Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Proyecto integrador - Ingeniería en Computación

EXTENSIÓN DEL MODELO SIMPLESEM A UN LENGUAJE PARALELO

Alumno: Lucas L. Diez de Medina Quintar
Director: Mg. Cr. Gustavo Wolfmann
Co-Director: Ing. Orlando Micolini

-2013-

A mi familia
A mi novia, Eugenia Cavaglia

Agradecimientos

Este trabajo, así como cada paso dado a lo largo de estos años en la carrera, no podrían haberse realizado de no ser por el apoyo y la ayuda de muchas personas a quienes quiero agradecer. A mi familia cercana por haberme educado desde niño con los valores que hoy en día definen quien soy. A mi novia, quien me acompaña hace ya siete años, apoyándome en cada decisión que tomo, y en cada paso que doy, y motivándome día a día para ser mejor persona. Al director de este trabajo, Gustavo Wolfmann, quien cedió su tiempo y su colaboración desde las primeras etapas del proyecto, hasta la finalización del mismo. Al co-director, Ing. Orlando Micolini, con quien comparto un profundo respeto por haberme guiado a lo largo de la carrera, y sobretodo en este último paso. A cada docente que nos brindó sus conocimientos, ayudándonos a crecer profesionalmente. A mis amigos, sin los cuales el paso por la facultad no hubiera tenido el sentido que tuvo. En especial quisiera agradecer a Fernando Saravia, quien comenzó el desarrollo de este trabajo conmigo, implementando una parte fundamental del mismo.

Resumen

El abordaje de la enseñanza de los lenguajes de programación, a menudo se centra en las características de alto nivel, transmitiendo a los alumnos las posibilidades que un lenguaje brinda en cuanto a las primitivas y mecanismos soportados. Pocos son los casos en los que se aborda la enseñanza del impacto a bajo nivel que produce una determinada primitiva, por ejemplo la manera en que un procesador ejecuta un bucle, o cuáles son los efectos de los saltos, o de las llamadas recursivas de una función.

Si bien para el desarrollo de software resulta indispensable conocer la manera en que un programa debe ser escrito en un lenguaje determinado (gramática), también es de suma importancia comprender qué impactos tiene el código escrito en dicho lenguaje de alto nivel sobre los recursos de la computadora, como la manera en que la memoria y los registros son utilizados. Esto permite al programador conocer cuáles son las limitaciones de su sistema de computación, como así también aprovechar de la mejor manera los recursos disponibles.

Este trabajo tiene como objetivo brindar una herramienta educativa que permita analizar de manera gráfica y sencilla el impacto de un programa paralelo (escrito en un lenguaje de alto nivel), sobre instrucciones de bajo nivel, que operan directamente sobre la memoria compartida de una máquina virtual. La herramienta presenta un lenguaje simple, que a través de algunas primitivas propias permite definir hasta 4 hilos que ejecutan instrucciones en paralelo. Mediante un parser desarrollado en Perl, el código escrito en lenguaje de alto nivel es compilado a un lenguaje similar al ensamblador, conocido como Simplesem, formado por un reducido número de instrucciones compuestas. Estas instrucciones luego son cargadas y ejecutadas en un entorno de simulación que permite observar gráficamente la memoria de instrucciones de cada hilo, la memoria de datos de la máquina virtual, y la manera en que estas interactúan.

Índice general

Dedicatoria II
Agradecimientos IV
Resumen VI

Introducción **2**
1.1. Motivación 2
1.2. Alcances y Objetivos del Proyecto 5

Marco Teórico **6**
2.1. Lenguajes de Programación 6
2.1.1. Alfabeto 7
2.1.2. Lenguajes y la jerarquía de Chomsky 7
2.1.3. Autómatas finitos 9
2.1.3.1. Definición formal 11
2.1.4. Sintaxis y Semántica. 11
2.1.4.1. Variables 12
2.1.4.2. Valores y referencias 14
2.1.4.3. Expresiones 15
2.1.5. Definición de un lenguaje de programación 15
2.1.5.1. Sintaxis 16
2.1.5.2. Semántica 18
2.1.6. Procesamiento del código 20
2.1.6.1. Lenguajes Interpretados 20
2.1.6.2. Lenguajes Compilados 21
2.1.7. Paradigmas de Programación. 22
2.1.7.1. Imperativo o por Procedimientos 23
2.1.7.2. Funcional 24
2.1.7.3. Lógico 24
2.1.7.4. Orientado a objetos 25
2.1.7.5. Orientada a componentes 26
2.2. Proceso de compilación 26
2.3. Expresiones regulares (RegExp) 28

2.3.1. Búsqueda y sustitución de caracteres 31
2.4. Modelo Simplesem 32
2.4.1. Instrucción set 33
2.4.2. Instrucciones jump y jumpt 34
2.4.3. Instrucción halt 35
2.4.4. Estructura del modelo Simplesem 35
2.4.4.1. Lenguajes estáticos 35
2.4.4.2. Lenguajes basados en pilas 36
2.4.4.3. Lenguajes completamente dinámicos 36
2.4.5. Máquina de Turing 36
2.4.6. Lenguaje C1 37
2.4.7. Lenguaje C2 39
2.4.8. Lenguaje C3 41

Desarrollo **48**
3.1. Requerimientos de la herramienta 48
3.2. Propuesta de extensión 50
3.2.1. Nuevas instrucciones Simplesem 50
3.2.2. Extensión del lenguaje C3 a C3P 51
3.2.3. Utilización de múltiples hilos 52
3.2.4. Semántica del lenguaje C3P 52
3.2.4.1. Nuevas reglas de producción 53
3.2.4.2. Reglas de declaración 55
3.2.4.3. Regla fun_decla 55
3.2.4.4. Regla statement 56
3.3. Implementación del compilador 61
3.3.1. Módulo Parse::RecDescent 62
3.3.1.1. Utilización 62
3.3.1.2. Definición de Reglas Sintácticas 63
3.3.1.3. Subreglas 64
3.3.1.4. Tokens - Terminales 64
3.3.1.5. Acciones 64
3.3.2. Consideraciones para la implementación 67
3.3.2.1. Variables 67
3.3.2.2. Saltos 68
3.4. Intérprete Simplesem 69
3.4.1. Requerimientos Funcionales del intérprete 69
3.4.2. Diseño de la interfaz gráfica 70
3.4.2.1. Biblioteca Qt 70
3.4.2.2. Interfaz gráfica 70
3.4.2.3. Signals y Slots 72

3.4.2.4. Arquitectura del intérprete 75
3.4.3. Casos de Uso 77
3.4.4. Diagrama de Clases 80
3.5. Pruebas con ejemplos en lenguaje C3P 83
3.5.1. Cálculo del factorial de un número 84
3.5.2. Máximo Común Divisor 87
3.5.3. Números primos 89
3.5.4. Suma de dos arrays en paralelo 91
3.5.5. Problema de los Filósofos 94
3.6. Conclusión 99
3.7. Trabajo a futuro 100

Bibliografía **102**

A. Gramática del lenguaje C3 **104**

B. Gramática del lenguaje C3P. **108**

C. Instalación **114**
C.1. Intérprete de Perl 116
C.2. Módulo Parse::RecDescent 117
C.3. Qt SDK 117
C.3.1. Instalación de las librerías para ejecución 118

Introducción

A lo largo de las carreras orientadas a la informática y computación, se estudian numerosos lenguajes de programación de alto nivel, que constituyen la herramienta fundamental para el desarrollo de software. El código fuente escrito en dicho lenguaje, debe ser traducido a instrucciones comprensibles por el procesador para poder ser ejecutado y cumplir finalmente la función para la que fue escrito. Este proceso, conocido como compilación, puede verse en la Figura 1.1. No es un proceso sencillo, y su comprensión requiere de un análisis profundo de cada etapa involucrada.

Para facilitar la enseñanza del proceso de compilación, Carlo Ghezzi y Mehdi Jazayeri propusieron en su libro "Programming Language Concepts" un modelo simplificado de una máquina que puede procesar un lenguaje formado por 4 instrucciones compuestas, utilizadas para ejecutar gran parte de las primitivas con las que cuentan la mayoría de los lenguajes de alto nivel. Dicho modelo permite ver la manera en que estas instrucciones de estilo ensamblador interactúan con la memoria de código, con la memoria de datos, y con los distintos punteros que maneja la mayor parte de los procesadores.

1.1. Motivación

La enseñanza y comprensión del proceso de compilación es una tarea complicada tanto para alumnos como para docentes. La arquitectura x86 suele ser el punto de partida en el estudio de los sistemas de cómputo, pero dicha arquitectura es demasiado compleja debido a la cantidad de instrucciones, de registros y de mecanismos que posee.

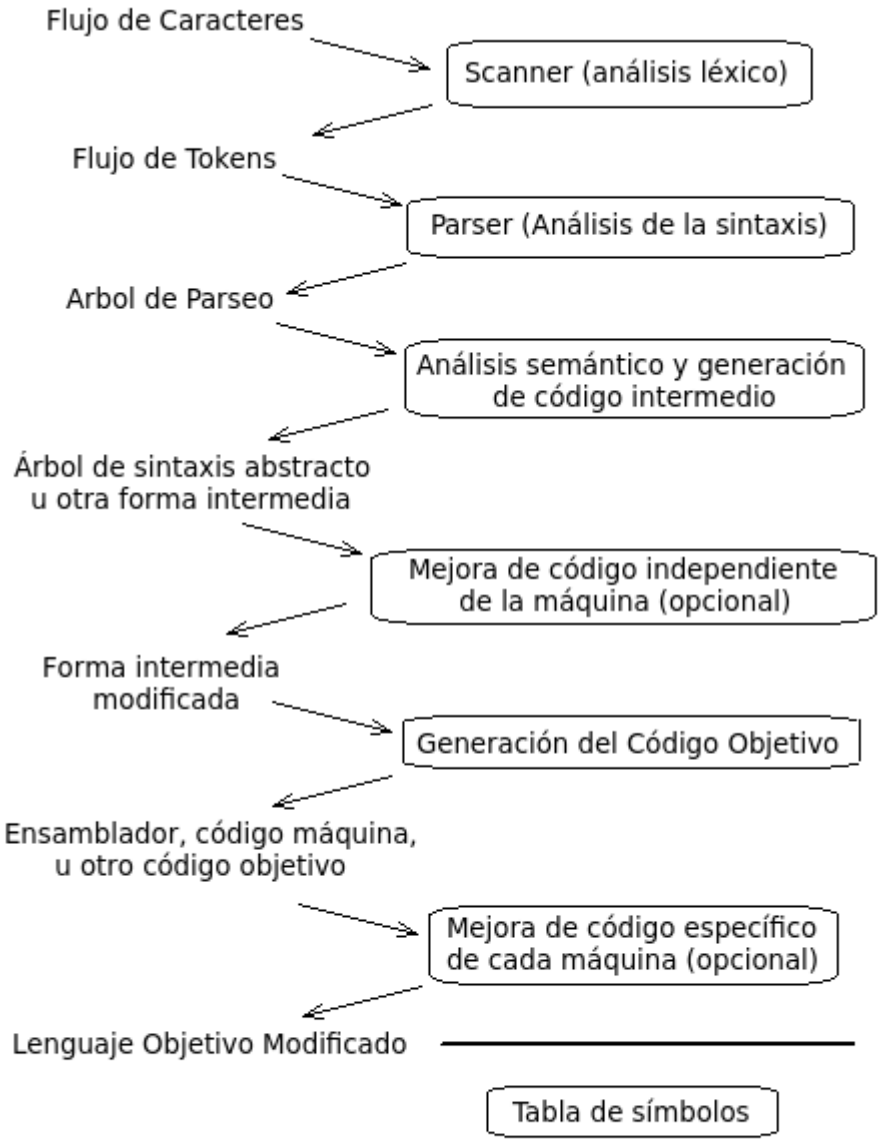


Figura 1.1: Proceso de Compilación

Se ha demostrado que el modelo Simplesem facilita el proceso de aprendizaje de los lenguajes de programación monohilo, y de la manera en que debe traducirse la semántica de estos a instrucciones en lenguaje máquina. Es utilizado en numerosas universidades del mundo, entre las que

se encuentran la Universidad de South Carolina, la Universidad de California en Riverside, la Universidad Técnica de Vienna, entre otras.

A lo largo de nuestro estudio, el modelo propuesto por Ghezzi y Jazayeri nos ayudó a comprender el impacto del código escrito en lenguajes de alto nivel sobre las instrucciones que finalmente ejecuta el procesador, de una manera mucho más sencilla, ya que simplifica al máximo la cantidad de instrucciones del ensamblador, la cantidad de registros, y los mecanismos de manejo de memoria, entre otros.

Sin embargo, este modelo se limita a lenguajes con ejecución secuencial, y no aborda la posibilidad de incorporar múltiples hilos en el código de alto nivel. Es por ello que consideramos necesaria la extensión de este modelo, para incorporar lenguajes paralelos en la enseñanza del proceso de compilación del código fuente. Esto supone un desafío que aborda gran parte de los conocimientos adquiridos en la carrera, como así también nuevos conocimientos que deberán ser incorporados a lo largo del proyecto.

Debido a los resultados favorables obtenidos con el modelo Simplesem existente en materia de educación, consideramos que realizar una extensión a un lenguaje paralelo de dicho modelo nos permitirá obtener resultados similares. Para poder realizar el estudio de los beneficios de utilizar una herramienta de estas características en el aprendizaje de los alumnos, primero debemos contar con dicha herramienta. El propósito de este trabajo es brindar esta herramienta a la comunidad educativa, a través de un proceso que consta de tres etapas:

- Definir un lenguaje de programación que permita ejecutar código en paralelo.
- Generar las instrucciones Simplesem que representan la semántica de dicho lenguaje.
- Desarrollar un intérprete que permita controlar la ejecución de estas instrucciones.

Dejamos para un estudio posterior el análisis de los beneficios obtenidos al utilizar esta herramienta en el proceso de aprendizaje.

La principal motivación de este proyecto es utilizar todo lo aprendido para generar una herramienta útil para futuros estudiantes y docentes, que se apoye en la tecnología para facilitar y mejorar el proceso de aprendizaje.

1.2. Alcances y Objetivos del Proyecto

El objetivo principal de este proyecto es generar una herramienta educativa destinada a la enseñanza de conceptos propios de los lenguajes de programación, que permita mostrar el funcionamiento de un modelo paralelo de memoria compartida. Luego de ello, pretendemos que la herramienta sea utilizada en instituciones educativas relacionadas a la enseñanza de la computación, con la intención de que a futuro pueda realizarse un análisis de los resultados obtenidos al incorporar esta herramienta al proceso de aprendizaje.

Podemos plantear entonces los siguientes objetivos generales:

- Generar una herramienta educativa que simplifique la enseñanza de los lenguajes de programación.
- Mostrar el funcionamiento de un modelo paralelo de memoria compartida de manera sencilla.
- Distribuir la herramienta generada junto con el código fuente, para que sea utilizada en las instituciones que lo requieran, y para que pueda realizarse un estudio sobre el impacto que logra en el proceso de aprendizaje de los alumnos.

Para ello es necesario cumplir con objetivos particulares, los cuales serán:

- Definir la sintaxis y la semántica de un lenguaje de alto nivel, que brinde la posibilidad de escribir código para ser ejecutado en paralelo.
- Desarrollar un parser para analizar el código escrito en el lenguaje propuesto, y generar el árbol de parseo.
- Obtener las instrucciones en lenguaje Simplesem a partir del árbol de parseo.
- Desarrollar una máquina virtual que permita ejecutar las instrucciones simplesem e interactuar con las distintas partes del sistema.

Marco Teórico

En el siguiente capítulo se introducen los contenidos teóricos necesarios para lograr la comprensión del trabajo realizado. Los temas aquí tratados pueden ser conocidos, incluso en mayor profundidad, por el lector, por lo que podrá saltarse este capítulo de ser este el caso.

Se abordan 4 tópicos generales que deberán comprenderse antes de estudiar la implementación de la herramienta, a saber: Lenguajes de programación; Proceso de Compilación; Expresiones Regulares; y Modelo Simplesem.

2.1. Lenguajes de Programación

En los comienzos de la computación, el desarrollo de software estaba orientado a resolver problemas científicos que necesitaban realizar gran cantidad de cálculos matemáticos, que requerían tiempo considerable para ser realizados por el hombre. Los primeros programas a menudo eran escritos por una sola persona, y resultaban simples comparados a los de hoy en día.

La intención de extender las aplicaciones de las computadoras llevó a la necesidad de incrementar la complejidad tanto de los lenguajes de programación, como de los entornos de desarrollo, de manera tal que pudiera trabajar un equipo de profesionales en forma clara y ordenada. Además, debía mejorarse la capacidad de transmitir a los miembros del equipo un mensaje en común sobre lo que se estaba intentando lograr.

Las fases de obtención de requerimientos y diseño del software adquirían cada vez más importancia, ya que debía existir una manera de comunicarse tanto con el cliente como con los demás integrantes del equipo de desarrollo. La necesidad de distribución del trabajo dio origen a la programación orientada a objetos, un paradigma que se basa en la representación de la realidad en un mundo virtual, donde cada objeto es crea-

do con un fin, y se relaciona con los demás objetos para lograr un fin en común.

Abordaremos la manera de definir formalmente un lenguaje, utilizando para ello un conjunto de definiciones más elementales, que componen lo que finalmente conocemos como `Lenguaje`.

2.1.1. Alfabeto

Todo lenguaje, como el que estamos utilizando para escribir este texto, está formado por un conjunto de símbolos. Estos son elementos indivisibles que representan la mínima parte de un lenguaje. El conjunto de símbolos que conforman un lenguaje se conoce como alfabeto, y para representar dicho alfabeto utilizaremos la notación Σ. Conocemos a los símbolos que conforman el alfabeto español como letras, por lo tanto podemos definir $\Sigma_{esp} = \{a, b, c...z\}$.

Los símbolos del alfabeto no tienen sentido por si mismos, pero colocados en un determinado orden, forman lo que se conoce como palabra. Utilizaremos la letra ω para representar una palabra, y la notación $|\omega|$ para representar la longitud de una palabra, dada por la cantidad de símbolos que la componen. El conjunto de todas las palabras que se pueden formar con un alfabeto Σ se denota como Σ^*. Este conjunto es infinito, pero enumerable. Más adelante, cuando abordemos las expresiones regulares, veremos que estas son utilizadas para definir conjuntos Σ^* dado un alfabeto.

Una de las operaciones más importantes de las palabras es la concatenación. Supongamos $\omega = hola$ y $\upsilon = mundo$ entonces $\omega\upsilon = holamundo$. Si agregamos a nuestro alfabeto el espacio, podremos entonces formar secuencias de palabras separadas por espacios, conocidas como frases o sentencias.

2.1.2. Lenguajes y la jerarquía de Chomsky

Un lenguaje se define como un conjunto de palabras, lo cual nos permite aplicar todas las operaciones de conjuntos (union, intersección, diferencia). Además se define la concatenación de lenguajes, escrita como L1 • L2 de la siguiente manera: $L1 \bullet L2 = \{\omega | \omega = xy, x \in L1, y \in L2\}$. Nótese que para ello es necesario que la concatenación de palabras (xy) esté definida.

En el año 1965, Chomsky propuso dividir los lenguajes en una jerarquía inclusiva de clases de lenguajes [Chomsky65]. La Figura 2.1 muestra las clases de lenguajes definidas por la jerarquía de Chomsky.

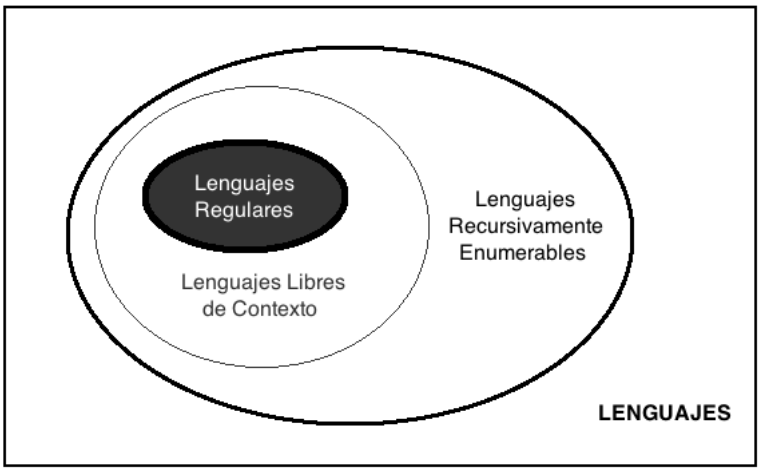


Figura 2.1: Clases de lenguajes según la jerarquía de Chomsky.

