=10;
10   int* ptr;
11   ptr = &a[0];
12   int i= 0;
13   printf("Los valores de los elementos del arreglo a son:\n");
```

```
14    while (i<5){
15      printf("%d\n",*(ptr+i));
16      i++;
17    }
18  }
```

Observamos el uso del operador + sobre un apuntador y un entero. `ptr` originalmente tiene como valor la dirección del primer elemento del arreglo y, por tanto, en la primera iteración del **`while`**, cuando se realiza la derreferencia, se imprime en pantalla el valor del primer elemento del arreglo; en la segunda iteración el valor de `i` es `1` y el resultado de `ptr+1` es que a la dirección de memoria almacenada en `ptr` se le suman el número de bytes correspondientes al tamaño de un elemento del arreglo (en este caso 4 por tratarse de enteros); de manera análoga sucede en el resto de las iteraciones: cuando `i` vale 2 se le suma 8, cuando vale 3, 12, y así sucesivamente; es interesante observar cómo internamente el compilador trata de igual forma a `a` que a un apuntador:

```
1  #include <stdio.h>
2
3  int main(void){
4    int a[5];
5    a[0]=2;
6    a[1]=4;
7    a[2]=6;
8    a[3]=8;
9    a[4]=10;
10   int* ptr;
11   ptr = &a[0];
12   int i= 0;
13   printf("Los valores de los elementos del arreglo a por medio"
14          " del apuntador ptr son:\n");
15   while (i<5){
16     printf("%d\n",*(ptr+i));
17     i++;
18   }
19   i=0;
20   printf("Los valores de los elementos del arreglo a utilizando"
21          " a como apuntador son:\n");
22   while (i<5){
23     printf("%d\n",*(a+i));
24     i++;
25   }
26 }
```

En ambos ciclos **while** se obtienen los mismos resultados y por tanto la asignación `ptr= &a[0]` es igual a la asignación `ptr=a`, ya que `a` es internamente un apuntador que tiene como valor la dirección del primer elemento del arreglo. Ahora nos preguntamos si podemos hacer uso de `ptr` como si se tratase de un arreglo y la respuesta es afirmativa: lo que hemos logrado es tener `ptr` como un sinónimo para `a`.

```
1  #include <stdio.h>
2
3  int main(void){
4    int a[5];
5    a[0]=2;
6    a[1]=4;
7    a[2]=6;
8    a[3]=8;
9    a[4]=10;
10   int* ptr;
11   ptr = &a[0];
12   int i= 0;
13   printf("Los valores de los elementos del arreglo a por medio"
14          " del apuntador ptr son:\n");
15   while (i<5){
16     printf(" %d\n",*(ptr+i));
17     i++;
18   }
19   i=0;
20   printf("Los valores de los elementos del arreglo a utilizando"
21          " a como apuntador son:\n");
22   while (i<5){
23     printf(" %d\n",*(a+i));
24     i++;
25   }
26   i=0;
27   printf("Los valores de los elementos del arreglo utilizando "
28           "a y ptr como arreglos son:\n");
29   while(i<5){
30     printf("a[ %d]= %d\n",i,a[i]);
31     printf("ptr[ %d]= %d\n",i,ptr[i]);
32     i++;
33   }
34
35 }
```

Los operadores válidos para realizar operaciones aritméticas con apuntadores son:

- + (incremento), del cual, como ya vimos anteriormente, aplicado a un apuntador le asigna a éste una dirección de memoria igual a la dirección de memoria que almacena actualmente, más el tamaño del elemento al que hace referencia por el operando utilizado.
- - (decremento) análogo al caso anterior, pero decrementando la dirección de memoria almacenada actualmente.

Debemos tener cuidado al decir que `a` es un apuntador, pues aunque esto es cierto internamente, no lo es así para el programador; por ejemplo, el enunciado `a++` es un intento de utilizar un arreglo como apuntador pero que no funciona y da como resultado un error en tiempo de compilación. En cambio, todo apuntador puede ser utilizado como un arreglo y, como vimos anteriormente, utilizar el operador *[]* aplicado a él, aunque debemos tener en cuenta algunos escenarios: cuando declaramos un arreglo debemos conocer al momento de inicializarlo el tamaño que éste ocupará; en comparación, un apuntador no necesariamente será utilizado como un arreglo y cuando su objetivo sea ser utilizado como un arreglo, puede ser utilizado para ser un sinónimo de otro arreglo (como en el ejemplo anterior) o bien, en el momento de su inicialización reservar espacio para el nuevo arreglo, como se muestra en el siguiente ejemplo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void){
5    int* ptrarr = malloc(sizeof(int)*4);
6    //int* ptrarr = {2,4,6,8}; //error no se permite inicializar
7    //un apuntador como un arreglo
8    ptrarr[0]=2;
9    ptrarr[1]=4;
10   ptrarr[2]=6;
11   ptrarr[3]=8;
12
13   printf("Los elementos del arreglo son:\n");
14   short i = 0;
15   while(i<4){
16     printf("%d\n",ptrarr[i]);
17     i++;
18   }
19   free(ptrarr); //liberamos espacio
20   int a[] ={1,3,5,7,9};
21   int b[] ={11,12,13,14}
22   ptrarr = a;
23   //b=a;  //error: no se permite utilizar b como un apuntador
24 }
```

Como se observa en el código anterior, `ptrarr` es un apuntador que utilizaremos como un arreglo de enteros y por lo tanto tenemos que reservar explícitamente espacio para dicho arreglo; esto lo hacemos a través de la función `malloc` que es parte la biblioteca estándar de C. `malloc` recibe como argumento el número de bytes a reservar y regresa un apuntador al inicio del espacio reservado, o -1 en caso de que no haya espacio disponible. Es importante notar que el espacio reservado resultante de una llamada a la función `malloc` se encuentra en el heap y no en la pila, y por tanto es responsabilidad del programador liberar el espacio cuando éste ya no se ocupe; esto se logra a través de la llamada a la función `free`, que recibe como argumento el apuntador al espacio que se va a liberar –ambas funciones (`malloc` y `free`) se encuentran en el archivo `stdlib.h`–. A continuación procedemos a inicializar los elementos del arreglo, accediendo a ellos mediante su índice (nótese que no es posible inicializar un apuntador con la sintaxis de un arreglo). También se puede utilizar el apuntador para almacenar otra dirección o, como es el caso del ejemplo anterior, para ser un sinónimo de otro arreglo. No está permitido utilizar una variable que originalmente era un arreglo para ser sinónimo de otro, porque, como ya mencionamos, un arreglo no es apuntador para el programador; si se omite la llamada a `free` el código aún compilará, aunque quedará espacio ocupado en la memoria que ya no es accesible mediante el programa y que puede ser causante de errores inesperados; además se considera un mal estilo de programación. A este tipo de escenario se le conoce como *memory leak* y desafortunadamente es un problema común y difícil de encontrar en los programas escritos en C y en C++. En Java el programador no se debe preocupar por este tipo de problemas ya que se cuenta con un recolector de basura que se hace cargo de liberar el espacio que ya no es accesible mediante el programa.[12] Los arreglos `a` y `b` son variables automáticas locales a la función `main` y se almacenan en la pila.

C y C++, al igual que Java, permiten el uso de arreglos multidimensionales; centremos ahora nuestra atención en los arreglos de arreglos o arreglos bidimensionales, con el siguiente ejemplo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void){
5    int bidi[3][4] = {{11,12,13,14},{21,22,23,24},{31,32,33,34}};
6    //equivalente a las siguientes declaraciones
7    //int bidi[3][4] = {11,12,13,14,21,22,23,24,31,32,33,34};
8    //int bidi[][4] = {11,12,13,14,21,22,23,24,31,32,33,34};
9
10   short i = 0;
11   short j = 0;
```

[12] Abordaremos a profundidad más adelante el manejo de memoria.

