UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Prácticas y proyectos para el curso de Compiladores

REPORTE DE ACTIVIDAD DOCENTE

QUE PARA OBTENER EL TÍTULO DE:

Licenciada en Ciencias de la Computación

P R E S E N T A :

Diana Olivia Montes Aguilar

TUTORA:

Dra. Elisa Viso Gurovich

2018

1. Datos del alumno

Montes
Aguilar
Diana Olivia
5551673680
Universidad Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
409071870

2. Datos del tutor

Dra.
Elisa
Viso
Gurovich

3. Datos del sinodal 1

Dra.
Sofía Natalia
Galicia
Haro

4. Datos del sinodal 2

M. en C.
Ángel Francisco
Zúñiga
Chávez

5. Datos del sinodal 3

Ing.
Adrián Ulises

Mercado
Martínez

6. Datos del sinodal 4

Dr.
José de Jesús
Galaviz
Casas

7. Datos del trabajo escrito

Prácticas y proyectos para el curso de Compiladores
64p
2018

Índice general

Agradecimientos

Gracias a Dios.

Gracias a mi tutora, la Dra. Elisa Viso Gurovich, que sin su constante guía, el término de este trabajo hubiera sido imposible.

Gracias a mi familia y amigos por su constante apoyo y comprensión.

Gracias a esta maravillosa institución que me abrió las puertas a la educación.

Y gracias a todas las personas que me han enseñado algo porque su conocimiento me ha hecho mejor.

1

Introducción

Las siguientes secciones proveen un manual de prácticas y proyectos para un curso introductorio de Compiladores adecuado a la comunidad de la Facultad de Ciencias. El eje sobre el que gira el manual es la construcción de un compilador con enfoque académico.

La construcción de un compilador involucra diversas áreas de las Ciencias de la Computación y demanda gran experiencia en la programación, pero obeciendo a la primicia de introductorio de este manual, se le dará mayor énfasis a la primera parte de la construcción; el análisis del lenguaje fuente. El lenguaje de implementación será *Java* y ensamblador para la arquitectura *mips* sera el lenguaje objetivo.

Para poner en práctica los conceptos teóricos que serán vistos en el curso, es necesario reducir el conjunto de instrucciones de algún lenguaje. Para este trabajo en particular, se eligió considerar un pequeño conjunto de instrucciones del lenguaje de programación *python*. A este conjunto de instrucciones se le llamará *p*. Se escogió *python* por tener una sintaxis sencilla y poco verbosa. Se indicará que un archivo tiene código de este lenguaje con la extensión `p`.

El lenguaje *p* cuenta con las siguientes características:

- Tipado estático.
- Indentación (sangría) significativa.

- Imperativo.
- Estructurado, aunque no comprende funciones.
    - Ciclo *while*.
    - Condicional *if*.

Las categorías léxicas y la gramática pueden revisarse en los apéndices A y B respectivamente. *p* será el lenguaje fuente.

Las arquitecturas modernas de los compiladores son modulares, ya que tienen mayor oportunidad de supervivencia a los cambios tecnológicos y proveen mayor flexibilidad en el soporte de distintas plataformas y distintos lenguajes. Los tres grandes módulos son el *Front-End*, *Back-end* y, para obtener código objetivo que cuente con alto desempeño, se requiere el módulo *Middle-End*. Cada uno de éstos puede pensarse como una pieza de *lego*, las cuales pueden cambiarse, en la misma secuencia *Front-End*, *Middle-End*, *Back-end*, para construir una estructura diferente.

El *Front-End* involucra todos los análisis que se hacen sobre el código fuente para corroborar que la entrada del compilador es un programa válido, según lo estipulado por el lenguaje fuente. En el *Back-End* se produce el código equivalente al programa de entrada expresado en el lenguaje objetivo, que típicamente es ensamblador de alguna máquina virtual o física. El *Middle-End* es opcional, pero un compilador que se preocupe por el desempeño del código objetivo considerará implementarlo. En esta etapa el código fuente se transforma en diversas representaciones intermedias. Cada una expone un conjunto de características y propiedades del código sobre las que se hacen optimizaciones[1].

Para empezar la implementación del compilador se trabaja en el *Front-End*. Toda la teoría referente a él se tiene bien estudiada y automatizada desde hace varias décadas. La mayoría de las prácticas y proyectos que se presentan en este trabajo son acerca del *Front-End*.

[1] Ejemplo de *Middle-End* `https://gcc.gnu.org/wiki/MiddleEnd`.

1.1. Etapas de construcción.

En una variante de implementación de [6] se presenta el orden que se siguió en la construcción del compilador para *p* junto con las prácticas y proyectos propuestos para cada etapa:

Análisis Léxico

Práctica 1:

El comportamiento que se espera de un analizador léxico, al reconocer expresiones regulares, es que siga el principio de *máxima mordida* y que la complejidad del reconocimiento se conserve lineal. En el artículo [4] se expone una modificación a la manera ingenua de implementar el analizador, para que sean garantizadas las características mencionadas. El objetivo principal de esta práctica es que se entienda e implemente la mejora que se propone en [4], así como entender las ventajas y desventajas de las diferentes maneras de implementar un analizador léxico.

Proyecto 1:

Para iniciar la construcción del compilador, se construye el analizador léxico con la ayuda de la herramienta JFlex. Las categorías léxicas que se definieron para *p* son las que se describen en el apéndice A.

Análisis Sintáctico

Práctica 2:

Se exponen las dos maneras más representativas de implementar un analizador sintáctico: descendente y ascendente. El objetivo de la práctica es apreciar la diferencias y características de cada una.

Proyecto 2:

Se provee una gramática, inspirada en la de *python*[2], a partir de la cual se elaborará un analizador sintáctico ascendente con tablas LALR, usando la herramienta generadora de analizadores sintácticos *byaccj*.

Práctica 3:

El analizador sintáctico construido en el proyecto 2 únicamente determina la pertenencia del programa a la gramática definida para el lenguaje. Esta práctica busca integrar al reconocimiento la formación de la representación intermedia, sobre la cual se realizará el análisis dependiente del contexto y la generación del código: el árbol de sintaxis abstracta *AST* (por sus siglas en inglés *Abstract Syntax Tree*). El AST será construido en cada reducción del reconocimiento sintáctico, lo cual significa que al final del reconocimiento sintáctico se tendrá el nodo raíz del AST.

Análisis dependiente del contexto

Proyecto 3:

En esta parte del análisis se validará que toda las variables usadas hayan sido declaradas anteriormente. Para lo cual implementaremos un recorrido sobre el *AST* con el patrón *Visitante* y la estructura que nos dará el contexto de las variables ya declaradas, la tabla de símbolos.

A la declaración de una variable le sumaremos la información del tipo con el que fue creada. Esto nos ayudará con el análisis de tipos. También se definirán los tipos del resultado de los operadores según sus operandos. Por ejemplo: *int + int :int*.

Generación de código

Práctica 4:

Se implementará un nuevo Visitante concreto que recorra el AST y genere el código objetivo correspondiente al programa fuente. La generación de código se hará

[2] `https://docs.python.org/2/reference/grammar.html`

de manera ingenua.

1.2. Herramientas.

Las herramientas que serán utilizadas son:

1. **Python 2.7** [3]

   El subconjunto que se escogió de Python es tan reducido que podría ser casi de cualquier lenguaje tipo *Algol* pero se decidió hacerlo con la versión 2.7 porque en ésta la impresión sigue siendo un enunciado y no una función como en la versión 3, lo que permite hacer pruebas de depuración con el mismo lenguaje.

2. **SPIM**

   Lenguaje Ensamblador para la arquitectura MIPS.

3. **JFlex(1.6)**[4]

   Es un generador de analizadores léxicos implementados en Java.

4. **BYACC/J**[5]

   Es un generador de analizadores sintácticos implementados en Java.

5. **Java (versión >1.7)**

   Será el lenguaje anfitrión.

6. **Git**

   Alojará el repositorio del código.

[3] `https://docs.python.org/2/`
[4] `http://jflex.de/`
[5] `http://byaccj.sourceforge.net/`

2

Práctica 1

2.1. Introducción a los analizadores léxicos.

Motivación

Las secuencias arbitrarias de caracteres en el código no tienen sentido para una computadora. Es porque ello que se necesita agrupar caracteres en lexemas de categorías léxicas definidas por los diseñadores del lenguaje. Este es justamente el trabajo de un analizador léxico.

