Universidad Nacional Autónoma de México

Facultad de Estudios Superiores Acatlán

Puma: Una herramienta de apoyo en la enseñanza de la programación

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA:
MARCOS DAVID MARÍN AMADOR

ASESOR:
ING. EFRÉN PABLO HECTOR GONZÁLEZ VIDEGARAY

Junio de 2010

Puma: Una herramienta de apoyo en la enseñanza de la programación

Marcos David Marín Amador
marcosmarin@gmail.com

8 de junio de 2010

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

RESUMEN

En esta tesis se describe el desarrollo de un programa llamado Puma que tiene como objetivo facilitar el aprendizaje de conceptos básicos de la programación.

El programa cuenta con un lenguaje imperativo con palabras reservadas en Español el cual es compilado a un lenguaje intermedio inspirado en el lenguaje ensamblador que es ejecutado por un intérprete en un ambiente virtual.

También cuenta con una interfaz gráfica donde el usuario puede desarrollar algoritmos sencillos y ejecutarlos. Al ejecutarlos la interfaz muestra el ambiente virtual donde se está procesando para que el usuario pueda visualmente comprender como funciona un lenguaje de programación imperativo. Es decir, atestiguar todo el proceso empezando con la traducción del algoritmo escrito en un lenguaje de alto nivel a un lenguaje de bajo nivel y como éste se ejecuta.

También se muestra el programa a un grupo de alumnos de un curso de programación para conocer su opinión respecto a Puma.

INTRODUCCIÓN

La hipótesis de la cual parte esta investigación es que basándose en la teoría de autómatas, compiladores y programación orientada a objetos, es factible construir una herramienta en la cual el usuario pueda programar algoritmos de sintaxis en Español y que permita verificar el comportamiento interno de la máquina al procesar el algoritmo.

Para esto, se desarrollará una aplicación llamada Puma. Puma contará con un lenguaje inspirado en C pero con palabras reservadas en Español y una sintaxis reducida para facilitar su aprendizaje. El usuario desarrollará algoritmos sencillos en este lenguaje y los podrá ejecutar línea por línea y al mismo tiempo ver un diagrama de la memoria utilizada para ver como es afectada.

El primer capítulo del trabajo es un resumen de diversas investigaciones realizadas sobre los problemas que tienen los alumnos al aprender a programar y cómo Puma pretende atacar dichos problemas.

El segundo capítulo son los antecedentes teóricos necesarios para la construcción del sistema Puma, principalmente habla sobre lo necesario para construir un compilador: teoría de autómatas, gramáticas libres de contexto, técnicas para construir el árbol de sintaxis del código fuente de un programa y finalmente discute algunas de las herramientas que facilitan la construcción de un compilador.

El tercer capítulo es el que describe el diseño y desarrollo del sistema Puma.

Para el cuarto capítulo Puma fue expuesto a un grupo de alumnos que sentían tener dificultades con el aprendizaje de la programación que tomaban un curso intersemestral de C y se les aplicó un cuestionario sobre su opinión tanto de las ideas detrás de Puma como qué tan bien Puma las llevaba a cabo.

El quinto capítulo son las conclusiones del autor y el trabajo que quedaría para el futuro de Puma.

1. DIFICULTADES EN EL APRENDIZAJE DE LENGUAJES DE PROGRAMACIÓN

Aprender a programar es difícil, varios estudios en los últimos años se han hecho para identificar qué aspectos de la programación la hacen complicada de aprender[1, 2], pero predominan los que se enfocan en buscar nuevas formas para introducir a alumnos a la programación y ciencias de la computación en general que sean más efectivas y menos costosas[3, 4, 5, 6, 7]. Muchos de estos estudios se enfocan en aprovechar tecnologías relativamente nuevas como el internet y cómputo móvil en la enseñanza. En esta sección haremos una revisión de los resultados de estos estudios y concluiremos con los aspectos de Puma que podrían ayudar.

En su estudio del uso de teléfonos celulares en la enseñanza, Chrisina Draganova[3] encontró que la mayor ventaja de estos sistemas era la inclusión de los alumnos al participar activamente en sesiones de preguntas y respuestas y la posibilidad de recibir retroalimentación casi instantánea.

En un panel de discusión para encontrar una mejor forma de dar la primer clase de programación[4], Joe Bergin propuso el uso de Karel J Robot[8] como alternativa por ser similar a Java con una sintaxis reducida pero que seguía siendo un lenguaje Turing-completo, esto simplifica el aprendizaje del alumno. En Puma hemos hecho una sintaxis reducida con pocas palabras clave que además están en Español.

En este mismo panel de discusión, Raymond Lister argumenta el uso de PigWorld[9], un pequeño mundo virtual donde dos puerquitos buscan alimentos y evitan depredadores. Este mundo virtual introduce a los alumnos a algoritmos sencillos como la ruta más corta de Dijkstra y sobre todo da a los alumnos la posibilidad de hacer pequeños cambios en el mundo virtual y ver los resultados inmediatos en la pantalla, alentando así a los alumnos a jugar con el sistema para ver cómo funciona. La interfaz de Puma contiene en la misma pantalla el código fuente de la aplicación, la salida de la aplicación y el diagrama de memoria para que los alumnos puedan hacer cambios en el código, ejecutarlo y ver qué cambió en la ejecución inmediatamente sin tener que hacer uso de múltiples herramientas como un editor, compilador y debugger.

Al rediseñar cursos de introducción a la programación, Michael Clancy et al.[5] proponen actividades dinámicas (como programación) que promuevan la

reflexión y colaboración de los alumnos además de más oportunidades para el profesor para interactuar con los alumnos e identificar los problemas que están teniendo de forma oportuna.

En un cuestionario aplicado a más de 500 estudiantes de distintos países realizado por Essi Lahtinen et al.[1] se encontró que los alumnos opinaron que los conceptos de programación más difíciles de aprender son apuntadores y referencias. Por otro lado Iain Milne y Glenn Rowe[2] encontraron que los tópicos más complicados de aprender son los relacionados con apuntadores y manejo de memoria como constructores copia y funciones virtuales, y concluyeron que es por causa de la inhabilidad de los estudiantes de comprender lo que está sucediendo en la memoria de su programa, ya que no forman un modelo mental claro de la ejecución del programa.

Debido a esto, la visualización del estado de la memoria en cualquier momento de ejecución del programa es uno de los enfoques principales de Puma, aunque no ataque problemas más complicados como los mencionados anteriormente de programación orientada a objetos, como son los constructores copia y funciones virtuales. A pesar de esto, es nuestra suposición que el entendimiento de cómo funciona la memoria a un nivel básico ayudará al alumno a comprender estas situaciones más avanzadas.

Además de la visualización de la memoria, Puma ayuda al alumno a hacer un modelo mental de la ejecución del programa a través de la posibilidad de ejecutar un programa línea por línea viendo cómo cambia el control de flujo de una línea a otra y los cambios de valores en la memoria. Como característica adicional, este modo de ejecución, que es similar al de una herramienta de depuración ayudará al alumno a identificar errores en su código que pueda corregir y aprender una lección a partir del error.

2. ANTECEDENTES

Puma es en esencia dos cosas, un compilador que traduce un programa escrito en un lenguaje de alto nivel a instrucciones de bajo nivel, y un intérprete que ejecuta las instrucciones de bajo nivel en secuencia para realizar la ejecución del programa escrito por el usuario.

De los dos componentes el más complejo es el compilador. Para entender cómo se construyó daremos en la siguientes secciones una breve introducción a la teoría necesaria.

La teoría discutida en este capítulo proviene principalmente de [12] y [13].

2.1. Compiladores

Un compilador es básicamente un traductor, normalmente cuando hablamos de un compilador pensamos en la traducción de un lenguaje de alto nivel como C++ a código objeto. Sin embargo la entrada no tiene que ser estrictamente un lenguaje de programación y la salida no necesariamente tiene que ser código objeto. Podemos, por ejemplo, tener un compilador que tome como entrada una serie de notas musicales y produzca un archivo mp3 como salida.

Sin embargo, en el caso de Puma tenemos un compilador que traduce de un lenguaje de alto nivel a una serie de instrucciones de bajo nivel. Cada una de estas instrucciones estará formada de una operación, uno o dos operandos y posiblemente un resultado. Estas instrucciones de bajo nivel posteriormente serán ejecutadas por un intérprete en secuencia. Así, aunque el compilador no generará estrictamente código objeto para ser ejecutado directamente por el sistema operativo, generaremos algo muy similar pero más simple.

Las acciones del compilador se pueden dividir en dos, el análisis y la síntesis. Durante el análisis el compilador descompone el código fuente de alto nivel y encuentra su estructura sintáctica, que es el significado del código. Con ésta y más información que guarda en una tabla, llamada la tabla de símbolos, posteriormente durante la síntesis genera la secuencia de instrucciones de bajo nivel.

Estas dos etapas del compilador son realizadas por distintos módulos que actúan en forma consecutiva, cada uno tomando la salida del anterior (o el códi-

go fuente intacto en el caso del primero) y generando una salida. En el caso de Puma tenemos únicamente los más básicos: el analizador léxico, que toma el código fuente que es una secuencia de caracteres y genera una secuencia de tokens, el analizador sintáctico que toma la secuencia de tokens y genera un árbol de sintaxis que representa el significado del código fuente. Un token es un objeto atómico en el código fuente, por ejemplo, un número o una palabra reservada. Por el tipo de analizador sintáctico que usa Puma, los nodos del árbol de sintaxis son generados en el mismo orden que si el árbol completo fuera recorrido en postorden. Debido a esto aprovechamos la generación del árbol para al mismo tiempo producir la secuencia de instrucciones de bajo nivel, haciendo así el proceso de la síntesis.

2.2. Analizador léxico

El primer paso en la traducción de un lenguaje de alto nivel a uno de bajo nivel es el análisis de la secuencia de símbolos pertenecientes a un alfabeto para identificar cadenas con significado. En el caso de Puma el alfabeto es un subconjunto de los símbolos en la tabla ASCII: las letras mayúsculas, minúsculas, los dígitos y algunos signos de puntuación. Las cadenas con significado que buscaremos identificar son las palabras reservadas, los nombres de variables y funciones, números enteros y reales, entre otras.

El Analizador léxico o lexer es quien se encarga de la identificación de estas cadenas y lo hace usando autómatas finitos deterministas. Un autómata finito determinista normalmente se define como una quíntupla $A = (Q, \Sigma, \delta, q_0, F)$ donde

Q Es un conjunto finito de *estados*.

Σ Es el alfabeto que son los *símbolos de entrada*.

δ Es una *función de transición* que determina el estado al que el autómata pasa dependiendo del estado actual y el siguiente símbolo de entrada.

q_0 Es un *estado inicial* perteneciente a Q.

F Es un subconjunto de Q formado por los *estados finales*, es decir, los estados en los que puede terminar el autómata.

El hecho de que sea determinista significa que en cada momento el autómata se encuentra en un solo estado y dado el siguiente símbolo de entrada hay un solo estado al que puede pasar. Todos los autómatas finitos no deterministas tienen uno o más autómatas finitos deterministas que son equivalentes.

Por simplicidad podemos representar un autómata finito determinista usando un grafo dirigido donde cada estado es un nodo y los nodos son conectados

por arcos etiquetados por un símbolo del alfabeto que significa que $\delta(n_i, a) = n_j$ si existe un arco del nodo i al nodo j etiquetado por el símbolo a. El nodo inicial q_0 es representado por una flecha que no sale de ningún nodo y apunta a q_0 y los estados pertenecientes a F son aquellos cuyo nodo tiene doble círculo.

Como ejemplo en la Figura 2.1 mostramos un grafo dirigido que representa un autómata finito determinista para reconocer la palabra reservada `haz`. Como se aprecia en el diagrama comenzamos con la cadena vacía, al encontrar una `h` pasamos al estado `h`, al encontrar una `a` pasamos al estado `ha` y finalmente al encontrar una `z` terminamos en el estado final `haz`.

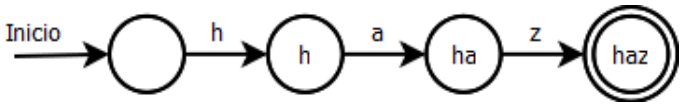


Fig. 2.1: Autómata finito determinista para reconocer la palabra reservada *haz*.

Para el compilador de Puma no crearemos manualmente todos los autómatas para reconocer las cadenas importantes del lenguaje, en vez de ello especificaremos *expresiones regulares* a partir de las cuales una herramienta llamada gplex[19] generará en forma de código en C# los autómatas finitos para el reconocimiento de cadenas dentro de la secuencia de símbolos ASCII, que es el código fuente. La sección 2.5 en la página 10 hablará más sobre esta herramienta.

Una *expresión regular* es una forma de especificar un conjunto de cadenas que pertenecen a un lenguaje donde un lenguaje es un subconjunto de las cadenas formadas por la concatenación de los símbolos de un alfabeto. Por ejemplo la expresión regular que usaremos para definir todas las cadenas que son números reales es

$$[0-9]+\backslash.[0-9]+((e|E)(\backslash+|-)?[0-9]+)?$$

Antes que nada hay que mencionar que el símbolo \ sirve para escapar caracteres con un significado especial en las expresiones regulares como el signo de suma `+` y el punto `.`. Es decir, quitarles su significado especial y tratarlos como cualquier caracter. Ahora bien, $[0-9]$ representa cualquier caracter entre 0 y 9 (cualquier dígito) y el $+$ que sigue significa que se puede repetir una o más veces. Esta secuencia de dígitos es seguida por un punto a su vez seguido por uno o más dígitos. A continuación usamos paréntesis para formar un grupo que termina con un símbolo de interrogación ? el cual significa que es opcional, es decir, el grupo puede aparecer cero o una vez. Este grupo está formado por una e o una E, seguida de un + o - opcional que finalmente es seguido por uno o más dígitos.