```
12    while( i < 3){
13      j=0;
14      while(j<4){
15        printf("Elemento en renglon %d columna %d del arreglo: %d\n",
16                i,j,bidi[i][j]);
17        j++;
18      }
19      i++;
20    }
21  }
```

En el ejemplo anterior se muestra la sintaxis para declarar e inicializar arreglos bidimensionales. Como se observa, se pueden definir explícitamente las dimensiones del arreglo: en este caso se declara un arreglo de tamaño `3` cuyos elementos son arreglos de tamaño `4` y por tanto cada uno de éstos está encerrado entre `{}`; sin embargo, el uso de `{}` es opcional, por lo que las declaraciones sucesivas son equivalentes. Esto se debe a que un arreglo de arreglos es internamente representado como un arreglo de una sola dimensión (con direcciones contiguas de memoria), donde el primer elemento del siguiente renglón se aloja exactamente después del último elemento del renglón anterior; sin embargo, la primera declaración es más legible para el programador. También se observa que se puede omitir el valor de la primera dimensión y que se infiera a partir del valor de la segunda. De esto, el arreglo `bidi` es internamente un arreglo lineal de 12 elementos, como se muestra a continuación:

```
1  #include <stdio.h>
2
3  int main(void){
4    int bidi[][4]={11,12,13,14,
5                   21,22,23,24,
6                   31,32,33,34};
7    int* ptr = (int*)bidi;
8    short i = 0;
9    printf("Elementos del arreglo vistos como arreglo lineal:\n");
10   while(i<12){
11     printf(" %d\n",*ptr);
12     ptr++;
13     i++;
14   }
15
16 }
```

Aquí se utiliza el apuntador `ptr` como sinónimo del arreglo `bidi` y se accede a sus elementos secuencialmente como si `bidi` fuese de un arreglo unidimensional, algo que es imposible realizar en Java.

A.3.6. Funciones

Una función es un bloque de código encargado de realizar una tarea específica. La declaración de una función consiste de el nombre de la función, los parámetros de entrada, tipo de regreso y el cuerpo de la función. En C++ toda función debe ser declarada antes de ser invocada como se muestra a continuación.

```
1  #include <stdio.h>
2
3  void imprimeval(int x, int y){
4    printf("El valor del primer entero es: %d\n", x);
5    printf("El valor del segundo entero es: %d\n", y);
6  }
7
8  int main(void){
9    int x=5;
10   int y=10;
11   imprimeval(x,y);
12 }
```

En este ejemplo, declaramos la función `imprimeval` que no regresa ningún valor, lo cual se indica con la palabra **`void`** como tipo de regreso; tiene como parámetros dos enteros `x` e `y` y lo que hace es imprimir en pantalla el valor de cada uno de ellos. Como se observa, `imprimeval` se define antes de ser invocada, algo que no siempre es deseable. Por ejemplo, podemos tener una función f que invoque a una función g y que ésta a su vez invoque a la función f; para aliviar este tipo de problemas se puede escribir el encabezado de la función antes de que ésta sea invocada y definir la función después en otra parte del código como se muestra a continuación:

```
1  #include <stdio.h>
2
3  void imprimeval(int x, int y);
4
5  int main(void){
6    int x=5;
7    int y=10;
8    imprimeval(x,y);
9  }
10
```

```
11 void imprimeval(int x, int y){
12   printf("El valor del primer entero es: %d\n", x);
13   printf("El valor del segundo entero es: %d\n", y);
14 }
```

Como podemos observar, primero se escribe el encabezado de la función `imprimeval` seguida de un "`;`", a lo que se le llama *prototipo* de la función (*function prototype*). Es común que los programas escritos en C contengan los prototipos de las funciones que se van a definir dentro del archivo al principio de éste (o en archivo de encabezados) y su definición se encuentre después. Cabe mencionar que el nombre de los parámetros en el prototipo es opcional y puede ser omitido, esto es, `void imprimeval(int x, int y)` es equivalente a `void imprimeval(int,int)`.

El paso de parámetros en C, al igual que en Java, es únicamente por valor, esto es, la función hace una copia local de los argumentos pasados en la invocación, como se ve al ejecutar el siguiente código:

```
1  include <stdio.h>
2
3  void intercambia(int,int);
4
5  int main(void){
6    int x = 10;
7    int y = 20;
8    intercambia(x,y);
9    printf("El valor de x es: %d\n",x);
10   printf("El valor de y es: %d\n",y);
11 }
12
13 void intercambia(int x,int y){
14   int tmp = x;
15   x = y;
16   y = tmp;
17 }
```

que imprime:

```
El valor de x es: 10
El valor de y es: 20
```

Definimos primero dos enteros `x` e `y` que usamos como argumentos para la función `intercambia`; la función `intercambia` copia el valor de los argumentos en los parámetros formales, obteniendo así sus propias copias locales de `x` e `y`, que se destruyen

cuando termina la ejecución de la función; dado el paso por valor, no se realiza ningún cambio a los valores de las variables `x` e `y` de la función `main`. Lo mismo sucede en Java como se muestra a continuación.

```
1  public class Inter{
2      public static void main(String args[]){
3          int x = 10;
4          int y = 20;
5          intercambia(x,y);
6          System.out.println("El valor de x es: "+x);
7          System.out.println("El valor de y es: "+y);
8      }
9
10     public static void intercambia(int x, int y){
11         int tmp = x;
12         x = y;
13         y = tmp;
14     }
15 }
```

Este programa, al igual que el escrito en C o C++, imprime:

```
El valor de x es: 10
El valor de y es: 20
```

Sin embargo, mediante el uso de apuntadores podemos obtener el resultado esperado.

```
1  #include <stdio.h>
2
3  void intercambia(int*,int*);
4
5  int main(void){
6    int x = 10;
7    int y = 20;
8    intercambia(&x,&y);
9    printf("El valor de x es: %d\n",x);
10   printf("El valor de y es: %d\n",y);
11 }
12
13 void intercambia(int* x,int* y){
14   int tmp = *x;
15   *x = *y;
16   *y = tmp;
17 }
```

En este caso, el resultado de ejecutar este programa es:

```
El valor de x es: 20
El valor de y es: 10
```

C++ permite el paso de parámetros tanto por valor como por referencia, donde en un paso de parámetros por referencia los parámetros formales de una función son un sinónimo de los argumentos; el ejemplo anterior, usando el paso por referencia de C++ se muestra a continuación:

```
1  include <iostream>
2
3  void intercambia(int&,int&);
4
5  int main(void){
6    int x = 10;
7    int y = 20;
8    intercambia(x,y);
9    std::cout << "El valor de x es: " << x << std::endl;
10   std::cout << "El valor de y es: " << y << std::endl;
11 }
12
13 void intercambia(int& x,int& y){
14   int tmp = x;
15   x = y;
16   y = tmp;
17 }
```

e imprime:

```
El valor de x es: 20
El valor de y es: 10
```

A.3.7. Argumentos por omisión

En C++, cuando se hace uso de paso de parámetros por valor se puede definir un valor por omisión para los parámetros formales y por tanto éstos ser omitidos en la invocación a la función, como se muestra en el siguiente código:

```
1  #include <iostream>
2
3  double volumen(double largo=1, double ancho=2, double alto=3);
4
```

```
5  int main(void){
6    double vol = volumen(4,5,10);
7    std::cout << "El volumen es: " <<  vol << std::endl;
8    vol = volumen(10);
9    std::cout << "El volumen es: " <<  vol << std::endl;
10 }
11
12 double volumen(double alto, double ancho, double largo){
13   return alto*ancho*largo;
14 }
```

En el código anterior se declara una función `volumen` que calcula el volumen de un cuerpo geométrico. Como se observa en la declaración del prototipo de la función, se declaran los valores por omisión de los parámetros `largo = 1`, `ancho = 2` y `alto = 3`. Así, en la segunda invocación de la función `volumen`, se omiten dos de los argumentos, los cuales toman los valores por omisión de la función, en este caso `largo` toma el valor del argumento pasado en la llamada (`10`) mientras que `ancho` y `alto` toman los valores `2` y `3` respectivamente; por lo tanto, la segunda línea que imprime el programa es:

```
El volumen es: 60
```

Como podemos notar, si se invoca a la función sin todos los argumentos, el o los argumentos pasados corresponden a los parámetros de la función de izquierda a derecha, en este caso `10` corresponde a `largo` y no a `ancho` ni a `alto`; es importante decir que los valores por omisión sólo se especifican una vez, esto es, ya sea en la declaración de la función ó en la definición de la función, pero no en ambas. En caso de que estén presentes ambas, se deben especificar en la declaración, ya que como se declara la función primero es ahí donde se definen los argumentos por omisión y no se repiten en la definición de la función; si no se usa el prototipo de la función y se define directamente la función, entonces es ahí donde se deben especificar los argumentos por omisión, como se muestra a continuación.

```
1  #include <iostream>
2
3  double volumen(double alto=1, double ancho=2, double largo=3){
4    return alto*ancho*largo;
5  }
6
7  int main(void){
8    double vol = volumen(4,5,10);
9    std::cout << "El volumen es: " <<  vol << std::endl;
```

```
10    vol = volumen(10);
11    std::cout << "El volumen es: " <<  vol << std::endl;
12  }
```

En caso de que no todos los parámetros de la función tengan valores por omisión, los que los tengan tienen que ir a la derecha, esto es, una vez que se declara un parámetros con valor por omisión, todos los que se encuentren a su derecha también deben tener valores por omisión; por ejemplo, el prototipo

```
double volumen(double alto=1, double ancho, double largo);
```

no está permitido ya que al tener `alto` un valor por omisión, necesariamente `ancho` y `largo` deben tenerlos también.

Existe una ligera diferencia entre Java y C++ en el orden de evaluación de los argumentos cuando se invoca a una función: en Java está definido que los argumentos se evalúan de izquierda a derecha, mientras que en C++ esto depende de la implementación del compilador. Veamos como ejemplo el siguiente código en Java.