Las palabras que componen los lenguajes regulares se caracterizan por estar formados de repeticiones de los mismos componentes. De acuerdo a la jerarquía de Chomsky, los lenguajes regulares son aquellos que pueden ser definidos por una gramática regular. Los lenguajes regulares sobre un alfabeto Σ están definidos recursivamente por las siguientes reglas:

- El conjunto vacío Φ es un lenguaje regular.
- Para cada $x \in \Sigma$, el lenguaje dado por x es un lenguaje regular.
- Si A y B son lenguajes regulares, entonces $A \cup B$, $A \bullet B$ y A^* son lenguajes regulares.
- No existen más lenguajes regulares sobre Σ.

Todo lenguaje regular está definido por una gramática regular. Estas se dividen en gramáticas regulares derechas e izquierdas. Una gramática regular derecha es una gramática formal (N, Σ, P, S) donde:

- N es un conjunto de símbolos no terminales.
- Σ es un conjunto de símbolos terminales, que es disjunto del conjunto N.
- P es un conjunto de reglas de producción de la forma:
  - $B \to a$ tal que $B \in N$ y $a \in \Sigma$.
  - $B \to aC$ tal que $B, C \in N$ y $a \in \Sigma$.

- $B \to \varepsilon$ tal que $B \in N$ y ε es la cadena vacía.

- $S \in N$ y representa el símbolo inicial.

Una gramática regular izquierda es una gramática formal (N, Σ, P, S), que se diferencia de la gramática regular derecha en que el conjunto P es un conjunto de reglas definidas de la siguiente manera:

- $A \to a$ tal que $B \in N$ y $a \in \Sigma$.

- $A \to Ba$ tal que $B, C \in N$ y $a \in \Sigma$.

- $B \to \varepsilon$ tal que $B \in N$ y ε es la cadena vacía.

El alcance de este trabajo se limita a la definición de un lenguaje regular, por lo que no se realizará una definición detallada de las demás categorías. Desde un punto de vista práctico, los lenguajes más útiles son aquellos que tienen una representación finita, por lo que los demás lenguajes son sólo de interés teórico. Mas adelante veremos que cada una de estas clases de lenguajes está asociada a un tipo de "autómata" capaz de procesarlos. Esto ha hecho pensar que las categorías de lenguajes de Chomsky no son completamente arbitrarias, sino que se relacionan a las posibilidades que un lenguaje brinda, de acuerdo a su autómata asociado.

2.1.3. Autómatas finitos

Los autómatas finitos, o máquinas de estado finitas, son abstracciones matemáticas que representan una secuencia de eventos que ocurren sobre una entrada, para producir una salida. En nuestro caso, para representar un lenguaje, utilizaremos un alfabeto, un conjunto de estados y un conjunto de transiciones que obtienen un nuevo estado a partir del estado actual, y del símbolo siguiente en la cadena de símbolos.

El funcionamiento de los autómatas finitos para determinar la pertenencia de una palabra a un lenguaje es el siguiente. Al comenzar el reconocimiento de una cadena de caracteres, el autómata se encuentra en un estado inicial. A medida que procesa cada símbolo, se va cambiando de estado a través de la función de transición, que como dijimos depende del estado actual y del símbolo que se está procesando. Cuando se terminan de procesar todos los símbolos, el autómata se detiene en un estado que, de ser un estado de aceptación, indica que la cadena pertenece al lenguaje reconocido por el autómata. Si bien el estado inicial es único, un autómata puede tener múltiples estados finales o de aceptación. También puede darse el caso de que

un estado final corresponda al mismo estado inicial. Este enfoque nos permite determinar que el alcance de un lenguaje estará determinado por la cantidad de estados finales o de aceptación que el autómata que lo representa posee.

Una máquina de estados que represente un lenguaje regular, puede ser visualizada como un dispositivo con tres elementos:

- Una cinta de entrada que contiene la cadena de símbolos a procesar.
- Un cabezal de lectura (y eventualmente escritura) que toma el símbolo que debe ser procesado.
- Un mecanismo de control, que implementa las funciones de transición entre estados.

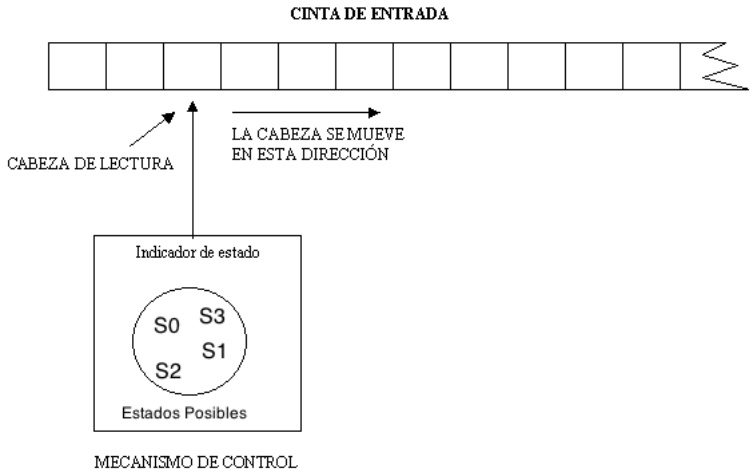


Figura 2.2: Componentes de un autómata.

El cabezal lector irá tomando los símbolos que componen la palabra de entrada desde la cinta. Cada vez que se lee un carácter, la información se manda a la unidad de control, la cual determina el estado siguiente. Debe existir algún mecanismo para indicar cuándo finaliza la entrada de caracteres, y un conjunto de estados finales. Cuando se detecta el fin de una palabra, debe determinarse si la máquina se encuentra en un estado final, en cuyo caso la palabra será aceptada como perteneciente al lenguaje. Desde un punto de vista teórico, si el conjunto de estados fuera infinito, y la cinta de entrada también lo fuera, todas las palabras pertenecerían al lenguaje, por lo que no habría límites de expresión con dicho lenguaje. Este es el modelo que se plantea en la máquina de Turing.

2.1.3.1. Definición formal

Una máquina de estados finitos M es un quíntuplo $(K, \Sigma, \delta, s, F)$ donde:

- K es un conjunto de identificadores de estados.
- Σ es el alfabeto de entrada.
- $s \in K$ es el estado inicial.
- $F \subseteq K$ es un conjunto de estados finales.
- $\delta : Kx\Sigma \to K$ es la función de transición, que a partir de un estado y un símbolo del alfabeto, obtiene un nuevo estado.

Como dijimos, los autómatas se utilizan para determinar si una palabra corresponde a un lenguaje, y para que esto ocurra, deben cumplirse las siguientes condiciones:

- Consumir todos los caracteres de la entrada, siguiendo las transiciones y pasando de un estado a otro.
- Una vez consumidos todos los caracteres, el autómata debe encontrarse en un estado final.

Ya hemos visto cómo podemos determinar si una palabra corresponde a un lenguaje. A partir de ahora abordaremos la definición de un lenguaje de programación. En estos, una palabra corresponde a una instrucción, que puede ser o bien una declaración de una variable, una llamada a una función, una expresión matemática, etc. Nuestro trabajo consistirá entonces en definir, mediante el uso de expresiones regulares, cómo deben estar formadas dichas instrucciones. Esto conformará la sintaxis del lenguaje, que nos permitirá decir si una determinada cadena de caracteres corresponde a una instrucción del lenguaje de programación.

Retomando el concepto de gramática regular, podemos decir ahora que existe una relación directa entre las reglas de una gramática formal izquierda estricta, y aquellas de un autómata finito, tal que la gramática genera exactamente el mismo lenguaje que acepta el autómata finito.

2.1.4. Sintaxis y Semántica.

Más allá de las capacidades de cualquier lenguaje de programación, todos están definidos con un conjunto de reglas que definen la forma que debe

tener una sentencia, como por ejemplo si debe colocarse un punto y coma al final de cada línea, o si deben delimitarse los bloques de instrucciones con llaves "{" y "}". Estas reglas son las que nos permiten distinguir un programa escrito en C++, por ejemplo, de otro escrito en Java.

El conjunto de reglas que definen la estructura de los programas en un determinado lenguaje, se conoce como `sintaxis` del lenguaje. No debe confundirse la sintaxis con la **semántica**. De acuerdo a [Ruiz01], la sintáxis y la semántica de un lenguaje se define de la siguiente manera:

- La sintaxis de un lenguaje de programación está relacionada con las reglas para escribir una sentencia válida en el lenguaje.
- La semántica de un lenguaje de programación está relacionada con el significado de los programas; por ejemplo, cómo ellos se comportarán cuando se ejecutan en una computadora.

La sintaxis de un lenguaje influye en cómo los programas son escritos por el programador, leídos por otro programador y traducidos por la computadora. La semántica de un lenguaje determina cómo los programas son compuestos por el programador, entendidos por otros programadores e interpretados por la computadora. Indica cómo se comporta el lenguaje en un determinado modelo, que puede ser matemático, lógico o computacional.

Veremos a continuación algunos aspectos de la semántica, que deberán ser tenidos en cuenta a la hora de definir el lenguaje de programación que se utilizará.

2.1.4.1. Variables

Las variables son el elemento principal de un lenguaje de programación. A cada variable le corresponde a una región de memoria que es usada para almacenar valores que son manipulados por el programa. Dentro del código fuente, accedemos a dichas variables a través de su nombre, el cual luego será traducido a la dirección de memoria que se le ha asignado.

Para todos los lenguajes existen reglas sintácticas sobre cómo las variables deben ser nombradas. Por ejemplo en lenguajes como C++, los nombres de variables deben comenzar con una letra o un guión bajo.

La declaración de una variable tiene un conjunto de propiedades semánticas asociadas. Al definir un lenguaje de programación, deberán tenerse en cuenta las siguientes propiedades para evitar ambigüedades.

- **Scope:** el alcance de una variable hace referencia a qué parte del programa tiene acceso a dicha variable para la interpretación semántica del mismo. Supongamos el ejemplo del Código 2.1.

Código 2.1: Alcance de la variable "j"

```
1 int j;
2 void main(){
3   int i = 0;
4   j = 20;
5   for(int j = 0; j < 5; j++){
6     i = j;
7   }
8   cout << i << j;
9 }
```

La línea 8, que imprime la salida en pantalla, imprimirá el número 420. Esto se debe a que existen dos variables con el mismo nombre en cuestión. La primera es una variable global, que fue declarada antes de ingresar a la función main, y por ende puede ser accedida desde main, para asignarle el valor 20.
Dentro del ciclo for se declara otra variable j, cuyo valor se inicializa a 0. La variable j que se modifica en cada iteración del ciclo for, corresponde a la que esta más "cercana" en el alcance, es decir a la que fue declarada dentro del ciclo for, por lo que la variable global no cambia. Al salir de este ciclo, la variable j a la que se hace referencia corresponde nuevamente a la variable global.
La variable i en este caso, es visible tanto desde fuera como desde dentro del ciclo for, por lo que se utiliza para almacenar el valor que vaya adquiriendo la variable j, interna al for, para luego imprimir su valor fuera del ciclo.

- **Tipo:** Cada variable puede almacenar datos que sean tratados de manera distinta en el programa. Puede ocurrir que al declarar una variable, sea necesario indicar qué tipo de valores guardará dicha variable. De esta forma, el programa entenderá de manera distinta un conjunto de bytes almacenados en memoria si estos corresponden a un entero (int) o a un número de punto flotante (float).
Los lenguajes en los que se debe declarar el tipo de variable se conocen como *lenguajes fuertemente tipados o de tipado estático*. Algunos ejemplos de estos lenguajes son C++, Java o Pascal.

Existen otros lenguajes en los que no se necesita declarar el tipo de variables, sino que se indica que existirá una variable, y el intérprete o el compilador serán los encargados de saber qué tipo de valores almacenan. Estos se conocen como *lenguajes débilmente tipados o de tipado dinámico*, y algunos ejemplos son Perl, Python o Smalltalk. El tipado dinámico permite que una misma variable almacene valores de distintos tipos en distintos momentos del programa.

- **Tiempo de vida:** esta propiedad hace referencia a cuándo una variable es creada en memoria y cuándo es destruida. Existen lenguajes como C/C++ u Objective-C que cuentan con mecanismos que permiten reservar y liberar memoria explícitamente cuando el programador lo desee.

2.1.4.2. Valores y referencias

Como dijimos anteriormente, cuando se define una variable se reserva un lugar en la memoria donde se almacenará el valor de dicha variable en un momento determinado del programa.

Si tenemos una asignación de la forma "`x=y;`", nos estamos refiriendo a la variable `x` de una manera, y a la variable `y` de otra. Esta asignación implica tomar el valor que hay almacenado en la posición de memoria que se reservó para la variable `y` y guardarlo en la posición de memoria que se le asignó a la variable `x`, por lo tanto nos interesa el *valor* almacenado en `y`, y la *referencia* a `x`, es decir la posición de memoria que se reservó para esta variable. No interesa en la asignación, qué valor tenía `x` antes, sino solo una referencia al lugar donde se almacena `x`.

Puede ocurrir también que nos interese guardar en `x` la dirección de memoria asignada a `y`. En tal caso, se utilizará una referencia tanto a `x` como a `y`. Las variables que almacenan direcciones de memoria en lugar de valores de algún tipo, se conocen como *punteros*. En este caso, `x` no guardaría el valor de la variable `y`, sino la posición de memoria donde esta variable es almacenada.

Al definir la sintaxis de un lenguaje de programación, debe tenerse en cuenta cuándo se está utilizando una referencia a una variable, y cuándo su valor. En el caso de una asignación a una variable que no sea un puntero, se utilizará el valor del operando de la derecha (conocido como *r-value*, que se refiere al contenido de una posición de memoria), y la referencia del operando

de la izquierda (conocido como *l-value*, que se refiere a una posición de memoria).

2.1.4.3. Expresiones

La última propiedad que veremos son las expresiones. Estas son construcciones sintácticas que permiten combinar valores y operaciones para obtener nuevos valores. La sintaxis del lenguaje nos determina cómo deben escribirse las expresiones, por ejemplo, operador1, operando, operador2. El sentido que tiene esa construcción sintáctica está dado por la semántica del programa, es decir, si el operador es "+", deberá sumarse el operador1 y el operador2, y devolver el resultado.

En la definición del lenguaje podría definirse que el símbolo "+" no realice la operación de suma cartesiana, sino una suma ponderada de los términos, o cualquier otra operación como una concatenación de cadenas, si ambos operadores fueran cadenas de caracteres. Nótese que el significado de las expresiones está dado por el compilador o el intérprete del lenguaje de programación, que será quien implemente las funcionalidades de cada expresión.

No debe confundirse la semántica del lenguaje con la semántica del programa. Para aclarar esto, veamos qué ocurre cuando se define una función que concatena dos cadenas de caracteres. Como se definió una función, la `semántica del lenguaje` indicará que deberá generarse el código correspondiente, almacenarse una referencia al comienzo de la función en una tabla de símbolos, y ejecutar dicho código cada vez que la función sea llamada. Por otro lado, la `semántica del programa` indica que cada vez que se llame a dicha función, se obtendrá una cadena de caracteres formada por dos cadenas unidas. La semántica del lenguaje compete al diseñador del mismo y al compilador, mientras que la semántica del programa compete a los programadores que utilizan dicho lenguaje.

2.1.5. Definición de un lenguaje de programación

Ya sabemos qué se entiende por sintaxis y semántica de un lenguaje. Veremos ahora de qué manera se definen formalmente dichos aspectos del lenguaje, con la ayuda de un ejemplo simple.

La definición de cualquier lenguaje de programación debería permitir determinar dos cosas de manera inequívoca. La primera, si un programa

está correctamente escrito en el lenguaje, y la segunda, cuál es el significado de ese programa, y qué efectos tendrá sobre el sistema. Como vemos, debe definirse tanto la *sintaxis* como la *semántica* del lenguaje.

2.1.5.1. Sintaxis

La sintaxis se describe a través de un conjunto de reglas que definen la estructura del lenguaje. Definen cómo deben estar formadas las instrucciones, como secuencias de elementos básicos llamados *palabras*. Usando estas reglas, conocidas también como *reglas de producción*, podemos determinar si una instrucción es correcta o no. Dichas reglas pueden verse como un árbol de sintaxis, en donde, donde las hojas contienen los símbolos terminales y las ramas contienen sentencias cada vez más complejas a medida que se asciende.

Los lenguajes de programación tienen palabras reservadas y símbolos reservados, como así también operadores o identificadores, que deben ser combinados de una manera específica para formar una construcción válida del lenguaje. Otra complicación que se agrega a la definición del lenguaje, es que las palabras están a su vez formadas por caracteres de un alfabeto, y debemos definir cómo esos caracteres deben ser utilizados para formar las palabras. Un ejemplo común de esto es la utilización de mayúsculas y minúsculas, ya que debe definirse si estas se tratarán de forma indistinta, o no. En algunos lenguajes como C o Java, se distingue entre mayúsculas y minúsculas en los nombres de variables, y si bien un nombre puede estar formado por letras, números y guiones bajos, no puede comenzar por un número.

A continuación veremos un ejemplo de la definición de la sintaxis de un lenguaje simple, que se utiliza para realizar una suma aritmética y almacenarla en una variable.

Código 2.2: Definición formal de la sintaxis de un lenguaje

```
1  start: statements
2
3  statements: statement ';' statements | statement
4
5  statement:  variable '=' expresion
6
7  expression: factor '+' factor
8
9  factor: number | variable
10
```

```
11 number:      /\d+/
12
13 variable:    /[a-zA-Z]+/
```

El árbol de la sintaxis de este lenguaje puede verse en la Figura 2.3. En la definición de la sintaxis se utilizan expresiones regulares, tema que trataremos con mayor profundidad más adelante. Como dijimos, las hojas del árbol definen los elementos más simples, en este caso las *variables* y los *números*. Los nombres de las variables en este lenguaje sólo pueden estar formados por letras, tanto mayúsculas como minúsculas. A partir de los elementos más simples, debemos comenzar el análisis del lenguaje. Un *factor* puede ser tanto un *número* como una *variable*. Una *expresión* está formada por un *factor* seguido de un signo '+' y seguido de otro factor. Un *statement* está formado por una *variable* seguida de un signo '=' seguido de una *expresión*. La regla *statements* está definida de manera recursiva como un *statement* seguido de otro conjunto de *statements*, o bien un *statement* individual, que corta la recursión. Finalmente, el programa está compuesto por un conjunto de *statements*, que constituyen el punto de entrada al programa. El siguiente es un ejemplo de un programa que corresponde al lenguaje definido:

Código 2.3: Programa escrito en el lenguaje definido en el Código 2.2

```
1 three=2+1;
2 six=three+3
```

Como se vio en la definición del lenguaje, algunas entidades son escritas entre comillas. Estos son llamados *terminales* del lenguaje, y corresponden a las palabras o símbolos propios del lenguaje. En la definición formal planteada, se incluyen todas las reglas sintácticas del lenguaje.