Implementación de analizadores léxicos

Existen típicamente tres maneras de implementar un analizador léxico; *ad hoc*, *dirigido por el código* y *dirigido por tablas*[2]. Esta última será en la que se pondrá mayor énfasis ya que será usada posteriormente. Todas ellas parten de la idea de reconocer presencias de expresiones regulares en un texto dado.

Ad hoc

Esta implementación está hecha especialmente para un conjunto de expresiones regulares y puede ser tan compleja o sencilla como el programador la decida hacer.

Con mucha pericia esta implementación puede llegar a ser la más eficiente de todas y puede implementar otras funcionalidades adicionales durante su ejecución.

Dirigido por el código

El código refleja el comportamiento de un autómata finito determinista que reconoce el conjunto de todas las expresiones regulares. Esto evita consultas a una tabla de transiciones por cada carácter que se lee, optimizando el desempeño del analizador con respecto al siguiente tipo de implementación.

Dirigido por tablas

Para la implementación de un analizador dirigido por tablas, necesitamos:

1. Un autómata. Este autómata debe ser un modelo reconocedor de todo el conjunto de expresiones regulares que necesitemos identificar. Para hacer eficiente el análisis este autómata debe ser determinista y debe estar minimizado. El autómata se representa a través de su tabla de transiciones.

2. Un motor. Este motor es una pieza de código que dada cualquier secuencia de caracteres y cualquier tabla (de transiciones) pueda obtener caracteres de la entrada y determinar cuál será la siguiente transición.

Si ya hemos podido implementar un analizador léxico para un determinado conjunto de expresiones, construir uno para otro conjunto se resume en obtener el autómata mínimo determinista que sea reconocedor de las expresiones nuevas y dejar que el motor haga su trabajo. De lo anterior observamos que este tipo de implementación nos da gran flexibilidad.

Máxima Mordida

Pensemos en el siguiente escenario:

En el programa hay un identificador llamado `fortuna`, los posibles comportamientos del analizador podrían ser:

1. Reconocer el identificador `fortuna`.

2. Reconocer el identificador `fortuna` y además la palabra reservada `for`.

3. Reconocer la palabra reservada `for` y un identificador `tuna`.

Evidentemente el comportamiento deseado es el primero, independientemente de la implementación. Al reconocimiento de la expresión que abarca el mayor número de caracteres consecutivos se le denomina *máxima mordida*. Implementar este comportamiento de manera ingenua puede tener un costo cuadrático, algo muy elevado si tomamos en cuenta que hay programas con miles de líneas y que ésta es apenas la primera fase en la compilación. El artículo [4] explica los casos en los que el reconocimiento tiene costo cuadrático y propone una modificación al algoritmo ingenuo de reconocimiento dirigido por tablas para que ante cualquier entrada la ejecución sea lineal.

2.2. Jflex

Jflex no es un analizador léxico, sino un generador de analizadores léxicos dirigidos por tablas y con comportamiento de máxima mordida. El código resultante está escrito en *Java*. Su funcionamiento a grandes rasgos es el siguiente:

1. Recibe un conjunto de expresiones regulares.

2. Construye un autómata no determinista que reconozca todas esas expresiones regulares.

3. Construye el autómata determinista.

4. Minimiza el autómata.

5. Regresa el analizador léxico para el conjunto de expresiones regulares dado.

Estructura del archivo

La estructura de la entrada para que *Jflex* pueda generar el analizador léxico consta de tres partes separadas por `%%`.

- **Código del usuario**

  Primera parte. En ésta se incluyen todas la bibliotecas de *Java* de las que haremos uso.

- **Opciones y declaraciones**

  Jflex provee varias opciones que modifican o extienden su funcionamiento. Se mencionarán a continuación las más comunes:

  - `%debug` imprime en la salida estándar información de depuración del reconocimiento léxico.
  - `%standalone` incluye un método `main` en la clase del analizador léxico que permite usarlo como componente independiente.
  - `%class <nombre clase>` especifica el nombre de la clase. Por omisión el nombre es `Yylex`.
  - `%unicode` da soporte a ese conjunto de caracteres.
  - `%line` provee a la clase de una variable llamada `yyline` mediante la cual se puede tener acceso a la línea del código que se está analizando.

  En esta sección, además de las opciones, también se puede escribir:

  - Código *Java* de funciones auxiliares escrito entre **%{** y **}%**
  - Asignaciones de nombres a patrones comunes o complicados para que posteriormente pueda hacerse referencia a ellos mediante un nombre. Ejemplo:

    ```
    DIGITO  =   [0-9]
    ```

    Cuando se requiere hacer uso de la declaración, debe escribirse entre `{` y `}`.

- **Reglas léxicas**

  En esta sección se escriben los patrones (reglas) y las acciones que se requiere llevar a cabo cuándo se reconozca esa regla. El formato es el siguiente:

  ```
  regla {acción}
  ```

  La sintaxis completa para escribir las reglas puede consultarse en la documentación oficial de *Jflex*. Las acciones son instrucciones en *Java* y pueden hacer uso de las variables o funciones provistas por *Jflex* como:

  - `yyline` que ya se mencionó anteriormente.
  - `yytext(void)` que regresa el lexema que empató con la regla correspondiente a esa acción.
  - `yypushback(int num)` que regresa al flujo de entrada `num` caracteres de la cadena que empató con esa regla. `num` no puede ser mayor a la longitud de la cadena completa.

  Todo los caracteres que no empaten con ninguna regla serán imprimidos en la salida estándar.

Ejemplo

A continuación se construirá un analizador léxico que reconozca números enteros y reales.

```
// AL.flex
%%
%public
%standalone
%unicode

PUNTO                   =       \.
CERO                    =        0+
ENTERO                  =       [1-9][0-9]* | 0+
%%
{ENTERO}  { System.out.print("ENTERO("+yytext() + ")");}
{ENTERO}? {PUNTO} {ENTERO} | {ENTERO} {PUNTO} {ENTERO}?
          {System.out.print("REAL(" + yytext() +  ")" );}
.         { }
```

Teniendo como entrada el siguiente archivo:

```
# numeros
123.122.21212
211.ccvvvv
.123
123
texto
```

Después *Jflex* se encargará de crear el analizador léxico.

```
$ jflex AL.flex
```

Después se compila la clase resultante y se prueba con el archivo `numeros`

```
$ javac Yylex.java
$ java Yylex numeros

REAL(123.122)REAL(.21212)
REAL(211.)
REAL(.123)
ENTERO(123)
```

2.3. Ejercicios

Máxima Mordida

Lee el artículo [4] e implementa un analizador léxico dirigido por tablas que reconozca la cadenas formadas por las siguientes expresiones regulares:

{ab, (ab) c}*.

Debe tener desempeño lineal ante todas las entradas y su funcionamiento debe seguir el principio de *mordida máxima*. La implementación deberá estar hecha bajo un paradigma estructurado u orientado a objetos. Si la cadena no está formada con átomos de las expresiones regulares mencionadas, deberá indicar que está mal formada y detendrá su funcionamiento.

La entrada puede ser desde un archivo o de la entrada estándar. A continuación un ejemplo del funcionamiento del analizador:

```
//entrada.txt
abababababcab
abababababbba
```

La ejecución se comportará de la siguiente manera:

```
$ ./atablas entrada.txt
cadena: abababababcab
Ok: [(AB)*C][AB]
cadena: abababababbba
Error: cadena mal formada
```

Uso de Flex

Extiende el ejemplo de *Jflex* para que se genere un analizador léxico para identificadores, enteros, números flotantes y las siguientes palabras reservadas: `for`, `if`, `else` y `print`. También utiliza las opciones brindadas *Jflex* y nombra la clase resultante *Analizador*.

Otros tipos de implementación.

El código del analizador léxico de *gcc* se encuentra aquí[1]. Contesta la siguiente pregunta a partir del código:

1. ¿Qué tipo de implementación es?

[1] `https://github.com/gcc-mirror/gcc/blob/master/libcpp/lex.c`

3

Proyecto 1: Análisis léxico

3.1. Descripción

En este proyecto se construirá el primer componente de un compilador, un analizador léxico. Para este trabajo se utilizará el generador de analizadores léxico *JFlex*. El lenguaje en el que estará escrito el código fuente es una versión minimizada de *python*; *p*. Los átomos presentes en la gramática del lenguaje son:

1. **IDENTIFICADOR**
2. **ENTERO**
3. **REAL**
4. **CADENA**
5. **SALTO**
6. **INDENTA**
7. **DEINDENTA**

Las palabras reservadas, operadores y separadores también tienen que ser reconocidos. Para mayor referencia revisar el archivo *atomos.pdf* (apéndice A).