A partir de esta expresión regular se generará un autómata finito determinista el cual reconocerá cadenas como las siguientes

- 1.0
- 23.98
- 1.4e48
- 1.4E48
- 1.3E+67
- 3.432e-654

Es decir números reales positivos, pero no números enteros ya que el . seguido de dígitos después de los primeros dígitos no es opcional. Cabe notar que no reconocerá números como .5 ya que se usó un + en vez de un $*$ en el primer grupo.

Al reconocer una cadena, el lexer le pasará al analizador sintáctico, o parser, un token. Este token es un par que consiste del nombre del token y un atributo opcional que puede ser un índice que hace referencia en la tabla de símbolos al símbolo correspondiente a la cadena que se reconoció. Esta tabla de símbolos contendrá información relevante a cada símbolo como el tipo en el caso de una variable.

2.3. Analizador sintáctico

El analizador sintáctico toma como entrada una secuencia de tokens generada por el lexer y los analiza para construir el árbol de sintaxis, que representa el significado de la serie de tokens.

Para construir el árbol de sintaxis primero hay que conocer la gramática del lenguaje. Existen varios tipos de gramáticas formales que han sido clasificadas en una jerarquía por Noam Chomsky[10]. En esta jerarquía existen cuatro niveles donde cada nivel es una clase de gramáticas que generan un subconjunto de los lenguajes generados por las gramáticas del nivel anterior.

Tipo 0 No tienen restricciones, incluye todas las gramáticas formales. Estas gramáticas generan todos los lenguajes que pueden ser reconocidos por una máquina de Turing.

Tipo 1 Son las gramáticas sensibles al contexto. Las reglas de producción de estas gramáticas tienen la forma $\alpha A\beta \rightarrow \alpha\gamma\beta$ donde A es un símbolo no terminal y α, β y γ son cadenas de símbolos terminales y no terminales. α y β pueden ser cadenas vacías pero γ no. El símbolo no terminal inicial puede ser vacío si y sólo si no aparece en el cuerpo de ninguna producción.

Tipo 2 Son las gramáticas libres de contexto. Todas las reglas de producción tienen un sólo símbolo no terminal en la cabeza y una cadena de símbolos terminales y no terminales en el cuerpo. Casi todos los lenguajes de programación, incluyendo el de Puma, usan una gramática de este tipo.

Tipo 3 Son gramáticas regulares. Los lenguajes generados por estas gramáticas pueden ser descritos también por expresiones regulares. Únicamente pueden tener producciones con un símbolo no terminal en la cabeza y un símbolo terminal en el cuerpo seguido opcionalmente de un símbolo no terminal. Es decir producciones del tipo $A \rightarrow \alpha$ y $A \rightarrow \alpha B$ si A y B son símbolos no terminales y α sí lo es. El símbolo no terminal inicial puede ser la cadena vacía si y sólo si no aparece en el cuerpo de ninguna producción.

La gramática del lenguaje de Puma la especificaremos en forma Backus-Naur Form (BNF)[11]. BNF es una metasintaxis para expresar una gramática libre de contexto usando símbolos terminales y no terminales. En la Figura 2.2 vemos un ejemplo de una gramática libre de contexto que reconoce expresiones aritméticas.

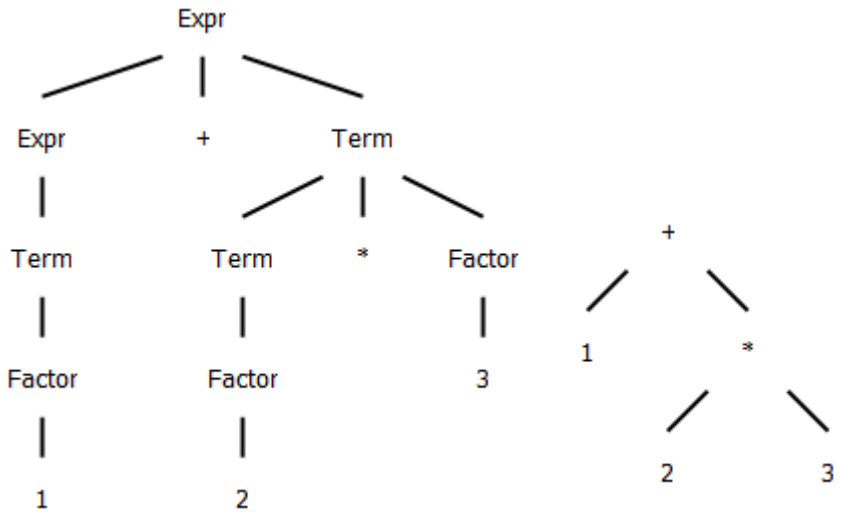
Expr	→	Expr + Term
	|	Expr − Term
	|	Term
Term	→	Term / Factor
	|	Term ∗ Factor
	|	Factor
Factor	→	**num** | **(** Expr **)**

Fig. 2.2: Gramática libre de contexto en forma BNF.

Expr, Term y Factor son símbolos no terminales mientras que $+$, $-$, $/$, $*$, (,) y **num** son los símbolos terminales, van a ser los tokens regresados por el lexer. A la izquierda de la flecha en cada regla de producción tenemos el símbolo no terminal conocido como la cabeza. En la parte derecha tenemos una cadena de símbolos terminales y no terminales conocidos como el cuerpo de la producción. En este caso hemos agrupado las producciones con el mismo símbolo no terminal en la cabeza separándolas con un |, que se lee como "o".

Si quisiéramos, por ejemplo, hacer el análisis sintáctico de $1+2*3$ crearíamos el árbol mostrado en la Figura 2.3.

En la Figura 2.3(a) se muestra el árbol de sintaxis concreto mientras que en 2.3(b) se muestra el abstracto. Empezaremos explicando el árbol de sintaxis concreto. Lo primero a notar es que todos los nodos hoja son símbolos no terminales (1, 2 y 3 son **num**) y que todos los nodos internos son símbolos no terminales. También, si se recorren los nodos hoja de izquierda a derecha obte-



(a) Árbol de sintaxis concreto

(b) Árbol de sintaxis abstracto

Fig. 2.3: Árbol de sintaxis.

nemos $1 + 2 * 3$.

El nodo raíz en el árbol es Expr, el símbolo no terminal inicial. Los hijos de todos los nodos son los símbolos en el cuerpo de alguna de las producciones en la gramática. Así es como se forma el árbol de sintaxis concreto de una frase. Dado que la gramática no es ambigua el árbol para cada frase es único, es decir, cada frase tiene un significado único. Esto no sería el caso si no hubiésemos ocupado distintos símbolos terminales para diferenciar entre términos, expresiones y factores. De no haber hecho esto podríamos encontrar dos árboles, uno que significaría $(1 + 2) * 3$ y otro que significaría $1 + (2 * 3)$.

El árbol de sintaxis abstracto, mostrado en 2.3(b), es un árbol donde cada nodo interno es un operador y los nodos hijos son los operandos. Podemos llegar a él simplemente quitando los nodos que no son ni operadores ni operaddos del árbol de sintaxis concreto.

Hay varias técnicas para construir el árbol de sintaxis, de forma muy amplia se separan en dos tipos, las que lo construyen de arriba hacia abajo, es decir, comenzando con el nodo raíz, seleccionando una producción y colocando los símbolos de esa producción como nodos hijo de la raíz hasta terminar el árbol completo. Y las que lo construyen de abajo hacia arriba. Los parsers que construyen el árbol de esta forma son llamados *shift-reduce* parsers ya que hacen dos operaciones principales. Comienzan examinando los tokens de izquierda a

derecha y colocándolos en una pila (la operación *shift*) y al tener en la pila los símbolos que formen una producción los reemplazan por el símbolo en la cabeza de esa producción (la operación *reduce*). De esta forma los símbolos que se redujeron son los nodos hijos y el símbolo no terminal de la cabeza de la producción es el padre. El árbol se va construyendo de abajo hacia arriba hasta llegar al nodo raíz, que será el símbolo no terminal inicial (Expr en nuestro ejemplo).

Es más sencillo construir el árbol de arriba hacia abajo, pero la cantidad de gramáticas con la que esta técnica funciona es menor a la cantidad de gramáticas que se pueden parsear construyendo el árbol de abajo hacia arriba. Para Puma usaremos un parser LALR(1), que significa que que es un parser que usa un símbolo de **L**ook **A**head (sólo ve un token hacia adelante mientras hace el parsing) **L**eft to right (analiza los tokens de izquierda a derecha) **R**ightmost derivation (escoge la derivación más a la derecha de una producción). Es un shift-reduce parser que construye el árbol de abajo hacia arriba.

2.4. Traducción dirigida por sintaxis

Una técnica bien conocida para hacer una traducción es embebiendo acciones en las producciones de la gramática libre de contexto. Estas acciones son conocidas como *acciones semánticas*. Por ejemplo, siguiendo con la gramática para reconocer expresiones aritméticas en la Figura 2.2, si quisiéramos traducir las expresiones infijas a expresiones postfijas podríamos agregar las acciones semánticas como se muestra en la Figura 2.4. Las acciones semánticas están entre llaves.

Expr	→	Expr + Term { imprime '+'}
	|	Expr − Term { imrpime '−'}
	|	Term
Term	→	Term / Factor { imprime '/'}
	|	Term ∗ Factor { imprime '∗'}
	|	Factor
Factor	→	**num** { imprime '**num**'} | **(** Expr **)**

Fig. 2.4: Gramática libre de contexto con acciones semánticas.

Al construir el árbol de sintaxis las acciones semánticas se colocan como cualquier nodo, en el orden que aparecen en la producción. El orden sí importa porque tendrá un impacto en el orden en el que se realizarán las acciones. Si generamos el árbol de sintaxis para la expresión $1 + 2 * 3$ usando la gramática con las acciones semánticas el árbol que construiremos será el mostrado en la Figura 2.5.

Si recorremos este árbol en orden postfijo realizando cada acción semántica

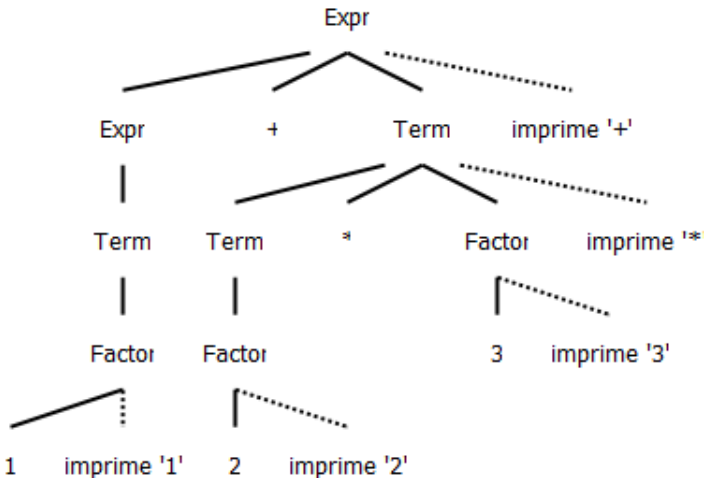


Fig. 2.5: Árbol de sintaxis con acciones semánticas.

cuando nos la topemos, la salida será $123*+$ que es justamente lo que buscábamos, la expresión en forma postfija. Nótese que el resultado fue la expresión en forma postfija no porque recorrimos el árbol en orden postfijo, sino por la posición de las acciones semánticas en la gramática. Si pusiéramos las acciones antes de los demás símbolos en cada producción el resultado hubiera sido $+1*23$, la expresión en forma prefija.

En Puma esta es la técnica que usamos para traducir el lenguaje de alto nivel a uno de bajo nivel, sólo que las acciones semánticas no son tan sencillas. En general lo que las acciones semánticas harán es recolectar información y guardarla en distintas estructuras, principalmente la tabla de símbolos y generar las instrucciones de bajo nivel que posteriormente serán ejecutadas.

2.5. *Herramientas para generar analizadores léxicos y sintácticos*

Para no escribir manualmente todo el código del analizador léxico y el sintáctico, existen varias herramientas que lo generan automáticamente. Para generar analizadores léxicos se tiene que proporcionar como entrada a la herramienta una serie de expresiones regulares que serán las cadenas a reconocer y para generar el analizador sintáctico la entrada es la gramática libre de contexto del lenguaje con acciones semánticas.

Dos herramientas populares de este tipo son Lex y Yacc[14], que sirven para generar el analizador léxico y el sintáctico, respectivamente. Ambas generan código C y son de software libre. Han sido usados en proyectos grandes como

gcc[1] y go[2].

Existen algoritmos como el de McNaughton y Yamada[15] para generar autómatas finitos deterministas a partir de una expresión regular. Ya con el autómata finito determinista se puede usar para reconocer cadenas en el texto. Ésta es una forma de implementar un generador de analizadores léxicos. También se podría generar un autómata finito no determinista usando el algoritmo de McNaughton-Yamada-Thompson[16] y simularlo directamente[16] o convertirlo a un autómata finito determinista, ya que existen uno o más equivalentes para cada autómata finito no determinista.

Lex y Yacc están diseñados para poder trabajar en conjunto fácilmente. Sin embargo, ya que generan código C y Puma está construido en C#, usaremos otro par de programas similares llamados gplex[19] y gppg[20].

[1] `http://gcc.gnu.org/`
[2] `http://golang.org/`

3. ARQUITECTURA Y DESARROLLO DE PUMA

Puma está desarrollado totalmente en C# sobre la plataforma .NET v3.5 usando Visual Studio 2008[17]. .NET nos permite fácilmente construir un ambiente virtual casi totalmente independiente del Sistema Operativo y la arquitectura de la máquina que ejecutará Puma. Esto es deseable porque no queremos distraer al usuario de Puma con estos detalles técnicos, y como bono adicional deja la puerta abierta para fácilmente migrar Puma a otras plataformas como Mac OSX y Linux a través de implementaciones alternativas de .NET como Mono[18].

3.1. Arquitectura general

Puma consta de 4 componentes principales, el analizador léxico, el analizador sintáctico, el intérprete y la interfaz gráfica. La Figura 3.1 muestra las interacciones entre estos componentes.