```
1  public class Imprime{
2      public static void main(String[] args){
3          String s = "hola";
4          imprime(s, s="mundo");
5      }
6
7      public static void imprime(String p, String s){
8          System.out.println(p+" "+s);
9      }
10
11 }
```

imprime

```
hola mundo
```

En C++, en cambio, el siguiente código

```
1  #include <iostream>
2  #include <string>
3
4  void imprime(std::string s, std::string p);
5
6  int main(void){
7    std::string s = "hola";
8    imprime(s, s="mundo");
9  }
```

```
10 void imprime(std::string s, std::string p){
11   std::cout << s << " " << p << std::endl;
12 }
```

imprime

```
mundo mundo
```

aunque, como ya mencionamos, en C++ es dependiente de la implementación y en algunos casos se puede obtener

```
hola mundo
```

Es posible pasar arreglos como argumentos a una función; en este caso el paso de parámetros es como si fuese por referencia, aunque en realidad a la función sólo se le pasa la dirección del primer elemento del arreglo y la función no conoce el tamaño del arreglo. Es por eso que en C y C++ se acostumbra pasar como argumento a la función el tamaño del arreglo o hacer uso del operador *sizeof* dentro de la función para determinar el tamaño del arreglo que se está pasando. Un ejemplo de la primer solución se muestra a continuación:

```
1  #include <iostream>
2
3  void duplica(int a[], int tam);
4
5  int main (void){
6    int a[] = {1,2,3,4,5};
7    duplica(a,5);
8    std::cout << "Los elementos del arreglo son: ";
9    for(int i=0; i< 5; i++){
10     std::cout << a[i] << " ";
11   }
12   std::cout << std::endl;
13 }
14
15 void duplica(int a[], int tam){
16   for(int i=0; i < tam; i++){
17     a[i] = a[i]*2;
18   }
19 }
```

En el ejemplo anterior se imprime como salida

```
Los elementos del arreglo son: 2 4 6 8 10
```

En algunas ocasiones, como en el caso anterior, es deseable que la función pueda modificar los elementos del arreglo que se pasa como argumento; no siempre es así, ya que a veces se desea que el argumento que se pasa a la función no sea modificado; para ello se cuenta con el modificador **const** utilizado dentro de este contexto. El modificador **const**, antepuesto a un parámetro, nos garantiza que el argumento correspondiente que se pasa a la función no sea modificado por la función. Aunque el modificador **const** se puede utilizar para cualquier tipo de parámetros, su uso es más común en arreglos y apuntadores, ya que en el paso por valor, como ya vimos anteriormente, no se modifican los argumentos dentro de la función porque se genera una copia local de ellos en los parámetros formales. Veamos un ejemplo :

```
1  #include <iostream>
2
3  double promedio(const double a[], int tam);
4
5  int main(void){
6    double calif[] = {6.5, 8.5, 7.5, 10};
7    double prom = promedio(calif,4);
8    std::cout << "El promedio de las calificaciones es: "
9              << prom << std::endl;
10 }
11
12 double promedio(const double a[], int tam){
13   double prom=0;
14   //a[0]=5; //error no se puede modificar el arreglo a
15   for(int i=0; i < tam; i++){
16     prom += a[i];
17   }
18   prom /= tam;
19 }
```

En este caso tenemos una función `promedio` que calcula el promedio de los elementos de un arreglo, así que se espera que dicha función no modifique los valores del arreglo; por tanto se antepone el modificador **const** al parámetro **double** `a` para asegurarse de ello. Así, cualquier intento de modificación a los elementos del arreglo, como en el enunciado `a[0]=5`, resulta en un error.

Si una función f tiene uno o varios de sus parámetros con el modificador **const** y ésta a su vez invoca a alguna función g pasando dichos parámetros como argumento a la función g, entonces los parámetros de g correspondientes a los argumentos pasados por f deben ser **const** para asegurarse la consistencia que f garantiza. Por ejemplo, en el siguiente código:

```
1  #include <iostream>
2
3  double promedio(const double a[], int tam);
4  void imprime(const double a[], int tam);
5
6  int main(void){
7    double calif[] = {6.5, 8.5, 7.5, 10};
8    double prom = promedio(calif,4);
9    std::cout << "El promedio de las calificaciones es: "
10             << prom << std::endl;
11 }
12
13 double promedio(const double a[], int tam){
14   imprime(a,tam);
15   double prom=0;
16   for(int i=0; i < tam; i++){
17     prom += a[i];
18   }
19   prom /= tam;
20 }
21
22 void imprime(const double a[], int tam){
23   std::cout << "Los elementos del arreglo son: ";
24   for(int i=0; i < tam; i++){
25     std::cout << a[i] << " ";
26   }
27   std::cout << std::endl;
28 }
```

en la función `promedio` el arreglo `a` es **`const`** y dicha función llama a la función `imprime`, pasándole como argumento el arreglo `a`; si en la función `imprime` el parámetro **`const`** **`double`** `a[]` no fuese **`const`**, entonces `imprime` podría modificar el argumento `a` y, por tanto, en `promedio` se modificaría `a` indirectamente a través de `imprime`, lo que resulta en una inconsistencia, pues promedio debe asegurar que no se modifique `a`. Esta inconsistencia debe ser detectada por el compilador.

A.3.8. Apuntadores a funciones

Es posible declarar apuntadores a funciones, que en ocasiones son útiles y proveen de flexibilidad al programador. Veamos el siguiente ejemplo:

```
1  #include <iostream>
2  #include <string>
3  #include <sstream>
4
5  using std::string;
6  using std::stringstream;
7  using std::cout;
8  using std::endl;
9
10 int cmpnum(string&, string&);
11 int cmpstring(string&, string&);
12
13 void bubblesort(int (*cmp) (string&, string&), string arr[],
14                              int length);
15 void swap(string& x, string& y);
16
17 int main(void){
18   int (*funptri) (string&, string&);
19   int (*funptrs) (string&, string&);
20   funptri = cmpnum;
21   funptrs = cmpstring;
22   string enteros[]={"52","36","21","15","74","110","96","232","14",
23                  "1","3","9","756","25","12","23","14","57","58",
24                  "22","9","100"};
25   bubblesort(funptri, enteros, 22);
26   string nombres[] ={"Veronica", "Alondra", "Berenice", "Dario",
27                      "Juana", "Ramon", "Roberto", "Heriberto",
28                      "Santiago", "Anastacia"};
29   bubblesort(funptrs,nombres,10);
30   cout << "Los elementos ordenados del arreglo enteros son: ";
31   for(int i=0; i < 22; i++){
32     cout << enteros[i] << " ";
33   }
34   cout << endl;
35   cout << "Los elementos ordenados del arreglo nombres son: ";
36   for(int i=0; i<10; i++){
37     cout << nombres[i] << " ";
38   }
39   cout << endl;
40 }
41
```

```
42 int cmpnum(string& a, string& b){
43   stringstream st;
44   st << a << " " << b;
45   double x;
46   double y;
47   st >> x;
48   st >> y;
49   if(x==y)
50     return 0;
51   return x < y;
52 }
53
54 int cmpstring(string& a, string& b){
55   if(a==b)
56     return 0;
57   return a < b;
58 }
59
60
61 void bubblesort(int (*cmp) (string&, string&), string arr[],
62                                 int length){
63   for(int i=0;i< length-1;i++){
64     for(int j=0;j<length-1;j++){
65       if((*cmp)(arr[j+1],arr[j])) {
66         swap(arr[j],arr[j+1]);
67       }
68     }
69
70   }
71 }
72
73 void swap(string& x, string& y){
74   string tmp = x;
75   x = y;
76   y = tmp;
77 }
```

El ejemplo anterior muestra cómo, haciendo uso de apuntadores a funciones, tenemos la posibilidad de generalizar una función; en este caso implementamos un algoritmo de ordenamiento y nuestro fin es que no sólo nos sirva para ordenar cadenas, sino también enteros. La función `bubblesort` realiza el ordenamiento sobre un arreglo de cadenas y para que pueda funcionar sobre enteros necesitamos proveer de una función de

comparación que sepa cómo tratar a las cadenas cuando éstas estén representando enteros y otra para cuando se traten de cadenas de texto en general. Las funciones `cmpnum` y `cmpstring` respectivamente, realizan este trabajo.[13] De esto, la función `bubblesort` tiene como primer parámetro un apuntador a función, donde pasaremos nuestra función de comparación según sea el caso; el enunciado