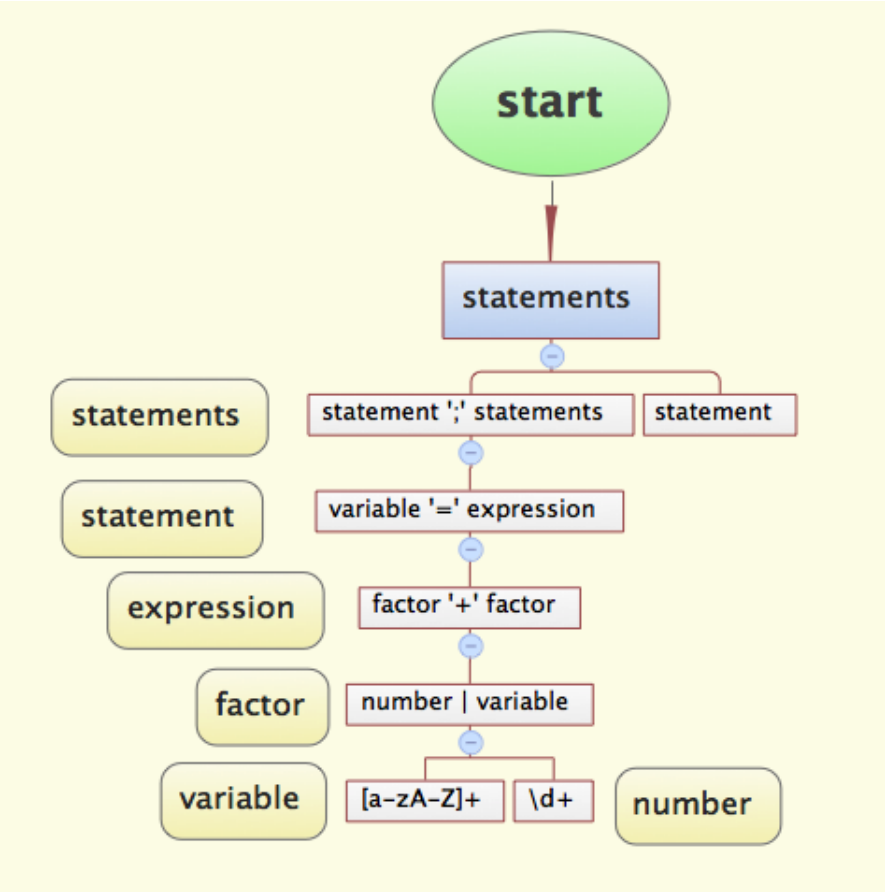


Figura 2.3: Árbol de la sintaxis del lenguaje.

2.1.5.2. Semántica

La sintaxis define programas correctos de un lenguaje. La semántica define el significado de programas sintácticamente correctos en un lenguaje. Cuando hablemos de semántica, tomaremos un enfoque operacional, desde el cual se utiliza el comportamiento de un procesador abstracto simple para describir los efectos de cada construcción del lenguaje. Para comenzar a entender este enfoque, presentamos un ejemplo tomado de [Ghezzi96, página 57].

La semántica de C nos indica que la declaración `int vector [10];` implica que se reserve memoria para diez elementos enteros, y la misma semántica de C nos indica que podemos referenciar dichos elementos con el nombre del arreglo `vector` y un índice. La semántica entonces es la que determina qué tarea realizar ante una expresión del lenguaje.

Vimos una manera de definir la sintaxis de un lenguaje de manera formal, utilizando reglas de producción. Ahora debemos dar un sentido a cada construcción sintáctica (es decir, definir la semántica operacional). Por el momento sólo describiremos las operaciones que deben realizarse ante cada sentencia. Luego, cuando presentemos las instrucciones Simplesem, veremos que estas pueden ser utilizadas para realizar las operaciones indicadas e implementar así la semántica. Veremos a continuación las operaciones que deben realizarse para cada regla del lenguaje presentado:

Código 2.4: Semántica operacional de un lenguaje

```
1  start: statements
2
3  statements: statement ';' statements | statement
4
5  statement:  variable '=' expresion
6  {
7    - Reservar una posición de memoria para almacenar la
       variable.
8    - Resolver la expresión.
9    - Guardar el resultado en la posición de memoria
       asignada a la variable.
10   - Volver a la regla superior.
11 }
12
13 expression: factor '+' factor
14 {
15   - Tomar el primer factor y guardarlo en un registro.
16   - Tomar el segundo factor y guardarlo en otro
       registro.
17   - Sumar los dos factores.
18   - Devolver el resultado a la regla superior.
19 }
20
21 factor: number | variable
22 {
23   - Devolver el número o la variable a la regla
       superior.
24 }
25
26 number:     /\d+/
27 {
```

```
28   - Devolver el número a la regla superior.
29 }
30
31 variable:   /[a-zA-Z]+/
32 {
33   - Devolver la variable a la regla superior.
34 }
```

Como vemos en la definición de la semántica, no todas las reglas sintácticas tienen un significado asignado. Esto se debe a que algunas reglas están compuestas por reglas más simples, y su significado se desprende de estas últimas. Ya hemos visto cómo definir la sintaxis y la semántica de manera formal. Ahora veamos cómo se procesa un programa escrito en un lenguaje determinado.

2.1.6. Procesamiento del código

Aunque en teoría es posible construir computadoras de propósito especial que permitan la ejecución directa de programas escritos en un lenguaje particular, las computadoras de hoy en día ejecutan de manera directa, sólo un lenguaje de muy bajo nivel, conocido como lenguaje máquina. Estos lenguajes son difíciles de comprender, de mantener y propenso a errores por parte del programador.

Los lenguajes máquina son diseñados para optimizar la velocidad de ejecución, el costo de realización y la flexibilidad para construir capas de software sobre ellos. En cambio los lenguajes de programación a menudo son diseñados teniendo en cuenta la facilidad de programación, mantenimiento y extensión del código. El problema surge entonces cuando pensamos cómo un código de un lenguaje de alto nivel puede ser ejecutado en una computadora cuyo lenguaje máquina es muy diferente, y a más bajo nivel. Bajo esta premisa, los lenguajes de alto nivel puede dividirse en *lenguajes interpretados*, o *lenguajes compilados*.

2.1.6.1. Lenguajes Interpretados

Los lenguajes de este tipo se basan en la ejecución directa del código. Usualmente se implementa un subprograma escrito en lenguaje máquina (o compilado al mismo) para cada posible acción que exista, y es este subprograma el que se ejecuta cuando se encuentra una acción determinada. Entonces, la interpretación de un programa puede verse como una secuencia apropiada

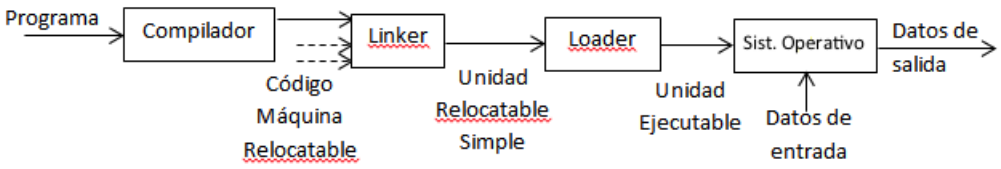


Figura 2.4: Proceso de ejecución para lenguajes compilados.

de llamadas a subprogramas escritos en lenguaje máquina. Precisamente, un intérprete es un programa que repetidamente ejecuta la siguiente secuencia:

- Tomar la siguiente instrucción.
- Determinar las acciones a ser ejecutadas.
- Llevar a cabo las acciones.

El intérprete de un lenguaje puede verse como una simulación de una máquina de propósito especial, cuyo lenguaje máquina es el lenguaje de alto nivel.

2.1.6.2. Lenguajes Compilados

Los programas escritos en lenguajes compilados requieren un paso previo antes de poder ser ejecutados. El código en lenguajes de alto nivel, debe ser *"traducido"* (compilado) a una versión equivalente en lenguaje máquina, comprensible por el procesador del dispositivo donde ese programa va a ser ejecutado (*target machine*). El proceso por el cual pasa un programa escrito en un lenguaje compilado, para ser ejecutado, es el siguiente.

Los módulos que conforman el programa primero deben ser traducidos por separado a código máquina relocatable. Esta traducción es llevada a cabo por el *compilador*. Luego, los módulos de código relocatable deben ser enlazados entre ellos en una única unidad relocatable. Esta tarea la realiza el *enlazador (Linker)*. Finalmente, el programa entero es cargado en memoria como código máquina ejecutable a través de un *cargador (Loader)*, y en general es el sistema operativo el que gestionará la ejecución de dicho programa. Gráficamente, el proceso necesario para ejecutar código de un lenguaje compilado puede verse en la Figura 2.4[Ghezzi96, pág. 65].

2.1.7. Paradigmas de Programación.

Existen diversos paradigmas de programación, a los que en distinta medida se adaptan los lenguajes existentes. Un paradigma provee la visión y los métodos que debe utilizar un programador en la construcción de un programa o subprograma para solucionar un problema determinado. Diferentes paradigmas resultan en diferentes estilos de programación, pero sobretodo en diferentes formas de pensar la solución a un problema. En general, los lenguajes de programación no suelen ser "puros", sino que soportan múltiples paradigmas.

Como dijimos, un paradigma define una forma de pensar una solución a un problema. Veremos un ejemplo del algoritmo Quick Sort en lenguaje C y en lenguaje Haskell, para demostrar de qué manera se piensa la solución con un paradigma procedural y con un paradigma funcional. Este ejemplo fue tomado de [Hernandez02].

Código 2.5: QuickSort implementado en C

```
1  qsort(a, lo, hi) int a[], hi, lo;
2  {
3    int h, l, p, t;
4
5    if (lo < hi) {
6      l = lo;
7      h = hi;
8      p = a[hi];
9
10     do {
11       while ((l < h) && (a[l] <= p))
12         l = l+1;
13       while ((h > l) && (a[h] >= p))
14         h = h-1;
15       if (l < h) {
16         t = a[l];
17         a[l] = a[h];
18         a[h] = t;
19       }
20     } while (l < h);
21
22     t = a[l];
23     a[l] = a[hi];
```

```
24      a[hi] = t;
25
26      qsort(a, lo, l-1 );
27      qsort(a, l+1, hi );
28    }
29  }
```

Código 2.6: QuickSort implementado en Haskell

```
1  qsort [] = []
2  qsort (x:xs) = qsort lt_x ++ [x] ++ qsort ge_x
3    where
4      lt_x = [y | y <- xs, y < x]
5      ge_x = [y | y <- xs, y >= x]
```

Como vemos, el algoritmo en lenguaje haskell puede ser implementado con mucho menos código que en C. Esto se debe a que en lenguaje C, al ser un paradigma principalmente procedural, debemos decirle a la máquina *cómo* debe hacer lo que nosotros queremos, mientras que Haskell, al ser un lenguaje funcional, debemos sólo indicarle *qué* debe hacer, es decir las funciones, y la máquina se encargará del cómo hacerlo. La manera de pensar la solución es completamente distinta en los dos paradigmas, teniendo que cambiar totalmente el enfoque del problema para encontrar una solución que en muchos casos puede ser más simple y más óptima. Veremos a continuación los principales paradigmas de programación existentes, haciendo énfasis en el modo en que debe pensar el programador al utilizar cada uno.

2.1.7.1. Imperativo o por Procedimientos

El paradigma imperativo o por procedimientos se basa en indicarle a la computadora *cómo* debe realizar una tarea, es decir, se le indican las instrucciones que deben ejecutarse. El lenguaje máquina de bajo nivel es inherentemente imperativo, ya que los programas están formados por un conjunto de instrucciones que el procesador ejecuta.

El programador debe pensar cuáles son los pasos que la computadora deberá seguir para solucionar un problema, e indicárselos utilizando un lenguaje de programación. Se corresponde con la idea que la mayoría tenemos sobre cómo solucionar un problema, siguiendo una serie de pasos que nos permitan ir avanzando hacia la solución final. Algunos ejemplos de lenguajes que soportan el paradigma procedural son: `C, Fortran o Pascal`.

2.1.7.2. Funcional

El paradigma de programación funcional es un paradigma declarativo basado en la utilización de funciones matemáticas. La programación funcional surge a partir del cálculo lambda, un sistema formal desarrollado en los años '30 para investigar la definición de función, la aplicación de funciones y la recursión.

La idea de este paradigma es pensar las soluciones a los problemas como una sucesión de funciones que deben realizarse. Utilizando funciones matemáticas, el programador debe indicarle a la computadora *qué* debe hacer, sin preocuparse por el cómo debe hacerlo. Los lenguajes de programación funcional, sobretodo aquellos lenguajes puramente funcionales, son utilizados casi exclusivamente en el ámbito académico, y no tanto en el desarrollo de software comercial.

Los programas escritos en un lenguaje funcional están formados por definiciones de funciones. No nos referimos aquí a funciones como subrutinas de un lenguaje imperativo, sino como funciones puramente matemáticas. En este tipo de lenguajes no existe la asignación de variables ni las construcciones estructuradas como la secuencia o la iteración (esto obliga a que en la práctica todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas). La no asignación de variables implica que se verifique la propiedad de transparencia referencial, que implica que el valor generado por una función depende exclusivamente de los argumentos pasados a la función. Esto se debe a que no hay variables que puedan tener valores distintos en distintos momentos del programa. Al eliminar los efectos secundarios se puede entender y predecir el comportamiento de un programa mucho más fácilmente, lo cuál es una de las principales ventajas de utilizar programación funcional. Un ejemplo de un lenguaje funcional puro es Haskell.

2.1.7.3. Lógico

La programación lógica está dentro de los paradigmas declarativos, al igual que el paradigma funcional. La diferencia con este último se basa en que la programación funcional tiene como unidad básica la función matemática, mientras que la programación lógica se basa en el concepto de predicado, o premisa.

Básicamente, lo que se hace es tener un conjunto de predicados, y a partir de ellos generar nuevos "conocimientos". El programador debe pensar

entonces cómo plantear estos predicados para que a partir de allí se generen nuevos predicados. La programación lógica suele encontrarse en aplicaciones de inteligencia artificial o relacionadas, donde se intenta generar nuevo conocimiento a partir del conocimiento anterior [Carloman76, capítulo IV].

2.1.7.4. Orientado a objetos

El paradigma orientado a objetos es quizás uno de los más difundidos de hoy en día, debido a su versatilidad a la hora de desarrollar software de cualquier escala. Se basa en la utilización de objetos que tienen un determinado estado (que define la identidad del objeto), y un comportamiento (definido por las operaciones que puede realizar).

El paradigma orientado a objetos se trata de que el programador piense una solución a un problema como si fuese el mundo real, dividiendo el problema en partes, y generando objetos cuyo fin sea solucionar cada una de estas partes. Todos estos objetos pueden ser programados por distintos programadores, por lo que este paradigma permite distribuir el trabajo entre múltiples desarrolladores. Un ejemplo ayudará a aclarar el concepto.

Supongamos que deseamos hacer una simulación de un mecanismo simple, por ejemplo una palanca que se utilice para levantar un cuerpo. Podemos dividir el problema en tres partes, la palanca, el cuerpo, y el agente que ejerce la fuerza para levantarlo. El agente tiene la responsabilidad de ejercer la fuerza (operación, o método, como se lo conoce formalmente). Además tiene una fuerza máxima que puede ejercer, y la fuerza que está ejerciendo en un determinado momento (estado). La palanca deberá recibir la fuerza ejercida por el agente, realizar un cálculo y aplicar la fuerza resultante sobre el cuerpo (método). El cuerpo finalmente deberá recibir la fuerza (método) y modificar su posición (estado) de acuerdo a otro cálculo.

Ahora bien, en el mundo real existen objetos que pueden agruparse bajo una misma categoría. En nuestro ejemplo, existen infinidad de cuerpos que podrían ser levantados por la palanca. La intención de agrupar objetos que realizan las mismas acciones y poseen las mismas propiedades se lleva al paradigma orientado a objetos con el concepto de *Clases*. En la práctica, lo que el programador define cuando utiliza el paradigma orientado a objetos son Clases, que representan al conjunto de todos los objetos que se agrupan por tener características y funcionalidades semejantes. Un ejemplo de lenguaje orientado a objetos puro es Smalltalk, pero el paradigma es soportado por la mayoría de los lenguajes más utilizados hoy en día.

2.1.7.5. Orientada a componentes

La programación orientada a componentes hace énfasis en la descomposición de un sistema en componentes funcionales o lógicos con interfaces bien definidas usadas para la comunicación entre componentes. Un componente es un elemento que cumple una función determinada, y que a la vez tiene la capacidad de comunicarse con otros componentes. Un concepto fundamental de este paradigma es la reusabilidad. A diferencia del paradigma orientado a objetos, el programador debe pensar a los componentes como entidades con un alto nivel de abstracción, que posean una interfaz para comunicarse con otros componentes, y que permitan su reutilización.

2.2. Proceso de compilación

En esta sección se estudiará el proceso de compilación. El trabajo realizado abarca desde la definición del lenguaje de programación, hasta la ejecución de código escrito por un programador, por lo que deberá comprenderse la manera en que se define un lenguaje, y cuál es el camino por el que pasa el código fuente hasta poder ser ejecutado por el procesador.

La Figura 1.1 que se presentó en la introducción muestra todo el proceso de compilación por el que debe pasar el código fuente hasta poder obtener código ejecutable por el procesador. En nuestro proyecto se utilizará un proceso simplificado, cuyos pasos pueden verse en la Figura 2.5.

La entrada al proceso de compilación, será desde luego el código fuente del programa. Lo primero que se hace es pasar el código por un *Scanner*, que se encarga de analizarlo para determinar si las palabras del código están bien escritas. Una vez que se determina que todas las palabras del código están bien escritas, se obtiene un flujo de *Tokens*. Los Tokens nos indicarán cuáles son las distintas porciones del código que se deberán tratar por separado. El flujo de Tokens ingresa al *parser* propiamente dicho, quien realiza un análisis de la sintaxis del lenguaje. El *parser* se encargará de verificar que las instrucciones estén correctamente formadas, detectando todo tipo de errores sintácticos de acuerdo a la definición formal que se haya realizado del lenguaje. El resultado del parser será un *árbol de parseo*. De este podemos obtener las distintas construcciones que contiene el código escrito, y la manera en que está formada una expresión.

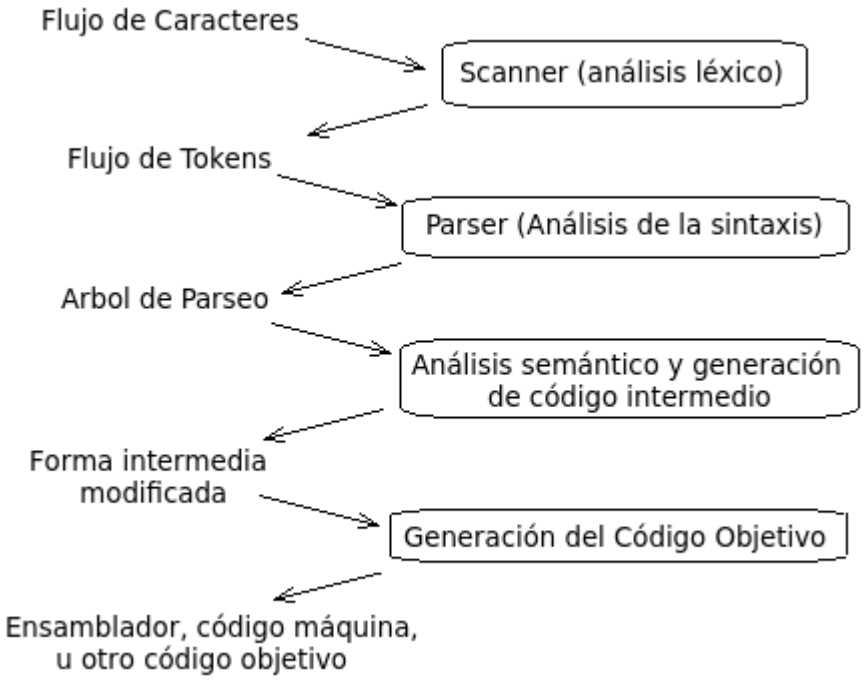


Figura 2.5: Proceso de compilación simplificado.