La interfaz gráfica consta de un editor donde el usuario escribe el código de su aplicación y menús que permiten al usuario ejecutarlo de distintas formas, ya sea en grupo o ejecutarlo línea por línea como si se estuviera usando un depurador. Esto permite al usuario ver el flujo de ejecución que tiene el código y entender cómo funcionan las estructuras de control del lenguaje como las condiciones, los ciclos y las llamadas a funciones.

Cuando Puma es iniciado el sistema crea un analizador léxico (o lexer) y un analizador sintáctico (parser) y le pasa la referencia del lexer. Una vez que el usuario indica que desea ejecutar su aplicación el código fuente es pasado al parser y se llama la subrutina para que comience el análisis (parsing). Si el parser no encuentra ningún error en el código fuente, la interfaz crea una instancia del intérprete y le pasa la salida del parser que es una unidad de código y la tabla de símbolos.

Dependiendo de la forma de ejecución que haya indicado el usuario, el intérprete ejecuta la aplicación y muestra la salida estándar de éste en la interfaz gráfica.

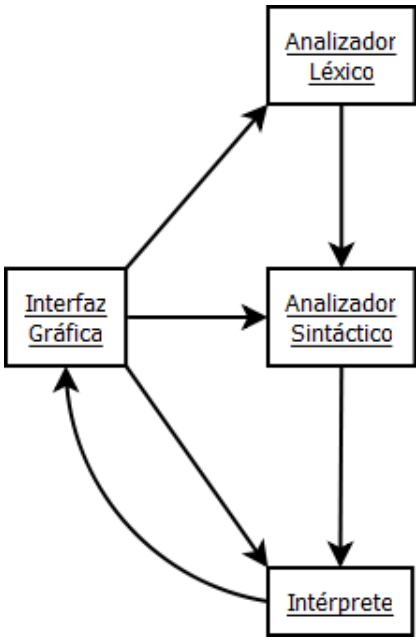


Fig. 3.1: Arquitectura general de Puma.

3.2. Analizador léxico

El analizador léxico es generado automáticamente por una herramienta llamada gplex[19] a partir de una especificación de expresiones regulares que representan los tokens que serán enviados al analizador sintáctico.

La forma en que funciona gplex es que toma un archivo con la especificación del lexer que se desea generar y genera el código fuente en C# que posteriormente es compilado para generar el analizador léxico en si. La estructura del archivo de entrada para gpplex se muestra en la Figura 3.2[21].

```
Declaraciones
%%
Reglas
%%
Funciones auxiliares
```

Fig. 3.2: Estructura del archivo de entrada para gpplex.

La Figura 3.3 muestra la sección de declaraciones del archivo de especificación del lexer de Puma. En las líneas 1–3 hacemos unas cuantas especificaciones sobre el código de C# que va a ser generado, primero especificamos que va a hacer uso del espacio de nombres `Puma.Runtime` que es donde está el código fuente que usaremos para generar símbolos e ingresarlos en la tabla de símbolos. En la línea dos definimos que el código generado estará en el espacio de nombres `Puma.Lenguaje` y finalmente en la línea 3 definimos que la clase generada tendrá un nivel de acceso `internal`.

```
(1)  %using Puma.Runtime;
(2)  %namespace Puma.Lenguaje
(3)  %visibility internal
(4)
(5)  ws       [ \t]
(6)  nl       \r?\n
(7)  digit    [0-9]
(8)  number   {digit}+
(9)  letter   [A-Za-z]
(10) id       {letter}({number}|{letter})*
(11) int      {number}
(12) float    {number}\.{number}((e|E)(\+|-)?{number})?
```

Fig. 3.3: Sección de declaraciones del archivo de definición del lexer de Puma.

Las líneas de 5–12 definen algunas expresiones regulares que posteriormente utilizaremos en la sección de Reglas para identificar los tokens en el código fuente y mandar el token correcto al analizador sintáctico. La sintaxis es de `nombre expresión`. El nombrar cada expresión regular nos permite hacer uso de ella posteriormente. Como ejemplo en la línea 8 la expresión regular `number` hace uso de la expresión regular `digit` definida la línea 7.

En la Figura 3.4 vemos partes de la sección de reglas del archivo de definición del lexer de Puma, a continuación se describirán algunas de ellas.

Las reglas están en formato `expresión acción` donde la expresión se refiere a una expresión regular y la acción a lo que hace el lexer cuando se encuentre con la expresión correspondiente. La acción se define en código C# y por lo general simplemente se regresa el token al que corresponde la expresión. Un ejemplo de esto se puede observar en la línea 5 donde al encontrarse el símbolo `+` simplemente se regresa el token `Tokens.SUM`.

También hay casos un poco más complicados donde se hacen más cosas además de regresar el token, como por ejemplo en la línea 39 está el caso de

```
(1)  {ws}        { }
(2)  /\/.*       { } // comentarios de una línea (ex. // comment)
(3)  no          { return (int)Tokens.BANG; }
(4)  \.          { return (int)Tokens.DOT; }
(5)  +           { return (int)Tokens.SUM; }
                   . . .
(14) ,           { return (int)Tokens.COMMA; }
(15) {nl}        { line_num++; return (int)Tokens.NEWLINE; }
(16) o           { return (int)Tokens.LOGIC_OR; }
(17) y           { return (int)Tokens.LOGIC_AND; }
(18) ==          { return (int)Tokens.EQUALS; }
(19) !=          { return (int)Tokens.NEQUALS; }
                   . . .
(24) {int}       { yylval = InstallLiteral(int.Parse(yytext));
(25)               return (int)Tokens.INT; }
(26) {float}     { yylval = InstallLiteral(float.Parse(yytext));
(27)               return (int)Tokens.FLOAT; }
(28) dir         { return (int)Tokens.DIR; }
(29) ref         { return (int)Tokens.REF; }
(30) imprln      { return (int)Tokens.PRINTLN; }
                   . . .
(37) haz         { return (int)Tokens.DO; }
(38) fin         { return (int)Tokens.END; }
(39) {id}        { yylval = InstallSym(yytext); return (int)Tokens.ID; }
(40) .           { return (int)Tokens.error; }
```

Fig. 3.4: Sección de reglas del archivo de defición del lexer de Puma.

encontrarse con un identificador, el cual como vimos en la sección de declaraciones del archivo consiste de {letter}({number}|{letter})*, es decir, una letra seguida de 0 o más letras o números. En este caso antes de regresar el token se llama la función InstallSym() pasando como parámetro yytext, más adelante en esta mismoa sección veremos la definición de esta función, por ahora es suficiente mencionar que agrega un símbolo nuevo a la tabla de símbolos con el lexema yytext, la variable yytext es una variable especial disponible a las acciones del lexer que contiene el texto que corresponde con la expresión regular de la acción.

InstallSym() regresa el índice de la tabla de símbolos que refiere al símbolo que posiblemente se acaba de agregar el cual se asigna a yylval, otra variable especial cuyo valor estará a disposición del parser.

Otra regla interesante es la mostrada en la línea 1, la expresión de esta regla se refiere a espacios blancos (excluyendo saltos de línea). Como podemos ver la acción no hace absolutamente nada, esto es el comportamiento correcto ya que la gramática del lenguaje de Puma no hace uso de espacios blancos en ningún momento. Por esto el lexer al encontrar un espacio blanco lo desecha sin informarle al parser de él. Algo muy similar sucede con los comentarios, como se ve en la línea 2.

Los saltos de línea no se agrupan con el resto de los espacios blancos ya que estos sí afectan la gramática del lenguaje de Puma, son usados para separar las sentencias entre si. Algo similar a los puntos y coma en la mayoría de los lenguajes derivados de C. Dado esto, la línea 15 contiene la regla para regresar el token Tokens.NEWLINE al parser. Además de esto dicha regla incrementa una variable llamada line_num que se usa para mantener el conteo de la línea en que se encuentra actualmente el lexer.

En la línea 24 está la regla para tratar a los números enteros que se encuentran en el código fuente, la acción que se ejecuta en este caso agrega el valor del entero encontrado a una tabla de literales usando la función InstallLiteral() la cual funciona de una forma muy similar a InstallSym() y también de la misma forma asigna el índice del valor que se acaba de agregar a la variable yylval para su uso posterior en el parser.

La última sección la veremos por partes, la Figura 3.5 muestra algunas declaraciones que serán copiadas al cuerpo de la clase del lexer, lo cual significa que serán propiedades de la clase.

En las líneas 1–3 se declaran propiedades públicas de escritura y lectura ya que serán leídas y modificadas por el parser. La lista Literals es donde se insertan los valores literales que se encuentren en el código fuente. Sym es la tabla de símbolos donde se insertan los identificadores que se encuentren en el código fuente, el parser cambiará la tabla de símbolos de acuerdo con el contexto actual

```
(1) public List<object> Literals { get; set; }
(2) public SymbolTable Sym { get; set; }
(3) public ErrorHandler Handler { get; set; }
(4) int line_num = 1;
(5)
(6) public int Line { get { return line_num; } }
```

Fig. 3.5: Declaraciones en la sección de funciones auxiliares del archivo de definición del lexer de Puma.

en el que se encuentre. De esta forma el nombre de una variable es local a la función donde es declarada.

`Handler` es un objeto que se encarga de reportar los errores, es utilizado por el parser ya que éste es el que detecta los errores sintácticos. En la línea 4 se declara un miembro privado usado para contar en que línea se encuentra el lexer, será incrementada cada vez que se encuentre un salto de línea en el código fuente. La propiedad pública de sólo lectura `Line` es usada por el parser para saber en que línea se encuentra el lexer.

La Figura 3.6 muestra la implementación de la función `InstallSym()` que se llama al encontrar un identificador, pasándole como parámetro el lexema de éste. En las líneas 3–7 recorre linealmente la tabla de símbolos buscando el identificador y si lo encuentra regresa su índice. Si no, en las líneas 9–11 lo agrega a la tabla y regresa su índice.

```
(1)  public int InstallSym(string id)
(2)  {
(3)      for (int i = 0; i < Sym.Count; i++) {
(4)          if (Sym[i].Lexeme == id) {
(5)              return i;
(6)          }
(7)      }
(8)
(9)      Sym.Add(new Symbol() { Lexeme = id });
(10)
(11)     return Sym.Count - 1;
(12) }
```

Fig. 3.6: Función `InstallSym()`.

3.3. Analizador sintáctico

Como vimos en el capítulo pasado, el analizador léxico (o lexer) analiza una secuencia de caracteres y genera una secuencia de tokens. Esta secuencia de tokens es analizada por el analizador sintáctico (o parser) y extrae el significado del código fuente, para de esta forma generar el código intermedio inspirado en ensamblador.

Para generar el parser usaremos una herramienta similar a gplex[19] llamada gppg[20]. El principio de la herramienta es básicamente el mismo, le damos un archivo que contiene la definición de la gramática del lenguaje que queremos analizar y la herramienta nos genera código fuente en C# del parser que luego compilaremos.

Gppg y gplex están diseñados para funcionar en conjunto (aunque no es un requerimiento). Gplex genera un lexer que puede fácilmente ser utilizado por el parser generado por gppg.

La Figura 3.7 muestra parte de la gramática del lenguaje de Puma. Los símbolos terminales están escritos en **negritas** y los símbolos no terminales están escritos con fuente normal. Se usa el símbolo | para separar las alternativas que tiene un símbolo no terminal, se lee como "o".

El símbolo `block` es el inicial del lenguaje de Puma, de él se deriva una secuencia de sentencias (la producción para `stmt`) las cuales pueden ser: la asignación de un valor a una variable, la declaración de una variable, un `si-entonces`, la instrucción de imprimir un valor a la salida estándar, un `mientras-haz`, la declaración de una nueva función, la llamada a una función o la instrucción `regresa` (para regresar de una función). La última producción de `stmt` sólo ocurre cuando el lexer regresa el token de error.

Cabe notar la producción para `newline` cuya única derivación es el token de salto de línea. Queremos guardar cuál línea del código fuente fue la que generó cada instrucción en el código intermedio (que es producido por el parser). Por lo tanto nos es útil llevar un conteo de la línea actual en que está el parser, que no es necesariamente la misma en la que se encuentra el lexer, es por esto que el parser lleva su propio conteo de la línea actual. Otra cosa a notar es que el símbolo `newline` es utilizado al final de cada sentencia, es decir, tiene un uso similar al punto y coma en lenguajes como C y C++.

La ventaja que tiene usar el salto de línea en vez de punto y coma es que hace que la sintaxis se vea menos "extraña"para alguien no familiarizado con los lenguajes de programación. La desventaja es que sólo se puede poner una instrucción por línea, pero es aceptable.

Las otras producciones mostradas en la Figura 3.7 son las que se refieren a

block	→	stmts
stmts	→	stmts stmt
	|	ϵ
newline	→	**\n**
stmt	→	var_attribute **=** bool newline
	|	decl newline
	|	**si** bool **entonces** newline block **fin** newline
	|	**imprln** bool newline
	|	**mientras** bool **haz** newline block **fin**
	|	func_decl newline
	|	func_call newline
	|	**regresa** bool newline
	|	**error**
func_decl	→	**funcion id (** opt_param_def_list **)**
		regresa def_type newline block **fin**
opt_param_def_list	→	opt_param_def_list **,** param_def
	|	para_def
	|	ϵ
param_def	→	def_type **id**
func_call	→	**id (** opt_param_list **)**
opt_param_list	→	opt_param_list **,** bool
	|	bool
	|	ϵ

Fig. 3.7: Primera parte de la gramática del lenguaje de Puma.

declarar una función, definir una lista de parámetros para la declaración de una función, llamar una función y una lista opcional de parámetros para la llamada de una función.

var_attribute	→	var_attribute **. dir**
	|	var_attribute **. ref**
	|	location
location	→	location [expr]
	|	**id**
decl	→	def_type **id** initializer
def_type	→	basic_type rank
basic_type	→	**apuntador tipo**
	|	**tipo**
rank	→	[**entero**] rank
	|	ϵ
initializer	→	= bool
	|	= [elem_list]
	|	ϵ
elem_list	→	elem_list **,** bool
	|	ϵ

Fig. 3.8: Segunda parte de la gramática del lenguaje de Puma.