Veamos un ejemplo de cómo debe comportarse nuestro analizador léxico. Tenemos el siguiente código en *p*:

```
if 9 > 7:
  v_x = -4
  while v_x < 1:
    v_x += 1
    print v_x * 3
else:
  v_x = "Hello␣world"
```

La salida esperada de nuestro analizador es:

```
RESERVADA(if)ENTERO(9)OPERADOR(>)ENTERO(7)SEPARADOR(:)SALTOINDENTA(2)
IDENTIFICADOR(v_x)OPERADOR(=)OPERADOR(-)ENTERO(4)SALTORESERVADA(while)
IDENTIFICADOR(v_x)OPERADOR(<)ENTERO(1)SEPARADOR(:)SALTOINDENTA(4)
IDENTIFICADOR(v_x)OPERADOR(+=)ENTERO(1)SALTORESERVADA(print)
IDENTIFICADOR(v_x)OPERADOR(*)ENTERO(3)SALTODEINDENTA(4)DEINDENTA(2)
RESERVADA(else)SEPARADOR(:)SALTOINDENTA(3)IDENTIFICADOR(v_x)OPERADOR(=)
CADENA(Hello world)SALTODEINDENTA(3)
```

A los ojos del analizador sintáctico, que será el consumidor de los átomos, basta que la salida preserve de manera ordenada las presencias de los mismos. Visualmente de esta otra forma es mejor:

```
RESERVADA(if)ENTERO(9)OPERADOR(>)ENTERO(7)SEPARADOR(:)SALTO
INDENTA(2)IDENTIFICADOR(v_x)OPERADOR(=)OPERADOR(-)ENTERO(4)SALTO
RESERVADA(while)IDENTIFICADOR(v_x)OPERADOR(<)ENTERO(1)SEPARADOR(:)SALTO
INDENTA(4)IDENTIFICADOR(v_x)OPERADOR(+=)ENTERO(1)SALTO
RESERVADA(print)IDENTIFICADOR(v_x)OPERADOR(*)ENTERO(3)SALTO
DEINDENTA(4)
DEINDENTA(2)
RESERVADA(else)SEPARADOR(:)SALTO
INDENTA(3)IDENTIFICADOR(v_x)OPERADOR(=)CADENA(Hello world)SALTO
DEINDENTA(3)
```

Cabe destacar que los espacios y saltos de línea son únicamente para hacer más legible la salida, no son átomos que deba producir el analizador léxico. Sólo para facilitar la lectura, salidas como la anterior con saltos de línea, son las que se esperan para este proyecto. Ahora veremos cómo se comporta bajo el siguiente error léxico:

```
# ejemplo.p
if 0 < 1:
    x = 2 + 9
    print x
 else:
   x = 8
```

Salida:

```
RESERVADA(if)ENTERO(0)OPERADOR(<)ENTERO(1)SEPARADOR(:)SALTO
INDENTA(4)IDENTIFICADOR(x)OPERADOR(=)ENTERO(2)OPERADOR(+)ENTERO(9)SALTO
RESERVADA(print)IDENTIFICADOR(x)SALTO
Error de indentacion, linea 5
```

El error fue producido por qué el bloque del `else`, al tener un nivel de indentación menor a la línea anterior, debería ser alguno de los que ya habían sido creados.

Veamos ahora el siguiente ejemplo:

```
if 3 < 1:
   print 2
      print 7
```

Salida:

```
RESERVADA(if)ENTERO(3)OPERADOR(<)ENTERO(1)SEPARADOR(:)SALTO
INDENTA(4)RESERVADA(print)ENTERO(2)SALTO
INDENTA(8)RESERVADA(print)ENTERO(7)SALTO
DEINDENTA(8)
DEINDENTA(4)
```

Entre las dos instrucciones de `print` no hay una que nos indique la creación de un nuevo bloque, por lo que no es un programa válido en *p*. Sin embargo, el analizador léxico no es el encargado de detectar este tipo de errores. La regla general es que el analizador léxico debe detectar átomos mal formados, mientras que el analizador sintáctico secuencias de átomos mal formadas respecto a la sintaxis del lenguaje.

La salida puede ser devuelta en algún archivo o a la salida estándar, pero en ambos casos debe reportar errores indicando el número de línea.

3.2. Jflex

Vista con detenimiento, la tarea de reconocer los átomos de bloques (**INDENTA** y **DEINDENTA**) no parece poderse solucionar únicamente con expresiones regulares, ya que:

- Los espacios que se encuentran al principio tienen un significado diferente a los que se encuentran entre las palabras del código
- Para saber si una línea pertenece o no a un bloque se debe conocer el nivel de indentación de las líneas previas.

Jflex provee una característica llamada *contexto* que puede ayudar a completar la tarea anterior.

Un contexto está definido por un identificador y el conjunto de reglas que se van a aplicar cuándo éste esté activo. Un contexto se encuentra activo si la variable interna `zzLexicalState` tiene el valor de su identificador.

Para definir un contexto se debe escribir en la segunda sección del archivo:

```
%s[tate] "identificador" [, "identificador", ... ]
```

ó bien,

```
%x[state] "identificador" [, "identificador", ... ]
```

En el primer caso, se definen *contextos inclusivos*, eso quiere que las reglas dentro del contexto y todas aquellas que no estén especificadas para algún contexto están activas.

En el segundo caso, se definen *contextos exclusivos*, eso quiere que únicamente las reglas dentro del contexto están activas.

En ambos casos los contextos se guardan como constantes enteras.

La sintaxis para indicar que una regla pertenece a un determinado contexto es la siguiente:

```
<Contexto>regla {acción}
```

Para indicar que todo un conjunto de reglas pertenecen a un contexto:

```
<Contexto>{
  regla_1 {acción_1}
  regla_2 {acción_2}
  ...
  regla_k {acción_k}
}
```

Las funciones relacionados con contextos son:

- `yybegin(int identificador_contexto)` inicia el contexto con identificador `identificador_contexto`.
- `yystate(void)` devuelve el contexto activo.

El contexto por omisión es `YYINITIAL`.

Ejemplo

Si tenemos el siguiente archivo de entrada para *Jflex*:

```
// Flexer.flex
%%
%public
%standalone
%s CONTEXTO_INCLUSIVO
%x CONTEXTO_EXCLUSIVO
%%
A                       { System.out.println("A sin contexto explícito.");
                          yybegin(CONTEXTO_INCLUSIVO);}
<CONTEXTO_INCLUSIVO>B   { System.out.println("B en CONTEXTO_INCLUSIVO.");
                          yybegin(YYINITIAL);}
C                       { System.out.println("C sin contexto explícito.");
                          yybegin(CONTEXTO_EXCLUSIVO); }
<CONTEXTO_EXCLUSIVO>D   { System.out.println("D en CONTEXTO_EXCLUSIVO.");
                          yybegin(YYINITIAL);}
```

Probando con la cadena :

```
AABCAD
```

El resultado del análisis es:

```
A sin contexto explícito.
A sin contexto explícito.
B en CONTEXTO_INCLUSIVO.
C sin contexto explícito.
A
D en CONTEXTO_EXCLUSIVO.
```

En el renglón en el que se encuentra únicamente la letra `A`, se puede notar que estando en un contexto exclusivo, se ignoran todas las reglas que no estén suscritas a ese contexto y el analizador toma el comportamiento por omisión; devolver el lexema a la salida estándar.

3.3. Proyecto

Escribir un archivo llamado `Flexer.flex` que genere un analizador léxico en una clase llamada `Flexer`. El analizador léxico debe reconocer todos los átomos de *atomos.pdf*, indicar la línea en la que haya átomos mal formados y detener su funcionamiento. Se probará de la siguiente manera:

```
$ jflex Flexer.flex
$ javac Flexer.java
$ java Flexer ejemplo.p
```

3.4. Extra

Detectar un error de sintaxis cuando las primeras líneas tienen indentación diferente de 0. Ejemplo:

```
   print 2
print 2
```

La salida deberá ser:

```
Error de indentación Línea 1.
```

4

Práctica 2

4.1. Introducción a los analizadores sintácticos

El análisis sintáctico consiste en determinar si una sucesión de símbolos (una cadena) de un lenguaje puede ser formada con las reglas de su gramática; es decir, pertenece al lenguaje.