El árbol de parseo tiene una estructura semejante a la del árbol de sintaxis, salvo que en las hojas se encuentra un valor específico, y no un conjunto infinito de valores, como todas las variables o todos los números en el ejemplo mostrado anteriormente. A modo de ejemplo, tomemos la primera línea del Código 2.2: `three=3;`. El árbol de parseo quedará formado como se muestra en la Figura 2.6

Nótese que aquí necesitaremos una rama semejante a esta para cada nueva instrucción que se incluya en el código. Una vez que obtuvimos el árbol de parseo, el siguiente paso será realizar un análisis semántico para generar un código intermedio. En este trabajo llamaremos código intermedio al código en lenguaje Simplesem (que luego se explicará) en el cual todavía no conocemos la posición de memoria que se asignará a las variables del programa, ni las direcciones de destino de los saltos.

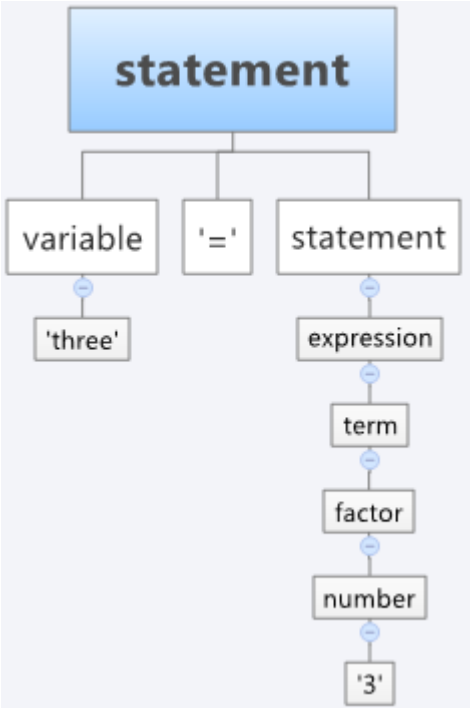


Figura 2.6: Árbol de parseo para una instrucción.

Al realizar el árbol de parseo, pueden determinarse los nombres de las variables y el alcance de las mismas (locales o globales), como así también las funciones involucradas en el programa, de manera tal que una vez finalizado el árbol de parseo y generado el código intermedio, podremos reemplazar los nombres de las variables y las direcciones de destino de los saltos con las posiciones de memoria efectivas, obteniendo de esa forma el código objetivo, que en este caso será código Simplesem.

2.3. Expresiones regulares (RegExp)

Ya vimos que un lenguaje puede ser representado por una máquina de estados finita, o por su gramática regular equivalente. Además vimos que el conjunto de palabras que forman un lenguaje puede ser infinito, pero enumerable. Debemos buscar alguna expresión que nos permita definir las máquinas de estado que procesarán el lenguaje, y para ello existen las expresiones regulares.

Las expresiones regulares permiten definir lenguajes regulares. Surgen de la teoría de conjuntos aplicada a los lenguajes. Las expresiones regulares

imponen ciertas condiciones que debe cumplir una palabra para pertenecer a un lenguaje. Desde un punto de vista teórico, nos permiten definir un autómata finito por cuyos estados pasarán los símbolos de una palabra, para determinar si dicha palabra pertenece o no al lenguaje..

A nivel práctico, las expresiones regulares se utilizan para procesamiento de texto, por ejemplo para reemplazar ciertas palabras, para eliminar múltiples espacios en blanco, para normalizar la entrada de texto, o bien para obtener ciertas partes que nos interesan de un texto. La mayoría de los lenguajes de programación de hoy en día incorporan algún mecanismo para trabajar con expresiones regulares en su estándar, o tienen librerías implementadas que lo permiten. En general, en todos los lenguajes la sintaxis definida para las expresiones regulares es similar. A continuación veremos cómo se utilizan las expresiones regulares en lenguaje Perl, para familiarizarnos con lo que veremos luego.

- **Delimitación de una expresión regular "/":** lo primero que necesitamos al trabajar con expresiones regulares es delimitar qué parte del código corresponde a una expresión regular y cuál a código fuente en lenguaje Perl. Las expresiones regulares van encerradas entre barras "/", por ejemplo `/esto es una expresión regular/`.

- **Operador =∼ y !∼ :** El operador =∼ permite buscar en una cadena de caracteres una expresión regular, devolviendo *True* si la expresión se encontró y *False* en caso contrario. Este operador también puede utilizarse para buscar la expresión en la cadena, y reemplazarla por otra en caso de encontrarla. Por el contrario, el operador `!∼` se utiliza para determinar si en una cadena de caracteres NO se encuentra una expresión regular, devolviendo *True* si la expresión no existe y *False* en caso contrario.

- **Operador ^ y $:** Si el operador `^` se encuentra al principio de la expresión regular, representa todas las cadenas que comiencen de una determinada manera. La expresión `/^Hola/` representa todas las cadenas de texto que comiencen por la palabra "Hola".

  El operador `$` se utiliza para representar las cadenas que terminen de una determinada manera. Por ejemplo la expresión `/$fin/` representa todas las cadenas finalizadas por la palabra "fin". Ambos caracteres pueden ser combinados para representar cadenas que empiecen y terminen de una determinada manera. La expresión `/^$/` puede utilizarse para representar una cadena vacía.

- **Operador *:** representa un carácter que puede aparecer cero o más veces. La expresión `/^H*/` representa todas las cadenas que comiencen por cero o más "H". Este operador se utiliza cuando un carácter puede existir o no. Por ejemplo en las expresiones matemáticas puede haber espacios separando los operandos del operador, pero también pueden no existir espacios, y la expresión seguirá siendo correcta.

- **Operador +:** representa un carácter que puede aparecer una o más veces. La expresión `/M+/` representa cadenas que tengan al menos una letra M mayúscula.

- **Operador ?:** indica que un carácter debe aparecer cero o una vez para pertenecer al conjunto. La expresión `/H?ola/` representa tanto a la palabra "Hola" como a la palabra "ola", pero no incluye por ejemplo a la palabra "HHola".

- **Operador . (punto):** este operador se utiliza como comodín, representando cualquier carácter excepto el salto de línea (\n). La expresión `/.+/` representará toda una oración hasta un punto y aparte (salto de línea).

- **Operador |:** permite listar un conjunto de posibles caracteres que pueden aparecer en la cadena. La expresión `/(H|h)ola/` representa la palabra "hola" con minúscula, u "Hola", con mayúscula.

- **Conjuntos de caracteres:** las expresiones regulares permiten representar conjuntos de caracteres para simplificar su definición. Los conjuntos de caracteres se representan entre corchetes "[" y "]". Por ejemplo, si se quiere representar cualquier cadena comenzada por vocal, puede utilizarse la expresión `/[aeiouAEIOU]/.*`.

  Si tenemos en cuenta la tabla de códigos ASCII, puede delimitarse un rango de caracteres utilizando el guión medio. La expresión `/[a-zA-Z]+/` representa cualquier cadena formada sólo por letras, mayúsculas y minúsculas, de al menos un carácter. Dentro de los conjuntos, puede utilizarse el operador ^ para indicar los caracteres que NO deben aparecer en la cadena. La expresión `[^abc]` indica que puede aparecer cualquier carácter excepto a, b o c.

- **Grupos de caracteres:** indica que un grupo de caracteres debe aparecer en el mismo orden. Los grupos de caracteres se colocan entre paréntesis. La utilización de grupos de caracteres suele utilizarse cuando quiere detectarse la repetición de una subcadena. Por ejemplo la

expresión `/(asd)+/` permite detectar cadenas como “asd”, “asdasd” o “asdasdasdasd”.

Hasta ahora vimos algunos operadores que nos permiten definir las expresiones regulares para abarcar un conjunto de expresiones de un lenguaje. Presentamos a continuación un conjunto de símbolos especiales conocidos como Backslash Sequences que representan un subconjunto de utilización frecuente, para no tener que definirlos nuevamente en cada aparición. La Tabla 2.1 muestra las backslash sequences y su significado.

\d	Conjunto formado por todos los dígitos del 0 al 9. Equivalente a [0-9].
\D	Conjunto formado por cualquier carácter excepto un dígito. Equivalente a [^0-9].
\s	Conjunto formado por cualquier carácter de espacio. Equivalente a [\t\n\f\r].
\S	Conjunto formado por cualquier carácter que no sea de espacio. Equivalente a [^\t\n\f\r].
\w	Conjunto formado por cualquier carácter alfanumérico y el guión bajo. Equivalente a [a-zA-Z0-9_].
\W	Conjunto formado por cualquier carácter que no sea ni alfanumérico ni guión bajo. Equivalente a [^a-zA-Z0-9_].

Tabla 2.1: Backslash Sequences

Como dijimos, las expresiones regulares son muy útiles cuando se desea procesar texto. En el desarrollo del trabajo, se utilizaron expresiones regulares para definir la sintaxis del lenguaje. Luego, una vez obtenido el código intermedio se utilizaron expresiones regulares para reemplazar los nombres de variables por las direcciones asignadas a cada una de ellas, y para reemplazar las direcciones de destino de los saltos. A continuación veremos algunas operaciones que posee el lenguaje Perl para el procesamiento de texto.

2.3.1. Búsqueda y sustitución de caracteres

Ya vimos cómo podemos armar expresiones regulares que representen un conjunto infinito de expresiones de manera formal. Ahora las utilizaremos con un fin, que será encontrar la ocurrencia de todas las expresiones que coincidan con la expresión regular. El operador **s///** realiza la búsqueda y sustitución de caracteres. El Código 2.7 muestra un ejemplo de utilización de este operador.

Código 2.7: Operador de sustitución **s///**

```
1 $cadena = ``Hola Hola Mundo'';
2 $cadena =~ s/Hola/Adios/;
3 #$cadena ahora contiene ``Adios Hola Mundo''.
```

El operador **s///** también puede contener sufijos que actúan como modificadores. Estos pueden ser:

- **g:** permite realizar un reemplazo global de la expresión, es decir en todas las ocurrencias de la expresión regular. Si en el ejemplo del Código 2.7 se hubiera utilizado el modificador **g** (s/Hola/Adios/**g**), $cadena contendría "Adios Adios Mundo".

- **e:** indica que la expresión de reemplazo es una operación matemática, y el resultado de dicha operación será la expresión que finalmente se utilice para reemplazar. En el Código 2.7, si se utiliza el operador `s/Hola/2+2/e`, la cadena resultante sería "4 Hola Mundo".

- **i:** este modificador sólo indica que no se distinga entre mayúsculas y minúsculas en la expresión de búsqueda. La expresión `s/hOLA/Adios/i` en la línea 2 del Código 2.7 producirá el reemplazo de la palabra "Hola" por "Adios" ya que no se distingue entre mayúsculas y minúsculas.

Estos tres modificadores pueden combinarse para reemplazar, por ejemplo, todas las ocurrencias de una palabra, sin importar si están escritas con mayúsculas o minúsculas. El operador `s/hola/2+2/gei` reemplazará todas las ocurrencias de la palabra hola, sin importar cómo esté escrita, por el número 4 (resultado de la expresión matemática).

2.4. Modelo Simplesem

En esta sección presentaremos la propuesta realizada por Carlo Ghezzi y Mehdi Jazayeri en su libro "Programming Language Concepts" sobre un procesador abstracto simple, que recibe el nombre de Simplesem. En realidad, la propuesta abarca más que el simple hecho de procesar instrucciones, puesto que involucra una forma de pensar los programas debido al impacto que un lenguaje de alto nivel genera sobre un lenguaje máquina de bajo nivel. De allí que en este trabajo se refiere a Simplesem como un modelo de ejecución, y no como un procesador abstracto solamente.

Cuando hablamos de la semántica de un lenguaje, dijimos que esta era la que determinaba qué significaba cada construcción del lenguaje para la computadora en su totalidad. La semántica operacional es la que indica, ante una determinada construcción, qué operación debe realizarse, cómo se debe trabajar con la memoria, cómo se deben tratar las iteraciones, etc.

Simplesem es un modelo de ejecución que nos permite describir dicha semántica a bajo nivel, es decir directamente dando instrucciones a la máquina, de manera similar a lo que suele hacerse con lenguaje ensamblador.

En su forma más básica, Simplesem consiste de un *puntero a instrucción (IP)* que apunta a la próxima instrucción que se debe ejecutar, una *memoria de datos (D)*, una *memoria de código (C)* y un *procesador*. Para simplificar el modelo, la memoria está organizada en celdas, y cada celda almacena una instrucción (memoria de código) o un valor entero (memoria de datos). El *puntero a instrucción* siempre apunta a alguna ubicación dentro de la *memoria de código*. Veremos más adelante que en la *memoria de datos* existen posiciones reservadas para la utilización de punteros con distintos fines.

El lenguaje cuenta con tres instrucciones compuestas (set, jump y jumpt), y una instrucción especial para indicar el final del programa (halt). Además existen dos registros especiales que se utilizan para dar soporte a la entrada/salida de datos, que son `read` y `write` respectivamente. A continuación se explica cada una de estas funciones.

2.4.1. Instrucción set

La instrucción *set* permite modificar el contenido de una celda de la memoria de datos. Está compuesta por dos parámetros: la dirección de la celda cuyo contenido quiere modificarse, y la expresión que evalúa el nuevo valor. Nótese que el segundo parámetro es una *expresión*, y no un valor, por lo tanto podría ser de la forma `1+5*D[20]`, que devolvería el valor de uno más cinco veces el valor que haya en la celda 20 de la memoria de datos. La instrucción

```
set 3, D[10] + 3
```

tiene el efecto de guardar en la celda 3 de la memoria de datos, el valor que haya almacenado en la celda 10 de la memoria de datos, más 3.

Al igual que en las asignaciones en lenguajes de alto nivel, el primer operando debe ser un *l_value*, es decir una dirección de memoria, que será el

destino donde se almacene el valor, mientras que el segundo operando debe ser un r_value, que será el valor que quiera almacenarse. Bajo estas reglas, cualquier posición de memoria puede utilizarse como puntero en el modelo Simplesem, ya que bastará con utilizar como l_value, un operando de la forma D[x] para tomar el valor de la celda x de la memoria de datos como una dirección de memoria. Esto se conoce como *direccionamiento indirecto*

El modelo Simplesem posee dos registros especiales que se utilizan para dar soporte a la *entrada/salida* de datos. Estos registros se conocen como `read` y `write`, y son utilizados en conjunto con la operación set. El registro `read` se utiliza como segundo operador de la instrucción set, e indica la toma de datos desde un dispositivo de entrada. La instrucción

```
set 4, read
```

lee un valor desde un dispositivo de entrada y lo coloca en la celda 4 de la memoria de datos.

Por otro lado, el registro write se utiliza para enviar un dato al dispositivo de salida. Así, la instrucción

```
set write, D[20]
```

se utiliza para enviar el contenido de la celda 20 de la memoria de datos, a un dispositivo de salida, por ejemplo la consola.

2.4.2. Instrucciones jump y jumpt

En el modelo Simplesem, la instrucción *jump* permite hacer un salto incondicional, y la instrucción *jumpt* permite realizar saltos condicionales. La primera instrucción recibe como parámetro una expresión cuyo resultado será el valor que adopte el IP, es decir, la celda de la memoria de código que contiene la siguiente instrucción a ejecutarse.

Por su parte, la instrucción *jumpt* recibe dos parámetros. El primero es una expresión cuyo resultado será la dirección de destino del salto, y el segundo es una expresión condicional, formada por dos expresiones y un operador de comparación. Si la condición es verdadera, se asigna el valor del primer parámetro al IP, en caso contrario, el IP no se modifica.

Al presentar el modelo simplesem, dijimos que el IP apunta a la próxima instrucción que se va a ejecutar. La máquina Simplesem opera siguiendo tres pasos fundamentales: (1) Búsqueda de la instrucción a ser ejecutada; (2)

Incremento del IP; (3) Ejecución de la instrucción buscada en (1). Ahora bien, como sabemos, la mayor parte de los programas contienen saltos. Estos saltos se basan en modificar el valor del IP (saltos no condicionales) o en evaluar una condición, y en caso de ser verdadera modificar el valor del IP (saltos condicionales). Incrementar el IP en el paso 2, permite que, si una instrucción modifica el IP, este no vuelva a ser modificado al finalizar la instrucción.

2.4.3. Instrucción halt

La instrucción halt es una instrucción especial que no recibe parámetros, y se utiliza únicamente para indicar que la ejecución del programa debe finalizar.

2.4.4. Estructura del modelo Simplesem

El modelo Simplesem es sumamente simple, y podrían irse agregando nuevas funcionalidades. La propuesta de Ghezzi y Jazayeri no sólo incluye lo explicado hasta aquí, sino que también incorpora un conjunto de lenguajes de complejidad ascendente, que permiten utilizar el modelo Simplesem para explicar el procesamiento de los lenguajes de programación en tiempo de ejecución.

En su libro, se presentan cinco lenguajes denominados C1, C2, C3, C4 y C5. Cada uno introduce nuevos conceptos que poseen los lenguajes de programación de propósito general, y que pueden verse reflejados en las instrucciones Simplesem que se generan. En nuestro estudio, sólo veremos hasta el lenguaje C3, ya que las características del lenguaje C4 y C5 no son necesarias para el propósito de este trabajo. Puede encontrarse más información a cerca del modelo Simplesem en [Ghezzi96, capítulo 2].

Los lenguajes de programación pueden clasificarse en distintas categorías de acuerdo a su estructura en tiempo de ejecución.

2.4.4.1. Lenguajes estáticos

Este tipo de lenguajes garantizan que los requerimientos de memoria para cualquier programa pueden ser calculados antes de que comience la ejecución del programa, permitiendo que toda la memoria necesaria sea reservada antes de que comience la ejecución. Este tipo de lenguajes no soporta recursión, ya que esto requiere una asignación dinámica de memoria,

de acuerdo al grado de profundidad de la recursión. Algunos ejemplos de este tipo de lenguaje son las primeras versiones de Fortran y Cobol.

2.4.4.2. Lenguajes basados en pilas

Esta clase de lenguajes implica un uso dinámico de la memoria, por lo que los requerimientos no pueden ser computados en tiempo de compilación. De todas maneras, el uso de la memoria es predecible en el sentido de que se comporta como una pila: el último registro de activación reservado, sera el primero en ser liberado.

Un registro de activación se utiliza cada vez que se llama a una función, y contiene información como los parámetros, la dirección de retorno, o el valor de retorno. Son estructuras de datos que almacenan la información de estado de la subrutina. Debido a que los registros de activación se encuentran en la memoria de datos, es posible utilizar la del modelo Simplesem para administrarla como una pila, permitiendo de esta forma emular el comportamiento de este tipo de lenguajes. El primer lenguaje que incorporó estas características fue Algol 60.

2.4.4.3. Lenguajes completamente dinámicos

Este tipo de lenguajes se caracteriza por tener un uso de la memoria impredecible en tiempo de compilación. Los datos son dinámicamente ubicados sólo cuando se necesitan durante la ejecución. El problema surge cuando queremos administrar eficientemente la memoria, ya que es difícil reconocer memoria que no esté en uso para reasignarla nuevamente.

2.4.5. Máquina de Turing

En los años 30, Alan Turing propuso un modelo de máquina abstracta como una extensión de los autómatas finitos. El modelo presentado constituye la máquina más poderosa de todas las máquinas planteadas hasta el momento. Decimos que una máquina M_A abstracta es más poderosa que otra M_B si el conjunto de lenguajes aceptados por M_B es un subconjunto de los lenguajes aceptados por M_A. A continuación veremos que la máquina de Turing puede aceptar todos los lenguajes posibles, incluido el lenguaje natural.

La máquina de Turing está compuesta por un control finito, un cabezal de lectura y escritura, y una cinta donde puede haber caracteres. La cinta es de longitud infinita hacia la derecha, pero no hacia la izquierda, es decir que tiene un comienzo pero no un fin. Las celdas que no poseen ningún carácter

se completan con un espacio en blanco.

Debido a que el cabezal puede ser utilizado para escritura, el contenido de la cinta puede ser modificado en tiempo de ejecución. El movimiento del cabezal es bidireccional, por lo que puede volverse sobre caracteres ya procesados. La operación de la máquina de Turing consta de tres pasos:

- Leer un carácter de la cinta.
- Efectuar una transición de estado.
- Realizar una acción e la cinta. Existen dos acciones posibles, mutumanete excluyentes, que son: escribir un símbolo en la cinta, o mover el cabezal hacia la izquierda o la derecha.