La Figura 3.8 empieza con la producción para `var_attribute` la cual es usada para reconocer el acceso al atributo de una variable, estos atributos son `dir`, para hacer referencia a la dirección de una variable y `ref` el cual sólo está presente en las variables de tipo apuntador, que se refiere a la variable a la que señala el apuntador.

Después le sigue la producción para `location` la cual es el acceso a un lugar en memoria, generalmente a través del identificador de una variable, pero también puede ser usando un índice junto con un arreglo o muchos índices para arreglos de muchas dimensiones.

La producción `decl` reconoce declaraciones de variables de tipo `def_type`, el cual es la definición de un tipo. Se requiere definir el tipo porque el tipo puede ser sencillo (como un entero o un real), un apuntador a un tipo sencillo o de hecho un arreglo de n dimensiones que contiene tipos sencillos o apuntadores a tipos sencillos. Para definir arreglos se usa la producción de `rank`.

La producción de `initializer` es usada para reconocer la inicialización de una variable. Una inicialización es la asignación de un valor a una variable al ser declarada. Si la variable es un arreglo el inicializador puede ser una lista de valores. Esta lista de valores se reconoce con la producción para `elem_list`.

bool	→	bool **o** join
	|	join
join	→	join **y** equality
	|	equality
equality	→	equality == rel
	|	equality ! = rel
	|	rel
rel	→	expr < expr
	|	expr <= expr
	|	expr > expr
	|	expr >= expr
	|	expr
expr	→	expr + term
	|	expr − term
	|	term
term	→	term ∗ unary
	|	term / unary
	|	unary
unary	→	− unary
	|	**no** unary
	|	factor
factor	→	**(** bool **)**
	|	**entero**
	|	**real**
	|	var_attribute
	|	**(tipo)** factor
	|	func_call

Fig. 3.9: Tercera parte del la gramática del lenguaje de Puma.

Las primeras cuatro producciones en la Figura 3.9 son de comparaciones de valores que dan como resultado cierto o falso. `bool` es un *or* lógico de un `join` y un `bool`, `join` es un *and* lógico de un `join` y un `equality`. Un `equality` es la equidad o desigualdad de un `equality` con un `rel`. Un `rel` es una comparación de dos `expr` (expresiones aritméticas).

Finalmente las últimas cuatro producciones son usadas para reconocer las expresiones aritméticas. Se separan los operadores en varias reglas para que se respete la precedencia de los operadores. Esto también se hizo en las cuatro reglas anteriores pero aquí es más evidente. Un `expr` se define como la suma o resta de un `expr` y un `term`, un `term` es una multiplicación o división de un `term` y un `unary`. Un `unary` es una negación lógica o aritmética de un `factor`, y finalmente un `factor` es una expresión entre paréntesis, un entero, un real, un valor en memoria (a través de `var_attribute`), una conversión del tipo de un factor o una llamada a una función.

Esta gramática la plasmaremos con un formato similar en un archivo para gppg y éste nos generará el código fuente en C# para el parser. En este archivo además de la gramática añadiremos acciones semánticas embebidas en las reglas de producción que se ejecutarán cuando el parser haga la reducción de un conjunto de símbolos de dicha producción, veremos esto más a detalle en la sección de generación de código, ya que este proceso se lleva a cabo en las acciones semánticas.

3.4. Generación de código

Debido a que el parser hace las reducciones en el mismo orden que si se recorriera el árbol de sintaxis en postorden, podemos usar acciones semánticas para la generación del código. Antes de entrar en detalle de cómo generamos el código hablaremos un poco sobre el código que queremos generar. Es un código inspirado en ensamblador donde cada instrucción contiene una operación binaria y 3 direcciones de memoria, dos para los operandos y el otro para el resultado.

La Figura 3.10 muestra las operaciones disponibles en el código generado por el compilador de Puma. `Noop` es una instrucción que no hace nada. `Param` sirve para empujar un parámetro en la sección de memoria del procedimiento cuando éste se va a llamar. `Call` hace una llamada al procedimiento (con los parámetros que se hayan empujado usando `Param`). `Return` regresa un valor de un procedimiento. `Assign` hace una asignación de un valor a una variable. `ArrayAssign` hace la asignación de un valor a un arreglo en una posición indicada por un índice. `GetAddress` guarda la dirección de memoria de una variable. `GetReference` se refiere a la variable a la que señala un apuntador. `Access` es el acceso a un arreglo en una dirección especificado por un índice. `Print` es una instrucción para imprimir un valor a la salida estándar. `Conv` realiza la conver-

```
public enum Kind
{
    Noop,
    Param,
    Call,
    Return,
    Assign,
    ArrayAssign,
    GetAddress,
    GetReference,
    Access,
    Print,
    Conv,
    Goto,
    IfFalseGoto,
    Div,
    Mult,
    Sub,
    Sum,
    Gte,
    Gt,
    Lte,
    Lt,
    Neq,
    Eq,
    And,
    Or,
    Negate,
    Not
}
```

Fig. 3.10: Operaciones disponibles en el código generado por el compilador.

sión de una variable de un tipo a otro (por ejemplo convertir un entero en un real o viceversa). `Goto` hace un salto a una instrucción (del código generado) especificado por una etiqueta. `IfFalseGoto` realiza un salto de la misma forma pero sólo en caso de que una expresión dada se evalúe a falso. `Div`, `Mult`, `Sub` y `Sum` se refieren a las cuatro operaciones aritméticas básicas división, multiplicación, sustracción y suma respectivamente. `Gte`, `Gt`, `Lte` y `Lt` son los operadores relacionales $\geq, >, \leq, <$ respectivamente. `Neq` es el operador $\neq$, `Eq` el operador $=$ y `And` y `Or` son los operadores lógicos de disyunción y conjunción, respectivamente. `Negate` es la negación aritmética, es decir, multiplicar un número por -1 y finalmente `Not` es la negación lógica.

```
class Instruction
{
    /* . . . */

    public Instruction(int line)
    {
        Labels = new List<int>();
        Line = line;

        Arg1 = -1;
        Arg2 = -1;
        Res = -1;
    }

    public Kind Op { get; set; }
    public int Arg1 { get; set; }
    public int Arg2 { get; set; }
    public int Res { get; set; }
    public int Line { get; set; }

    public List<int> Labels { get; set; }
}
```

Fig. 3.11: Estructura para guardar las instrucciones del código generado por el compilador de Puma.

La Figura 3.11 muestra la estructura donde guardamos las instrucciones generadas por el compilador que posteriormente serán ejecutadas por el intérprete. El código omitido es la enumeración mostrada en la Figura 3.10. El constructor recibe únicamente un entero que es la línea del código fuente que generó esta instrucción. Además inicializa `Labels` que es una lista de etiquetas (usadas por `Goto` y `IfFalseGoto`). `Op` es la operación que realizará la instrucción y finalmente `Arg1`, `Arg2` y `Res` son los operandos y el resultado de la operación,

respectivamente. Generalmente si son una variable o constante son el índice que hace referencia a su entrada en la tabla de símbolos. Pero esto no es un requerimiento, por ejemplo en la instrucción `Goto`, `Arg1` será la etiqueta a la que hay que brincar.

Hasta ahora nos hemos referido a la tabla de símbolos como si hubiera únicamente una. Esto no es completamente cierto, en realidad hay una por cada *contexto*. Un contexto consiste en esencia de una tabla de símbolos y una unidad de código. Una unidad de código es un conjunto de instrucciones que serán ejecutadas por el intérprete. Se comienza con un contexto principal llamado `main` y se crea uno nuevo cada vez que se define una nueva función. El nombre que se le da es el mismo que el de la función. Cada uno tiene una referencia al contexto padre, el cual es donde está declarada la función. Se le llama nombre completo de un contexto a los nombres de los ancestros concatenados con puntos al nombre de éste. Por ejemplo, una función llamada `suma` que fue declarada en el contexto principal, `main`, su nombre completo es `main.suma`.

A continuación analizaremos la generación de código para algunas de las construcciones del lenguaje de Puma.

3.5. Generación de código para una expresión aritmética

Recordemos las reglas de producción para expresiones aritméticas mostradas en la Figura 3.9. Estas producciones las plasmamos en el archivo de entrada de gppg como se muestra en la Figura 3.12.

```
expr
    : expr SUM term
        {
           $$ = DoBinOp(Instruction.Kind.Sum, $1, $3);
        }
    | expr MINUS term
        {
            $$ = DoBinOp(Instruction.Kind.Sub, $1, $3);
        }
    | term { $$ = $1; }
    ;
```

Fig. 3.12: Producción para el símbolo `expr` en el archivo de entrada para gppg.

Como se había mencionado al final de la sección 3.3, en las producciones también embebimos acciones semánticas que serán ejecutadas por el parser. Las

acciones se colocan como código C# entre llaves tal como si fueran un símbolo de la producción, y serán ejecutadas cuando el parser haga el shift de dicho símbolo. Dentro de las acciones semánticas tenemos a nuestra disposición los símbolos especiales $$ y $n que son *valores semánticos*. $n es el valor semántico del *n-ésimo* símbolo dentro del cuerpo de la producción mientras que $$ es el valor semántico de la cabeza de la producción (un símbolo no terminal). ¿Cuál es el valor semántico de un símbolo terminal? Recordemos que en la sección del lexer (Figura 3.4) mencionamos el uso de una variable especial `yylval` en las reglas para identificadores y literales. Esta variable es el valor semántico de un símbolo no terminal que en el caso de identificadores será el índice que refiere a la entrada del identificador en la tabla de símbolos.

Regresando al tema actual, si la producción es por ejemplo una suma, el parser hace una llamada a la función `DoBinOp`. Los parámetros que recibe esta función son la instrucción a generar y los operandos sobre los cuales va a actuar la instrucción generada. La instrucción es obviamente `Instruction.Kind.Sum` y los operandos son los valores semánticos de `expr` y `term`. Los valores semánticos de estos símbolos serán el índice que refiere a la entrada de los operandos en la tabla de símbolos. `DoBinOp` genera una variable temporal y la inserta en la tabla de símbolos, el valor que regresa es el índice de esta nueva entrada y por eso es asignado al valor semántico de la producción actual (la expresión aritmética).

Aquí mostramos que esto se hace para sumas y restas, también se hace para divisiones y multiplicaciones en sus respectivas producciones. Así podemos ver que una expresión aritmética puede posiblemente generar muchas instrucciones, una por cada operador, y estas instrucciones serán generadas en el orden adecuado que respete la precedencia de los operadores.

3.6. Generación de código para una llamada a función

En la Figura 3.7 se aprecia la regla de producción para la llamada a una función. El objetivo es generar código como el siguiente para una expresión que llama una función `suma` pasando los valores 1 y 2.

```
param 1
param 2
call suma 2
```

Las primeras dos instrucciones son para empujar los valores 1 y 2 en la memoria del contexto al que se va a cambiar (por la llamada a función) y la tercer instrucción es la llamada a la función. El dos se refiere al número de parámetros que recibe la función. Esto es necesario ya que como parmámetro de una función se puede hacer una llamada a una función que regrese un valor y en este caso

no estaría claro cuantos parámetros son para cada función. Para mostrar esto imaginémonos que se tiene la expresión *suma(1, mult(2, 3))*. Esto es, queremos multiplicar 2 por 3 y sumarle 1 al producto. Esto generaría el código

```
param 1
param 2
param 3
t1 = call mult 2
param t1
call suma 2
```

De esta forma, al llamar a `mult` le pasamos únicamente 2 y 3 como parámetros, porque generaron las últimas dos instrucciones `param`, y cuando llamamos posteriormente a `suma`, dado que los parámetros 2 y 3 ya fueron consumidos, le pasamos 1 y t1 que es una variable temporal donde se guardó el resultado de la llamada a `mult`. Esto es justo lo que buscábamos con la expresión.

```
func_call
    : ID OPEN_PAREN
        {
            param_counter_stack.Push(0);
        }
      opt_param_list CLOSE_PAREN
        {
            int num_params = param_counter_stack.Pop();
            Function function = context.FindFunction(
                context.Sym[$1].Lexeme, num_params);

            /* código omitido por brevedad */

            Puma.Runtime.Type expr_type = function.Type;
            int new_sym = CreateSymbol(expr_type, true);

            context.Unit.PushCall(parser_line, new_sym, $1, num_params);

            $$ = new_sym;
        }
    ;
```

Fig. 3.13: Producción de una llamada a función.

Ahora sí, veamos cómo se genera este código. Las Figuras 3.13 y 3.14 muestran las reglas de producción de los símbolos pertinentes más las acciones

```
opt_param_list
    : opt_param_list COMMA bool
        {
            context.Unit.PushParam(parser_line, $3);
            param_counter_stack.Push(param_counter_stack.Pop() + 1);
        }
    | bool
        {
            context.Unit.PushParam(parser_line, $1);
            param_counter_stack.Push(param_counter_stack.Pop() + 1);
        }
    | /* nada */
    ;
```

Fig. 3.14: Producción de la lista de parámetros para una función.

semánticas que generan el código. El código omitido en la Figura 3.13 es sólo unos cuantos chequeos de errores, por ejemplo checar que una función se llame con la cantidad de parámetros que recibe y que la función en efecto exista.

Empecemos con la producción para `func_call` en la Figura 3.13, primero identificamos los símbolos terminales de un identificador (el nombre de la función) y un paréntesis que abre, inmediatamente en la acción semántica que sigue agregamos un contador con valor cero en una pila. Esta pila se usa para contar el número de parámetros con los que se va a llamar la función. Se usa una pila porque como parámetro a una función se puede poner una llamada a otra función que regrese un valor, por lo tanto al empezar a empujar parámetros para la otra función agregamos un nuevo contador a la pila y al terminar lo sacamos, por lo que se sigue contando con el contador anterior para la función anterior.