Los lenguajes formales cuentan con modelos reconocedores teóricos, pero sin perder de vista que esos modelos serán implementados como parte de un compilador, necesitan tener buen desempeño.

Podemos garantizar un reconocimiento lineal para lenguajes regulares, pero éstos no cuentan con la expresividad suficiente que demanda un lenguaje de programación. Es por ello que los lenguajes libres del contexto son una buena opción para la implementación de los lenguajes de programación. Como recordaremos, en general los lenguajes libres del contexto tiene reconocimiento de $O(n^3)$ y algunos de ellos son inherentemente ambiguos. Pero si nos limitamos a los lenguajes libres del contexto que no son ambiguos, se puede garantizar un reconocimiento lineal [3, 1].

Tradicionalmente existen dos tipos de analizadores sintácticos [2]:

1. **Descendentes** (*Top-Down*). Los lenguajes libres del contexto que pueden ser reconocidos en tiempo lineal por uno de estos analizadores son identificados

como LL (*Left-to-right Left-most-derivation*). Como su nombre lo indica buscan la derivación empezando por el símbolo inicial.

2. **Ascendentes** (*Bottom-UP*). Los lenguajes libres del contexto que pueden ser reconocidos en tiempo lineal por estos analizadores se les conoce como LR(*Left-to-right Right-most-derivation*) y contienen propiamente a todos los lenguajes *LL*. Dentro de esta clasificación existen otros, por ejemplo *LALR*. Es importante mencionar esta clasificación ya que los lenguajes que son reconocidos por analizadores generados por herramientas como *bison* [1] , *yacc* ó *byaccj* caen en esta clasificación. Entender el funcionamiento de estos analizadores es menos intuitivo, ya que empieza con la cadena completa y por medio de reducciones llega al símbolo inicial, si es que la cadena pertenece al lenguaje.

4.2. Byacc/J

Es un generador de analizadores sintácticos, con la J denotando que los analizadores que genera están implementados en *Java*, a diferencia de sus predecesores (*yacc*, *bison*) que los generaban en el lenguaje C. Hay muy poca documentación pero copia casi idénticamente el funcionamiento y la sintaxis de *yacc*. En general este tipo de herramientas recibe los elementos de la gramática (símbolo inicial, símbolos terminales, símbolos no terminales y producciones) y construye el analizador sintáctico. En la construcción de un compilador, el único componente que trata directamente con el código fuente es el analizador léxico; el analizador sintáctico recibe los átomos reconocidos por el primero.

A continuación se describirá brevemente su instalación, sintaxis, funcionamiento e interacción con el analizador léxico.

Instalación

La siguientes instrucciones son para una instalación típica de *Ubuntu* en una arquitectura *x86*. Para otras plataformas la instalación es muy similar.

[1] Aunque en sus versiones recientes ya trabaja con LR (y no únicamente con LALR) `https://www.gnu.org/software/bison/manual/bison.html#Language-and-Grammar`.

1. Descargar el comprimido que contiene el binario[2].
2. Instalar bibliotecas de compatibilidad con software para arquitectura de 32 bits[3]. En algunos sistemas operativos ya están instaladas.
3. Extraer el binario, `yacc.linux`.
4. Se sugiere que se mueva el binario a alguna ruta en el PATH para que se pueda ejecutar desde cualquier ruta.

```
# cp yacc.linux /usr/local/bin/byaccj
$byaccj
  usage:
   byaccj [-dlrtvj] [-b file_prefix] [-Joption] filename
    where -Joption is one or more of:
     -J
     -Jclass=className
     -Jvalue=valueClassName (avoids automatic value class creation)
     -Jpackage=packageName
     -Jextends=extendName
     -Jimplements=implementsName
     -Jsemantic=semanticType
     -Jnorun
     -Jnoconstruct
     -Jstack=SIZE   (default 500)
     -Jnodebug (omits debugging code for better performance)
     -Jfinal (makes generated class final)
     -Jthrows (declares thrown exceptions for yyparse() method)
```

Sintaxis

El archivo que *byaccj* recibe tiene las siguientes secciones:

- **Declaraciones**

  Entre `%{` y `%}`, si es necesario, se importan bibliotecas; si no se hace se pueden omitir los delimitadores. Los símbolos terminales y no terminales también se describen en esta sección de la siguiente manera:

  ```
  %token "nombre" [ "nombre" ... ] /* Terminales */
  %type "nombre" [ "nombre" ... ]  /* No Terminales */
  ```

[2] `http://byaccj.sourceforge.net/#download`
[3] `http://askubuntu.com/questions/454253/how-to-run-32-bit-app-in-ubuntu-64-bit`

Como se muestra en el ejemplo, los comentarios están permitidos. Internamente a cada uno de esos símbolos se les asigna un objeto de tipo `ParserVal` en el que pueden guardar su valor semántico. Sin importar si durante el reconocimiento se utiliza el valor semántico de los símbolos, siempre es necesario declarar los símbolos terminales, ya que definen la manera en la que ambos analizadores (léxico y sintáctico) se van a comunicar.

- **Acciones**

  Las producciones de la gramática se describen de la siguiente manera:

```
símbolo_no_terminal: <forma_sentencial>        { }
                   | <otra_forma_sentencial>   { }
                   ;
```

  Con forma sentencial debe entenderse una secuencia de símbolos terminales y/o símbolos no terminales. Se pueden escribir tantas formas sentenciales como sea necesario, sólo deben estar separadas por `|`, y la lista debe terminar con `;`. Si se requiere hacer algo más que sólo el reconocimiento sintáctico, puede ponerse código de *java* dentro de una pareja de llaves (`{` y `}`). Este código se encuentra a la derecha de las reglas. Con la directiva `$n` se puede tener acceso al objeto que aloja el valor semántico del `n`–ésimo símbolo de la forma sentencial.

  La descripciones de las gramáticas por lo general se hacen en notación *EBNF*[4] la cual incluye en su sintaxis cerradura de Kleene, cerradura positivas y `[e]` para denotar que la expresión `e` puede estar presente una o cero veces. En este sentido la sintaxis de *byaccj* no es compatible ya que no acepta ese tipo de azúcar sintáctica. La recursión será la opción en *byaccj* que hace uso de *BNF*.

- **Código**

  En la última sección se puede escribir código en *java* que puede ser usado en las acciones de las reglas o para modificar alguna funcionalidad del analizador.

Las secciones están separadas por`%%` .

[4] `https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form`

Funcionamiento e interacción con Jflex

El analizador sintáctico solicitará átomos al analizador léxico cada vez que requiera hacer una operación *shift*. Es decir, solicita átomos sobre demanda. Por lo tanto se requiere sólo una pasada del código fuente para realizar ambos reconocimientos. El analizador léxico y el analizador sintáctico deben interactuar. Para ello deben estar conscientes de la existencia del otro y establecer los términos de la comunicación.

- Para que el analizador léxico sepa de la existencia del analizador sintáctico, se le agregará un atributo y un nuevo constructor que permita la inicialización de ese nuevo atributo.

```
//Flexer.flex Segunda sección
private Parser yyparser;

/* Constructor original */
public Flexer(java.io.Reader in) {
  this.zzReader = in;
}

/* Nuevo constructor */
public Flexer(java.io.Reader r, Parser parser){
  this(r);
  this.yyparser = parser;
}
```

- Para que el analizador sintáctico esté enterado de la existencia del analizador léxico se hace básicamente lo mismo que en el caso anterior:

```
//Parser.y Tercera sección
/* Atributo nuevo */
private Flexer lexer;

/* lexer is created in the constructor */
public Parser(Reader r) {
    lexer = new Flexer(r, this);
}
```

Es importante recalcar que el código puede cambiar de acuerdo a como se nombren las clases de los analizadores (en el ejemplo `Flexer` y `Parser`).

- Para acordar los términos bajo los cuales se hará la comunicación se necesitan agregar las siguientes líneas en `Parser.y`:

```
//Primera sección
//Definición del vocabulario .
%token IDENTIFICADOR ENTERO REAL
%%
...
%%
//Tercera sección
/* Definición del método mediante el cual se le solicita
   al analizador léxico un átomo y el analizador léxico
   trabaja para devolver el siguiente átomo del código. */
private int yylex () {
  int yyl_return = -1;
  try {
    yyl_return = lexer.yylex();
  }
  catch (IOException e) {
    System.err.println("IO␣error␣:"+e);
  }
  return yyl_return;
}
```

Los átomos declarados en la primera sección se ponen como constantes estáticas dentro de la clase `Parser`.