Al iniciarse la operación, la palabra de entrada se encuentra en la cinta. Esta palabra es el programa que desea ejecutarse. Decimos que en la máquina de Turing se llega al final del programa cuando se alcanza una instrucción especial llamada *halt*. Si se llega a esta instrucción se considera aceptada la palabra de entrada. Como vemos, a diferencia de los autómatas finitos no hay múltiples estados finales, sino uno solo. Si se quiere que una palabra no sea aceptada por el lenguaje, debe evitarse que la máquina de Turing llegue al estado final. El hecho de que todas las palabras puedan llegar al estado final, implica que todas las palabras pueden pertenecer al lenguaje, y por ende una máquina de Turing abarca todos los lenguajes posibles.

2.4.6. Lenguaje C1

El primer lenguaje propuesto por Ghezzi y Jazayeri, al que llamaron C1, es sumamente simple. Este lenguaje puede ser visto como una variante léxica de un subconjunto de C, que sólo posee tipos de datos simples e instrucciones simples (no tiene soporte para funciones). Sólo pueden utilizarse tipos de datos que permitan determinar los requerimientos de memoria de manera estática, como valores enteros y de punto flotante, arreglos de tamaño fijo, y estructuras.

El programa entero está formado por una rutina principal (`main()`), donde se declaran los datos que se utilizarán y un conjunto de instrucciones para operar con esos datos. La entrada y salida de datos se realiza a través de dos operaciones: `get` para leer de un dispositivo de entrada, y `print` para escribir en un dispositivo de salida. El Código 2.8 muestra un ejemplo de un programa escrito en lenguaje C1 tomado de [Ghezzi96, página 94].

Código 2.8: Programa escrito en lenguaje C1

```
1  main ()
2  {
3    int i, j;
4    get (i, j);
5    while (i != j)
6      if (i > j)
7        i -= j;
8      else
9        j -= i;
10   print (i);
11 }
```

La Figura 2.7 muestra las instrucciones Simplesem equivalentes al Código 2.8, y la manera en que se organiza la memoria de datos y la memoria de código.

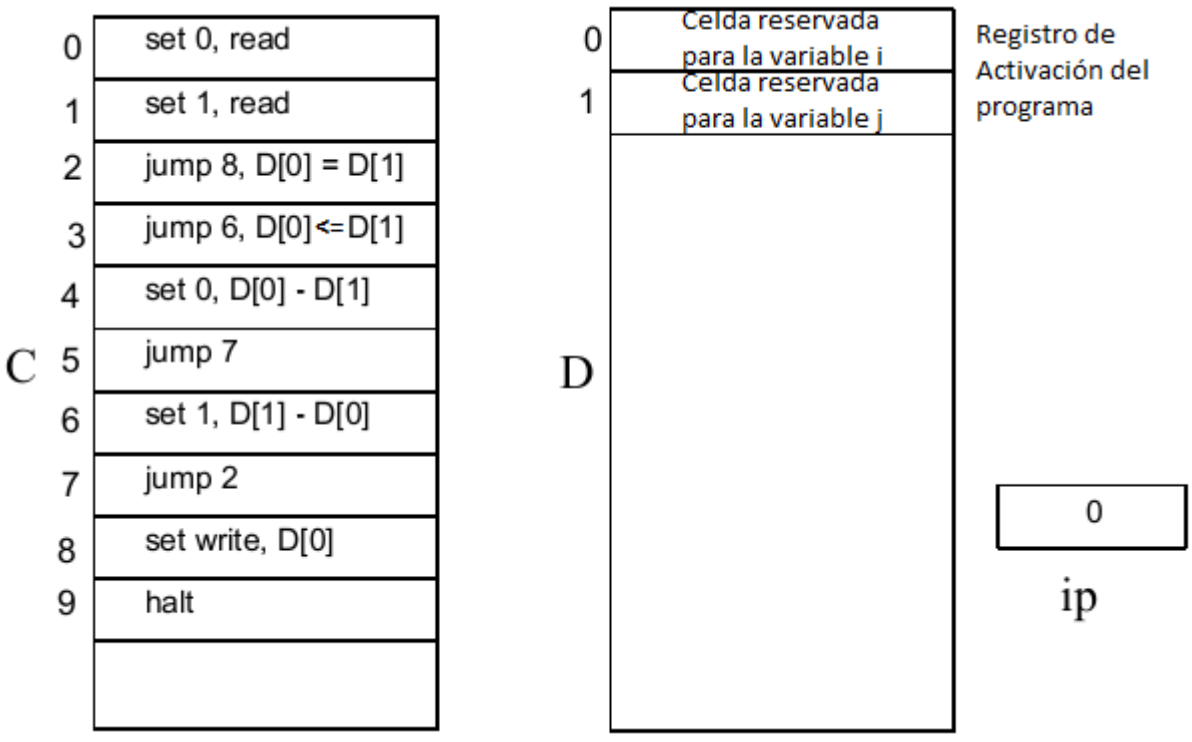


Figura 2.7: Instrucciones Simplesem equivalentes al programa 2.8

2.4.7. Lenguaje C2

El lenguaje C2 permite definir rutinas en un programa, y permite que en cada rutina se declaren datos locales. Un programa en lenguaje C2 está formado por la siguiente secuencia de elementos:

- Declaración del conjunto de datos globales (si no se requieren variables globales, esta parte puede no existir).
- La definición y/o declaración de las funciones (también puede no existir si no se utilizan funciones).
- Una función principal (`main()`) que contiene primero la declaración de variables locales y luego un conjunto de instrucciones que se ejecutarán en orden al comenzar la ejecución del programa. La función `main()` no puede ser llamada por otras funciones.

Cada función puede acceder a los datos locales y a cualquier dato global que no esté redeclarado dentro de la función.

Las funciones del lenguaje C2 no pueden recibir parámetros, no pueden entregar ningún valor de retorno, ni soportan llamadas recursivas. El lenguaje C3 eliminará estas restricciones. El Código 2.9 muestra un ejemplo de un programa escrito en lenguaje C2, con dos funciones declaradas. El ejemplo fue tomado de [Ghezzi96, página 95].

Código 2.9: Programa escrito en lenguaje C2

```
1  int i=1, j=2, k=3;
2  alpha()
3  {
4    int i=4, l=5;
5    ...
6    i+=k+l;
7    ...
8  };
9
10 beta()
11 {
12   int k=6;
13   ...
14   i=j+k;
15   alpha();
16   ...
```

```
17 };
18
19 main()
20 {
21   ...
22   beta();
23   ...
24 }
```

Con las restricciones que posee el lenguaje C2, el tamaño de los registros de activación de cada función puede ser determinado en tiempo de compilación, y el espacio en memoria para cada registro de activación puede ser reservado antes de la ejecución (*reserva estática de memoria*), ya que se conoce cuántas funciones existen, y cuántas veces serán llamadas. Esto permite que pueda asignarse una celda en la memoria de datos para cada variable del programa, ya sea local o global.

Un registro de activación se utiliza cada vez que se llama a una función. En cada llamada a función, en la memoria de datos se apila un registro de activación correspondiente a la función llamada. Cada vez que se vuelve de una función, su registro de activación es desapilado. Los registros de activación conforman en la memoria lo que se conoce como la pila de llamadas, o call stack.

En el lenguaje C2, puede desperdiciarse espacio debido a que se utiliza asignación estática de memoria, lo que implica que puede reservarse memoria para una función que nunca es llamada. La Figura 2.8 muestra el estado de la máquina Simplesem luego de ejecutar la instrucción de la línea 6.

La primera celda de cada registro de activación (offset 0) se reserva para la dirección de retorno, que contiene la dirección de la siguiente instrucción que deberá ser ejecutada al terminar la función. A partir de la celda con offset 1 de cada registro de activación, se reserva espacio para las variables locales. La función `main()` no contiene una dirección de retorno debido a que al finalizar esta función, finaliza el programa. En una computadora real, la función main es llamada por el sistema operativo, por lo que una vez finalizada la ejecución del programa, se devuelve el control al sistema operativo.

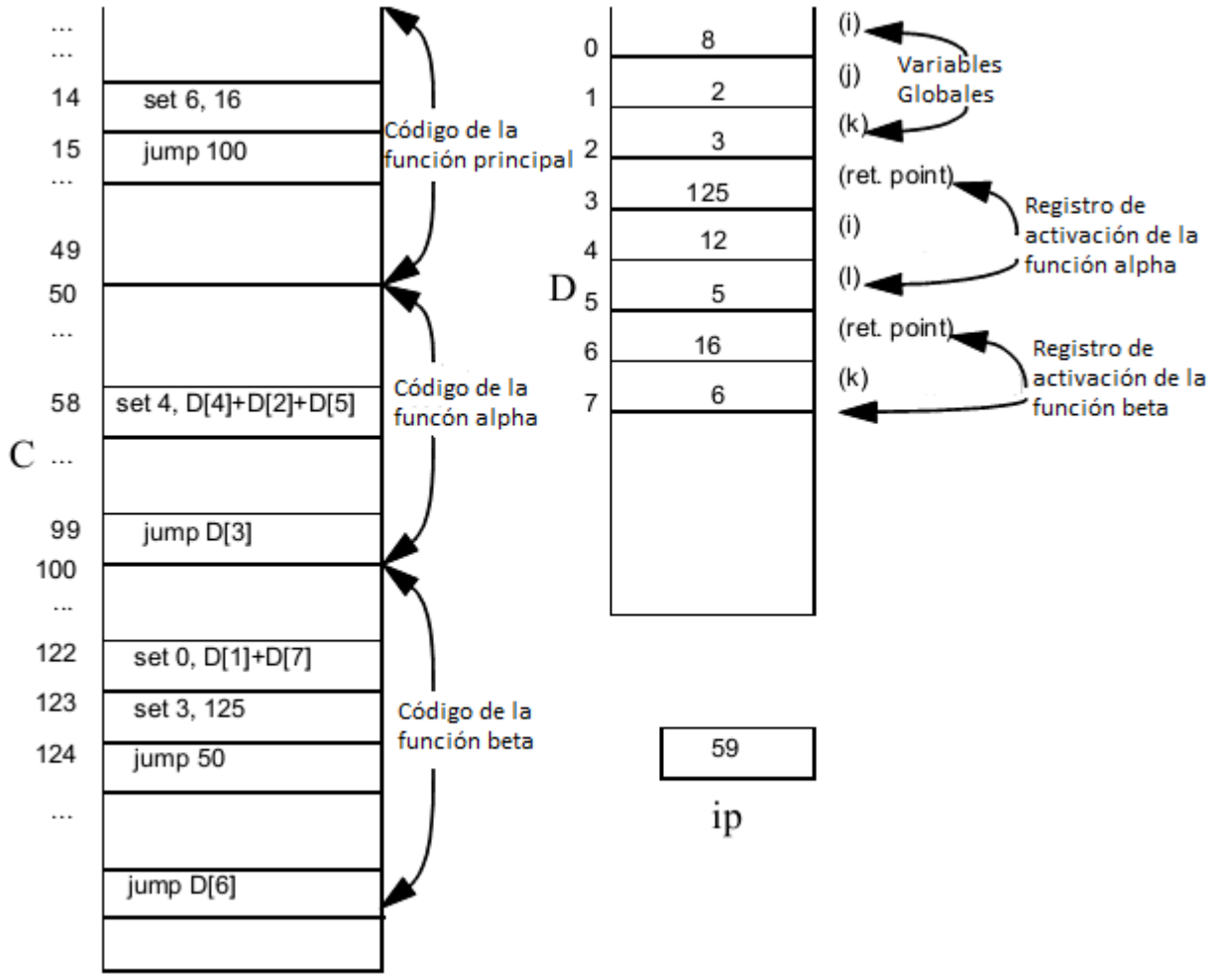


Figura 2.8: Instrucciones Simplesem equivalentes al programa 2.9

2.4.8. Lenguaje C3

El lenguaje C3 agrega nuevas características al trabajo con funciones. Constituye una extensión al lenguaje C2, que agrega la posibilidad de realizar llamadas recursivas dentro de una función, y la posibilidad de devolver algún valor como resultado. El Código 2.10 muestra un programa escrito en lenguaje C3 tomado de [Ghezzi96, página 99]. El objetivo de este programa es calcular el factorial de un número ingresado por teclado.

Código 2.10: Cálculo del factorial de un número - Lenguaje C3

```
1 int n;
2 fact()
```

```
3  {
4    int loc;
5    if (n > 1){
6      loc = n--;
7      return loc * fact();
8    }
9    else {
10     return 1;
11   }
12 }
13
14 main(){
15   read(n);
16   if (n >= 0){
17     write(fact());
18   }
19   else{
20     write("input error");
21   }
22 }
```

Cuando incorporamos recursión, se pierde la posibilidad de determinar qué cantidad de memoria necesitaremos para el programa, por lo que deberemos utilizar un mecanismo de asignación dinámica de memoria.

A partir de la definición de la función, podemos determinar cuál será el tamaño de cada registro de activación, pero no podemos saber cuántas veces se llamará recursivamente la función, es decir no sabemos cuántas unidades del registro de activación necesitaremos. En el ejemplo del Código 2.10, la cantidad de entradas recursivas dependerá del valor de `n`.

Si bien cada llamada recursiva ejecuta el mismo bloque de código, necesitamos distintos registros de activación ya que las variables locales no tienen los mismos valores. Al igual que en el lenguaje C2, se asignará un offset dentro del registro de activación a cada variable local, pero no será posible realizar dicha asignación en tiempo de compilación. Deberemos entonces agregar código Simplesem dentro de la función llamante, que realice la reserva de memoria para el registro de activación de la función que se desea llamar.

Si bien conocemos el offset de la dirección de retorno, del valor de retorno y de las variables locales de las funciones, no podemos conocer

cuál será la base del registro de activación, ya que esta cambiará de una ejecución a otra. Para resolver esto, necesitamos mantener un registro para saber cuál es la base del registro de activación actual, y cuál es la siguiente celda libre. Comencemos definiendo una manera de saber cuál es la base del registro de activación actual. Para este fin, usaremos la celda de la posición 0 de la memoria de datos. Esta celda almacenará la dirección de la base del registro de activación actual, y por ende lo llamaremos *CURRENT*.

Cuando finaliza una función, no necesitaremos más su registro de activación, por lo que podremos liberar el espacio que este ocupaba para que esté disponible para otra función. Por ejemplo, si A llama a B, y B llama a C, los registros de activación se ordenan como A, B, C en la pila de llamadas. Una vez que C finaliza, su registro de activación puede ser descartado. Lo mismo ocurre cuando finaliza B. Ahora bien, ¿Cómo podemos implementar el retorno de una función?. Para hacer esto posible, es necesario almacenar información extra en el registro de activación. Específicamente, debemos almacenar lo siguiente:

- La dirección de la instrucción que deberá ejecutarse al volver de la función (*puntero de retorno*). Esta dirección será almacenada en la primera celda del registro de activación (*offset 0*).
- La dirección de la base del registro de activación que deberá activarse al volver a la función llamante. Esta dirección será almacenada en la segunda celda del registro de activación (*offset 1*), y se conoce como *dynamic link*.

Para manejar la memoria de datos como una pila, debemos conocer cuál es la primera celda que se encuentra disponible en la parte superior de la pila. Para ello utilizaremos la celda en la posición 1 de la memoria de datos, a la que llamaremos *FREE*. Descartar un registro de activación consistirá en modificar las celdas 0 y 1 de la memoria de datos, colocando en *FREE* (1) el actual valor de *CURRENT*, y luego en *CURRENT* (0) el valor del *dynamic link* del registro de activación actual.

Lo último que necesitamos implementar es algún mecanismo que permita devolver un valor de retorno. Debido a que se descartará el registro de activación de la función que va a finalizar, está claro que debemos colocar el valor de retorno en algún lugar del registro de activación de la función llamante, que será al que se vuelva cuando finalice la ejecución de la función que produce el resultado.

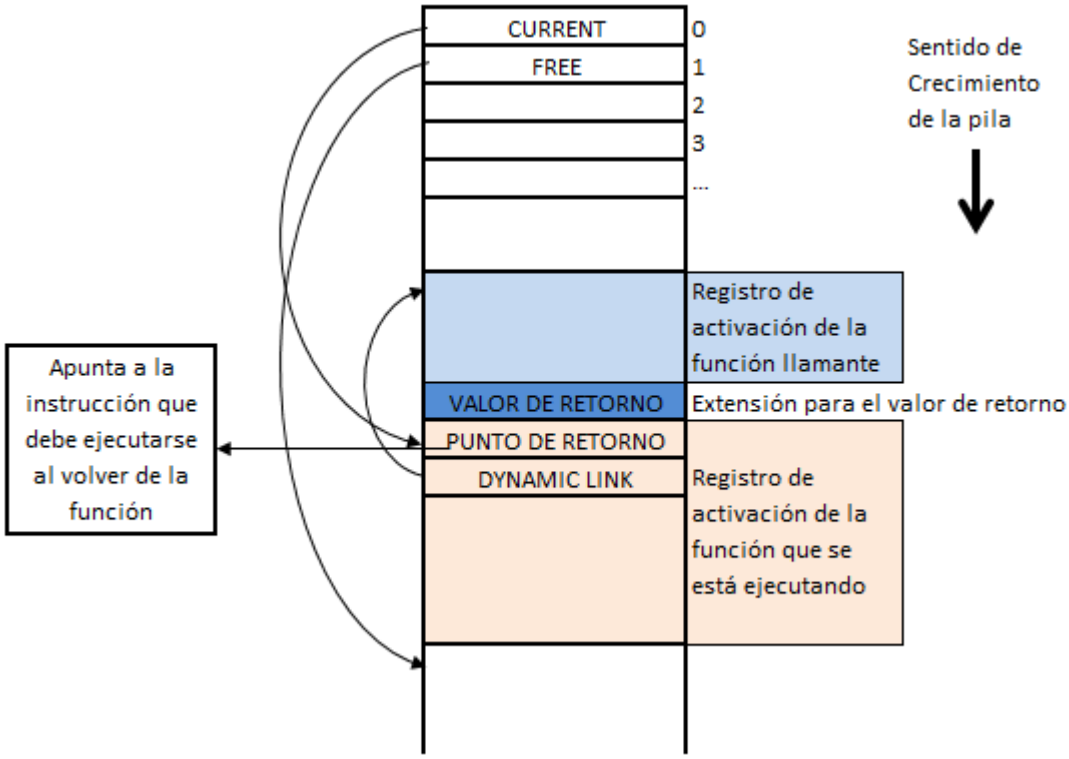


Figura 2.9: Estado de la pila de registros de activación.

Cuando una función sea llamada, lo que haremos será extender una celda el registro de activación de la función llamante, para colocar en esta celda el valor de retorno. Así, la función llamada colocará el valor de retorno en la celda anterior a la base de su registro de activación, utilizando para esto un offset de -1. La Figura 2.9 muestra cómo queda conformada la pila de registros de activación en la memoria de datos.

Las características fundamentales introducidas en el lenguaje C3 son la recursividad y la posibilidad de utilizar valores de retorno. La Tabla 2.2 muestra la secuencia de instrucciones que representan la semántica de una llamada a función en lenguaje Simplesem.

set 1, D[1]+1	Modificar el FREE a la siguiente celda para extender el registro de activación actual.
set D[1], ip + 4	Modificar el valor del punto de retorno en el registro de activación de la función llamada, a la instrucción siguiente al salto al código de la función.
set D[1] + 1, D[0]	Modificar el valor del dynamic link del registro de activación de la función llamada, para apuntar al registro de activación de la función llamante.
set 0, D[1]	Modificar el CURRENT para que apunte al registro de activación de la función que se está llamando.
set 1, D[1] + AR	Modificar el FREE para que apunte a celda que sigue de la última del nuevo registro de activación.
jump start_addr	Saltar a la primera instrucción de la función llamada.

Tabla 2.2: Secuencia de instrucciones Simplesem para una llamada a función.