```
int (*funptri) (string&, string&)
```

declara un apuntador a función,[14] pues de otra forma, el enunciado

```
int *funptri (string&, string&)
```

se interpretaría como un intento de declaración de una función, que regresa un `int*`, con nombre `funptri`, con la intención que almacene la dirección de la función `cmpnum`. Esto se logra con el enunciado `funptri=cmpnum`. Hacemos lo análogo para la función `cmpstring`. Ahora estamos listos para invocar a la función `bubblesort`, dependiendo del criterio con el que queramos ordenar; si deseamos ordenar enteros, el primer argumento de la función es el apuntador a la función `funptri`, que trabaja con cadenas que representan enteros; si lo que se desea es ordenar cadenas con el orden lexicográfico usual, el primer argumento es el apuntador a la función `funptrs`; el segundo argumento es un arreglo de cadenas que en el primer caso se trata de enteros representados como cadenas y en el segundo caso se tomarán las cadenas tal cual; el tercer argumento, en ambos casos, será el tamaño del arreglo.

Como mencionamos anteriormente, el uso de apuntadores a funciones ofrece al programador flexibilidad y son útiles en diferentes escenarios; por ejemplo, podemos definir un arreglo de apuntadores a funciones y crear una función que invoque alguna de las funciones del arreglo sobre los argumentos que recibe según convenga.

A.3.9. Clases

Las clases en C++ son similares a la clases en Java aunque tienen algunas diferencias que veremos a continuación. Un ejemplo de clase en Java es el siguiente:

```
1 public class Persona{
2     String nombre;
3     double altura;
4
5     public Persona(String nombre, double altura){
6         this.nombre = nombre;
7         this.altura = altura;
8     }
9
```

[13] En particular, la función `cmpnum` convierte a las cadenas de enteros en enteros para que el ordenamiento sea numérico y no de cadenas.

[14] Es necesario el uso de paréntesis en `(*funptri)`.

```
10     public void setNombre(String nombre){
11          this.nombre=nombre;
12       }
13
14       public void setAltura(double altura){
15          this.altura=altura;
16       }
17
18  }
```

Una clase en C++ similar a la anterior es

```
1  #include <string>
2
3  class Persona{
4
5  public:
6    Persona(std::string nom, double alt){
7      nombre = nom;
8      altura = alt;
9    }
10
11   void setNombre(std::string nom){
12     nombre=nom;
13   }
14
15   void setAltura(double alt){
16     altura = alt;
17   }
18
19   std::string getNombre() const{
20     return nombre;
21   }
22
23   double getAlura() const{
24     return altura;
25   }
26
27
28 private:
29   std::string nombre;
30   double altura;
31
32 };
```

El primer detalle a notar es que una clase en C++ termina con `";"`. Las clases en Java pueden incluir el modificador **`public`** que denota un nivel de protección público y cuando éste no está presente el nivel de protección es de paquete –se puede usar también alguno de los modificadores **`private`** o **`protected`**–; en C++, en cambio, clase[15] por lo que todas las clases son públicas. En cuanto a los atributos o variables propias de clase, en Java se puede anteponer un modificador de acceso a cada uno de ellos, ya sea **`public`**, **`private`** o **`protected`** –si no hay modificador el acceso es de paquete–; en C++ en lugar de esto tenemos secciones que comienzan con con uno de los tres modificadores de acceso seguido de "`:`", en donde los atributos, también llamados variables miembro, tienen el nivel de acceso dictado por el modificador de la sección en la que se encuentran; la sección termina en donde comienza la siguiente, esto es, donde se encuentra el siguiente modificador seguido de `:` o, alternativamente, donde termina la clase; la sección por omisión es **`private`** y si bien se pueden alternar secciones con diferentes modificadores un número arbitrario de veces, es poco usual y es más claro y limpio utilizar sólo una sección del modificador deseado; no hay una regla en cuanto a si la primer sección debe ser **`public`** o **`private`**, aunque es más usual que sea **`public`** para facilitar al programador ver la interfaz que provee la clase y es la convención que utilizaremos en el presente texto. Por otra parte, observamos que las funciones miembro, al igual que los atributos, aparecen dentro de las secciones antes mencionadas y de igual forma se aplica el modificador de acceso de la sección en la cual están declaradas. Es conveniente, en ocasiones, declarar dentro de la clase sólo el prototipo de la función y afuera de ella la implementación de dicha función, como se muestra en el siguiente ejemplo para el constructor y la función `imprime`.

```
1  #include <iostream>
2
3  class Racional{
4  public:
5    Racional(int num, int den);
6    void imprime();
7    void setNumerador(int num){
8      numerador=num;
9    }
10   void setDenominador(int den){
11     denominador=den;
12   }
13
14 private:
15   int numerador;
16   int denominador;
17 };
```

[15] A menos de que se trate de una clase anidada.

```
18 Racional::Racional(int num, int den){
19   numerador=num;
20   denominador=den;
21 }
22
23 void Racional::imprime(){
24   std::cout << "El numero racional es: " << numerador
25             << "/" << denominador << std::endl;
26 }
27
28 int main(void){
29   Racional r(3,4);
30   r.imprime();
31 }
```

En este caso declaramos dentro de la clase el prototipo de la función `imprime` y el prototipo del constructor con dos parámetros y realizamos la implementación fuera de ella; para tal propósito debemos colocar el nombre de la clase de la cual es miembro la función que estamos implementando seguida del operador `::` (conocido como operador de resolución de alcance) antes del nombre de la función, donde estemos definiendo dicha función; también podemos implementar la función dentro de la clase como en el caso de la funciones `setNumerador` y `setDenominador`, sin embargo esto tiene una semántica diferente ya que implementar una función dentro de la clase implica que dicha función será **`inline`** lo que significa que cuando el compilador genere el código del programa cuando se encuentre una invocación a dicha función, en lugar de realizar un llamada a la función, substituirá la invocación por las instrucciones de la función, lo cual es una estrategia de optimización para evitar la sobrecarga de la llamada a la función; sin embargo sólo debe usarse cuando la función sea muy pequeña (una o dos líneas de código), debe evitarse para funciones grandes y nunca usarse en funciones recursivas; también puede utilizarse explícitamente el modificador **`inline`** cuando se implementa una función, como sigue:

```
1 inline void Racional::imprime(){
2   std::cout << "El numero racional es: " << numerador
3             << "/" << denominador << std::endl;
4 }
```

Comparemos ahora cómo se crea un objeto en ambos lenguajes; en Java se realiza como se ve en la línea 15 del listado en la siguiente página.

```
1  public class Racional{
2      int numerador;
3      int denominador;
4      public Racional(int numerador, int denominador){
5          this.numerador = numerador;
6          this.denominador = denominador;
7      }
8
9      public void imprime(){
10         System.out.println("El numero racional es: " + numerador
11                         + "/" + denominador);
12     }
13
14     public static void main(String[] args){
15         Racional r = new Racional(3,4);
16         r.imprime();
17     }
18 }
```

Mientras que en Java los objetos sólo se crean en el heap, en C++ se pueden crear en la pila de ejecución o en el heap.

En el enunciado en la línea 15 se crea un nuevo objeto en el heap invocando al constructor con dos parámetros; lo que almacena la variable `r` es una referencia a dicho objeto. Fijémonos ahora en el caso de C++. En el código mostrado con anterioridad de la clase `Racional`, el enunciado `Racional r(3,4)` crea un nuevo objeto invocando al constructor de dos parámetros, pero dicho objeto se crea en la pila, en este caso en el marco de pila correspondiente a la función `main`, y se destruye cuando la llamada de dicha función termina. Una forma más parecida a Java de crear un objeto en C++, que se ubica en el heap, es la siguiente:

```
1  int main(void){
2    Racional* r = new Racional(3,4);
3    r->imprime();
4    delete(r);
5  }
```

En este caso se crea a través del operador *new* un nuevo objeto en el heap y dicho operador regresa la dirección de memoria del objeto recién creado, la cual es alojada en el apuntador a `Racional` de nombre `r`, que es exactamente igual a lo que sucede en Java, sólo que en Java se oculta el manejo de apuntadores. Aquí el apuntador `r` es el que se destruye al terminar la llamada a la función `main`, aunque el objeto sigue estando en el heap pero ya no hay apuntadores que se refieran a él. En C++ es el programador quien

debe encargarse de liberar la memoria que ocupan los objetos que ya no son útiles y esto se logra con el uso del operador *delete*; en Java esto no es necesario ya que el recolector de basura lo hace por nosotros.

Métodos de consulta y actualización

Todo lenguaje orientado a objetos presenta métodos que permiten únicamente ver el estado de las variables propias o atributos y que se conocen como *métodos de consulta* (*accessors*). También distinguen estos lenguajes a los métodos o funciones propias que permiten cambiar el estado del objeto y que se conocen como métodos de actualización (en C++, *mutators*). Hemos visto ya que cuando deseamos pasar como argumento a una función un objeto que no queremos que sea modificado, lo pasamos por referencia constante; esto nos asegura que el objeto no será modificado en el cuerpo de la función, como se muestra en el siguiente ejemplo.