`lexer.yylex()` es la función del analizador léxico que hace el trabajo de análisis. Si se deseara leer los átomos de otra fuente (consola, por ejemplo), el método `Parser.yylex()` es el que debe ser modificado.

```
public class Parser {
 ...
  public final static short IDENTIFICADOR=259;
  public final static short ENTERO=260;
  public final static short REAL=261;
...
}
```

Estas constantes definen la interfaz del analizador sintáctico, es decir, cualquier analizador léxico que desee interactuar con él, debe hacerlo en términos de esos átomos. En nuestro caso:

```
//Flexer.flex Tercera sección
{REAL}                  { return Parser.REAL;}
{ENTERO}                { return Parser.ENTERO;}
{IDENTIFICADOR}         { return Parser.IDENTIFICADOR;}
```

Finalmente para poner a trabajar al analizador sintáctico sobre un archivo determinado:

```
//Parser.y  - Tercera sección
public static void main(String args[]) throws IOException {
  Parser yyparser = new Parser(new FileReader(args[0]));
  yyparser.yyparse();
}
```

4.3. Ejercicios

1. Instalar *byaccj*

2. Implementa un intérprete para cada una de las siguientes dos gramáticas descritas en formato *EBNF*. Utiliza: *byaccj* y *jflex*.

   - Gramática 1

     ```
     E : (E (+|-))* T
     T : (T (*|/))* F
     F : [-] NUMBER
     ```

   - Gramática 2

     ```
     E : T ((+|-) E)*
     T : F ((*|/) T)*
     F : [-] NUMBER
     ```

Las gramáticas implementadas en sintaxis *EBNF* deben conservar la dirección en la que crecen sus cadenas (hacia la derecha o hacia la izquierda). Con intérprete debe entenderse que no basta el reconocimiento sintáctico sino que al final del mismo se debe obtener el valor de la expresión.

Los intérpretes debe tener el siguiente comportamiento ante una expresión aritmética bien construida:

```
// entrada.txt
1 + 2 + 4

$ java interprete entrada.txt
$ [ok]  7
```

Comportamiento ante una expresión aritmética mal construida:

```
// entrada.txt
1 + 2 texto

$ java interprete
$ [ERROR] La expresión aritmética no está bien formada.
```

3. Encuentra una manera de imprimir la pila de reconocimiento cada que se hace una reducción. Pista, revisa el código generado por *byaccj*

    - ¿Qué resultado da la evaluación de la expresión **3 - 2 + 8**? Explica el motivo de los resultados.

Extra:

Modifica las gramáticas para que acepten un número arbitrario de expresiones aritméticas separadas por un salto de línea.

5

Proyecto 2: Análisis sintáctico

5.1. Descripción

El presente proyecto tiene como objetivo construir el analizador sintáctico que verificará que la secuencia de átomos, que reconoce el analizador léxico, tenga sentido bajo la gramática de *p* (Apéndice B).

Las herramientas que utilizaremos para construir el analizador sintáctico son *byaccj* y *jflex*.

Analizador léxico

Actualmente el analizador léxico no devuelve átomos, sólo los imprime. Como parte de la realización de este proyecto deben regresarse los átomos en el formato especificado en el analizador sintáctico y uno a la vez.

Analizador sintáctico

El analizador sintáctico no incluye la generación del *árbol de sintaxis abstracta* pero hay que tener en cuenta su futura existencia a la hora de transformar la gramática de *p*, que está en formato *EBNF*, a la sintaxis de *byaccj* ya que internamente no es lo

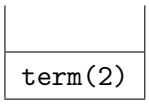
mismo reconocer una gramática con recursión a la derecha que una con recursión a la izquierda, como pudo observarse en la práctica anterior. Veamos un ejemplo:

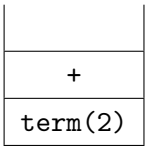
```
/* arith_expr: term (('+'|'-') term)* */

/*    test: (term ('+'|'-'))* term   */
test: term
    | aux8 term
;

/*    aux8: (term ('+'|'-'))+        */
aux8: term MAS  { }
     | term MENOS { }
     | aux8 term MAS  { }
     | aux8 term MENOS  { }
;
```

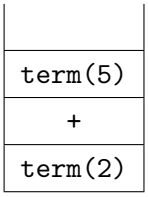
La expansión natural de esa regla sería:

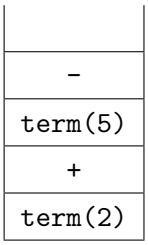
```
/* arith_expr: term (('+'|'-') term)* */
arith_expr: term
          | term aux8
;
aux8: MAS term
    | MENOS term
    | MAS term aux8
    | MENOS term aux8
;
```

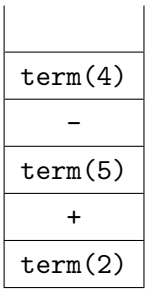
Recordando que el reconocimiento utiliza una pila para almacenar los símbolos que en algún momento va a reducir, el comportamiento, obviando algunos pasos, sería el siguiente ante la cadena `2 + 5 - 4`:



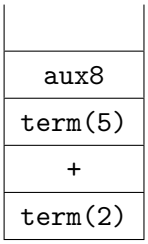


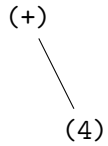






reduciendo con regla (aux8 : MENOS term)





```
reduciendo con regla (aux8 : MAS term aux8)
```


```
reduciendo con regla (arith_expr : term aux8)
```

De lo anterior podemos observar dos inconvenientes:

1. No tan grave pero innecesario, es guardar todos los símbolos en la pila antes de hacer alguna reducción.

2. Lo más grave es que se tiene asociatividad derecha.

Para este proyecto el reconocimiento sintáctico no se ve afectado por la dirección de la recursión, pero es importante tomar en cuenta que en las siguientes etapas sí afectará la dirección de la recursión. Por lo que se sugiere que desde esta etapa se expandan adecuadamente las reglas (con recursividad izquierda y entendiendo el uso de la pila y las reducciones).

5.2. Byacc/J

Para facilitar la realización del proyecto se expondrán las siguientes características de *byaccj*:

- **Depuración**. La clase del analizador sintáctico cuenta con un atributo booleano para cada objeto que cuando es verdadero imprime las acciones que va realizando el analizador sintáctico. Por omisión es falsa.

```
public static void main(String args[]) throws IOException {
    Parser yyparser = new Parser(new FileReader(args[0]));
    yyparser.yydebug = true; //Activamos la depuración
    yyparser.yyparse();
}
```

- **Descripción completa de la gramática y del analizador**. *byaccj* puede generar un archivo, `y.output`, en el que escribe de manera legible la gramática y expone los posibles conflictos que pueda tener la misma. Especifica el tipo de conflicto, el estado y la regla que lo causó.

```
$ byacc -v -J  Parser.y
$ ls
Parser.y   ParserVal.java  Parser.java  y.output
```

5.3. Ejercicios

1. Modificar el analizador léxico para que regrese un único átomo por cada regla empatada.

2. Construir un archivo, que pueda ser recibido por *byaccj* para generar el analizador sintáctico del compilador para *p*. El analizador sintáctico no debe tener ningún conflicto [1] (shift/reduce ó reduce/reduce [1]).

[1] https://www.gnu.org/software/bison/manual/bison.html#Reduce_002fReduce, https://www.gnu.org/software/bison/manual/bison.html#Mysterious-Conflicts

6

Práctica 3: Árbol de sintaxis abstracta

6.1. Introducción

El analizador sintáctico que se tiene ya construido únicamente valida que la cadena (el programa fuente) sea una de las que se pueden generar con la gramática del lenguaje. La presente práctica tiene como objetivo agregarle al reconocimiento sintáctico la generación del árbol de sintaxis abstracta (*AST Abstract Syntax Tree*). Nos basaremos en el patrón Compuesto para darle estructura al AST y en el despacho dinámico que ofrece Java para obtener el comportamiento específico de cada nodo.

Patrón Compuesto

Las partes del patrón compuesto son las siguientes:

1. *Componente*. Será nuestra clase Nodo. Define una interfaz para todos los elementos de la composición (el AST) e implementa un comportamiento por omisión.
2. *Compuesto*. Define el comportamiento específico para los nodos que tienen que manejar hijos.
3. *Hoja*. Representación de los elementos que no tienen hijos y definición del comportamiento específico para estos objetos.