La Tabla 2.3 muestra las tres instrucciones Simplesem necesarias para volver de una función.

set 1, D[0]	Modificar el FREE para que apunte donde ahora apunta el CURRENT
set 0, D[D[0] + 1]	Modificar el CURRENT y asignarle el valor del dynamic link.
jump D[D[1]]	Saltar a la instrucción a la que apunta el FREE, que será el return pointer.

Tabla 2.3: Secuencia de instrucciones Simplesem para el retorno de una función.

Al comenzar la ejecución del programa, lo primero que debe hacerse es asignar el valor del CURRENT (celda 0), y del FREE (celda 1). Debido a que el registro de activación de la función principal siempre comienza en la celda 2 de la memoria de datos, la primera instrucción siempre será `"set 0, 2"`, lo que posiciona el CURRENT en la celda 2. El valor de la celda 1 (FREE) dependerá de la cantidad de variables globales declaradas, ya que la primer celda libre siempre estará después de las celdas reservadas para las variables globales. La estructura de la memoria de datos, ni bien comienza la ejecución de la función principal, debería quedar como en la Figura 2.10.

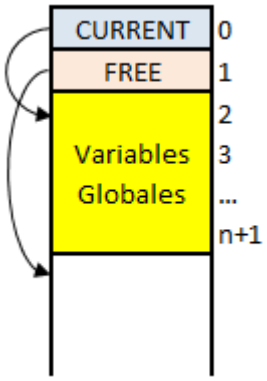


Figura 2.10: Estado inicial de la memoria de datos - Lenguaje C3.

Para finalizar, veremos el código Simplesem equivalente al ejemplo del Código 2.10, que calcula el factorial de un número. La secuencia de instrucciones Simplesem puede verse en el Código 2.11.

Código 2.11: Instrucciones Simplesem equivalentes al programa del Código 2.10

```
0  set 0, 2
1  set 1, 3
2  set 2, read
3  jumpt 12, D[2] < 0
4  set 1, D[1] + 1
5  set D[1], ip + 4
6  set D[1] + 1, D[0]
7  set 0, D[1]
8  set 1, D[1] + 3
9  jump 14
10 set write, D[D[1] - 1]
11 jump 13
12 set write, ``input error''
13 halt
14 jumpt 25, D[2] <= 1
15 set D[0] + 2, D[2]
16 set 2, D[2] - 1
17 set 1, D[1] + 1
18 set D[1], ip + 4
19 set D[1] + 1, D[0]
20 set 0, D[1]
21 set 1, D[1] + 3
22 jump 14
```

```
23 set D[0] - 1, D[D[0] + 2] * D[D[1] - 1]
24 jump 26
25 set D[0] - 1, 1
26 set 1, D[0]
27 set 0, D[D[0] + 1]
28 jump D[D[1]]
```

Las primeras dos instrucciones setean el valor del CURRENT y el FREE respectivamente. Luego se lee el valor de `n`, que está ubicada en la celda 2 de la memoria de datos, y se verifica que el valor de n sea mayor o igual a 0. Si el valor de n es menor que 0, se imprime un mensaje de error y finaliza el programa, sino se inicia el llamado a la función fact().

Desde la instrucción de la línea 4 hasta la instrucción de la línea 8 se encuentra la configuración del registro de activación de la función que vamos a llamar, y de los punteros CURRENT y FREE. La instrucción de la línea 9 realiza el salto a la primera instrucción de la función.

La línea 14 corresponde a la evaluación de la condición en la función fact(). Las instrucciones desde la línea 15 a la 22 corresponden al bloque if de la función fact(). Nótese que aquí vuelven a aparecer las instrucciones que realizan la llamada a la función de manera recursiva (desde la línea 17 a la 21) y el salto al siguiente nivel de recursión (línea 22).

La línea 23 escribe el valor de retorno en la extensión del registro de activación de la función llamante, y luego se salta a la línea 26, donde aparecen las 3 instrucciones necesarias para volver de la función. Lo mismo ocurre con la línea 23, que devuelve el valor 1 a la celda de extensión del registro de activación llamante. Resulta complicado realizar un seguimiento de la ejecución con una imágen estática, pero alcanzado este punto, el lector cuenta con el conocimiento necesario para realizar la ejecución del Código 2.11 por su cuenta.

Desarrollo

El modelo Simplesem ayuda a los estudiantes a comprender el impacto que tiene un lenguaje de alto nivel sobre el lenguaje máquina, finalmente ejecutado por el procesador. Además de ello, la propuesta realizada por Ghezzi y Jazayeri permite analizar el proceso por el que pasa un programa escrito en un lenguaje de programación de alto nivel, antes de poder ser ejecutado por el procesador. Este proceso se conoce como proceso de compilación, y en el participan numerosas herramientas que realizan distintas etapas, como el reconocimiento de símbolos, el parseo del código, el armado del árbol de parseo o, finalmente, la generación del código máquina.

El trabajo aquí presentado se divide en tres etapas. La primera consiste en extender el lenguaje Simplesem a un modelo de ejecución paralelo. Para ello será necesario incorporar nuevas instrucciones que permitan trabajar con múltiples hilos, procurando mantener la simpleza que caracteriza al modelo. Llamamos C3P a la extensión propuesta, debido a que posee las mismas funcionalidades que le lenguaje C3, incorporando cierto grado de paralelismo.

Completada la primera etapa, el segundo paso será desarrollar un compilador para generar las instrucciones Simplesem correspondientes para un programa escrito en el lenguaje de alto nivel definido.

Una vez que hayamos obtenido las instrucciones Simplesem, se deberá desarrollar un intérprete que muestre la ejecución de las instrucciones de una manera comprensible para el usuario, y que permita tener un control sobre qué instrucciones ejecutar, para que sea el usuario quien dirija la simulación de la ejecución.

3.1. Requerimientos de la herramienta

Lo primero que debemos hacer es definir la gramática del lenguaje de alto nivel que podrá procesar el compilador. Dicho lenguaje deberá ser una

extensión del lenguaje C3 propuesto por Ghezzi y Jazayeri, que cumpla con las siguientes características:

- Permitir la definición de una función que reciba múltiples parámetros y pueda devolver un valor de retorno.
- Permitir la definición de variables globales y locales.
- Poseer algún mecanismo de Entrada/Salida para comunicarse con el usuario.
- Poseer una construcción que permita definir un bloque de instrucciones que se ejecutarán en múltiples hilos.
- Cada hilo debe tener variables compartidas y variables privadas.
- El lenguaje debe poseer una función principal, que será donde comience la ejecución.

Una vez definida la gramática del lenguaje (sintaxis y semántica), deberá desarrollarse el compilador, cuyos requerimientos son:

- Abrir un archivo llamado `source.c3p` que contenga código fuente escrito en el lenguaje de alto nivel definido.
- Determinar si el programa escrito es sintácticamente correcto.
- Identificar el orden y tipo de reglas sintácticas que aparecen en el programa cargado.
- Generar las instrucciones Simplesem correspondientes al programa escrito en el archivo `source.c3p`, y guardar dichas instrucciones en un archivo llamado SimpleSemSource.txt
- A través de las instrucciones Simplesem, definir la semántica del lenguaje.

Finalmente, se desarrollará un evaluador del lenguaje C3P didáctico. Este consiste en una interfaz gráfica que permitirá al usuario analizar el impacto de la ejecución de un programa en varios hilos, entre los cuales existen variables compartidas, y variables privadas. Dicho intérprete deberá:

- Presentar la memoria de código de cada hilo y la memoria de datos de la máquina Simplesem.
- Posibilitar la ejecución de cada hilo en particular.

- Posibilitar la ejecución de una instrucción en todos los hilos, en paralelo.
- Brindar un mecanismo que permita escribir código fuente en lenguaje de alto nivel, y compilarlo a lenguaje Simplesem.
- Mostrar la salida en pantalla del programa que se está ejecutando.
- Obtener un valor de entrada del teclado cuando aparezca una instrucción que así lo requiera.
- Poseer ejemplos que muestren las distintas funcionalidades del lenguaje.

3.2. Propuesta de extensión

El propósito de este trabajo es generar una herramienta educativa que permita mostrar un modelo de ejecución paralelo basado en el modelo Simplesem. Dicha herramienta tiene como objetivo apoyar el aprendizaje de ciertos conceptos relacionados a los lenguajes de programación.

3.2.1. Nuevas instrucciones Simplesem

Al definir la semántica del lenguaje incorporamos un conjunto de instrucciones al modelo Simplesem para soportar la ejecución de múltiples hilos. Estas instrucciones son:

- `processors n`: esta instrucción provoca que se creen n hilos, cada uno con su propio IP, y con una memoria de código independiente del resto. Al momento de su creación, todos los hilos estarán formados por las mismas instrucciones, y será el programador quien deba dotar de un comportamiento distinto a cada hilo.
- `barrier`: cada vez que el procesador deba ejecutar la instrucción barrier de un hilo, deberá verificar si todos los otros hilos están en la misma instrucción. De no ser así, la ejecución deberá bloquearse hasta que todos los hilos estén en la misma instrucción.
- `wait(n)`: esta instrucción es utilizada para lograr la implementación de semáforos a través de instrucciones Simplesem. Cada vez que una instrucción `wait` se ejecuta sobre una variable `n`, si `n == 0` la ejecución del hilo se bloquea. De no ser así, se decrementa el valor de `n` en 1, y se continua con la ejecución. Esto podría hacerse a través de múltiples

instrucciones Simplesem, pero la modificación de los permisos de un semáforo debe ser una operación atómica, por lo que se necesita una única instrucción para tal fin.

- `numHilo`: el indicador numHilo se utiliza como palabra clave dentro de las instrucciones Simplesem para referirnos al hilo que actualmente se está ejecutando. Cada hilo posee un identificador (de 0 a 3) que se utilizará para generar un offset de una posición en la memoria de datos, para acceder a los datos que corresponden a cada hilo.

3.2.2. Extensión del lenguaje C3 a C3P

La gramática del lenguaje C3 puede verse definida en el apéndice A del presente informe. En esta sección mostramos sólo aquellas reglas de producción que se han incorporado al lenguaje C3P.

El Código 3.1 muestra las nuevas reglas de producción que se incorporaron a la gramática del lenguaje C3 para definir el lenguaje C3P.

Código 3.1: Nuevas reglas del lenguaje C3P

```
1  statement : assign ";"
2            | "read" "(" iden ")" ";"
3            | "write" "(" expr ")" ";"
4            | iterate
5            | condition
6            | "return" expr ";"
7            | "wait" "(" iden ")" ";"
8            | "notify" "(" iden ")" ";"
9            | call ";"
10           | parallel_statement
11           | comment
12
13 parallel_statement : parallel_for
14                    | "barrier" ";"
15
16 parallel_for : parallel_for_beginning statement_block
     parallel_for_end
17
18 parallel_for_beginning : "par_for" numbers
     for_definition
19
20 parallel_for_end : "end_par_for" ";"
```

La gramática completa del lenguaje C3P puede verse en el Apéndice B del presente informe[1].

3.2.3. Utilización de múltiples hilos

Cuando múltiples hilos se crean en el procesador, los mismos comparten la memoria de datos (memoria compartida). El programador deberá definir, al comienzo del programa, un conjunto de variables globales compartidas por todos los hilos (globales), y una variable que será independiente para cada uno. Se decidió utilizar una memoria compartida para guardar un único registro de activación (de la función en la cual se crean los hilos) que contenga una única copia de las variables compartidas, y tantas instancias como hilos se hayan creado, de la variable independiente.

En el presente trabajo se abordó una única primitiva que permite ejecución en paralelo: `parallel for`. La sintaxis de un parallel for se muestra a continuación:

```
par_for [numero_de_hilos] (i = 0;i <= 7;i++){
        bloque_de_instrucciones
}
```

`numero_de_hilos` representa la cantidad de hilos que quieren crearse. Entre paréntesis, se coloca el rango de valores que queremos procesar. Debe tenerse en cuenta que dicho rango debe ser divisible entre la cantidad de hilos que se creen, de manera tal que cada hilo tome la misma cantidad de valores (en este caso, cada hilo procesará dos valores de`i`). Si el programador desea dar un comportamiento distinto a cada hilo, en el bloque de instrucciones puede preguntarse qué hilo está ejecutándolo y en función de eso realizar una u otra tarea. De esta manera, con la misma primitiva existen numerosas alternativas de ejecución en paralelo.

3.2.4. Semántica del lenguaje C3P

Una vez definida la sintaxis (presente en el apéndice B), el siguiente paso es definir la semántica del lenguaje. Como se dijo en la introducción, la semántica del lenguaje deberá ser implementada a través de instrucciones Simplesem. Se explicará cada regla de la sintaxis con la semántica correspondiente.

[1] Puede verse una imagen del árbol de la sintaxis completa en http://goo.gl/gZHaU

En general tenemos tres tipos de reglas. Algunas, las más generales como `programm`, tendrán una semántica compuesta de la semántica de otras reglas. A estas las llamaremos reglas compuestas. Luego existen reglas cuya semántica es definible con instrucciones Simplesem, como por ejemplo una asignación, o una operación. A estas las llamaremos reglas elementales. Finalmente, existen reglas que no tienen una semántica, sino que conforman el léxico del lenguaje, es decir que sólo definen la forma en la que están compuestos algunos términos, como los nombres de variables, los números o los identificadores de función. A estas últimas las llamaremos reglas léxicas.

Para definir la semántica de las reglas compuestas, seguiremos el criterio utilizado por Ghezzi y Jazayeri al definir la semántica del lenguaje C3, para la cual la semántica de las reglas compuestas dependerá de la semántica de las reglas que las conforman.

3.2.4.1. Nuevas reglas de producción

Las reglas de producción agregadas permiten implementar en el lenguaje C3P, un conjunto de elementos propios de la programación concurrente, como semáforos e hilos. A continuación explicamos las nuevas reglas de producción incorporadas al lenguaje C3P.

- **wait:** una sentencia `wait` sobre una variable deberá bloquear la ejecución del hilo si la variable tiene valor 0, o decrementar su valor en caso contrario. Si bien este comportamiento puede ser implementado con un conjunto de instrucciones Simplesem, la necesidad de que esta sea una instrucción atómica nos llevó a definir una nueva instrucción Simplesem, `wait`, cuyo funcionamiento se detallará más adelante.

- **notify:** la sentencia notify implica incrementar en 1 el valor de una variable. Cumple la misma función que el incremento (variable++) y se implementa con la misma instrucción Simplesem, pero fue definida con el objetivo de mantener un identificador de utilización frecuente en la implementación de semáforos.

- **parallel_statement:** esta regla representa instrucciones que deben ejecutarse en paralelo. Está compuesta por dos producciones: un bloque `parallel_for` y la instrucción `barrier`. La instrucción barrier deberá bloquear la ejecución del hilo que la alcance hasta que todos los demás hilos lleguen a dicha instrucción.
  El bloque parallel_for representa un conjunto de instrucciones que deberán ejecutarse en paralelo. Para poder utilizarlo, es necesario definir

una variable que poseerá el mismo nombre para todos los hilos, pero adoptará diferentes valores para cada uno. Esta variable debe ser global ya que será compartida por todos los hilos.
Supongamos una variable `var_par`. Esta debe definirse como `var_par range 1..4`, indicando que habrá 4 hilos. Luego debe iniciarse un bloque `par_for`, donde se indica la cantidad de hilos (4 en este caso), y el rango de variación de `var_par` (que en este caso, deberá ser múltiplo de 4). En el ejemplo del Código 3.2 se muestra la sintaxis en lenguaje C3P que deberá utilizarse para ejecutar instrucciones en paralelo. Dicho ejemplo imprime el identificador de cada hilo en pantalla. La Figura 3.1 muestra las instrucciones Simplesem que se utilizan para la ejecución en paralelo. En esta figura, ya se han ejecutado todas las instrucciones, de allí que la memoria de datos tenga valores asignados.

Código 3.2: Ejemplo de ejecución en paralelo en lenguaje C3P

```
1 var: var_par range 1..4;
2 program{
3     par_for 4 (var_par = 0; var_par <= 3; var_par
          ++){
4         write(var_par);
5     }
6     end_par_for;
7 }
```

	Hilo 0	Datos
0	set 0, 0	0
1	set 1, 6	6
2	processors 4	1
3	parallel_for 4, 0	2
4	set 2 + numHilo, 0 + numHilo*1	3
5	jumpt 9, D[2+numHilo] > 1*(numHilo+1) + 0 - 1	4
6	set write, D[2+numHilo]	
7	set 2 + numHilo, D[2+numHilo]+1	
8	jump 5	
9	end_par_for	
10	halt	
11		

Figura 3.1: Instrucciones Simplesem para el Código 3.2

La instrucción de la celda 2 indica a la máquina Simplesem que se crearán 4 hilos. La siguiente instrucción indica el rango de la variable independiente (var_par en el ejemplo dado). La instrucción de la celda 4 colocará el identificador de cada hilo en la celda que lo identifica. Luego se ejecuta el código del bloque parallel for. La instrucción 5 corresponde al salto condicional que marcará el final de la ejecución de todos los hilos, excepto del hilo 0 que será el hilo principal. La instrucción 6 se compila igual en todos los hilos, pero debemos notar que en tiempo de ejecución será distinta debido al valor de `numHilo`. La instrucción end_par_for marca el final de la ejecución de un hilo, y su destrucción, excepto para el hilo 0, para el cual no tiene efecto.

A continuación veremos la semántica de algunas reglas que ya estaban definidas en el lenguaje C3.

3.2.4.2. Reglas de declaración

Las reglas que representan la declaración de variables (o de arreglos), como lo son `declare` y todas las subreglas que la componen, nos indican que debemos guardar un espacio en memoria para almacenar el valor de dichas variables. Además, cada vez que queramos referirnos a esa posición de memoria debemos tener la posibilidad de hacerlo utilizando el mismo nombre con el que fueron definidas.

En el compilador desarrollado, se utiliza un diccionario que almacena para cada variable, la posición de memoria que le fue asignada, para que cada vez que la variable sea referenciada, se reemplace su nombre por su posición de memoria. La regla declare, entonces, deberá agregar al diccionario todas las variables que sean declaradas, para poder ser referenciadas posteriormente.

3.2.4.3. Regla fun_decla

La regla `fun_decla` implica una declaración de función. Debido a que el lenguaje sólo soporta la declaración de una función, esta regla deberá obtener todo el código de la función en instrucciones Simplesem, y colocarlo en el lugar correcto en la memoria de código.

Sin importar cual sea el código específico de la función, existe un conjunto de instrucciones que siempre deberán estar presentes al final de cualquier función. Estas instrucciones son:

- set D[0]-1, [valor_de_retorno] (si la función devuelve algún valor)

- set 1, D[0] (apunta el FREE donde ahora está el CURRENT)
- set 0, D[D[0]+1] (pone en CURRENT el valor que tiene el enlace dinámico)
- jump D[D[1]] (vuelve a la ejecución desde donde se llamó a la función)

La declaración de una función también puede implicar que esta tome parámetros. Para ello, debe conocerse también la cantidad de parámetros, sus nombres, y sus desplazamientos dentro del registro de activación de la función. Si en la declaración de la función existen parámetros, la información mencionada deberá ser almacenada para ser utilizada en el momento en que se llame a la función. Toda esta configuración debe ser realizada al llamar la función, por lo que las instrucciones Simplesem correspondientes deberán generarse en la regla `call`.

3.2.4.4. Regla statement

Esta es quizás la regla más importante, ya que será la que define cuáles son las instrucciones que pueden darse a la máquina Simplesem. Si bien es una regla compuesta, las distintas producciones que puede tener esta regla, constituyen la mayor parte de las reglas elementales del lenguaje. Analizaremos para cada una de las posibilidades, cuáles son las instrucciones que deben utilizarse para representar la semántica del lenguaje.