```
1  #include <iostream>
2  #include <string>
3
4  class Persona{
5
6  public:
7    Persona(std::string nom, double alt){
8      nombre = nom;
9      altura = alt;
10   }
11
12   void setNombre(std::string nom){
13     nombre=nom;
14   }
15
16   void setAltura(double alt){
17     altura = alt;
18   }
19
20   std::string getNombre(){
21     return nombre;
22   }
23
24   double getAltura(){
25     return altura;
26   }
27
```

```
28    bool compara(const Persona & otra){
29      return (this->nombre == otra.getNombre())
30        && (this->altura == otra.getAltura());
31    }
32  private:
33    std::string nombre;
34    double altura;
35
36  };
37
38  int main(void){
39
40    Persona uno("Genaro",1.60);
41    Persona dos("Genaro",1.60);
42    if( uno.compara(dos))
43        std::cout << "Son la misma persona" << std::endl;
44  }
```

Aquí, el método `compara` de la clase `Persona` se encarga de comparar si dos objetos de la clase `Persona` describen a la misma persona. Ahora bien, dicho método recibe como argumento una referencia a un objeto de la clase `Persona` y esto es deseable para evitar el costo del paso por valor; por otra parte, queremos asegurarnos que el argumento que pasemos no sea modificado dentro del cuerpo del método, por lo que pasamos una referencia constante –ver línea 28–. Hasta aquí todo luce bien, pero el código anterior no compila. El problema radica en los enunciados `otra.getNombre()` y `otra.getAltura()`, donde se invoca a los métodos `getNombre()` y `getAltura()` respectivamente, sobre el objeto que es una referencia constante, por lo que el método `compara` no debe modificar el estado del objeto; aunque ninguno de los dos métodos modifica al objeto, el compilador no sabe que dichos métodos no lo harán; por tanto, necesitamos un mecanismo para indicar que un método no modificará los atributos de la clase y esto se logra poniendo el modificador **`const`** después de la lista de parámetros en la declaración de un método, como se ilustra a continuación.

```
1  #include <iostream>
2  #include <string>
3
4  class Persona{
5  public:
6    Persona(std::string nom, double alt){
7      nombre = nom;
8      altura = alt;
9    }
```

```
10
11    void setNombre(std::string nom){
12      nombre=nom;
13    }
14
15    void setAltura(double alt){
16      altura = alt;
17    }
18
19    std::string getNombre() const{
20      return nombre;
21    }
22
23    double getAltura() const{
24      return altura;
25    }
26
27    bool compara(const Persona & otra){
28      return (this->nombre == otra.getNombre())
29        && (this->altura == otra.getAltura());
30    }
31
32  private:
33    std::string nombre;
34    double altura;
35
36  };
37
38  int main(void){
39
40    Persona uno("Genaro",1.60);
41    Persona dos("Genaro",1.60);
42    if( uno.compara(dos))
43        std::cout << "Son la misma persona" << std::endl;
44  }
```

Los métodos `getNombre` y `getAltura`, definidos como se muestra en el código anterior, aseguran que no modificarán los atributos de la clase y, por tanto, el código anterior compila y funciona correctamente. En general, todo método que no hace modificaciones sobre los atributos de la clase a la que pertenece se llama de consulta, en comparación con los métodos que hacen modificaciones sobre los atributos de la clase y que se llaman de actualización. En Java no hay forma de indicar en la declaración de un método si éste se trata de uno de consulta o uno de actualización; en C++ todo método es de actualización a menos que se indique lo contrario, por lo que siempre que se trate de uno de

consulta debe indicarse, ya que en C++ definir si se trata de uno u otro es parte del diseño de la clase y un uso inadecuado puede presentar errores como el visto anteriormente. Ocasionalmente, aun cuando un método sea de consulta, necesitará modificar un atributo de la clase; para que esto se permita dicho atributo de la clase debe ser declarado anteponiendo el modificador **mutable**, como se muestra a continuación

```
1  #include <iostream>
2  #include <string>
3
4  class Persona{
5
6  public:
7    Persona(std::string nom, double alt){
8      nombre = nom;
9      altura = alt;
10     numSolAlt = 0;
11   }
12
13   void setNombre(std::string nom){
14     nombre=nom;
15   }
16
17   void setAltura(double alt){
18     altura = alt;
19   }
20
21   std::string getNombre() const{
22     return nombre;
23   }
24
25   double getAltura() const{
26     numSolAlt++;
27     return altura;
28   }
29
30   int getNumSolAlt() const{
31     return numSolAlt;
32   }
33
34   bool compara(const Persona & otra){
35     return (this->nombre == otra.getNombre())
36       && (this->altura == otra.getAltura());
37   }
38
```

```
39 private:
40   std::string nombre;
41   double altura;
42   mutable int numSolAlt;
43 };
44
45 int main(void){
46
47   Persona uno("Genaro",1.60);
48   uno.getAltura();
49   uno.getAltura();
50   uno.getAltura();
51   std::cout << "El numero de solicitudes de la altura es: "
52             << uno.getNumSolAlt() << std::endl;
53 }
```

En la línea 42 tenemos un atributo de la clase, `numSolAlt`, que es un contador del número de veces que se ha solicitada la altura de una persona; el método `getAltura()` es de consulta ya que usualmente sólo regresa el atributo `altura`, pero en esta ocasión debe actualizar el contador `numSolAlt`; por ello dicho atributo debe ser **`mutable`**.

Constructores

Tanto en C++ como en Java, cuando se crea un objeto a los atributos de dicho objeto se les asigna un valor inicial antes de que se ejecute el cuerpo del constructor. Los valores iniciales para los atributos en el caso de Java son `null` para los objetos, `0` para los numéricos y **`false`** para los booleanos. En C++ esto cambia, ya que para los atributos que son objetos se invoca a su constructor sin parámetros[16] y los atributos con tipos primitivos se inicializan con su valor por omisión, `0` para los numéricos y **`false`** para los los booleanos;[17] en el caso que los atributos sean apuntadores[18] se inicializan con `NULL`, donde en *C++* `NULL` es una constante que tiene el valor de cero. Todo lo anterior se lleva a cabo sólo si el objeto es un objeto global, ya que si se trata de un objeto local entonces la inicialización de sus atributos será indeterminado y puede contener valores basura.[19] Veamos un ejemplo.

[16] En el caso que no exista un constructor sin parámetros habrá un error en tiempo de compilación.
[17] Representando internamente como 0 por algunos compiladores.
[18] Referencias en terminología Java.
[19] Esto si no se define un constructor explícitamente por el programador.