4. *Cliente*. Manipula los objetos de la composición a través de la interfaz Componente.

El patrón compuesto ofrece una jerarquía de clases, estas van desde la clase más general hasta la más particular; entre más alejada de la raíz se encuentre la clase, tendrá un comportamiento más específico. Aunque es flexible para implementar comportamiento específico a cada elemento, ofrece una interfaz al exterior, de tal modo que los objetos de las clases más particulares pueden tener el mismo comportamiento (los mismos métodos) que las otras clases.

Es importante recalcar que el patrón Compuesto tiene muchas otras aplicaciones. Pero en esta práctica fue adaptado al *AST* y que con la palabra interfaz usada en el componente no hace referencia a una interfaz de Java (clase con todos los métodos abstractos, *interface*), sino a una plantilla de métodos que pueden ser aplicados a cualquier elemento de la composición.
Para mayor información del patrón, consultar [5].

Despacho dinámico

Veamos el siguiente ejemplo:

```
class Nodo{
    public String getHijos(){
        return "";
    }
}
class Compuesto extends Nodo{
    public String getHijos(){
        return "Tengo␣n␣hijos";
    }
}
class Hoja extends Nodo{

}

class Prueba{
```

```
    public static void main(){
        Nodo h = new Hoja();
        Nodo c = new Compuesto;
        System.out.println(h.getHijos());
        System.out.println(c.getHijos());
    }
}
```

Podemos observar en el ejemplo que tanto el objeto `c` como el objeto `h` en tiempo de compilación tienen tipo `Nodo`, pero al momento de ejecución se hace el despacho dinámico, es decir, se resuelve el tipo en tiempo de ejecución y se determina que `h` es tipo `Hoja`, pero como no hay una implementación propia del método `getHijos()`, se ejecuta la que se definió por omisión en la interfaz de la composición. En cambio para el nodo `c`, que se creó con el constructor de la clase `Compuesto`, sí hay una implementación específica.

Valores semánticos

En la versión actual del analizador sintáctico no hay manejo de valores semánticos para símbolos terminales o no terminales. Dado que lo que nos interesa crear son nodos e irlos relacionando unos con otros, necesitamos que los símbolos de la gramática tengan asociadas referencias a Nodos. Una manera de hacerlo, provista por *byaccj*, es mediante la opción de compilación *-Jsemantic=<SemanticValue>* que reemplaza el tipo *ParserVal* por lo que esté en *<SemanticValue>*. En nuestro caso, el tipo semántico que nos interesa es `Nodo`. Eso quiere decir que los valores que sean pasados por el analizador léxico en el atributo *yylval* del parser serán de tipo `Nodo`.

```
$ byacc -Jsemantic=Nodo Arith.y
```

Cuando un átomo se forma, el analizador léxico sabe la categoría sintáctica a la que pertenece y el valor asociado a él. El analizador léxico ya regresa el átomo con el nombre de su categoría sintáctica, pero no se ha visto cómo pasar el valor. El analizador sintáctico cuenta con un atributo llamado `yylval` en el que guarda el valor del último átomo solicitado para posteriormente guardarlo en una pila. El analizador

léxico será el encargado de ponerle valor a esa variable que, como mencionamos anteriormente, siempre será de tipo `Nodo`.

```
//Flexer.flex
{ENTERO}  { yyparser.yylval = new IntHoja
                                (Integer.parseInt(yytext()));
            return Parser.ENTERO; }
```

Recursividad izquierda y derecha

En la práctica 2 notamos que la recursividad derecha implica asociatividad derecha y la recursividad izquierda implica asociatividad izquierda, por lo que en el momento de interpretar una expresión aritmética el resultado puede ser diferente en cada uno de los casos. También se observó que la pila crece más con la recursividad derecha porque no se hacen las reducciones hasta el final.

Observemos las siguientes reglas de la gramática en su versión para *byaccj* y con recursividad izquierda y derecha:

Recursividad izquierda:

```
1  stmt : aux0

2  aux0 : simple_stmt
3       | aux0 simple_stmt

4  simple_stmt : small_stmt NEWLINE

5  small_stmt : expr_stmt
6             | print_stmt
```

Recursividad derecha:

```
1  stmt : aux0

2  aux0 : simple_stmt
3       | simple_stmt aux0

4  simple_stmt : small_stmt NEWLINE

5  small_stmt : expr_stmt
6             | print_stmt
```

Pensemos que el archivo fuente es

```
print x
print y
z = 87 + y
print 'cadena'
```

Cada una de esas líneas de código serán eventualmente reducidas a un `small_stmt`, (un nodo para cada una de ellas) para su reducción con el `simple_stmt`; no va a ser necesario la creación de un nuevo nodo, ya que el átomo `NEWLINE` no será integrado al árbol porque no proporciona información necesaria para los siguientes análisis. La meta es que al hacer la reducción de la regla 1, se tenga la referencia de la raíz del árbol. Esta referencia será a un nodo `Compuesto` (o alguna de las clases que hereden de `Compuesto`) con una lista de hijos y en ella vivirán cada uno de los 4 nodos antes mencionados. La diferencia sutil entre la manera en la que se construye el árbol en la gramática con recursividad izquierda y la de derecha radica en el orden en el que se van agregando los hijos. En la recursividad izquierda los hijos se van agregando en el orden que se van encontrando ya que las reducciones se hacen inmediatamente; en cambio en la recursividad derecha, los hijos conforme se van reconociendo se meten a la pila y el último que se reconoció se agrega primero a la lista, entonces si no se agregan con atención a la lista, el orden en el que se escribió el código fuente será el inverso al de las reducciones.

6.2. Ejercicios

1. Construir las clases necesarias para construir el *AST* según el patrón *Compuesto* y las necesidades de la gramática.

2. Integrar al analizador léxico y sintáctico el código necesario para que se pueda construir el *AST* durante el reconocimiento sintáctico y, al fin del mismo, se obtenga la raíz del árbol.

7

Proyecto 3: Análisis dependiente del contexto

7.1. Descripción

Hasta el momento hemos verificado que el código fuente sea válido de acuerdo a la gramática del lenguaje, pero tenemos que recordar que el análisis libre del contexto tiene un enfoque local. Por ejemplo, valida que aparezca un átomo identificador en el lugar adecuado según la gramática, pero no se asegura que la instancia de ese identificador ya haya sido definida anteriormente. Si se quisiera validar la *definición antes del uso* de una variable únicamente con la sintaxis, la gramática no podría ser libre del contexto; se requeriría un nivel mayor de detalle. Pero esto implicaría mayor complejidad del reconocimiento. Por ello se opta por técnicas *ad hoc* para llevar a cabo esta tarea.

La etapa anterior concluyó con la elaboración de una representación intermedia, el *AST*. En abstracto, cada uno de los componentes del árbol es un *Nodo*, pero cada uno pertenece a una subclase con comportamientos y atributos muy particulares.

Sobre esta estructura, el *AST*, realizaremos, con recorridos, el análisis dependiente del contexto y la generación de código. Es claro que cada una de las instancias de las diferentes subclases debe ser analizada según su tipo. La manera ingenua es:

identificar el tipo particular del nodo y operarlo en consecuencia. La implementación correspondiente sería un *switch case* con casos en igualdad de número a las subclases de nodos para cada uno de los recorridos que se realicen.
El patrón Visitor provee una manera modular y transparente de definir las acciones específicas que se requieren realizar con cada uno de los tipos específicos de nodos en cada recorrido realizado sobre el árbol.

7.2. Patrón Visitante

El patrón Visitante será la interfaz encargada de personalizar el análisis dependiente del contexto por cada nodo.

El patrón Visitante está conformado por las siguientes clases:

1. **Visitante Abstracto**: define un método abstracto de visita para cada uno de los elementos concretos.

```
public interface Visitante {
    public void visita(Nodo n);
    public void visita(HojaEntera h);
    ...
    public void visita(StmtNodo s);
}
```

2. **Visitante Concreto**: implementa la interfaz `Visitante` con las acciones que se desean realizar cuando se esté visitando cada elemento concreto.

```
public class VisitantePrint implements Visitante {
    public void visita(Nodo n){
        System.out.print("Nodo␣Genérico");
    }
    public void visita(HojaEntera h){
        System.out.print("Hoja␣Entera");
        System.out.print("valor:␣" + h.valor);
    }
```

```
    ...
    public void visita(StmtNodo s){
        System.out.print("Nodo Sentencia");
          // código para imprimir los hijos que
          // pueda tener.
    }
}
```

3. **Elemento**: Es la clase Nodo. Es decir, el objeto genérico que se visita.

```
public class Nodo{
  ...
  public void acepta(Visitante v){
      v.visita(this);
  }
}
```

4. **Elemento Concreto**: Cada uno de los tipos específicos de nodos. En cada una de las clases deberá haber un método que permita la entrada del `Visitante`.

```
public class HojoEntera{
  ...
  public void acepta(Visitante v){
      v.visita(this);
  }
}
```

Para mayor información del patrón, consultar [5].