Después de esto sigue el símbolo no terminal `opt_param_list` que será la lista de parámetros que se le pasan a la función, lo cual nos lleva a la producción para `opt_param_list` mostrada en la Figura 3.14. Aquí lo que hacemos en las acciones semánticas es: primero por cada parámetro en la lista generamos una instrucción `Param` y la agregamos a la unidad de código del contexto actual a través de la función `PushParam`, a esta función le pasamos la línea actual del parser y el valor del parámetro en sí, en la forma del índice que refiere al símbolo asociado con el símbolo no terminal `bool`. Este índice es el tercer valor semántico en la primera producción y el primero en la segunda. Y segundo aumentamos en uno el contador más arriba en la pila de contadores que cuentan el número de parámetros que le estamos pasando a la función.

Regresando a la producción de `func_call` en la Figura 3.13, siguiendo el símbolo `opt_param_list` encontramos un paréntesis que cierra, en la acción

semántica que sigue primero buscamos la función a la que se hace referencia usando su nombre y la cantidad de parámetros que se le están pasando, luego creamos un nuevo símbolo que será el que guardará el valor que regrese la función, generamos la instrucción `Call` para llamar a la función y la agregamos a la unidad de código del contexto actual a través de la función `PushCall` y finalmente asignamos el índice que refiere a la entrada del símbolo que acabamos de generar en la tabla de símbolos del contexto actual al valor semántico del símbolo `func_call`.

Este proceso se muestra en la Figura 3.15.

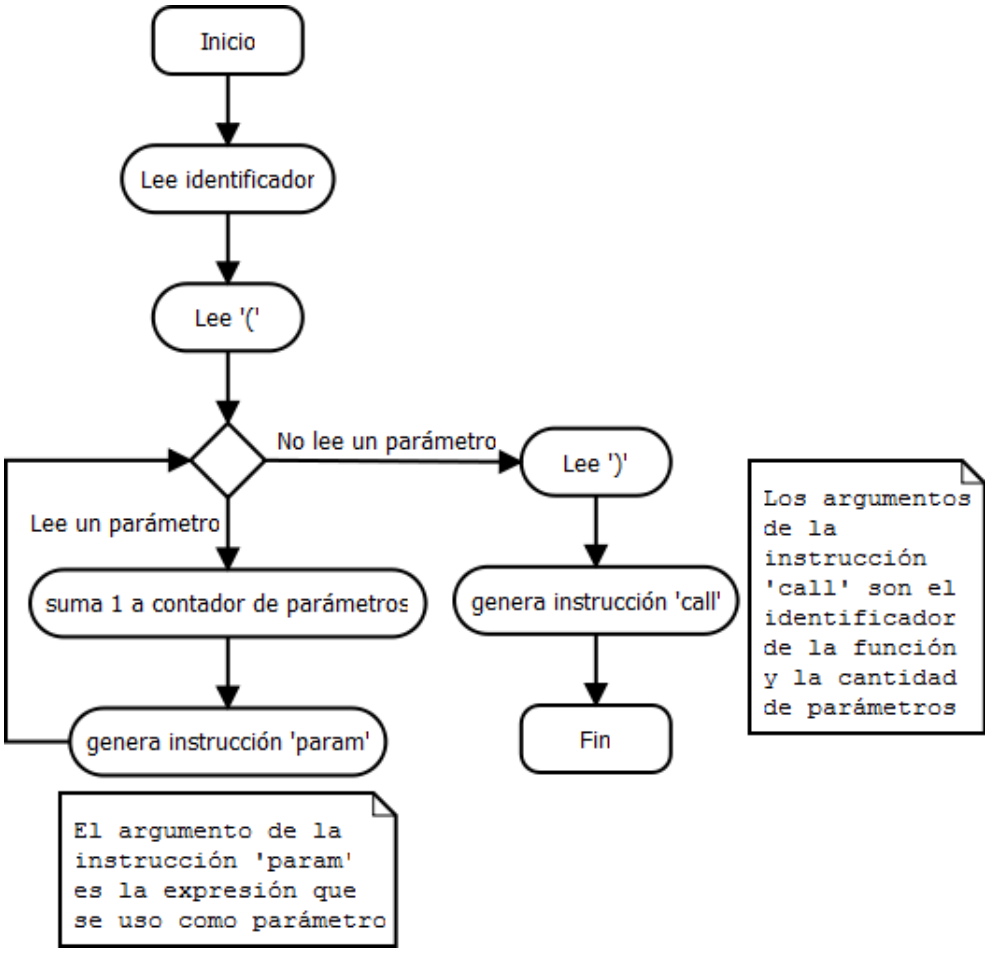


Fig. 3.15: Proceso para generar el código para la llamada a una función.

3.7. *Generación de código para un* mientras-haz

En la Figura 3.7 podemos ver las producciones para `stmt`, una de las alternativas es un ciclo *mientras-haz*, es decir, un ciclo que se repite en tanto se cumpla la condición del mientras. Por ejemplo, si tenemos el código

```
mientras i < 10 haz
  /* . . . */
  i = i + 1
fin
```

Queremos generar código como el que sigue

```
L1: noop
    t1 = i < 10
    IfFalseGoto t1 L2
    /* . . . */
    i = i + 1
    goto L1
L2: noop
```

Donde L# son etiquetas usadas por las instrucciones `IfFalseGoto` y `Goto` para brincar a otra instrucción. Las instrucciones `noop`, como ya habíamos visto al principio de la sección 3.4 en la página 22, son instrucciones que no hacen nada. Las generamos porque el no tener que seleccionar una instrucción existente al colocar la etiqueta nos facilita la generación del código. El problema con hacer esto es que el código resultante tendrá instrucciones `noop` no necesarias que lo harán menos eficiente. Sin embargo, esto es aceptable ya que no es el objetivo de Puma generar el código más eficiente posible, sino ser una herramienta didáctica.

Hay que notar que en este caso la condición es muy sencilla y sólo generó una instrucción, eso no va a ser siempre el caso, la evaluación de la condición puede ser tan compleja como sea y no hay límite teórico de la cantidad de instrucciones que pueda generar. También hay que notar que dentro del cuerpo del ciclo puede haber otros ciclos o construcciones de código que también generen etiquetas, por lo tanto hay que tomar esto en cuenta al generar las etiquetas.

En la Figura 3.16 podemos ver la regla de producción para un *mientras-haz* y las acciones semánticas para la generación de código. Como ya vimos anteriormente en el ejemplo del código que queremos generar, vamos a necesitar generar dos etiquetas, una para poder regresar al código donde se evalúa la condición, y otra para poder saltarnos por completo el cuerpo del ciclo (en caso de que no su cumpla la condición). Lo primero que hacemos al encontrar la palabra

```
stmt
    : /* . . . */
    | WHILE
      {
          Queue<int> label_queue = new Queue<int>();
          label_queue_stack.Push(label_queue);
          int label = ++label_num;
          label_queue.Enqueue(label);
          context.Unit.PushNoop(parser_line, label);
      }
      bool
      {
          Queue<int> label_queue = label_queue_stack.Peek();
          int label = ++label_num;
          label_queue.Enqueue(label);
          context.Unit.PushIfFalseGoto(parser_line, $3, label);
      }
      DO newline block
      {
          Queue<int> label_queue = label_queue_stack.Pop();
          context.Unit.PushGoto(parser_line - 1, label_queue.Dequeue());
          context.Unit.PushNoop(parser_line, label_queue.Dequeue());
      }
      END newline
    | /* ... */
    ;
```

Fig. 3.16: Reglas de producción y acciones semánticas para un *mientras-haz*.

reservada *mientras* (con el token `WHILE`) es generar una cola donde pondremos ambas etiquetas. Esta cola la ponemos en una pila de colas. Esta pila es necesaria porque como ya mencionamos dentro del mismo ciclo puede haber otros ciclos, por lo tanto de ser éste el caso empujaríamos una nueva cola en la pila y trabajaríamos con esa para el ciclo embebido; al terminar de generar ese código la sacaríamos de la pila para continuar trabajando con la cola del ciclo anterior. Después generamos la primer instrucción `Noop` asignándole la etiqueta que acabamos de generar.

El reconocer el símbolo no terminal `bool` nos generará el código para la evaluación de la condición y nos generará una variable temporal en la tabla de símbolos con el resultado de la evaluación. Ese código es generado casi de la misma forma en que generamos código para la evaluación de expresiones aritméticas pero usando además operadores relacionales, lógicos y de igualdad.

Después generamos la segunda etiqueta, tomamos la cola de etiquetas que se encuentra en el tope de la pila y enfilamos la nueva etiqueta. Además de esto generamos la instrucción `IfFalseGoto` que brincará a la etiqueta que acabamos de generar en caso de que la condición haya evaluado a falso.

El símbolo no terminal `block` generará el código que hay en el cuerpo del ciclo así que no nos queda más que quitar la cola de la pila de colas y usar las etiquetas en esta cola para generar la instrucción `Goto` que saltará incondicionalmente a donde se evalúa la condición del ciclo y finalmente la instrucción `Noop` con la segunda etiqueta que generamos para poder salir del ciclo.

Con esto habremos generado instrucciones que repiten un cuerpo de código mientras una cierta condición se cumpla.

3.8. *Generación de código para un* si-entonces

En la Figura 3.7 se encuentran las producciones de `stmt`, una de las alternativas en esta producción es la de un *si-entonces*, esto es, un cuerpo de código que se ejecuta una vez si y sólo si una condición dada se cumple. Por ejemplo, si tenemos el código

```
si i < 10 entonces
  /* . . . */
fin
```

Vamos a querer generar las siguientes instrucciones

```
    t1 = i < 10
    ifFalseGoto t1 L1
    /* . . . */
L1: Noop
```

Es decir, si la condición evalúa a verdadero ejecutamos el código generado por el cuerpo del *si-entonces*, de lo contrario, nos saltamos todo el cuerpo para no ejecutarlo.

```
stmt
    : /* . . . */
    | IF bool THEN
        {
            label_stack.Push(++label_num);
            context.Unit.PushIfFalseGoto(
                parser_line, $2, label_stack.Peek());
        }
      newline block END newline
        {
            context.Unit.PushNoop(parser_line - 1, label_stack.Pop());
        }
    | /* . . . */
    ;
```

Fig. 3.17: Reglas de producción y acciones semánticas para un *si-entonces*.

En la Figura 3.17 podemos ver las reglas de producción y acciones semánticas de un *si-entonces*. Es similar a un *mientras-haz* pero considerablemente más sencillo. En vez de usar una pila de colas de etiquetas aquí únicamente necesitamos una pila de etiquetas, la cola no es necesaria porque sólo necesitamos generar una etiqueta y usarla al generar la instrucción `IfFalseGoto` y en un `Noop` al final del código generado por el cuerpo del *si-entonces*. Naturalmente la pila sigue siendo requerida porque es posible que dentro del cuerpo de este *si-entonces* hayan más *si-entonces*.

El reconocimiento del símbolo no terminal `bool` generará el código necesario para evaluar la condición, así como una variable temporal en la tabla de símbolos que posteriormente usamos como parámetro al generar la instrucción `IfFalseGoto`. De la misma forma el símbolo no terminal `block` producirá las instrucciones del código en el cuerpo del *si-entonces*.

Podemos observar este proceso en la Figura 3.18.

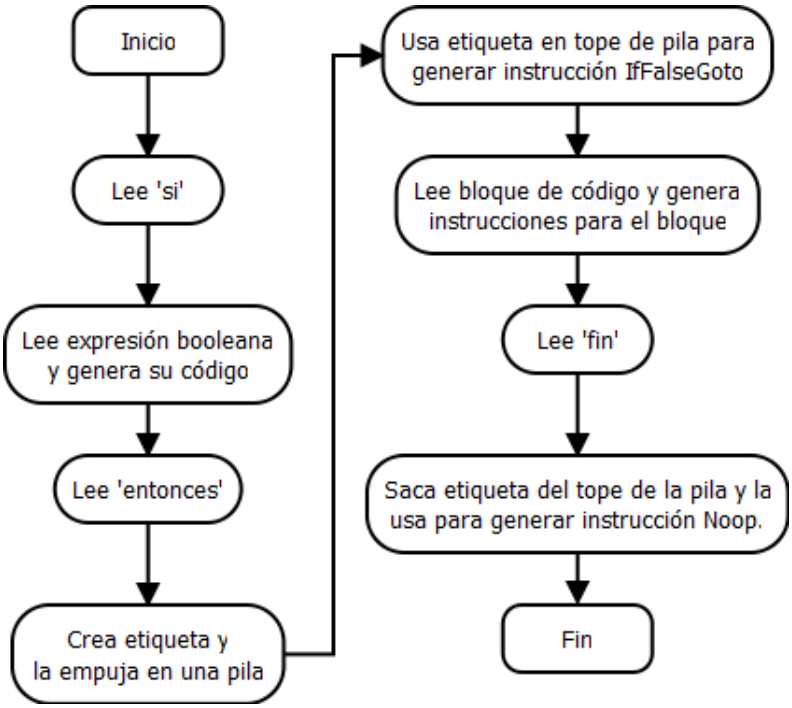


Fig. 3.18: Proceso para generar el código de un *si-entonces*.

3.9. Entorno de ejecución

Una vez que el compilador de Puma ha generado una serie de instrucciones de bajo nivel a partir del lenguaje de alto nivel, el entorno de ejecución carga estas instrucciones y las ejecuta en un ambiente virtual. El entorno de ejecución está implementado como una clase que contiene los siguientes campos

```
internal byte[] memory = new byte[1024 * 100]; // 100KB
Dictionary<string, Context> program;
internal CallStack stack;
TextWriter stdout;
int param_offset = 0;
```

El campo `memory` es un arreglo de bytes que será la memoria usada para la ejecución, aquí se guardarán todas las variables del programa, incluyendo las declaradas en el código fuente, pero además las temporales generadas por el compilador y los valores literales que se encontraban en el código fuente. Por

ejemplo al hacer una expresión como *entero i = 4 / 2*, 4 y 2 son valores literales que se tienen que poner en memoria para poder ejecutar la división.