```
1  #include <iostream>
2
3  class Aux{
4  public:
5    int i;
6    bool b;
7  };
8
9  class ejemplo{
10 public:
11
12   int uno;
13   int dos;
14
15   bool b;
16
17   int* ptru;
18   int* ptrd;
19
20   bool* ptrb;
21
22   Aux* ptra;
23
24   Aux a;
25 };
26
27 ejemplo ejglobl;
28 int main(void){
29   std::cout << "Los valores por omision en el ejemplo global son: "
30           << ejglobl.uno << " " << ejglobl.dos << " " << ejglobl.b
31           << " "  << ejglobl.a.i << " " << ejglobl.a.b << " "
32           << ejglobl.ptru <<  " " << ejglobl.ptrd <<  " "
33           << ejglobl.ptrb << " " << ejglobl.ptra << std::endl;
34   ejemplo ejlocal;
35   std::cout << "Los valores por omision en el ejemplo local son: "
36           << ejlocal.uno << " " << ejlocal.dos << " " << ejlocal.b
37           << " " << ejlocal.a.i << " " << ejlocal.a.b << " "
38           << ejlocal.ptru << " " << ejlocal.ptrd << " "
39           << ejlocal.ptrb << " " <<
40     ejlocal.ptra << std::endl;
41 }
```

En el ejemplo anterior, el objeto `ejglobal` –línea 27– es un objeto global y por tanto su inicialización es como se describió anteriormente; en cambio, el objeto `ejlocal` –línea 34– es un objeto local y los valores iniciales de sus atributos pueden ser valores basura. La ejecución del código anterior imprime en pantalla las siguientes líneas

```
Los valores por omision en el ejemplo global son: 0 0 0 0 0 0 0 0 0 0
Los valores por omision en el ejemplo local son: 1581118160 32767
   66 1581118456255 0x2b7e4cbd6540 0x400c90 0 0x4007e3
```

En Java no hay forma de cambiar esta inicialización, a diferencia de C++ en la cual sí se puede cambiar esta forma de inicialización por medio de inicializadores, los cuales veremos más adelante.

Cabe señalar que en C++, aun cuando no se llame al constructor de manera explícita cuando se declara un objeto, como es el caso de los objetos `ejglobl` y `ejlocal` en este ejemplo, se ejecuta el constructor (en este caso el constructor por omisión que se crea implícitamente), lo que no es así en Java, cuando si sólo se declara un objeto, pero no se llama al constructor explícitamente, éste no se ejecuta.

En C++, al igual que en Java, si no se declara ningún constructor, se crea uno por omisión sin parámetros, que es análogo a definir en Java un constructor sin parámetros y con el cuerpo vacío como se ilustra a continuación.

```
1  class Par{
2      int uno;
3      int dos;
4
5      Par(){
6      }
7
8      public static void main(String[] args){
9          Par p = new Par();
10     }
11 }
```

Es importante notar que se crea un constructor sin parámetros implícitamente para una clase sólo cuando el programador no define ningún otro de manera explícita; en el caso que el programador defina alguno ya no se creará uno sin parámetros por omisión. En el ejemplo mostrado con anterioridad se crea un constructor sin parámetros por omisión para la clase `ejemplo`, que inicializa los tipos primitivos con su valor por omisión, los apuntadores con `NULL` y llama al constructor sin parámetros de la clase `Aux` para inicializar el atributo `a`; en este caso también se creó implícitamente un constructor por omisión para la clase `Aux`. Ahora, si en C++ definimos un constructor para la clase `Aux` de la siguiente forma:

```
1  class Aux{
2  public:
3    Aux(int ent, bool bol){
4      i = ent;
5      b = bol;
6    }
7
8    int i;
9    bool b;
10 };
```

no se construirá un constructor sin parámetros por omisión y el programa no compilará. Para que el código compile necesitamos una forma para que el constructor por omisión de la clase `ejemplo` llame al constructor con dos parámetros de la clase `Aux`, lo que se logra por medio de *inicializadores*, como se muestra a continuación:

```
1  class ejemplo{
2  public:
3    ejemplo():a(0,false){
4    }
5
6    int uno;
7    int dos;
8
9    bool b;
10
11   int* ptru;
12   int* ptrd;
13
14   bool* ptrb;
15
16   Aux* ptra;
17
18   Aux a;
19 };
```

El constructor sin parámetros, definido como se muestra en el código anterior, llama al constructor de dos parámetros de la clase `Aux` para inicializar el atributo `a`, aunque el programador de Java habrá pensado hacer lo anterior de la siguiente forma:

```
1  ejemplo(){
2     a = Aux(0,false);
3   }
```

Lo anterior no funcionará ya que en C++, antes de ejecutarse el cuerpo del constructor, se inicializan todos los atributos a su valor por omisión; por ello, en éste, antes de que se ejecute el cuerpo del constructor se intentará inicializar `a` llamando al constructor sin parámetros de la clase `Aux` y, como ese constructor no existe, ocurrirá un error; la única forma de cambiar el valor por omisión de los atributos es por medio de inicializadores.

Inicializadores

La inicialización de variables y atributos se manejan muy distinto en C++ y Java. Como ya mencionamos, en Java, cuando se crea un objeto los atributos que pertenecen a él y que son de tipos primitivos, se inicializan con su correspondiente valor por omisión y todos aquellos que son objetos (referencias) se inicializan con `null` antes de que se ejecute el cuerpo del constructor; no hay forma de que esta inicialización se cambie.

Veamos otro ejemplo:

```
1  class Entero{
2  public:
3    Entero(int ent){
4      entero = ent;
5    }
6  private:
7    int entero;
8  };
9
10 int main(void){
11   Entero e = 10;
12 }
```

En este ejemplo utilizamos la clase `Entero` como un envoltorio para un tipo primitivo **`int`**. El enunciado `Entero e = 10` está permitido, porque en C++ siempre que sea posible se intenta realizar una conversión implícita de tipos, en este caso de **`int`** a `Entero`.

Con esto damos por terminada esta breve introducción a C++ desde Java. Cubrimos los principales aspectos necesarios para la implementación de los algoritmos necesarios para compilación.

APÉNDICE B

Gramática de Python

FILE_INPUT	$\longrightarrow$	(***newline*** | STMT)*
DECORATOR	$\longrightarrow$	**@** DOTTED_NAME [**(** [ARGLIST] **)**] ***newline***
DECORATORS	$\longrightarrow$	DECORATOR$^+$
FUNCDEF	$\longrightarrow$	[DECORATORS] **def** ***name*** PARAMETERS **:** SUITE
PARAMETERS	$\longrightarrow$	**(** [VARARGSLIST] **)**
VARARGSLIST	$\longrightarrow$	(FPDEF [**=** TEST] **,**)* (***** ***name*** [**,** ****** ***name***] | ****** ***name***)
VARARGSLIST	$\longrightarrow$	FPDEF [**=** TEST] (**,** FPDEF [**=** TEST])* [**,**]
FPDEF	$\longrightarrow$	***name*** | **(** FPLIST **)**
FPLIST	$\longrightarrow$	FPDEF (**,** FPDEF)* [**,**]
STMT	$\longrightarrow$	SIMPLE_STMT | COMPOUND_STMT
SIMPLE_STMT	$\longrightarrow$	SMALL_STMT (**;** SMALL_STMT)* [**;**] ***newline***
SMALL_STMT	$\longrightarrow$	EXPR_STMT | PRINT_STMT | DEL_STMT | PASS_STMT
SMALL_STMT	$\longrightarrow$	FLOW_STMT | IMPORT_STMT | GLOBAL_STMT
SMALL_STMT	$\longrightarrow$	EXEC_STMT | ASSERT_STMT
EXPR_STMT	$\longrightarrow$	TESTLIST (AUGASSIGN TESTLIST | (**=** TESTLIST)*)
AUGASSIGN	$\longrightarrow$	**+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | ****=** | **//=**
PRINT_STMT	$\longrightarrow$	**print** ([TEST (**,** TEST)* [**,**]] | **>>** TEST [(**,** TEST)$^+$ [**,**]])
DEL_STMT	$\longrightarrow$	**del** EXPRLIST
PASS_STMT	$\longrightarrow$	**pass**

FLOW_STMT ⟶ BREAK_STMT | CONTINUE_STMT | RETURN_STMT
FLOW_STMT ⟶ RAISE_STMT | YIELD_STMT
BREAK_STMT ⟶ **break**
CONTINUE_STMT ⟶ **continue**
RETURN_STMT ⟶ **return** [TESTLIST]
YIELD_STMT ⟶ **yield** TESTLIST
RAISE_STMT ⟶ **raise** [TEST [**,** TEST [**,** TEST]]]
IMPORT_STMT ⟶ IMPORT_NAME | IMPORT_FROM
IMPORT_NAME ⟶ **import** DOTTED_AS_NAMES
IMPORT_FROM ⟶ **from** DOTTED_NAME **import**
(***** | **(** IMPORT_AS_NAMES **)** | IMPORT_AS_NAMES)
IMPORT_AS_NAME ⟶ ***name*** [***name name***]
DOTTED_AS_NAME ⟶ DOTTED_NAME [***name name***]
IMPORT_AS_NAMES ⟶ IMPORT_AS_NAME (**,** IMPORT_AS_NAME)* [**,**]
DOTTED_AS_NAMES ⟶ DOTTED_AS_NAME (**,** DOTTED_AS_NAME)*
DOTTED_NAME ⟶ ***name*** (**.** ***name***)*
GLOBAL_STMT ⟶ **global** ***name*** (**,** ***name***)*
EXEC_STMT ⟶ **exec** EXPR [**in** TEST [**,** TEST]]
ASSERT_STMT ⟶ **assert** TEST [**,** TEST]
COMPOUND_STMT ⟶ IF_STMT | WHILE_STMT | FOR_STMT | TRY_STMT
COMPOUND_STMT ⟶ FUNCDEF | CLASSDEF
IF_STMT ⟶ **if** TEST **:** SUITE (**elif** TEST **:** SUITE)* [**else :** SUITE]
WHILE_STMT ⟶ **while** TEST **:** SUITE [**else :** SUITE]
FOR_STMT ⟶ **for** EXPRLIST **in** TESTLIST **:** SUITE [**else :** SUITE]
TRY_STMT ⟶ (**try :** SUITE (EXCEPT_CLAUSE **:** SUITE)+ [**else :** SUITE]
| **try :** SUITE **finally :** SUITE)
EXCEPT_CLAUSE ⟶ **except** [TEST [**,** TEST]]
SUITE ⟶ SIMPLE_STMT | ***newline indent*** STMT+ ***dedent***
TEST ⟶ AND_TEST (**or** AND_TEST)* | LAMBDEF
AND_TEST ⟶ NOT_TEST (**and** NOT_TEST)*
NOT_TEST ⟶ **not** NOT_TEST | COMPARISON
COMPARISON ⟶ EXPR (COMP_OP EXPR)*
COMP_OP ⟶ **<** | **>** | **==** | **>=** | **<=** | **<>** | **!=** | **in** | **not in** | **is** | **is not**

EXPR ⟶ XOR_EXPR (**|** XOR_EXPR)*
XOR_EXPR ⟶ AND_EXPR (**^** AND_EXPR)*
AND_EXPR ⟶ SHIFT_EXPR (**&** SHIFT_EXPR)*
SHIFT_EXPR ⟶ ARITH_EXPR ((**<<** | **>>**) ARITH_EXPR)*
ARITH_EXPR ⟶ TERM ((**+** | **-**) TERM)*
TERM ⟶ FACTOR ((***** | **/** | **%** | **//**) FACTOR)*
FACTOR ⟶ (**+** | **-** | **~**) FACTOR | POWER
POWER ⟶ ATOM TRAILER* [****** FACTOR]
ATOM ⟶ **(** [TESTLIST_GEXP] **)** | **[** [LISTMAKER] **]** | **{** [DICTMAKER] **}**
ATOM ⟶ **‘** TESTLIST1 **‘** | ***name*** | ***number*** | ***string***+
LISTMAKER ⟶ TEST (LIST_FOR | (**,** TEST)* [**,**])
TESTLIST_GEXP ⟶ TEST (GEN_FOR | (**,** TEST)* [**,**])
LAMBDEF ⟶ **lambda** [VARARGSLIST] **:** TEST
TRAILER ⟶ **(** [ARGLIST] **)** | **[** SUBSCRIPTLIST **]** | **.** ***name***
SUBSCRIPTLIST ⟶ SUBSCRIPT (**,** SUBSCRIPT)* [**,**]
SUBSCRIPT ⟶ **...** | TEST | [TEST] **:** [TEST] [SLICEOP]
SLICEOP ⟶ **:** [TEST]
EXPRLIST ⟶ EXPR (**,** EXPR)* [**,**]
TESTLIST ⟶ TEST (**,** TEST)* [**,**]
TESTLIST_SAFE ⟶ TEST [(**,** TEST)+ [**,**]]
DICTMAKER ⟶ TEST **:** TEST (**,** TEST **:** TEST)* [**,**]
CLASSDEF ⟶ **class** ***name*** [**(** TESTLIST **)**] **:** SUITE
ARGLIST ⟶ (ARGUMENT **,**)*
(ARGUMENT [**,**] | ***** TEST [**,** ****** TEST] | ****** TEST)
ARGUMENT ⟶ TEST [GEN_FOR] | TEST **=** TEST [**(** GEN_FOR **)**]
LIST_ITER ⟶ LIST_FOR | LIST_IF
LIST_FOR ⟶ **for** EXPRLIST **in** TESTLIST_SAFE [LIST_ITER]
LIST_IF ⟶ **if** TEST [LIST_ITER]
GEN_ITER ⟶ GEN_FOR | GEN_IF
GEN_FOR ⟶ **for** EXPRLIST **in** TEST [GEN_ITER]
GEN_IF ⟶ **if** TEST [GEN_ITER]
TESTLIST1 ⟶ TEST (**,** TEST)*

Bibliografía

[Age95] Ole Agesen. "The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism". En: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, págs. 2-26. ISBN: 3-540-60160-0. URL: http://dl.acm.org/citation.cfm?id=646153.679533.

[Age96a] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Inf. téc. Mountain View, CA, USA, 1996.

[Age96b] Ole Agesen. "Concrete type inference: delivering object-oriented applications". UMI Order No. GAX96-20452. Tesis doct. Stanford, CA, USA, 1996.

[Aho+07] Alfred V. Aho y col. *Compilers: Principles, Techniques, and Tools*. Second. Boston, MA: Addison-Wesley, oct. de 2007.

[AHU83] Alfred V. Aho, John E. Hopcroft y Jeffrey Ullman. *Data Structures and Algorithms*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201000237.

[App04] Andrew W. Appel. *Modern Compiler Implementation in ML*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521607647.

[App98] Andrew W. Appel. "SSA is Functional Programming". En: *ACM SIGPLAN Notices* 33.4 (abr. de 1998), págs. 17-20.

[AU72] Alfred V. Aho y Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Vol. I. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1972. ISBN: 0-13-914556-7.

[AU73] Alfred V. Aho y Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Vol. II. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1973. ISBN: 0-13-914564-8.

[BD94] Manuel E. Benitez y Jack W. Davidson. "The Advantages of Machine-Dependent Global Optimization". En: *Programming Languages and System Architectures*. 1994, págs. 105-124.

[BL89] M. E. Bermudez y G. Logothetis. "Simple computation of LALR(1) lookahead sets". En: *Inf. Process. Lett.* 31.5 (jun. de 1989), págs. 233-238. ISSN: 0020-0190. DOI: 10.1016/0020-0190(89)90079-3. URL: http://dx.doi.org/10.1016/0020-0190(89)90079-3.

[Boi+09] Benoit Boissinot y col. "Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency". En: *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, págs. 114-125. ISBN: 978-0-7695-3576-0. DOI: 10.1109/CGO.2009.19. URL: http://dx.doi.org/10.1109/CGO.2009.19.

[Bri+98] Preston Briggs y col. "Practical improvements to the construction and destruction of static single assignment form". En: *Softw. Pract. Exper.* 28.8 (jul. de 1998), págs. 859-881. ISSN: 0038-0644. DOI: 10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8. URL: http://dx.doi.org/10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8.

[Bri06] Philip Brisk. "Advances in static single assignment form and register allocation". AAI3254798. Tesis doct. Los Angeles, CA, USA, 2006.

[Bud00] Timothy Budd. *Classic Data Structures in Java: A Visual and Explorational Approach*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201700026.

[BZ10] Sebastian Buchwald y Andreas Zwinkau. "Instruction selection by graph transformation". En: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*. CASES '10. Scottsdale, Arizona, USA: ACM, 2010, págs. 31-40. ISBN: 978-1-60558-903-9. DOI: 10.1145/1878921.1878926. URL: http://doi.acm.org/10.1145/1878921.1878926.

[Car04] Luca Cardelli. "Type Systems". En: ed. por Allen B. Tucker. Second. CRC Press, feb. de 2004. Cap. 97.