7.3. Análisis dependiente del contexto

Lo que se verificará durante este análisis es lo siguiente:

- Que todo identificador sea definido antes de ser usado. En caso contrario, reportar error.

- Que los operadores se utilicen con tipos adecuados, en caso contrario, reportar error. Es decir que las reglas del sistema de tipos sean respetadas.

Durante la creación de algunos nodos hoja, se puede conocer el tipo del mismo. Es decir, si construyo una `HojaEntera` se puede determinar que su tipo es entero. Pero hay nodos que en el momento de su creación no pueden determinar el tipo al que pertenecen, por ejemplo `x+4`, ya que no se conoce si el identificador `x` ya fue definido previamente. La etapa del análisis dependiente del contexto también se encarga de dotar de tipos a los nodos que no son hojas pero que son susceptibles a tener un tipo, como el ejemplo mencionado anteriormente. Lo anterior es posible porque se cuenta con un depósito de información de los identificadores y sus tipos, la tabla de símbolos.

La característica de cambiar el tipo de un identificador no estará permitida, ya que requiere de un nivel de análisis más sofisticado, como un algoritmo de inferencia de tipos. El valor con el que fue creado sí podrá cambiar, pero deberá conservar el tipo.

7.4. Tabla de símbolos

La tabla de símbolos es la estructura que utilizaremos para almacenar el contexto en el que aparece cada identificador y conocer el tipo con el que fue creado.

Las funciones que debe soportar la tabla de símbolos `h` son las siguientes:

1. `LookUp(name)`: regresa el valor asociado a `name` en la tabla de símbolos o `null` en caso de que no haya sido encontrado en la tabla.

2. `Insert(name,info)`: guarda `info` en `h(name)`.

La tabla de símbolos será implementada con tablas hash de la biblioteca `import java.util.Hashtable` de java.

7.5. Sistema de tipos

Un sistema de tipos consiste en la definición de los mismos y las reglas de uso. Los tipos soportados por el lenguaje son :

1. Booleanos
2. Enteros
3. Reales
4. Cadenas

Las reglas de su uso deben definirse para cada operador. Por ejemplo, para el operador suma (+):

	Booleano	**Entero**	**Real**	**Cadena**
Booleano	Booleano	Booleano	Booleano	Booleano
Entero		Entero	Real	Cadena
Real			Real	Cadena
Cadena				Cadena

Sólo se define la mitad de la matriz porque es simétrica.

7.6. Ejercicios

1. Implementar la Tabla de símbolos.
2. Implementar las reglas del sistema de tipos del lenguaje.
3. Implementar el Visitante Abstracto.
4. Implementar el Visitante Concreto que se encargue de hacer el análisis dependiente del contexto sobre el *AST*.

8

Práctica 4: Generación de código

8.1. Descripción

La segunda parte en la tarea de un compilador es la síntesis del lenguaje objetivo. Este proyecto será el responsable de cumplir con esa tarea. Nuestro lenguaje objetivo será ensamblador para la arquitectura MIPS, de la cual las características más importantes son:

- MIPS tiene arquitectura RISC.
- Tiene registros de 32-bits.
- Tiene 32 registros para enteros. Algunos son reservados para el procesador.
- Tiene otros 32 registros de punto flotante.
- La mayoría de sus operaciones son de tres direcciones:
  `add $t0, $t1, $2` se traduce a `$t0 := $t1 + $t2`.

La generación del código se realizará de manera ingenua, es decir se aplica ningún tipo de optimización. Los registros sólo preservarán las datos mientras se esté operando con ellos. Si se necesita preservar más información se hará en memoria.

Para implementar la síntesis se creará un nuevo `Visitante Concreto`. Para facilitar la asignación de registros se sugiere crear una clase que abstraiga el manejo de los registros `Registros`.

```
public class Registros{

  int objetivoEntero;
  int objetivoFlotante;
  // Todos los registros enteros disponibles
  String[] E_registros = {"$t0", ... ,"$t9"};
  // Todos los registros flotantes disponibles
  String[] F_registros = {"$f4", ... ,"$f10"};

  public void setObjetivo(int o, boolean entero){
      if(entero){
          objetivoEntero = o % E_registros.length;
      }else{
          objetivoFlotante = o % F_registros.length;
      }
  }

  public void setObjetivo(String o, boolean entero){
      String[] registros;

      if(entero){
         registros = E_registros;
      }else{
         registros = F_registros;
      }
      int nvo_objetivo = Arrays.asList(registros)
                           .indexOf(o);
      setObjetivo(nvo_objetivo, entero);
  }

  public int getObjetivo(boolean entero){
      if(entero){
          return objetivoEntero;
      }else{
          return objetivoFlotante;
      }
  }
```

```
    /* Regresa los n registros siguientes "disponibles" */
    public String[] getNsiguientes(int n,
                                   boolean enteros){
        String[] siguientes = new String[n];
        int objetivo;
        String[] registros;

        if(entero){
            objetivo = objetivoEntero;
            registros = E_registros;
        }else{
            objetivo = objetivoFlotante;
            registros = F_registros;
        }

        for(int i = 0; i < n; i++){
            siguientes[i] = registros[(objetivo + i)
                                % E_registros.length];
        }
        return siguientes;
    }
}
```

Esta clase provee básicamente las siguientes funciones:

- `getObjetivo(boolean entero)`: Recupera el registro en el que se espera que sea guardado el resultado de la operación principal.

- `getNSiguientes(int s, boolean entero)`: Recupera los `#\s#` registros siguientes que pueden ser usados de manera auxiliar para llevar a cabo los cálculos.

El código en la siguiente página es un ejemplo de implementación para la generación de la operación de la resta.

```
public class VisitanteGenerador implements Visitante {
  Registros reg = new Registros();

  public void visit(DifNodo n){
      Nodo hi = n.getPrimerHijo();
      Nodo hd = n.getUltimoHijo();

      // Tipo de registro objetivo
      int tipo = n.getType().type;
      boolean entero =  n==2 ? false : true;

      String objetivo = reg.getObjetivo(entero);
      String[] siguientes = reg.getNsiguientes(2,entero);

      // Genero el código del subárbol izquierdo
      reg.setObjetivo(siguiente[0],entero);
      hi.accept(this);

      // Genero el código del subárbol derecho
      reg.setObjetivo(siguiente[1], entero);
      hd.accept(this);

      String opcdode =  n==2 ? "sub.s" : "sub";

      System.out.println(opcode + "␣" + objetivo + ",␣" +
                         siguientes[0] + ",␣"
                         + siguientes[1]);
  }
}
```

8.2. Ejercicios

1. Instalar QtSpim Version 9.1 o mayor, un simulador de MIPS.
2. Programar un nuevo Visitante Concreto que recorra el AST y genere el código correspondiente en ensamblador MIPS.

Punto extra

1. Crear un binario con nombre `<cmd>` para linux que, colocado en path, ejecute el código de su compilador y regrese un archivo con extensión `.asm`.

   Ejemplo:

   ```
   #\$# <cmd> <archivo>.py
   ```

   ```
   #\$# ls <archivo>.asm
   ```

Apéndice A

Átomos

A.1. Átomos.

Para definir las categorías sintácticas de *p* (subconjunto de Python) se va a usar, como ya dijimos, *EBNF*.