El campo `program` es un diccionario que mapea de strings a objetos de tipo `Context`. El compilador lo que produce es el código de una serie de contextos, un contexto por cada función además del contexto principal. En este diccionario se encuentran los contextos y el string es el nombre del contexto. Recordemos que un objeto de tipo `Context` consta de una tabla de símbolos para las variables locales a la función y un conjunto de instrucciones de bajo nivel.

El campo `stack` es una pila de contextos, al iniciar un programa la pila contiene únicamente el contexto principal y cada vez que se hace una llamada a una función, el contexto de esa función se empuja sobre la pila. Al regresar la función, el contexto de tope se saca y se continúa ejecutando el contexto que hizo la llamada.

El campo `stdout` es donde el entorno de ejecución escribirá la salida estándar del programa. Este campo será asignado por la interfaz gráfica y puede ser un objeto de cualquier tipo siempre y cuando cumpla con la interfaz especificada por `TextWriter`.

Finalmente el campo `param_offset` es un entero usado al hacer una llamada a una función. Como veremos a detalle más adelante en esta misma sección, al llamar una función, los parámetros que se le van a pasar se ponen al principio de la memoria donde se ejecutará, que es también el final de la memoria de la función actual. Entonces para saber exactamente donde colocar el parámetro, lo que hacemos es ponerlo en el final de la memoria actual más el valor de `param_offset` e incrementamos el valor de `param_offset` para que el siguiente parámetro no sobrescriba el parámetro que acabamos de escribir. Después de que se haya llamado la función este valor es regresado a 0.

Antes de ver como se hace la ejecución del programa vale la pena también ver algunas de las funciones auxiliares. Veremos ahora las funciones para escribir y leer de la memoria. En la Figura 3.19 podemos ver al función usada para leer números enteros. El parámetro `offset` es la dirección donde se encuentra el entero en memoria que queremos leer. El entero se guarda en 4 bytes contiguos, así que los leemos y luego los empaquetamos en un solo entero de 32 bits.

Leer flotantes se hace de una forma muy similar pero un poco más compleja ya que si únicamente hacemos lo que hicimos anteriormente para empaquetar los cuatro bytes, C# intentaría hacer una conversión de entero a flotante y terminaríamos con un número totalmente distinto al que inicialmente guardamos. En la Figura 3.20 vemos el código de la función que lee flotantes. Para evitar la conversión primero empaquetamos los cuatro bytes en un `uint` que es un entero sin signo de 32 bits y luego creamos un apuntador a flotante y lo asignamos a la dirección del `uint`. De esta forma no sucederá ninguna conversión implícita

```
internal int ReadInt(int offset)
{
   byte b0, b1, b2, b3;

    b0 = memory[offset + 0];
    b1 = memory[offset + 1];
    b2 = memory[offset + 2];
    b3 = memory[offset + 3];

    return (int)(b0 << 24 | b1 << 16 | b2 << 8 | b3 << 0);
}
```

Fig. 3.19: Función `ReadInt`.

entre entero y real. La función está marcada como `unsafe` porque esto es un requerimiento al usar apuntadores en C#.

```
internal unsafe float ReadFloat(int offset)
{
    byte b0, b1, b2, b3;

    b0 = memory[offset + 0];
    b1 = memory[offset + 1];
    b2 = memory[offset + 2];
    b3 = memory[offset + 3];

    uint bytes = (uint)(b0 << 24 | b1 << 16 | b2 << 8 | b3 << 0);
    float* result = (float*)&bytes;

    return *result;
}
```

Fig. 3.20: Función `ReadFloat`.

Naturalmente para poder leer enteros y flotantes de la memoria, es necesario primero escribirlos en ella. En la Figura 3.21 está la función para escribir un flotante en la memoria. Para desempaquetar los bytes asignamos primero a la dirección del flotante un apuntador a `uint` para evitar conversiones implícitas. Luego tomamos los 4 bytes y los guardamos en la memoria de forma contigua empezando en la dirección especificada por el parámetro `offset`. No vamos a mostrar aquí la función para escribir enteros ya que es básicamente lo mismo

que escribir flotantes pero sin tener que preocuparnos de usar apuntadores para evitar conversiones implícitas.

```
unsafe void WriteFloat(float f, int offset)
{
    uint* f_bytes = (uint*)&f;
    byte b0 = (byte)((*f_bytes & 0xFF000000) >> 24);
    byte b1 = (byte)((*f_bytes & 0x00FF0000) >> 16);
    byte b2 = (byte)((*f_bytes & 0x0000FF00) >> 8);
    byte b3 = (byte)((*f_bytes & 0x000000FF) >> 0);

    memory[offset + 0] = b0;
    memory[offset + 1] = b1;
    memory[offset + 2] = b2;
    memory[offset + 3] = b3;
}
```

Fig. 3.21: Función `WriteFloat`.

Ahora sí, para ejecutar las instrucciones lo que hacemos es que tenemos un entero en cada contexto que es el índice de la siguiente instrucción a ejecutar. Después de ejecutar dicha instrucción incrementamos este entero para ejecutar la instrucción que sigue. Si ya no hay más instrucciones sacamos el contexto actual de la pila de llamadas (el campo `stack`) y continuamos con el contexto en el tope de la pila. Si la pila está vacía significa que el programa ha terminado. En la Figura 3.22 vemos la función que ejecuta cada instrucción pero por espacio omitimos todos los casos dentro del switch. Hay un caso por cada tipo de instrucción.

Dijimos que después de ejecutar una función el índice de instrucción se aumenta en uno, esto pasa a menos de que la instrucción que se haya ejecutado sea un `Goto` o un `IfFalseGoto` (y la condición evalúe a falso), en este caso el índice cambia al valor de la instrucción que tenga la etiqueta a la que se haya hecho el salto.

Por ejemplo, en el caso para un `IfFalseGoto` hacemos lo que se muestra en la Figura 3.23. Aquí `Sym` es la tabla de símbolos del contexto actual y `Arg1` es el índice del símbolo que contiene el resultado de la evaluación de la condición del `IfFalseGoto`. Si es igual a cero (falso) cambiamos el índice de la instrucción a ejecutar del contexto actual (el del tope de la pila) a que haga referencia a la instrucción que tenga la etiqueta que buscamos (el argumento 2 de la instrucción). Esto lo hacemos con la función `FindLabelIndex`.

```
void DoInstruction()
{
    int ip = CurrentContext.InstructionPointer;
    Instruction ins = CurrentContext.Unit[ip];
    CurrentContext.InstructionPointer++;

    switch (ins.Op) {
    case Instruction.Kind.Noop:
        break;
    /* . . . */
    default:
        break;
    }
}
```

Fig. 3.22: Función `DoInstruction`.

```
case Instruction.Kind.IfFalseGoto:
    int expr = ReadInt(CurrentContext.Sym[ins.Arg1].Address);
    if (expr == 0) {
        CurrentContext.InstructionPointer =
            CurrentContext.Unit.FindLabelIndex(ins.Arg2);
    }
    break;
```

Fig. 3.23: Caso para un `IfFalseGoto`.

Para casi todas las demás instrucciones, lo que hacemos es llamar una función auxiliar pasándole los argumentos relevantes de la instrucción. Por ejemplo, para realizar una división llamamos la función auxiliar mostrada en la Figura 3.24. Lo primero que hacemos es checar que tanto los dos operandos como el resultado tengan el mismo tipo. Esto en teoría siempre debería pasar porque cuando el compilador ve una operación entre dos variables genera una instrucción de conversión para que todas tengan el tipo adecuado. Por ejemplo si estás multiplicando un entero con un real, convierte el entero a un real. De no existir ninguna conversión posible el compilador emite un error de compilación. Una vez que se han checado los tipos se lee el valor de ambos operandos y se escribe en la dirección del resultado el cociente resultante. Si el denominador es cero, C# lanzará una excepción y de esta forma se maneja el error.

```
void DoDiv(Symbol arg1, Symbol arg2, Symbol res)
{
    CheckTypes(arg1, arg2);
    CheckTypes(arg2, res);

    if (arg1.Type == Type.IntType) {
        int a = ReadInt(arg1.Address);
        int b = ReadInt(arg2.Address);
        WriteInt(a / b, res.Address);
    } else if (arg1.Type == Type.FloatType) {
        float a = ReadFloat(arg1.Address);
        float b = ReadFloat(arg2.Address);
        WriteFloat(a / b, res.Address);
    } else {
        throw new TypeException();
    }
}
```

Fig. 3.24: Función `DoDiv`.

Las demás operaciones aritméticas, lógicas, relacionales y de igualdad se realizan básicamente de la misma forma.

Mencionamos hace dos párrafos que las operaciones aritméticas siempre se ejecutan sobre datos del mismo tipo, ya que si no son iguales el compilador emite una conversión previa a la operación de tal forma que ambos operandos sean del mismo tipo. Esta conversión se hace usando un *type-casting* explícito en C#, como se muestra en la Figura 3.25. Primero leemos el dato a convertir, luego hacemos la conversión al tipo que queremos y finalmente esto lo escribimos en la dirección del resultado.

```
void DoConv(Symbol arg1, Symbol arg2, Symbol res)
{
    if (arg1.Type == Type.IntType && arg2 == Type.FloatType) {
        int a = ReadInt(arg1.Address);
        float b = (float)a;
        WriteFloat(b, res.Address);
    } else if (arg1.Type == Type.FloatType && arg2 == Type.IntType) {
        float a = ReadFloat(arg1.Address);
        int b = (int)a;
        WriteInt(b, res.Address);
    } else {
        throw new TypeException();
    }
}
```

Fig. 3.25: Función para hacer conversiones de entero a flotante y viceversa.

Ahora hablaremos un poco sobre apuntadores y arreglos. Para poder utilizar apuntadores vamos a necesitar una forma de accesar la dirección de todas las variables y poder a partir de un apuntador accesar la variable a la que el apuntador hace referencia. Como vimos en el lenguaje esto lo hacemos usando los atributos `dir` y `ref`, para la dirección y la referencia, respectivamente.

Cuando queremos la dirección de una variable hacemos uso de la función auxiliar `DoGetAddress` mostrada en la Figura 3.26. Primero checa que el argumento de la instrucción sea un apuntador, ya que no queremos asignar direcciones a variables que no sean apuntadores, y que además sea un apuntador al mismo tipo de la variable de la cual queremos la dirección.

```
void DoGetAddress(Symbol arg, Symbol res)
{
    if (!(res.Type is Pointer) && (res.Type as Pointer).To != arg.Type) {
        throw new TypeException();
    }

    WriteInt(arg.Address, res.Address);
}
```

Fig. 3.26: Función `DoGetAddress`.

Para la referencia es un poco más interesante. Cuando el compilador generó la variable temporal para hacer la referencia de un apuntador, a esta variable temporal no le asignó dirección de memoria. Entonces lo que el intérprete hace es asignarle la dirección de esta variable temporal el valor del apuntador, el cual es naturalmente la dirección a la variable a la que el apuntador hace referencia. Esto se hace con la función mostrada en la Figura 3.27. Además de lo que acabamos de explicar hacemos chequeos de sanidad para asegurarnos de que el argumento de la instrucción es un apuntador y el tipo de la variable temporal de resultado es el mismo que el de la variable al que el apuntador señala.

```
void DoGetReference(Symbol arg, Symbol res)
{
    if (!(arg.Type is Pointer) && (arg.Type as Pointer).To != res.Type) {
        throw new TypeException();
    }

    res.Address = ReadInt(arg.Address);
}
```

Fig. 3.27: Función `DoGetReference`.

En cuanto a arreglos, lo que necesitamos es poder hacer referencia a un elemento dentro de él usando un índice tanto para leer su valor como para cambiarlo. El acceso lo hacemos usando la función `DoArrayAccess` mostrado en la Figura 3.28. Esta función primero checa que los tipos sean correctos. Luego lee el valor del índice y el tamaño en bytes del tipo de dato del arreglo. Finalmente la dirección de la variable temporal que va a almacenar el valor del acceso lo asigna al índice multiplicado por el tamaño del tipo del arreglo más la dirección éste donde empieza. Es decir, si se quiere accesar el tercer elemento dentro de un arreglo de enteros, el índice que vale 2 (ya que los arreglos empiezan en 0) es multiplicado por 4 (el tamaño en bytes de un entero) y la dirección de la variable temporal se asigna a la del arreglo más 8, que es la dirección del elemento que buscamos. De este acceso resulta una variable temporal con la misma dirección que el elemento que queremos accesar, por lo tanto podemos usarla tanto para leer el valor del elemento como para cambiarlo.

Lo más interesante que hace el intérprete es cambiar de contexto cuando se llama una función. Recordemos que cuando llamamos una función primero se genera una secuencia de instrucciones `Param` para especificar los parámetros que le pasaremos a la función. La función que se llama cuando se encuentra una instrucción `Param` es `DoParam` mostrada en la Figura 3.29. Esta función primero calcula el final de la memoria del contexto actual, la cual será memoria libre y copia el valor del parámetro a pasar aquí sumándole el valor del cam-

```
void DoArrayAccess(Symbol arg1, Symbol arg2, Symbol res)
{
    if (arg2.Type != Type.IntType) {
        throw new TypeException();
    }

    int idx_val = ReadInt(arg2.Address);
    int type_width = (arg1.Type as Arr).Of.Width;

    res.Address = arg1.Address + (idx_val * type_width);
}
```

Fig. 3.28: Función `DoArrayAccess`.

po `param_offset`, que, como ya habíamos mencionado, es usado para que los parámetros que se empujen después de éste no lo sobrescriban. Por lo tanto también aumentamos el valor de `param_offset` en el tamaño del tipo del parámetro que se pasó.

```
void DoParam(Symbol arg)
{
    int stack_frame_end =
        stack.StackFrameStart() + CurrentContext.MemoryOffset;

    CopyMem(arg.Address,
            stack_frame_end + param_offset,
            arg.Type.Width);

    param_offset += arg.Type.Width;
}
```

Fig. 3.29: Función `DoParam`.