- **assign:** esta regla representa una instrucción de alto nivel que puede ser de tres maneras: una asignación de la forma `variable = expresion;`, un incremento de la forma `variable++;` o un decremento de la forma `variable--;`. Cualquiera sea el caso, la semántica puede ser representada con una instrucción `set` de Simplesem. Por ejemplo la asignación `x = y;` se representará con la instrucción Simplesem `set 3, D[4]`, suponiendo que la posición de `x` es 3, y la de `y` es 4. Un incremento puede representarse como `set 3, D[3]+1`, y un decremento como `set 3, D[3]-1`.
- **read y write:** la instrucción `read` en Simplesem se utiliza para tomar un valor desde un dispositivo de entrada (en nuestro caso, el teclado), y colocar dicho valor en una variable. Por lo tanto, la sentencia `read(variable)` en el lenguaje C3P deberá hacer uso de la instrucción read de Simplesem para tomar el valor del teclado y guardarlo en una variable.
  Lo mismo ocurre con la instrucción `write`, utilizada para imprimir un valor en dispositivo de salida (en nuestro caso una consola). A diferencia

de la regla `read`, en la regla `write` puede pasarse como parámetro una expresión, por lo que el resultado de todas las instrucciones Simplesem que representen esa expresión, será lo que finalmente se imprima en pantalla.

- **iterate:** En el lenguaje C3P existen dos tipos de iteraciones: while y for. Los ciclos while (`while_iteration`) poseen una expresión booleana y un bloque de instrucciones, que deberá ejecutarse hasta que la expresión booleana sea falsa. En Simplesem, las instrucciones que representan la semántica de una iteración while dependen del bloque de instrucciones. La estructura de un ciclo while en Simplesem puede verse en el Código 3.3.

  Código 3.3: Estructura de las instrucciones que representan un ciclo while en Simplesem

```
3 jumpt 9, D[2] >= 5  //Salto a la instrucción
4                      //siguiente a la última del
                          while
5 ...
6 statement_block    //Instrucciones del bucle
7 ...
8 jump 3             //Volver a ejecutar el ciclo.
9 ...
```

  Para un ciclo for, la estructura será la misma, excepto que antes del jump de la línea 3 se debe asignar el valor inicial a la variable de control, y antes de volver a ejecutar el ciclo debe modificarse dicha variable.

- **condition:** esta regla puede ser representada por dos subreglas: un bloque `if`, y un bloque `if-else`. Cuando nos encontramos con un bloque `if`, deberíamos determinar si la condición es verdadera, y sólo si así fuere, ejecutar las instrucciones que están dentro de las llaves "{" y "}".
  Una sentencia `if-else` está formada por una condición, y dos bloques de instrucciones. Si la condición es verdadera, se ejecutará el bloque `if`, mientras que si es falsa, se ejecutará el bloque `else`. Las instrucciones Simplesem que representan la sentencia `if-else` del Código 3.4 se muestran en el Código 3.5,

  Código 3.4: Sentencia if-else en lenguaje C3P

```
1 if(a > 5){
2         ...
```

```
3 }
4 else {
5         ---
6 }
```

Código 3.5: Instrucciones Simplesem para una sentencia if-else

```
3 jumpt 7, D[2] <= 5  //Ejecutar el bloque else si a
4             //(almacenado en D[2]) es mayor que 5.
5 ...         //Instrucciones del bloque if.
6 jump 3      //Saltar después del bloque else.
7 ---         //Instrucciones del bloque else.
```

- **call:** la regla call define cuál es la sintaxis para una llamada a función. Cuando esto ocurre, existen una serie de instrucciones Simplesem que deben utilizarse para implementar la ejecución de la rutina.
  Para explicar este caso, utilizaremos el Código 3.6 escrito en lenguaje C3P, y su equivalente Simplesem, mostrado en la Figura 3.2. Se utilizará este ejemplo para reflejar cómo se manejan múltiples parámetros en el registro de activación, y cómo deberá ser el código Simplesem que implemente la semántica de una llamada a función y su ejecución.

Código 3.6: Declaración de una función con múltiples parámetros y una variable local

```
0  var: n, j;
1  suma(num1, num2, num3, num4)
2  var: loc;
3  {
4      loc = -4;
5      num1 = num1 + num2 + num3 + num4 + loc;
6      return num1;
7  }
8
9  program{
10     n = 5;
11     j = 6;
12     write(suma(n, 4, j, 3));
13 }
```

Cuando se realiza una llamada a función, independientemente de dónde se encuentre el código que la implementa, debemos preparar el

registro de activación de la función. Este registro almacenará toda la información necesaria a cerca de la función, como el puntero de retorno, un enlace al registro de activación que debe manejarse al terminar la función, y las variables locales a la función. Las instrucciones desde la celda 4 a la 13 realizan esta tarea.

- Lo primero que haremos es modificar el valor del Free, para que apunte a la celda siguiente a la última libre. Recordemos que cuando se llama a una función, debemos dejar una celda libre antes del registro de activación para colocar el valor de retorno.
- Luego, en la base del registro de activación debemos guardar el puntero de retorno, que será la dirección de la instrucción siguiente a la última utilizada para preparar el llamado a función (en este caso deberá volverse a la celda 14).
- Con la instrucción de la celda 6 guardamos el link dinámico al registro de activación que deberá manejarse al volver de la función.
- Desde la instrucción de la celda 7 a la celda 10 colocamos los valores de los parámetros pasados a la función en el registro de activación.
- Finalmente modificamos el valor del CURRENT (para que apunte al nuevo registro de activación) y del FREE (que apuntará a la siguiente celda libre), y luego realizamos el salto al código de la función (desde la celda 16).

Una plantilla para la llamada a funciones en lenguaje Simplesem se muestra a continuación:

```
set 1, D[1]+1
set D[1], ip + 4 + [cant. de parámetros]
inicialización de parametros
set 0, D[1]
set 1, D[1] + 3 + [cant. de parámetros]
jump [primera instrucción de la función]
—o—
código específico de la función
set D[0]-1, [valor_de_retorno]
set 1, D[0]
set 0, D[D[0]+1]
jump D[D[1]]
```

	Hilo 0	Datos
0	set 0, 0	CURRENT
1	set 1, 4	FREE
2	set 3, 5	j
3	set 2, 6	n
4	set 1, D[1]+1	return value
5	set D[1], ip + 8	return pointer
6	set D[1]+1, D[0]	dynamic link
7	set D[1]+2,D[3]	num1
8	set D[1]+3,4	num2
9	set D[1]+4,D[2]	num3
10	set D[1]+5,3	num4
11	set 0,D[1]	loc
12	set 1,D[1] + 7	
13	jump 16	
14	set write, D[D[1]-1]	
15	halt	
16	set D[0]+ 6, -4	
17	set D[0]+ 2, D[D[0]+2] + D[D[0]+3]...	
18	set D[0]-1, D[D[0]+2]	
19	set 1,D[0]	
20	set 0, D[D[0]+1]	
21	jump D[D[1]]	
22		

Figura 3.2: Instrucciones Simplesem para el Código 3.6

- **return:** la regla `'return' expresion ';'` deberá aparecer sólo dentro de una función, y será la última instrucción que se ejecute en ese scope. Deberá colocarse el resultado de la expresión en la posición de memoria reservada para tal fin, que es la posición anterior a la primera del registro de activación de la función. Recordando que en la posición 0 de la memoria de datos se almacena un puntero a la base del registro de activación, las instrucciones correspondientes a un `return` son:

Código 3.7: Instrucciones Simplesem para una sentencia return

```
3 set D[0]-1, [valor_de_retorno]
4 set 1, D[0]
5 set 0, D[D[0]+1]
6 jump D[D[1]]
```

- **comment:** cuando se encuentre la regla comment, el compilador deberá ignorarla, de manera tal que no produzca ninguna instrucción como resultado. Los comentarios fueron agregados para facilitar la lectura del código de alto nivel, pero no tienen ningún impacto sobre las instrucciones Simplesem generadas.

3.3. Implementación del compilador

Una vez definida la gramática del lenguaje, comenzamos con la implementación del compilador. El mismo consiste en un parser que, a grandes rasgos, deberá analizar el código fuente escrito en lenguaje C3P, encontrar las reglas gramaticales que se presenten, y generar instrucciones Simplesem que implementen la semántica de dichas reglas. Para ello, se utilizó el módulo Parse::RecDescent de Perl, con el cual puede generarse un parser de descenso recursivo a partir de la definición de una gramática. En esta sección se presentan las partes más importantes del compilador. La Figura 3.3 muestra un esquema general de la estructura del compilador.

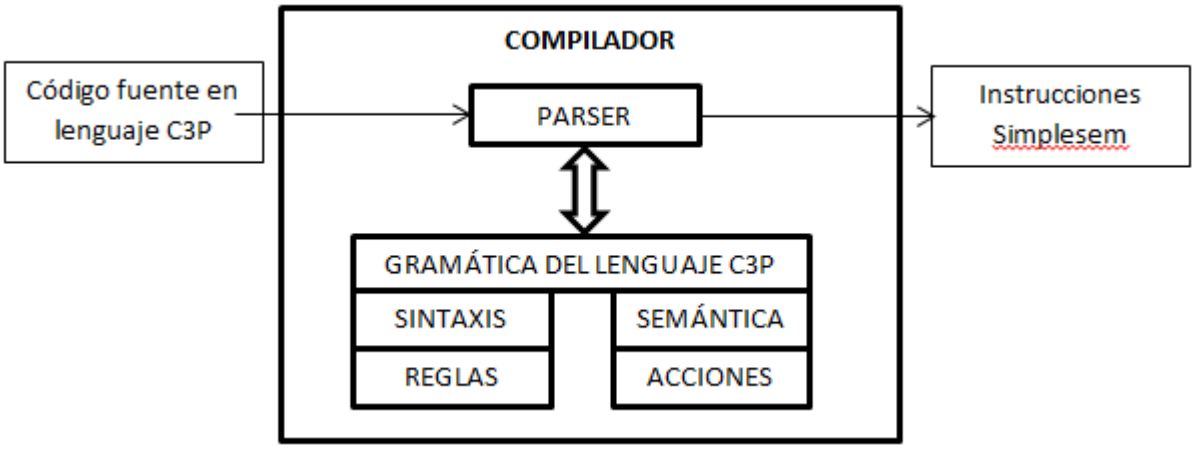


Figura 3.3: Estructura del compilador

El código fuente ingresa al parser para ser analizado. El parser posee una gramática definida por una sintaxis y una semántica. La sintaxis del lenguaje está definida por un conjunto de reglas, cada una de las cuales representa un elemento perteneciente al lenguaje. Si el parser determina que el código fuente cumple con la regla raíz decimos que el código está escrito en lenguaje C3P. Se utilizaron instrucciones Simplesem para implementar la semántica del lenguaje. Las instrucciones correspondientes a cada regla se generan a través de

acciones que se ejecutan cuando el parser determina la ocurrencia de una regla.

3.3.1. Módulo Parse::RecDescent

El módulo Parse::RecDescent es utilizado para generar parsers de texto de descenso recursivo en sentido top-down. Por descenso recursivo, se entiende que se intentarán matchear las producciones de las reglas gramaticales en el órden en que son definidas, y la primera coincidencia será la "mejor". Otros parsers determinan que la mejor coincidencia es la más larga, por lo que deben buscar en todas las producciones de cada regla para determinar cuál es la mejor coincidencia, ya que la primera encontrada podría no ser la más larga.

El sentido top-down se refiere a que, dado un árbol de la sintáxis del lenguaje, el parser comenzará a analizar la entrada a partir de lo más general (la raíz del árbol) a lo más particular (las hojas, que contienen los símbolos terminales del lenguaje). En un análisis en sentido bottom-up, el parser comenzaría a analizar la entrada desde de los elementos más simples, y a partir de allí intentará inferir las estructuras de mayor orden a partir de dichos elementos.

Algunas de las características del módulo Parse::RecDescent son:

- Los terminales de la gramática pueden ser expresados como literales o con expresiones regulares.
- Múltiples producciones para una regla (es decir que, por ejemplo, un operando puede estar formado por un número, una variable, una expresión, etc.).
- La posibilidad de utilizar todas las potencialidades de Perl para definir las acciones que forman la semántica de un lenguaje.
- La posibilidad de pasar datos hacia arriba y abajo en el árbol de parseo.

3.3.1.1. Utilización

Los objetos que realizarán el parseo de texto deben ser creados llamando a la función `Parse::RecDescent::new`, que recibe como parámetro la especificación de una gramática. Si la gramática es correcta, `new` devuelve una referencia que puede ser utilizada para iniciar el parseo desde cualquier regla especificada en la gramática original. El Código 3.8 muestra una plantilla que puede ser utilizada para trabajar con el módulo Parse::RecDescent.

Código 3.8: Plantilla para la definición de una gramática

```
$grammar = q {
  #Especificacion de la gramatica
}

$parser = Parse::RecDescent->new($grammar) or die ``
   Gramatica Erronea\n'';

#completar el texto que sera parseado en alguna
   variable, por ejemplo $text

defined $parser->startrule($text) or print ``Texto
   incorrecto\n'';
```

La regla con la cual el parseo es inicializado debe estar explícitamente declarada en la gramática (para el ejemplo dado, la regla "startrule" debería estar definida en la gramática). Si la regla de inicio es correcta, su valor es devuelto. Una falla al generar el parser, o al realizar el parseo del texto, es indicada devolviendo `undef`, una valor especial similar a NULL. Al escribir una gramática, no debemos considerar "0", 0 ni "" como resultados que indiquen errores, ya que son resultados que pueden obtenerse incluso utilizando una gramática correctamente definida.

3.3.1.2. Definición de Reglas Sintácticas

En la gramática a partir de la cual se construye el parser, las reglas son especificadas utilizando un identificador, seguido de dos puntos (":"), seguido por una o más producciones, separadas estas últimas por barras verticales. Un ejemplo de regla gramatical es el siguiente: `rule1: production1 | production2 | production3`. En cualquier momento pueden agregarse nuevas producciones a las reglas gramaticales previamente definidas redeclarando la regla con las nuevas producciones.

Cada producción en una regla puede estar formada por cero o más items, cada uno de los cuales puede ser:

- El nombre de otra regla (una subregla).
- Una expresion regular, o una cadena que debe ser matcheada directamente (conocida como "token" o "terminal").

- Un bloque de código en lenguaje Perl que debe ser ejecutado (una acción).
- Una instrucción especial al parser (una directiva).
- Un comentario de Perl, que será ignorado.

Se dice que una regla coincide con un texto cuando una de sus producciones coincide, mientras que una producción coincide si cada uno de sus items, en orden, coinciden con las subcadenas del texto. Las producciones de una regla se van probando en el mismo orden en que fueron definidas, y ante la primera coincidencia la búsqueda termina. Si todas las producciones fueron probadas y ninguna coincide, se dice que la búsqueda falla.

3.3.1.3. Subreglas

Una subregla que aparece en una producción implica que el parser intente encontrar una producción de dicha subregla que coincida con el texto que se está analizando. Si la subregla no está definida, automáticamente se produce un fallo en la búsqueda. Una regla puede llamarse a si misma recursivamente, es decir que una regla puede ser subregla de ella misma. Cuando esto se haga, debe tenerse el cuidado de poner una condición de corte para la entrada recursiva, ya que no hacerle provocaría un ciclo infinito.

3.3.1.4. Tokens - Terminales

En las reglas de producción pueden aparecer cadenas de caracteres encerradas entre comillas dobles, o bien expresiones regulares. Este tipo de elementos se conocen como `"Tokens"` o `Terminales`. Cuando un `Terminal` aparece en algún elemento de la producción, el parser intenta buscar dicho terminal en ese punto en el texto. Por ejemplo, la regla `typedef: "typedef" typename identifier ";"` arrojará una coincidencia con el texto `"typedef int Entero;"`, implicando que siempre la primera palabra deberá ser `typedef` y la última un punto y coma.

3.3.1.5. Acciones

Las acciones son quizás la herramienta más importante del módulo, ya que permiten definir operaciones que se realizan cuando se encuentra una determinada regla. Debido a que tomaremos un enfoque operacional al definir la semántica del lenguaje, las acciones se utilizarán para determinar las instrucciones Simplesem que deberán generarse al encontrar la ocurrencia

de una determinada regla.

En el módulo Parse::RecDescent, una `Acción` es un elemento de la producción que se representa con un bloque de código en lenguaje Perl que deberá ser ejecutado al alcanzarlo. Los elementos de la producción coinciden uno a uno con el texto procesado. Cuando se encuentra una acción, en lugar de buscar una coincidencia con el texto, se ejecutan una serie de instrucciones, de allí que las acciones por lo general sean el último elemento de la producción.

Dentro de la definición de una acción, el programador tiene acceso a un conjunto de variables que estarán disponibles sólo durante el parseo de esa producción. Estas variables nos permitirán acceder a los elementos de la producción para utilizarlos y procesarlos. Veremos estas variables a continuación.

3.3.1.5.1. @item y %item: El arreglo `@item[1..$#item]` almacena el valor asociado a cada elemento de la producción, es decir cada subregla, terminal o acción. Debido a que el parser realiza el apareo de izquierda a derecha, `@item` sólo contiene los valores de los elementos que están antes del punto actual en la producción. Es decir, si una acción aparece como el segundo elemento de la producción, `@item` no contendrá el valor de ninguno de los elementos que le siguen.

El primer elemento (`$item[0]`), almacena el nombre de la regla en la que actualmente se encuentra. `@item` se comporta como un arreglo estándar de Perl, por lo que puede ser indexado con números negativos representando los indices de los elementos desde el final hasta el principio.

Por otro lado, el hash `%item` es un complemento al arreglo `@item` que provee acceso a los mismos elementos que `@item`, pero utilizando un nombre en vez de un índice. El resultado de las subreglas que posean nombre es almacenado en el hash `%item` bajo el nombre de cada subregla, mientras que el resto de elementos son almacenados bajo un nombre generado de acuerdo a su posición y al tipo de elemento del que se trate. El siguiente ejemplo aclarará la manera en que el hash *%item* almacena los datos:

```
stuff: /various/ bits 'and' pieces 'then' data
'end' {save}
{
    print $item{data}, #Imprime data
    print $item{__PATTERN1__}, #Imprime 'various'
```

```
    $item{__STRING2__}, #Imprime 'then'
    $item{__ACTION1__} #Imprime el valor de
                          #retorno de save.
}
```

Así como en el arreglo `@item` podíamos acceder al nombre de la regla actual a través de `@item[0]`, también podemos utilizar la entrada `$item__RULE__` del hash para el mismo fin. La ventaja de utilizar el hash `%item` en lugar del arreglo `@item` es que se elimina la dependencia de la posición de los elementos, que puede cambiar a medida que la gramática evoluciona. Sin embargo, una limitación del hash `%item` es que almacena sólo el último valor de una subregla particular. Por ejemplo, en la siguiente regla: `range: '(' number '..' number ')'` la entrada `number` del hash contendrá el valor de la segunda subregla number.

3.3.1.5.2. $return: Si un valor es asignado a la variable `$return` dentro de una acción, ese valor es devuelto al finalizar la acción. Si la variable `$return` no es asignada en ningún momento dentro de la producción, el valor del último componente de la producción es devuelto cuando se determina una coincidencia completa. El último componente a menudo es una instrucción en una acción, y el resultado de esa instrucción es el que será devuelto como resultado de la producción.

3.3.1.5.3. $text: El texto que todavía no fue parseado se almacena en la variable `$text`. Esta variable puede ser modificada, lo cual significa que podemos alterar dinámicamente el contenido del texto que está siendo parseado.

3.3.1.5.4. $thisline y $prevline: Cuando estamos parseando un texto, a menudo necesitamos saber sobre qué linea estamos ubicados en un momento determinado. La variable `$thisline` almacena el número de línea en el parseo actual, comenzando desde 1. Por otro lado, `$prevline` almacena el número de línea del último caracter que fue parseado con éxito, por lo que será distinto de `$thisline` sólo al final de cada línea.

3.3.1.5.5. $thiscolumn y $prevcolumn: Al igual que las variables `$thisline` y `$prevline`, `$thiscolumn` y `$prevcolumn` almacenan los números de columna del carácter actual que se está parseando, y del último carácter que fue parseado con éxito. En general,

`$prevcolumn==$thiscolumn-1`, excepto al final de la línea. A diferencia de `$prevline` y `$thisline`, a las variables `$thiscolumn` y `$prevcolumn` no se les puede asignar un valor.