Las categorías léxicas son:[1]

```
Identificador : ([a-zA-Z] |'_' ) ([a-zA-Z] | [0-9] | '_')*
Booleano : 'True' | 'False'
Entero : [1-9][0-9]* | 0+
Real : '.'[0-9]+ | Entero'.'[0-9]+ | Entero '.'
Cadena : '"'<cualquier carácter, excepto '\' o '"'>*'"'
Palabra reservada : 'and' | 'or' | 'not' | 'for' |
                    'while' | 'if' | 'else' | 'print'
Operador : '+' | '-' | '*' | '**' | '/' | '//' | '\%' |
           '<' | '>' | '>=' | '<=' | '==' | '!='
Separador : ':' | '(' | ')'
```

En *p* la indentación es significativa, lo que quiere decir que también debemos tener átomos que distingan los niveles de indentación. El nivel de indentación está dado por el número de espacios y tabuladores al inicio de línea. Tomaremos cada tabulador como un número constante de espacios, 4 (el valor de los tabuladores en

[1] https://docs.python.org/2/reference/lexical_analysis.html#line-structure

python es variable; no seguiremos esa convención). A diferencia de los átomos de las categorías léxicas anteriores, éstos no dependen únicamente de empatar patrones, sino también del contexto en el que fueron reconocidos. A continuación los detalles de cada uno:

- INDENTA: El nivel de indentación de cada línea determinará el bloque al que pertenece.

  Se emitirá uno de estos átomos cada vez que se reconozca el nivel de indentación y éste sea mayor que el de la línea anterior. Eso quiere decir que se ha empezado un nuevo bloque de código.

- DEINDENTA: Se emitirán estos átomos cada vez que se reconozca el nivel de indentación actual y sea menor que los anteriores. Por cada nivel anterior y mayor al actual, se emitirá un DEINDENTA. También se emitirá un DEINDENTA por cada nivel que haya entre el del 0 y el de la última línea de código.

- SALTO: Indica el final de una línea (dependiente del sistema operativo; en Linux es "\n"). Las líneas que solo contengan espacios en blanco serán ignoradas. Es decir no generarán átomos SALTO.

De lo anterior puede deducirse que si el nivel de indentación de una línea es igual al de su predecesora, entonces no será emitido ningún átomo de indentación, ya que pertenecen al mismo bloque.

A.2. Comentarios

A pesar de que los comentarios no representan átomos de la gramática, indicaremos cómo deben ser manejados. Los comentarios inician con un # y terminan con un salto de línea. Puede haber comentarios después de código en la misma línea. Esto significaría el fin de la línea. Se ignoran las líneas que empiezan con comentario.

Apéndice B

Gramática

B.1. Gramática

Esta gramática es el resultado de la modificación de la gramática oficial del lenguaje *python* 2.7 [1]. Está especificada en notación *EBNF* por lo que no es aceptada tal cual por generadores de analizadores sintácticos, ya que ellos hacen uso de *BNF*.

```
file_input: (SALTO | stmt)* ENDMARKER

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt SALTO
small_stmt: expr_stmt | print_stmt
expr_stmt: test '=' test]
print_stmt: 'print' test

compound_stmt: if_stmt | while_stmt
if_stmt: 'if' test ':' suite ['else' ':' suite]
while_stmt: 'while' test ':' suite
suite: simple_stmt | SALTO INDENTA stmt+ DEINDENTA

test: or_test
or_test: and_test ('or' and_test)*
```

[1] https://docs.python.org/2.7/reference/grammar.html

```
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'!='
expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-') factor | power
power: atom ['**' factor]
atom: IDENTIFICADOR | ENTERO | CADENA | REAL | BOOLEANO | '(' test ')'
```

Apéndice C

Tipos

C.1. Tipos

El lenguaje *p* contempla los siguientes tipos:

- Booleano
- Entero
- Real
- Cadena

C.2. Operadores

A continuación se define el sistema de tipos para el lenguaje y sus operadores. La definición se dará en una matriz tringular superior, ya que la matriz completa es simétrica.

And

	Entero	**Real**	**Booleano**	**Cadena**
Entero	No Definido	No Definido	No Definido	No Definido
Real		No Definido	No Definido	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

Diferencia

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Entero	Real	Entero	No Definido
Real		Real	Real	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

División

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Entero	Real	No Definido	No Definido
Real		Real	No Definido	No Definido
Booleano			No Definido	No Definido
Cadena				No Definido

División Entero

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Entero	Entero	No Definido	No Definido
Real		Entero	No Definido	No Definido
Booleano			No Definido	No Definido
Cadena				No Definido

Mayor que

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Booleano	Booleano	Booleano	No Definido
Real		Booleano	Booleano	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

Mayor igual que

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Booleano	Booleano	Booleano	No Definido
Real		Booleano	Booleano	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

Menor que

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Booleano	Booleano	Booleano	No Definido
Real		Booleano	Booleano	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

Menor igual que

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Booleano	Booleano	Booleano	No Definido
Real		Booleano	Booleano	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

Módulo

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Entero	No Definido	No Definido	No Definido
Real		No Definido	No Definido	No Definido
Booleano			No Definido	No Definido
Cadena				No Definido

Or

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Booleano	Booleano	Booleano	No Definido
Real		Booleano	Booleano	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

Potencia

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Entero	No Definido	No Definido	No Definido
Real		No Definido	No Definido	No Definido
Booleano			No Definido	No Definido
Cadena				No Definido

Suma

	Entero	**Real**	**Booleano**	**Cadena**
Entero	Entero	Real	Entero	No Definido
Real		Real	Real	No Definido
Booleano			Booleano	No Definido
Cadena				No Definido

9

Epílogo

9.1. Conclusiones

Llevar a cabo un proyecto como el trabajo presente es formativo, ya que es un proyecto de dimensiones un poco mayor a las estándar, se construye de manera modular y requiere del uso de buenas prácticas de programación e ingenio para ser llevado a cabo.

Por otro lado, el trabajo incursiona en el área de compiladores de manera un tanto tímida, ya que si bien ataca ampliamente el análisis sobre el lenguaje fuente, la parte de síntesis es elaborada de manera ingenua y sin ningún tipo de optimización.

Es precisamente ahí donde se encuentran las áreas de oportunidad de este trabajo:

1. Incurrir en la generación de código para procedimientos.

2. Después del análisis dependiente del contexto realizado sobre el *AST* podría generarse otra representación intermedia con menor grado de abstracción, para que la generación de código sea susceptible a alguna optimización.

3. Cambiar el lenguaje objetivo de ensamblador para *MIPS* a ensamblador para *x86*, lo que haría posible no usar un simulador.

En cuanto a mejoras al trabajo en la parte técnica he pensado en las siguientes:

1. Automatizar el proceso de evaluar prácticas. Es decir incluir pruebas unitarias para que el alumno sepa si su trabajo está funcionando como se espera y la persona que tiene que revisar las prácticas tenga que invertir menos tiempo en la revisión de las mismas.

2. Utilizar una herramienta como Vagrant o Docker para automatizar la instalación de todas las herramientas que se utilizan en la construcción del compilador.

En un futuro me agradaría que los cursos de compiladores incluyeran a los lenguajes que no son tipo *Algol*, como los funcionales, ya que tiene particularidades muy interesantes.

Otro enfoque interesante y enriquecedor de llevar un laboratorio de compiladores sería tomar los compiladores usados actualmente y de código abierto e ir analizando cada una de los módulos para que al final del análisis se solicite implementar alguna modificación pequeña.

Este enfoque requiere una fuerte inversión de tiempo para estudiar la complejidad técnica y teórica de estos compiladores, pero creo que sería una manera de aterrizar mejor los conocimientos del amplio programa de Compiladores ya que se tendría acceso a los detalles de implementación de los mismos.

Comparando los cursos de compiladores de universidades extranjeras con los de las nacionales pude observar que tratan temas (en la teoría y en la práctica) muchos más avanzados que en México. Pero esto es debido a que en el extranjero hay grupos de personas dedicadas al estudio e investigación en el área, lo cual sería deseable que pasara en México también. Asimismo, en muchas de estas universidades los estudiantes llegan a la licenciatura con una amplia experiencia en programación.

9.2. Distribución del código

Con el fin de contar con mayor experiencia en la redacción de los ejercicios y el nivel de dificultad de cada uno de ellos, realicé su implementación.

Pero se decidió que el código no será publicado ya que, en caso de que el presente trabajo sea empleado en algún curso, es mejor que los alumnos desarrollen sus propias ideas y sea un reto provechoso para ellos.

El código base que se les distribuye a los alumnos para orientarlos en la implementación puede ser consultado en `https://github.com/dianamontes1516/Compiladores`.

Bibliografía

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[2] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[3] Elisa Viso Gurovich. *Introduccion a la Teoria de la Computacion*. UNAM, 2008.

[4] Thomas Reps. "maximal-munch" tokenization in linear time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(2):259–273, 1998.

[5] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[6] Ángel Francisco Zúñiga Chávez. *Diseño e implementación de un compilador para un subcojunto de python*. Tesis de Licenciatura. UNAM, FC, 2013.