Una vez que se han empujado todos los parámetros para la función tenemos que encontrar el contexto de ésta y empujarlo en la pila de llamadas para que se convierta en el contexto actual. Esto lo hacemos con la función `FunctionStart` mostrada en la Figura 3.30. Por brevedad omitimos un poco de código de esta función. El código omitido lo que hace es buscar el contexto de la nueva función. Luego al nuevo contexto le asigna su índice de la instrucción actual a 0 y lo empuja sobre la pila de llamadas.

Cuando una función se llama y su valor es usado en otra expresión, el compilador genera una variable temporal donde guardar el valor que regresó, así que cuando llamamos una nueva función también asignamos la dirección donde se copiará el valor de retorno a la dirección de la variable temporal generada por el compilador. Luego tenemos que decrementar el valor del campo `param_offset` en el tamaño de la suma de las variables que se pasaron como parámetros. No asignamos `param_offset` a 0 porque recordemos que es posible que hayan, por ejemplo, 3 instrucciones `Param` continuas y sin embargo luego llamemos a una función pasándole únicamente los últimos 2 parámetros. El parámetro restante sería para una función que seguramente se llamará posteriormente.

El campo `MemoryOffset` de un contexto es el tamaño en memoria que éste ocupa, es decir, la suma del tamaño de todas las variables y constantes usadas dentro de él. Incrementamos este campo en `param_offset` para que al mover el contexto actual no se sobrescriban los parámetros no usados en la llamada a esta función. Para explicar lo que es mover un contexto primero tenemos que recordar que al asignarle direcciones de memoria a las variables de un contexto el compilador asumió que éste se encontraba en la dirección de memoria cero, sin embargo esto no es cierto más que para el principal. No podemos ejecutar un contexto en la misma memoria donde se está ejecutando otro que aún no ha terminado, ya que esto seguramente causaría que varias variables del anterior se sobrescriban con los valores de las variables del actual. Por lo tanto, al comenzar a ejecutar un nuevo contexto tenemos que moverlo a un lugar de memoria libre. Esto lo hacemos simplemente sumando a las direcciones de todas las variables que le pertenecen un offset que será donde se ejecutará el nuevo contexto.

La función `StackFrameStart` suma el `MemoryOffset` de todos los contextos en la pila con la excepción del actual, es ahí donde se ejecutará el nuevo, en la primer dirección de memoria aun no utilizada. Finalmente recordamos que todos los valores literales encontrados en el código fuente deben de escribirse en memoria, esto lo hacemos para el nuevo contexto llamando la función `WriteLiterals`.

Después de entrar en el nuevo contexto, la ejecución del programa sigue normal, pero usando las instrucciones del contexto actual, el cual es el que está en el tope de la pila. Es decir el que acabamos de empujar. Una vez que se han ejecutado todas las instrucciones o se ejecutó una instrucción `Return`, lo sacamos de la pila y continuamos ejecutando el que estaba inmediatamente debajo. Esto lo hacemos con la función `FunctionFinish` mostrada en la Figura 3.31. Esta función recibe como parámetro un entero que puede no valer nada que es el índice que hace referencia al símbolo con el valor que va a regresar. Este entero puede no valer nada porque recordemos que una función no necesariamente va a regresar un valor. Si el entero sí tiene valor, es decir, sí va a regresar un valor, copiamos el valor de este símbolo a la dirección que especificamos como de retorno.

Luego quitamos el contexto actual de la pila convirtiendo así el anterior en el actual, esto es si es que había uno anterior. Si sí lo había movemos el que

```
void FunctionStart(int func_idx, int num_parameters, Symbol result)
{
    /* . . . */

    new_context.InstructionPointer = 0;
    stack.Push(new_context);
    new_context.ReturnAddress = result.Address;

    foreach (Type t in function.Parameters) {
        param_offset -= t.Width;
    }

    CurrentContext.Parent.MemoryOffset += param_offset;
    CurrentContext.Relocate(stack.StackFrameStart());

    WriteLiterals();
}
```

Fig. 3.30: Función `FunctionStart`.

acabamos de quitar de regreso a la dirección 0 con el objetivo de que la próxima vez que lo usemos lo movamos al lugar adecuado, ya que al usarlo lo movemos asumiendo que se encuentra en la dirección 0. También decrementamos el campo `MemoryOffset` en `param_offset` y con esto regresamos el estado del intérprete al que estaba antes de hacer la llamada a la nueva función, pero con el índice de la instrucción a ejecutar después de la instrucción `Call` realizó la llamada a la nueva función.

Si todavía hay al menos un contexto en la pila de llamadas regresamos en la función `True` lo cual significará que aún no ha terminado de ejecutarse el programa. `False` significa que hemos terminado.

3.10. Interfaz gráfica

Puma utiliza una interfaz gráfica bastante sencilla que consta básicamente de las tres secciones mostradas en la Figura 3.32.

Sección A En esta caja de texto es donde se escribe el código de la aplicación. Esta caja de texto nos brinda con coloreado de sintaxis para las palabras clave del lenguaje y los comentarios. Cuando se está ejecutando el programa línea por línea, al lado izquierdo del código se podrá ver una caja azul que indica la línea de código que se ejecutará a continuación.

```
bool FunctionFinish(int? return_symbol)
{
    if (return_symbol.HasValue) {
        Symbol symbol = CurrentContext.Sym[return_symbol.Value];
        CopyMem(symbol.Address,
                CurrentContext.ReturnAddress,
                symbol.Type.Width);
    }

    int current_stack_frame_start = stack.StackFrameStart();
    Context last_context = stack.Pop();

    if (stack.Count > 0) {
        last_context.Relocate(-current_stack_frame_start);
        CurrentContext.MemoryOffset -= param_offset;
        return true;
    }

    return false;
}
```

Fig. 3.31: Función `FunctionFinish`.

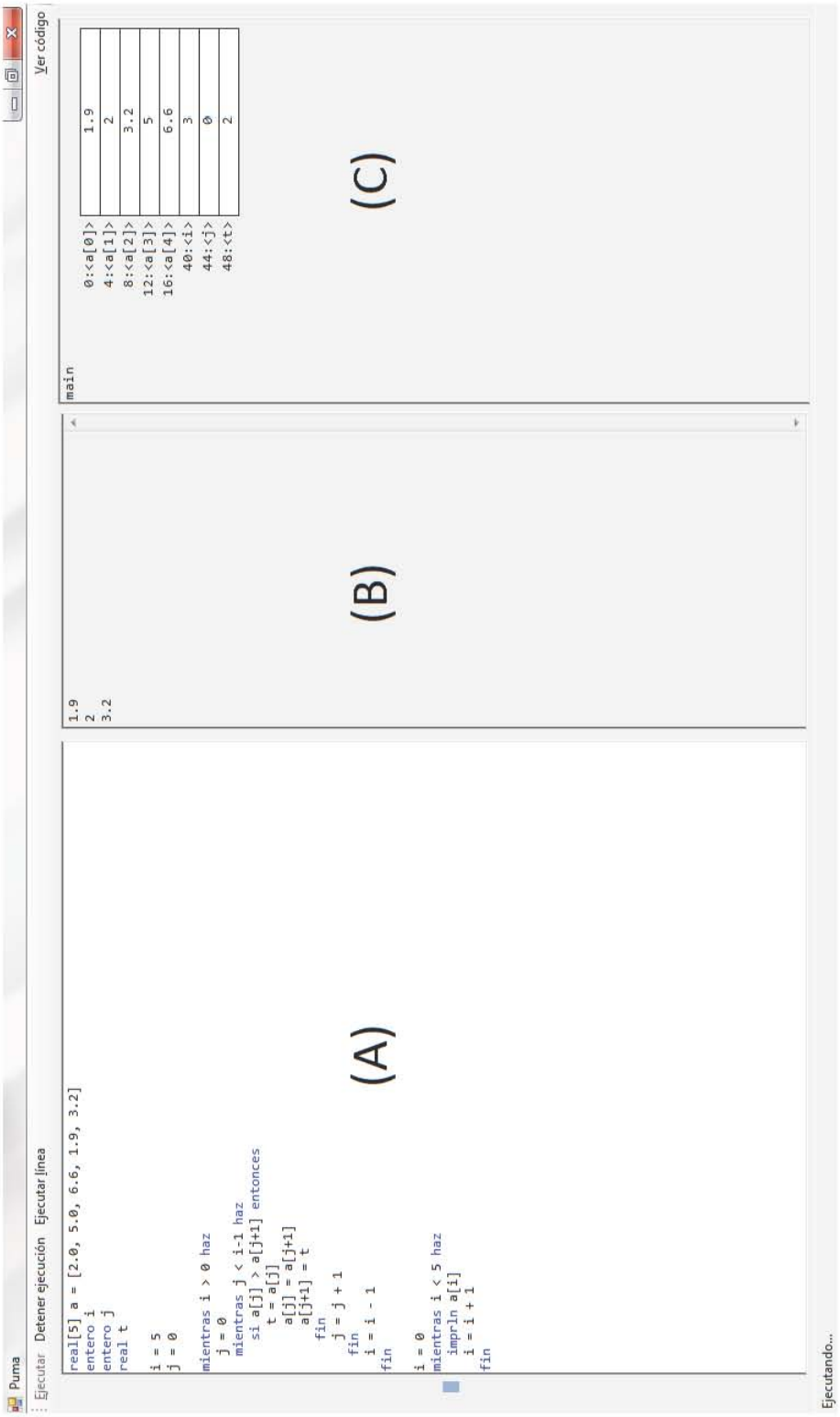


Fig. 3.32: La interfaz de Puma.

Sección B Aquí Puma imprimirá toda la salida de la aplicación al ejecutarla, es decir, todo lo que la aplicación imprima usando la instrucción `imprln`. Además de esto, si hubo un error de sintaxis o de cualquier tipo durante la compilación del programa, es aquí donde Puma reportará los problemas, haciendo su mejor esfuerzo de indicarnos la línea de código donde está el error.

Sección C Finalmente en esta sección es donde podemos ver el diagrama de la memoria. Esto es útil cuando ejecutamos el programa línea por línea, ya que podemos ver en cualquier momento de ejecución el valor de las variables que usa el programa. No muestra la memoria completa, en vez, muestra únicamente las variables declaradas por el usuario y omite las variables temporales generadas por el compilador y las constantes literales que están en el código fuente de la aplicación. La razón para esto es simplificar lo que ve el usuario para entender cómo funciona su propio código y no los detalles del cómo funciona el compilador. En el diagrama de memoria se muestra el nombre de la variable, su dirección de memoria y su valor. Para apuntadores se muestra una → junto a su valor indicando que se está apuntando a esa dirección de memoria.

Puma puede estar en dos estados: ejecutando o editando código. Dependiendo del estado de Puma, la barra de herramientas mostrará o tendrá habilitadas diferentes operaciones. En modo de edición de código las operaciones posibles son:

Ejecutar Compila y ejecuta la aplicación, si hay errores de compilación se reportarán y si no simplemente se ejecuta el programa y se muestra su salida.

Comenzar Ejecución Compila el código y entra en modo de ejecución de código línea por línea. Hablaremos más sobre esto cuando hablemos de las operaciones posibles en modo de ejecución.

Ver Código Esta operación mostrará el código de bajo nivel generado por el compilador.

En cambio cuando se está en modo de ejecución, las operaciones disponibles son

Detener Ejecución Detiene la ejecución del programa y regresa a modo de edición de código.

Ejecutar Línea Esta operación ejecuta una sola línea de código y actualiza el diagrama de memoria por si esa línea hizo algún cambio al valor de las variables.

Cabe mencionar que en modo de ejecución no se puede modificar el código fuente.

4. EFECTIVIDAD DE PUMA

Puma fue presentado a 16 alumnos y una maestra de un curso intersemestral de C para alumnos principiantes en la programación. Se les dio una presentación que explicaba el objetivo de Puma además de varios ejemplos que mostraban cómo funcionaba Puma. Se les pasó un cuestionario con 8 afirmaciones y una sección para comentarios y se les pidió que calificaran cada afirmación del 1 al 5 dependiendo de que tan de acuerdo estaban con la afirmación donde 5 era muy de acuerdo y 1 muy en desacuerdo.

Cabe mencionar que este cuestionario se realizó porque se dio la oportunidad y es puramente informativo, no pretende demostrar que Puma es de hecho una herramienta que ayuda en la enseñanza de la programación.

Las afirmaciones fueron las siguientes

1. Ver el flujo del programa me ayudará a entender la programación
2. Ver el diagrama de memoria de mi programa me ayudará a entenderlo
3. Puma parece hacer un buen trabajo de mostrarme el flujo de mi programa
4. Puma parece hacer un buen trabajo de mostrarme el diagrama de la memoria de mi programa
5. La sintaxis del lenguaje que usa Puma me pareció más sencilla que la de C
6. Me gusta que las palabras reservadas del lenguaje de Puma estén en Español
7. La interfaz gráfica de Puma me pareció intuitiva y amigable
8. Pienso que Puma se debería usar en la materia de programación de primer semestre

Las respuestas de los alumnos (mostradas en las Figuras 4.1 y 4.2) son en general positivas. Una afirmación con la que la mayoría no estuvo completamente de acuerdo es que la interfaz gráfica de Puma fuera amigable e intuitiva, un alumno tuvo esto que decir en sus comentarios: *"Me da gusto que hayas creado tu propio lenguaje y me parece un buen trabajo aunque me gustaría ver*

más color en la pantalla." lo cual me indica que la interfaz es demasiado sencilla.

Aunque con las primeras afirmaciones la mayoría pensó que Puma era una buena idea y cumplía con sus objetivos relativamente bien, la afirmación 8 muestra que no están del todo de acuerdo con que se use en la materia de programación de primer semestre, un alumno comentó: *"Bueno me parece muy amigable y fácil de manejar sin embargo en vez de ser una materia sería mejor mejorarlo y elaborar una especie de taller para conocer este grandioso sistema."* Lo cual es una muy buena idea, dada la importancia de que los alumnos no sólo entiendan la programación pero además aprendan lenguajes importantes como C, que es el que regularmente se enseña en esa materia.