[CCF91] Jong-Deok Choi, Ron Cytron y Jeanne Ferrante. "Automatic construction of sparse data flow evaluation graphs". En: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '91. Orlando, Florida, United States: ACM, 1991, págs. 55-66. ISBN: 0-89791-419-8. DOI: 10.1145/99583.99594. URL: http://doi.acm.org/10.1145/99583.99594.

[CH94] William D. Clinger y Lars Thomas Hansen. "Lambda, the ultimate label or a simple optimizing compiler for Scheme". En: *SIGPLAN Lisp Pointers* VII.3 (jul. de 1994), págs. 128-139. ISSN: 1045-3563. DOI: 10.1145/182590.156786. URL: http://doi.acm.org/10.1145/182590.156786.

[Cha+81] Gregory J. Chaitin y col. "Register allocation via coloring". En: *Comput. Lang.* 6.1 (ene. de 1981), págs. 47-57. ISSN: 0096-0551. DOI: 10.1016/0096-0551(81)90048-5. URL: http://dx.doi.org/10.1016/0096-0551(81)90048-5.

[Cona] Flying Frog Consultancy. *C++ vs OCaml: Ray tracer comparison*. URL: http://www.ffconsultancy.com/languages/ray_tracer/comparison.html.

[Conb] Flying Frog Consultancy. *C++ vs SML: Ray tracer comparison*. URL: http://www.ffconsultancy.com/languages/ray_tracer/comparison_cpp_vs_sml.html.

[Cyt+91] Ron Cytron y col. "Efficiently computing static single assignment form and the control dependence graph". En: *ACM Trans. Program. Lang. Syst.* 13.4 (oct. de 1991), págs. 451-490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[DF84] Jack W. Davidson y Christopher W. Fraser. "Code Selection through Object Code Optimization". En: *ACM Transactions on Programming Languages and Systems* 6.4 (1984), págs. 505-526.

[DP82] Frank DeRemer y Thomas Pennello. "Efficient Computation of LALR(1) Look-Ahead Sets". En: *ACM Trans. Program. Lang. Syst.* 4.4 (oct. de 1982), págs. 615-649. ISSN: 0164-0925. DOI: 10.1145/69622.357187. URL: http://doi.acm.org/10.1145/69622.357187.

[DS09] Charles Donnelly y Richard Stallman. *Bison: the Yacc-compatible parser generator*. Free Software Foundation, Inc. 2009, pág. 169.

[Ebn+08] Dietmar Ebner y col. "Generalized instruction selection using SSA-graphs". En: *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. LCTES '08. Tucson, AZ, USA: ACM, 2008, págs. 31-40. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375663. URL: http://doi.acm.org/10.1145/1375657.1375663.

[EG03] M. Anton Ertl y David Gregg. "The Structure and Performance of Efficient Interpreters". En: *Jornal of Instruction-Level Parallelism* 5 (2003), págs. 1-25.

[Fou] Python Software Foundation. *The Python Language Reference*. URL: `http://docs.python.org/reference/`.

[Ghu] Abdulaziz Ghuloum. *Ikarus Scheme*. URL: `https://launchpad.net/ikarus`.

[GJR79] S. L. Graham, W. N. Joy y O. Roubine. "Hashed symbol tables for languages with explicit scope control". En: *Proceedings of the 1979 SIGPLAN symposium on Compiler construction*. SIGPLAN '79. Denver, Colorado, United States: ACM, 1979, págs. 50-57. ISBN: 0-89791-002-8. DOI: `10.1145/800229.806953`. URL: `http://doi.acm.org/10.1145/800229.806953`.

[HP06] John L. Hennessy y David A. Patterson. *Computer Architecture: A Quantitative Approach*. Fourth. Morgan Kaufmann, 2006.

[HU79] John E. Hopcroft y Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Reading, Mass.: Addison Wesley, 1979.

[Hud+07] Paul Hudak y col. "A history of Haskell: being lazy with class". En: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. HOPL III. San Diego, California: ACM, 2007, ISBN: 978-1-59593-766-7. DOI: `10.1145/1238844.1238856`. URL: `http://doi.acm.org/10.1145/1238844.1238856`.

[Int99] SPARC International. *SPARC COMPLIANCE DEFINITION 2.4.1*. SPARC International, Inc., 1999.

[Joh75] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Technical Report CSTR32. Murray Hill, NJ: Bell Laboratories, 1975.

[Kes95] Richard A. Kesley. "A Correspondence between Continuation Passing Style and Static Single Assignment Form". En: *ACM SIGPLAN Workshop on Intermediate Representations*. 1995, págs. 13-22.

[Knu65] D. Knuth. "On the Translation of Languages from Left to Right". En: *Information and Control* 8 (1965), págs. 607-639.

[Knu97] Donald E. Knuth. "The Art of Computer Programming". En: Third. Vol. 1. Addison-Wesley, 1997. Cap. 1, págs. 216-228.

[KR88] Brian W. Kernighan y Dennis M. Ritchie. *The C programming language*. Upper Saddle River, NJ, USA: Prentice Hall Press, 1988. ISBN: 0-13-110362-8.

[LS79] Michael E. Lesk y Eric Schmidt. "Lex – A Lexical Analyzer Generator". En: *UNIX Programmer's Manual*. Vol. 2. AT&T Bell Laboratories Technical Report in 1975. pub-HRW:adr: Holt, Rinehart, y Winston, 1979, págs. 388-400.

[Mat+12] Michael Matz y col., eds. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. 2012.

[Mor98] Robert Morgan. *Building an optimizing compiler*. Newton, MA, USA: Digital Press, 1998. ISBN: 1-55558-179-X.

[Pel88] Eduardo Pelegri-Llopart. *Rewrite Systems, Pattern Matching, and Code Generation*. Inf. téc. Berkeley, CA, USA, 1988.

[PH08] David A. Patterson y John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Fourth. Morgan Kaufmann, 2008.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0-262-16209-1.

[PM07] Vern Paxson y John Millaway. *Lexical Analysis with Flex*. 2.5.35. Sep. de 2007.

[PQ96] Terence J. Parr y Russell W. Quong. "LL and LR translators need k > 1 lookahead". En: *SIGPLAN Not.* 31.2 (feb. de 1996), págs. 27-34. ISSN: 0362-1340. DOI: 10.1145/226060.226066. URL: http://doi.acm.org/10.1145/226060.226066.

[Pro92] Todd Alan Proebsting. "Code generation techniques". UMI Order No. GAX92-31217. Tesis doct. Madison, WI, USA, 1992.

[PS91] Jens Palsberg y Michael I. Schwartzbach. "Object-oriented type inference". En: *Conference proceedings on Object-oriented programming systems, languages, and applications*. OOPSLA '91. Phoenix, Arizona, United States: ACM, 1991, págs. 146-161. ISBN: 0-201-55417-8. DOI: 10.1145/117954.117965. URL: http://doi.acm.org/10.1145/117954.117965.

[Qui08] Fernando Magno Quintao Pereira. "Register allocation by puzzle solving". AAI3354420. Tesis doct. Los Angeles, CA, USA, 2008. ISBN: 978-1-109-12399-9.

[RS69] D. J. Rosenkrantz y R. E. Stearns. "Properties of deterministic top down grammars". En: *Proceedings of the first annual ACM symposium on Theory of computing*. STOC '69. Marina del Rey, California, USA: ACM, 1969, págs. 165-180. DOI: 10.1145/800169.805431. URL: http://doi.acm.org/10.1145/800169.805431.

[Sco09] Michael L. Scott. *Programming Language Pragmatics*. Third. Morgan Kaufmann, 2009.

[SG95] Vugranam C. Sreedhar y Guang R. Gao. "A linear time algorithm for placing &phgr;-nodes". En: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '95. San Francisco, California, United States: ACM, 1995, págs. 62-73. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199464. URL: http://doi.acm.org/10.1145/199448.199464.

[SH87] Peter Steenkiste y Jhon Hennessy. "Tags and Type Checking in LISP: Hardware and Software Approaches". En: *ASPLOS-II Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. Ed. por Randy Katz. Los Alamitos, California: IEEE Computer Society Press, oct. de 1987, págs. 50-59.

[SIK09] Masataka Sassa, Yo Ito y Masaki Kohama. "Comparison and evaluation of back-translation algorithms for static single assignment forms". En: *Comput. Lang. Syst. Struct.* 35.2 (jul. de 2009), págs. 173-195. ISSN: 1477-8424. DOI: 10.1016/j.cl.2007.03.001. URL: http://dx.doi.org/10.1016/j.cl.2007.03.001.

[Sre+99] Vugranam C. Sreedhar y col. "Translating Out of Static Single Assignment Form". En: *Proceedings of the 6th International Symposium on Static Analysis*. SAS '99. London, UK, UK: Springer-Verlag, 1999, págs. 194-210. ISBN: 3-540-66459-9. URL: http://dl.acm.org/citation.cfm?id=647168.718132.

[SS90] Seppo Sippu y E. Soisalon-Soininen. *Parsing theory volume 2: LR(K) and LL(K) parsing*. New York, NY, USA: Springer-Verlag New York, Inc., 1990. ISBN: 0-387-51732-4.

[SS98] Gerald Jay Sussman y Guy L. Steele Jr. "Scheme: A Interpreter for Extended Lambda Calculus". En: *Higher-Order and Symbolic Computation* 11.4 (1998), págs. 405-439.

[Ste78] Guy Lewis Steele Jr. *Rabbit: A Compiler for Scheme*. MIT AI Memo 474. Cambridge, Massachusetts: Massachusetts Institute of Technology, mayo de 1978.

[TA97] Inc. The Santa Cruz Operation y AT&T. *SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement*. Fourth. The Santa Cruz Operation, Inc., 1997.

[TC11] Linda Torczon y Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012088478X.

[Vis08] Elisa Viso Gurovich. *Introducción a la teoría de la computación (Autómatas y lenguajes formales)*. las prensas de ciencias, 2008. ISBN: 978-970-32-5415-6.