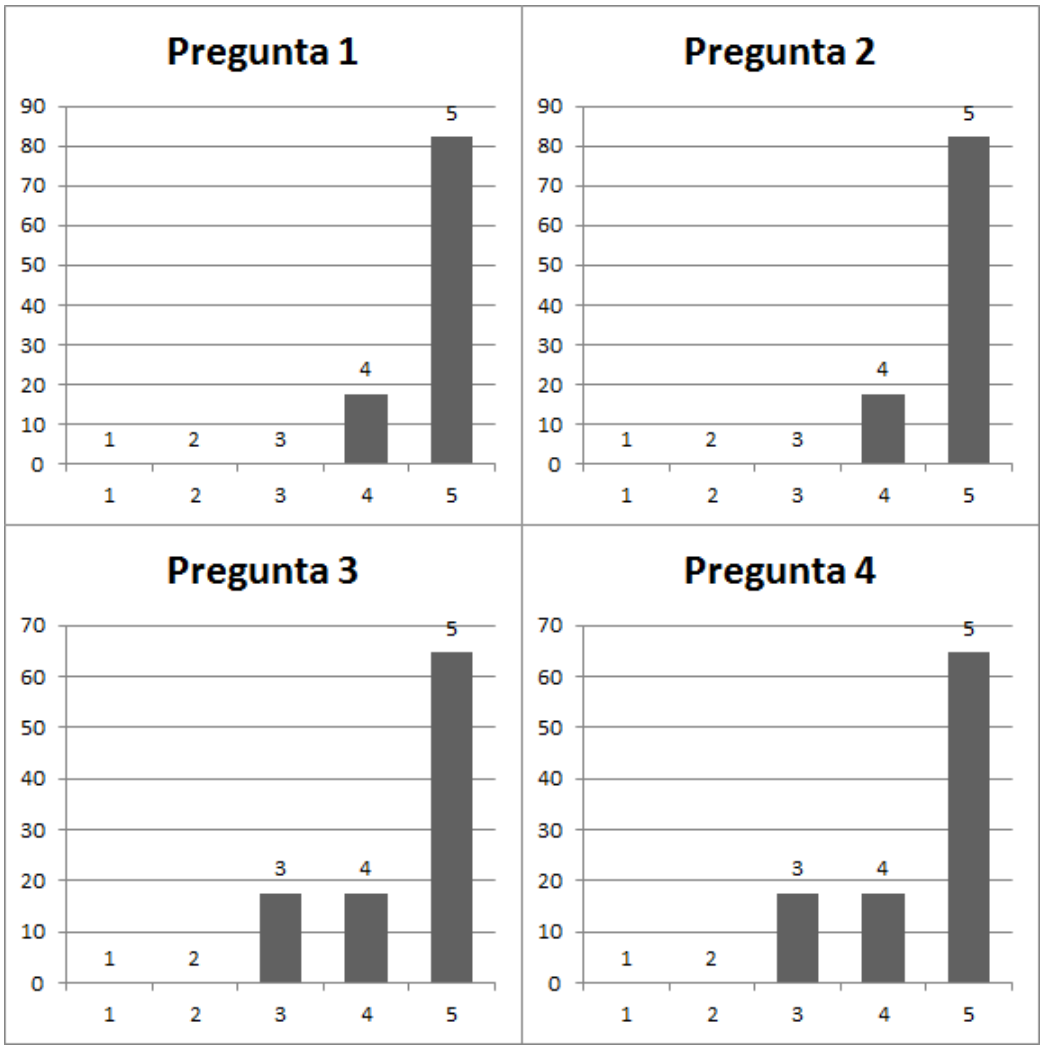


Fig. 4.1: Respuestas al cuestionario de las preguntas 1–4. La barra muestra el porcentaje de alumnos que seleccionaron esa respuesta.

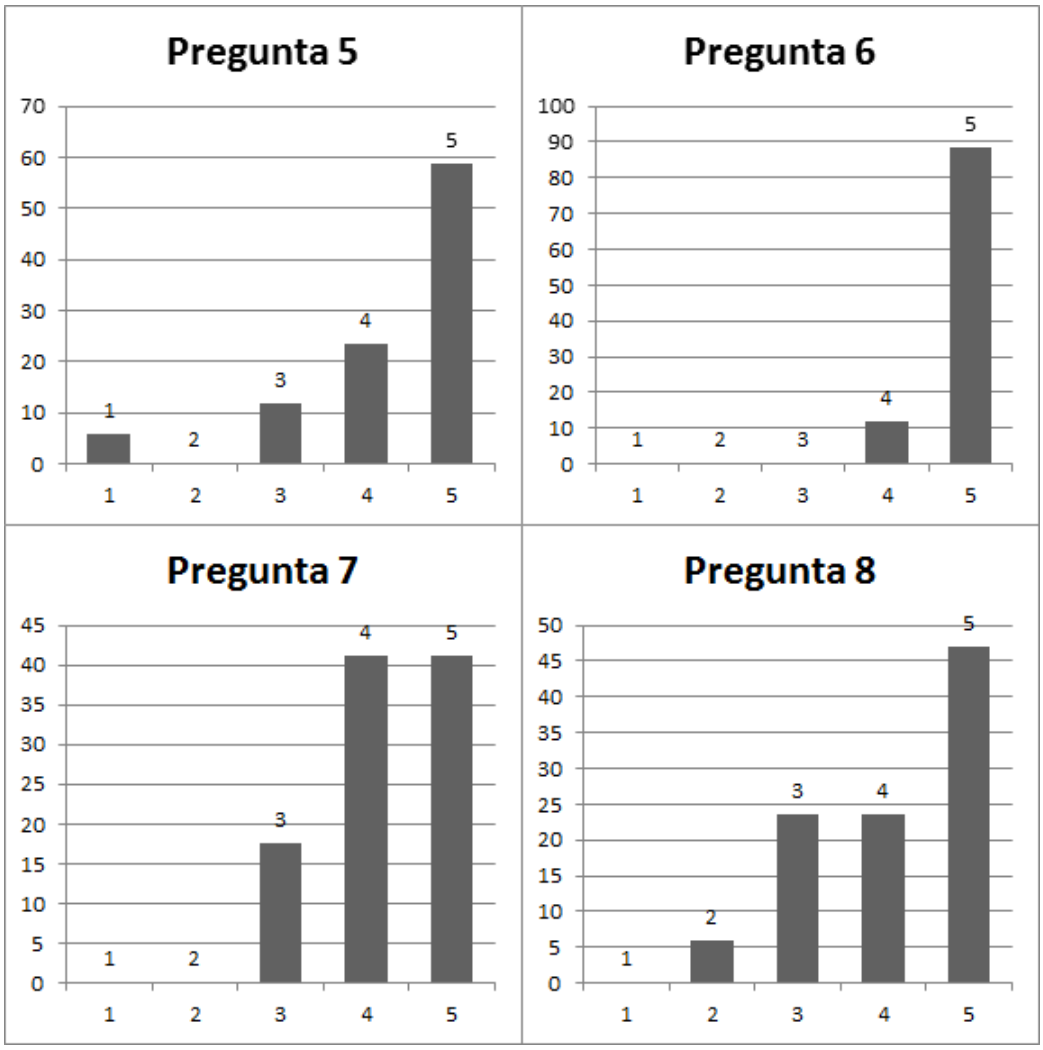


Fig. 4.2: Respuestas al cuestionario de las preguntas 5–8. La barra muestra el porcentaje de alumnos que seleccionaron esa respuesta.

5. CONCLUSIONES Y TRABAJO A FUTURO

El objetivo de Puma es ser un sistema que facilite a los alumnos el aprendizaje de lenguajes de programación de alto nivel. Esto se trata de hacer con un lenguaje imperativo con una sintaxis reducida cuyas palabras reservadas están en Español, mostrando fácilmente el flujo del programa al ejecutarlo y un diagrama de la memoria en cualquier punto de ejecución.

Aunque la sintaxis del lenguaje de Puma es sencilla, cuenta con todo lo básico de cualquier lenguaje imperativo como son las estructuras de control para repetición y decisión, arreglos, funciones y apuntadores.

La evaluación que se realizó sugiere que las ideas detrás de Puma son buenas pero la implementación podría ser mejorada, principalmente en lo que a la interfaz gráfica se refiere. Puma fue sólo desarrollado como una prueba de concepto y creo que en ese aspecto fue exitoso, los alumnos mostraron interés en él y en sus comentarios varios recomendaron continuar trabajando en él para que un día se usara en instituciones educativas.

Hay muchas cosas de las que Puma se podría beneficiar, una sería separar más la generación del código intermedio del análisis semántico y sintáctico. En el caso de Puma todo esto se hacía en las acciones semánticas de las producciones de la gramática. Sería mejor usar estas acciones semánticas para únicamente construir un árbol de sintaxis abstracto el cual fuera primero analizado para detectar errores y posteriormente se generaría el código intermedio a partir de él. Esto haría a Puma más robusto y más hábil en detectar errores en el código fuente.

Otra cosa en la que habría que trabajar, que probablemente sea más necesaria, es hacer mejor la interfaz gráfica. Principalmente crear un editor de texto más flexible y más específico a la gramática de Puma que pudiera completar código y hacer sugerencias al usuario, resaltar la línea de código actual cuando se está ejecutando el programa de una forma más elegante que el cuadrado azul que se usa actualmente. También el diagrama de memoria podría ser más ilustrativo siendo dinámico para que se pudieran ver secciones específicas de la memoria y posiblemente hacer animaciones con flechas para representar los apuntadores, o para resaltar una celda cuando su valor cambie.

Puma cuenta con un sistema de tipos, pero los únicos tipos primitivos con

los que cuenta son los números enteros y reales, sería bueno también soportar caracteres y crear funciones estándar para manipular arreglos de caracteres de tal forma que el usuario pudiera fácilmente trabajar con cadenas de texto.

Como alumno de la carrera de Matemáticas Aplicadas y Computación me siento muy satisfecho con los conocimientos que aquí he adquirido, los profesores hacen un gran trabajo mejorando la calidad educativa de sus alumnos. Es por esto que quiero agradecer a todas las personas de la universidad que han hecho mis estudios posibles.

BIBLIOGRAFíA

[1] Lahtinen, E., Ala-Mutka, K. y Järvinen, H., *"A Study of the Difficulties of Novice Programmers"*, ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, pp. 14–18, ACM, 2005.

[2] Milne, I. y Rowe G., *"Difficulties in Learning and Teaching Programming—Views of Students and Tutors"*, Volumen 7, Número 1, pp. 55–66, Springer Netherlands, Marzo, 2002.

[3] Draganova, C., *"Use of Mobile Phone Technologies in Learning"*, ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education, pp. 299–299, ACM, 2009.

[4] Bergin, J., Lister, R., McNally, M. y Owens B. B., *"The First Programming Course: Ideas to End the Enrollment Decline"*, ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, pp. 301–302, ACM, 2006.

[5] Clancy, M., Titterton, N., Ryan, C., Slotta, J. y Linn, M., *"New Roles for Students, Instructors, and Computers in a Lab-based introductory Programming Course"*, SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education, pp. 132–136, ACM, 2003.

[6] Kerren, A., *"Generation as Method for Explorative Learning in Computer Science Education"*, ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, pp. 77–81, ACM, 2004.

[7] Gonzales-Dholakia, G., *"Computing Competencies—Ensuring Student Success"*, SIGUCCS '03: Proceedings of the 31st annual ACM SIGUCCS conference on User services, pp. 18–20, ACM, 2003.

[8] Bergin, J., Stehlik, M., Roberts, J. y Pattis, J., *"Karel J Robot: A gentle introduction to the art of object oriented programming"*, url: `http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html`, Accesado 29 Dic. 2009.

[9] Raymond, L., *"Teaching Java First: Experiments with a Pigs-Early Pedagogy"*, Sixth Australasian Computing Education Conference (ACE2004), Volumen 30, pp. 177–183, ACS, url: `http://crpit.com/Vol30.html`, 2004.

[10] Chomsky, N., *"Three models for the description of language"*, IRE Transactions on Information Theory, url: `http://chomsky.info/articles/195609--.pdf`, 1956.

[11] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., Van Wijngaarden, A. y Woodger, M., *"Revised report on the algorithm language ALGOL 60"*, Commun. ACM, Volumen 6, Número 1, pp. 1–17, ACM, 1963.

[12] Aho, V. A., Lam, M. S., Sethi, R. y Ullman, J. D., *"Compilers: Principles, Techniques, & Tools"*, Segunda Edición, Addison Wesley, 2007.

[13] Hopcroft, J. E., Motwani, R. y Ullman, J. D., *"Introducción a la Teoría de Autómatas, Lenguajes y Computación"*, Segunda Edición, Addison Wesley, 2002.

[14] Levine, J. R., Mason, T. y Brown, D., *"lex & yacc"*, Segunda Edición, O'REILLY, 1995.

[15] McNaughton, W. S. y Yamada, H., *"Regular expressions and state graphs for automata"*, IRE Trans. on Electronic Computers, 1960.

[16] Thompson, K., *"Regular expression search algorithm"*, Comm. ACM 11:6, 1968.

[17] MSDN Library, *"C# Language Specification"*. `http://msdn.microsoft.com/en-us/library/ms228593.aspx`, Accesado 18 de Ene. 2010,

[18] *"The Mono Project"*. `http://mono-project.com/Main\_Page`, Accesado 18 de Ene. 2010,

[19] Queensland University of Technology, *"GPLEX"* `http://plas.fit.qut.edu.au/gplex/`, Accesado 18 de Ene. 2010,

[20] Queensland University of Technology, *"GPPG"* `http://plas.fit.qut.edu.au/gppg/`, Accesado 18 de Ene. 2010,

[21] Gough, J., *"The GPLEX Input Language"*. `http://plas.fit.qut.edu.au/gplex/files/gplex.pdf`, Accesado 18 de Ene. 2010, 3 de Marzo, 2009.