Hasta aquí hemos visto una introducción al uso del módulo Parse::RecDescent, abordando todas las potencialidades que serán utilizadas en el desarrollo del trabajo. Se pueden obtener más detalles a cerca de este módulo ingresando al sitio oficial[2].

3.3.2. Consideraciones para la implementación

El proceso de compilación comienza analizando el código fuente. Ni bien comienza dicho proceso, y se detecta que se han declarado variables, no podemos saber dónde serán utilizadas. Tampoco podemos saber si al momento de ser utilizada, habrá otra variable con un scope más cercano que esté siendo referenciada. Cuando realizamos un salto relativo a la posición de la instrucción (por ejemplo al declarar una función) no podemos saber cuál será el destino hasta no tener todas las instrucciones generadas. Para solucionar estos problemas, se utiliza un conjunto de marcadores que deben ser reemplazados por los valores correspondientes al finalizar el parseo.

3.3.2.1. Variables

Para el caso de las variables, se guarda la información en una tabla hash que dado un nombre de variable y su scope, se obtiene la dirección de memoria en que fue guardada. Para variables globales, se utiliza el scope GLOBAL, mientras que para variables locales a funciones, se utiliza el nombre de la función.

Existe una variable especial que puede utilizarse para referenciar al hilo actual. Esta variable es `numHilo`, y si se observa la tabla hash, es una variable global cuyo valor es `RRRnumHilo`. Este valor será traducido a un indicador que, al encontrarlo, el intérprete lo reemplazará por el identificador de cada hilo, en tiempo de ejecución.

Cada vez que en el código en lenguaje C3P se hace referencia a una variable, la instrucción Simplesem relacionada a dicha variable se genera manteniendo el nombre de la variable, y anteponiendo el símbolo "%%%" o "???" de acuerdo a si las variables están como destino u origen,

[2] http://search.cpan.org/~jtbraun/Parse-RecDescent-1.967009/lib/Parse/RecDescent.pm

respectivamente. Esto se debe a que si la variable es el destino de la operación, reemplazaremos este nombre por su dirección de memoria, pero si la variable es el orígen de la operación (y por ende nos interesa su valor), debemos reemplazar el nombre de la variable por D[posicion_de_memoria], ya que esta es la manera que tiene una instrucción Simplesem para obtener el valor almacenado en una posición de memoria. Una vez procesado todo el código y conocidas las posiciones de las variables, estos indicadores, junto con el nombre de cada variable, se modifican por su posición de memoria en el modelo Simplesem.

3.3.2.2. Saltos

El manejo de los saltos presenta otro problema. En general, cuando el parser detecta un salto no puede conocer cuántas instrucciones deben saltarse ya que todavía no las generó. Los saltos pueden ocurrir básicamente en tres reglas: llamadas a función (regla `call`), bloques if o if-else (regla `condition`), o en bucles while, for, o parallel for.

De acuerdo a la definición del lenguaje C3P, luego de declaradas las variables, puede declararse una función. Sin embargo, al generar las instrucciones Simplesem, el código correspondiente a la función deberá ser colocado a continuación de las instrucciones del programa principal (de acuerdo a cómo está definido el modelo Simplesem), por lo que el compilador almacena todas las instrucciones de una función para colocarlas una vez generadas las instrucciones de la función principal. La dirección de destino de un salto utilizado para una llamada a función, será la de la primera instrucción de la función, por lo que podremos conocerla fácilmente si llevamos un conteo de la cantidad de instrucciones generadas, y detectamos cuando todas las instrucciones de la función principal fueron generadas.

En los bloques if-else, los saltos siempre serán hacia adelante. No podemos conocer la dirección en la que aparecerá la instrucción de salto en tanto no conozcamos cuántas instrucciones hay antes de esta, y no contamos con este dato hasta no finalizar la generación de las instrucciones del scope donde se encuentra la instrucción de salto. Sin embargo, podemos conocer la cantidad de instrucciones que forman el bloque if o else, ya que pertenecen al mismo scope donde debe aparecer la instrucción de salto. Por ello, en las instrucciones de salto hacia adelante se coloca el indicador +++[offset], para que una vez conocida la posición en la que aparece la

instrucción de salto se reemplace este indicador por dicha posición más el offset.

Con los bucles while, for y parallel for ocurre algo similar, solo que los saltos ahora pueden ser hacia adelante (en la evaluación de la condición de los bucles) o hacia atrás (cuando se retorna al comienzo del bucle). Nuevamente, al llevar un conteo de la cantidad de instrucciones que hay dentro del bucle podemos conocer el offset necesario para volver al comienzo del bucle o para salir del mismo. Sin embargo no conocemos la posición absoluta de la instrucción de salto, por el mismo motivo que en el caso de los bloques if-else, por lo que se coloca el indicador +++[offset+1] cuando debe ser un salto hacia adelante, para salir del bucle, y —[offset] cuando debe ser un salto hacia atrás, para la siguiente iteración del bucle.

3.4. Intérprete Simplesem

Con las instrucciones Simplesem obtenidas, el siguiente paso es desarrollar un entorno de simulación que nos permita seguir la ejecución del programa. Para ello se diseñó e implementó un intérprete integrado con una interfaz gráfica, que en su conjunto permiten escribir un programa en lenguaje C3P, compilarlo a instrucciones Simplesem y ejecutarlo paso a paso.

3.4.1. Requerimientos Funcionales del intérprete

Para que la herramienta cumpla el objetivo de ser utilizada con fines educativos, el intérprete debe tener ciertas características que permitan realizar un análisis de la relación existente entre el código escrito en lenguaje C3P, y las instrucciones Simplesem equivalentes. Para ello, se definió un conjunto de requerimientos con los que el intérprete debió cumplir:

- Presentar la memoria de código de cada hilo y la memoria de datos de la máquina Simplesem en una interfaz gráfica.
- Posibilitar la ejecución de cada hilo por separado.
- Posibilitar la ejecución de una instrucción en todos los hilos, en paralelo.
- Brindar un mecanismo que permita escribir código fuente en lenguaje de alto nivel, y compilarlo a lenguaje Simplesem.
- Mostrar la salida en pantalla del programa que se está ejecutando.

- Obtener un valor de entrada del teclado cuando aparezca una instrucción que así lo requiera.
- Poseer ejemplos que muestren las distintas funcionalidades del lenguaje.

3.4.2. Diseño de la interfaz gráfica

Al momento de comenzar con este proyecto se planteó la posibilidad de lograr una herramienta multiplataforma, por lo cual se utilizó lenguaje `Perl` para el desarrollo del compilador, y lenguaje C++ con las librerías `Qt` para el desarrollo del intérprete. `Qt` cuenta con un conjunto de librerías para implementar una interfaz gráfica multiplataforma, utilizando lenguaje C++, como así también otras funcionalidades que agilizarían el desarrollo, como por ejemplo librerías para procesamiento de cadenas con expresiones regulares, o para ejecución de scripts.

La siguiente sección constituye una introducción a la biblioteca de clases de Qt, que presenta conceptos fundamentales incorporados por el framework de Qt, como el manejo de señales y slots.

3.4.2.1. Biblioteca Qt

En esta sección presentaremos una breve introducción a la utilización de la biblioteca de clases `Qt`. El contenido de esta sección fue tomado del sitio oficial de Qt [QtSite]. No pretendemos lograr un conocimiento profundo de estas librerías, sino sólo lo necesario para comprender las características utilizadas en el desarrollo de intérprete Simplesem. Esta biblioteca de clases conforma un framework en lenguaje C++ que permite incluir características fundamentales para cualquier producto de software, de manera sencilla y ordenada, como por ejemplo el desarrollo de interfaces gráficas multiplataforma. Comenzaremos analizando cómo puede armarse la interfaz gráfica de manera interactiva, y luego veremos cómo dar funcionalidades a los elementos de la interfaz.

3.4.2.2. Interfaz gráfica

Para el desarrollo con librerías Qt utilizaremos el entorno de desarrollo brindad por Nokia en [QtSite]. El SDK que puede descargarse del sitio incluye la biblioteca de clases Qt, un entorno de desarrollo integrado (Qt Creator) y un conjunto de herramientas utilizadas para el proceso de compilación del código. En el apéndice C, **Instalación** se indican los pasos necesarios para

instalar y configurar el SDK en distintos sistemas operativos.

Una vez iniciado Qt Creator, podemos crear un nuevo proyecto en el menú `File >New File or Project`. Para utilizar el editor de interfaz gráfica, debemos crear una aplicación del tipo `"Qt Gui Application"`. Una vez creado el proyecto, en la estructura de archivos veremos tres carpetas: `Headers`, `Sources` y `Forms`, además de un archivo con extensión `.pro` que se utilizará para configurar el proyecto.

Si analizamos el código fuente, veremos que la aplicación está compuesta por una pantalla principal, que posee un objeto que representa la interfaz gráfica, al cual puede referirse por defecto como `ui`. Esta pantalla principal será la que se muestre ni bien se ejecute el programa. Podemos modificar la interfaz gráfica, utilizando el editor brindado por `Qt Creator`. Para ello, abrimos el archivo con extensión `.ui` de la carpeta `Forms` y automáticamente se ejecutará el editor. La Figura 3.4 muestra cómo se ve el editor de Qt Creator.

El editor tiene cuatro secciones fundamentales, que utilizaremos con frecuencia:

- La primera sección, a la izquierda, se denomina `Widget Box`. De esta "caja" podremos sacar todos los elementos que queramos que compongan nuestra interfaz gráfica, como botones, listas desplegables, etiquetas, cuadros de texto, etc. Para colocarlos en la interfaz, bastará con arrastrarlos a la ventana de nuestra aplicación.

- La sección central es el `WorkSpace`, o espacio de trabajo, donde podremos ver cómo queda el diseño de nuestra aplicación

- En la parte inferior derecha tenemos una sección denominada `Property Editor`. Esta sección es sumamente importante ya que nos permitirá editar las propiedades de cualquier elemento de la interfaz, como el tipo de letra que utilice, el color, el tamaño, habilitarlo o deshabilitarlo, entre otras. Esta sección puede utilizarse sobretodo para definir el estado inicial de los elementos, pero a medida que la aplicación pase por distintos estados, será sumamente necesario utilizar las funciones que establecen estas propiedades a través del código fuente.

- Finalmente, la sección superior derecha, conocida como `Object Inspector` permitirá ver la estructura de organización de los elementos dentro de la ventana de la aplicación.

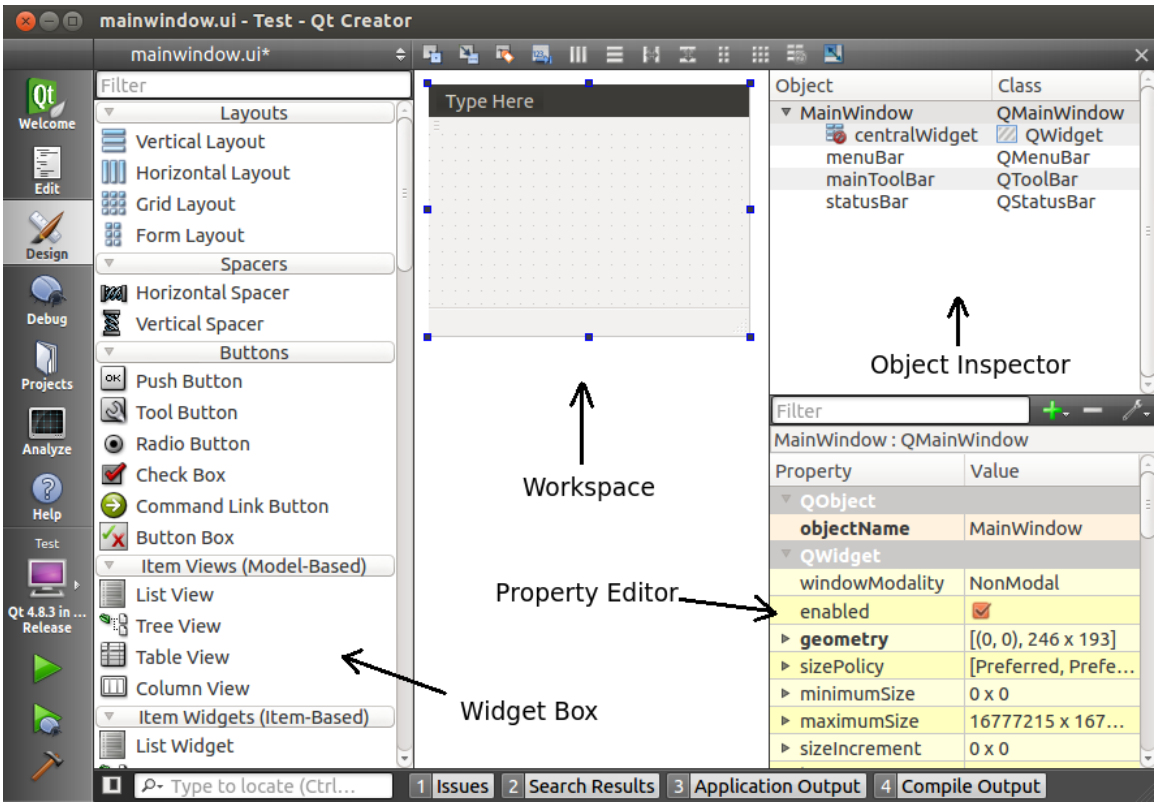


Figura 3.4: Editor de Interfaz Gráfica de Qt Creator.

Es fundamental tener en cuenta los datos presentados en el inspector de objetos. Bajo el título de `.°bject"`, podemos ver que aparece el nombre que se le dará a cada elemento de la interfaz gráfica. En el código fuente de la aplicación, el objeto llamado `ui`, que representa la interfaz gráfica del programa, posee un puntero a cada elemento de la ventana, cuyo nombre será el mismo con el que aparece en el editor de interfaz gráfica.

Una vez definidos los elementos de la interfaz gráfica, necesitaremos implementar la manera en que interactúa el programa con dicha interfaz. Para ello haremos uso de un mecanismo que utiliza Qt, conocido como `"Signals and Slots"`.

3.4.2.3. Signals y Slots

El mecanismo de Signals and Slots, consiste en un intercambio de mensajes entre dos objetos. Es una de las características más importantes de Qt, y su uso nos permitirá implementar la comunicación entre objetos de

manera sumamente simple. Cuando trabajamos con interfaces gráficas de usuario, es común que al interactuar con algún elemento, queramos modificar el estado de otros objetos del programa. Para ello necesitamos un mecanismo que nos permita que un objeto notifique a otro de dicho cambio, de manera tal que el receptor ejecute algún método para modificar su estado actual.

Una *señal* puede ser emitida por cualquier objeto cuando ocurre un evento en particular. La mayoría de las clases de Qt ya tienen implementadas muchas de las señales más utilizadas. Por ejemplo un botón emite señales cuando se lo presiona y otra cuando lo soltamos.

Un *slot* es una función que es llamada cuando se recibe una señal en particular. Las clases de Qt también tienen definidos numerosos slots por defecto, pero en general deberemos definir nuevos Slots ya que el comportamiento deberá amoldarse a nuestra aplicación. Cabe aclarar que los slots son métodos que pueden ser llamados explícitamente, y que poseen un valor de retorno que sólo será utilizado en este caso.

Cuando emitimos una determinada señal, suele ser necesario enviar ciertos parámetros al objeto receptor, para que este los utilice al modificar su estado. La signatura de las señales debe tener por lo menos la misma cantidad de parámetros que el slot que recibe dicha señal, y del mismo tipo. Si el slot tiene menos cantidad de parámetros que la señal, se ignoran los parámetros extra. Para realizar la conexión de una señal con un slot, se deberá utilizar la función connect, que está definida en la clase QObject, de la cual deben heredar todos los objetos utilizados en Qt. Además, ni bien comience la definición de la clase deberá ejecutarse la macro `Q_OBJECT`, que se encargará de preparar el objeto para su utilización.

Supongamos ahora que tenemos un objeto `objectA`, que emite una señal llamada `signalA` que recibirá un objeto `objectB`, y que al recibirla ejecutará el código que se encuentra en `slotB`. Además, supongamos que deseamos pasar un parámetro a través de la señal, para que sea utilizado en el slot. La llamada al método `connect` por lo general se realiza en el constructor del objeto que recibe la señal (`objectB`). A continuación se muestra un ejemplo para el escenario propuesto.

```
QObject::connect(&objectA, SIGNAL(signalA(int)),
&objectB, SLOT(slotB(int)));
```

Pueden conectarse tantas señales como se quiera a un slot, y una misma señal puede ser conectada a tantos slots como se desee. La sintaxis para definir los slots y las señales de una clase puede verse en el Código 3.9.

Código 3.9: Declaración de una clase que implementa Signals and Slots

```
1  #include <QObject>
2
3  class Counter : public QObject
4  {
5      Q_OBJECT
6
7  public:
8      Counter() { m_value = 0; }
9
10     int value() const { return m_value; }
11
12 public slots:
13     void setValue(int value);
14
15 signals:
16     void valueChanged(int newValue);
17
18 private:
19     int m_value;
20 };
```

Nótese que la clase hereda de QObject y ejecuta la macro Q_OBJECT antes de comenzar la definición de la clase. Como puede verse en la signatura de la señal y del slot implementado en este ejemplo, el valor de retorno de una señal siempre deberá ser void, pues una señal consiste únicamente en un mensaje emitido para dar aviso de un evento ocurrido, y no es una función en si. Por otro lado, los slots puede tener un valor de retorno. Cuando un slot se ejecuta tras haber recibido una señal, el valor de retorno no se utilizará, ya que no es llamado siempre desde el mismo lugar (problema de indeterminismo). Sin embargo los slots pueden ejecutarse como métodos tradicionales, llamados explícitamente, por lo que el valor de retorno podrá ser utilizado en estos casos.

Supongamos ahora que llega un momento de nuestro programa en el que queremos emitir la señal `signalA` del objeto `objectA` para que todos los slots conectados a esta señal, como el `slotB` sean ejecutados. Para ello, en el momento en que queramos emitir la señal bastará con utilizar la instrucción `emit` como muestra el Código 3.10.

Código 3.10: Emisión de una señal dentro de un método

```
1 void objectA::changeValue(int value){
2     if(value != actualValue){
3         actualValue = value;
4         emit valueChanged(value);
5     }
6 }
```

La línea 4 se encargará de emitir una señal indicando que el valor ha cambiado, y pasando el nuevo valor como parámetro, para que cualquier objeto que tenga un slot conectado a esta señal, tome el nuevo valor y modifique su estado de acuerdo a ello.

Hasta aquí hemos visto las características fundamentales que se utilizaron en el desarrollo del intérprete Simplesem: el diseñador de interfaces gráficas, y el mecanismo de Signals and Slots. Durante el desarrollo de la herramienta se utilizaron numerosas clases de la biblioteca Qt, por lo que, de requerir mayor información, puede dirigirse a la documentación oficial de Qt en [QtSite].

3.4.2.4. Arquitectura del intérprete

Una vez definida la tecnología que íbamos a utilizar, comenzamos con el diseño del intérprete. Se planteó una arquitectura Model-View-Controller debido a la existencia de una interfaz gráfica, y un conjunto de objetos que representan a la máquina Simplesem (modelo). El controlador de esta interfaz gráfica será el que conecte todos los elementos de la misma con las tareas que deben ser realizadas, como por ejemplo la compilación del código, o la ejecución de una instrucción.

Qt brinda un mecanismo que permite tomar los elementos definidos en la interfaz gráfica y utilizarlos como si fueran objetos programáticamente definidos (a través del código). De esta manera podemos diseñar la interfaz gráfica con una herramienta visual, conectar los elementos de la interfaz a objetos de nuestro programa, e interactuar entre la interfaz gráfica y la lógica de la aplicación a través del controlador.

La interfaz gráfica se diseñó para cumplir con los requerimientos, y brindar una manera intuitiva y sencilla de realizar la escritura del código, la compilación del mismo y la ejecución de las instrucciones Simplesem. La Figura 3.5 muestra el resultado final del diseño de la interfaz gráfica.

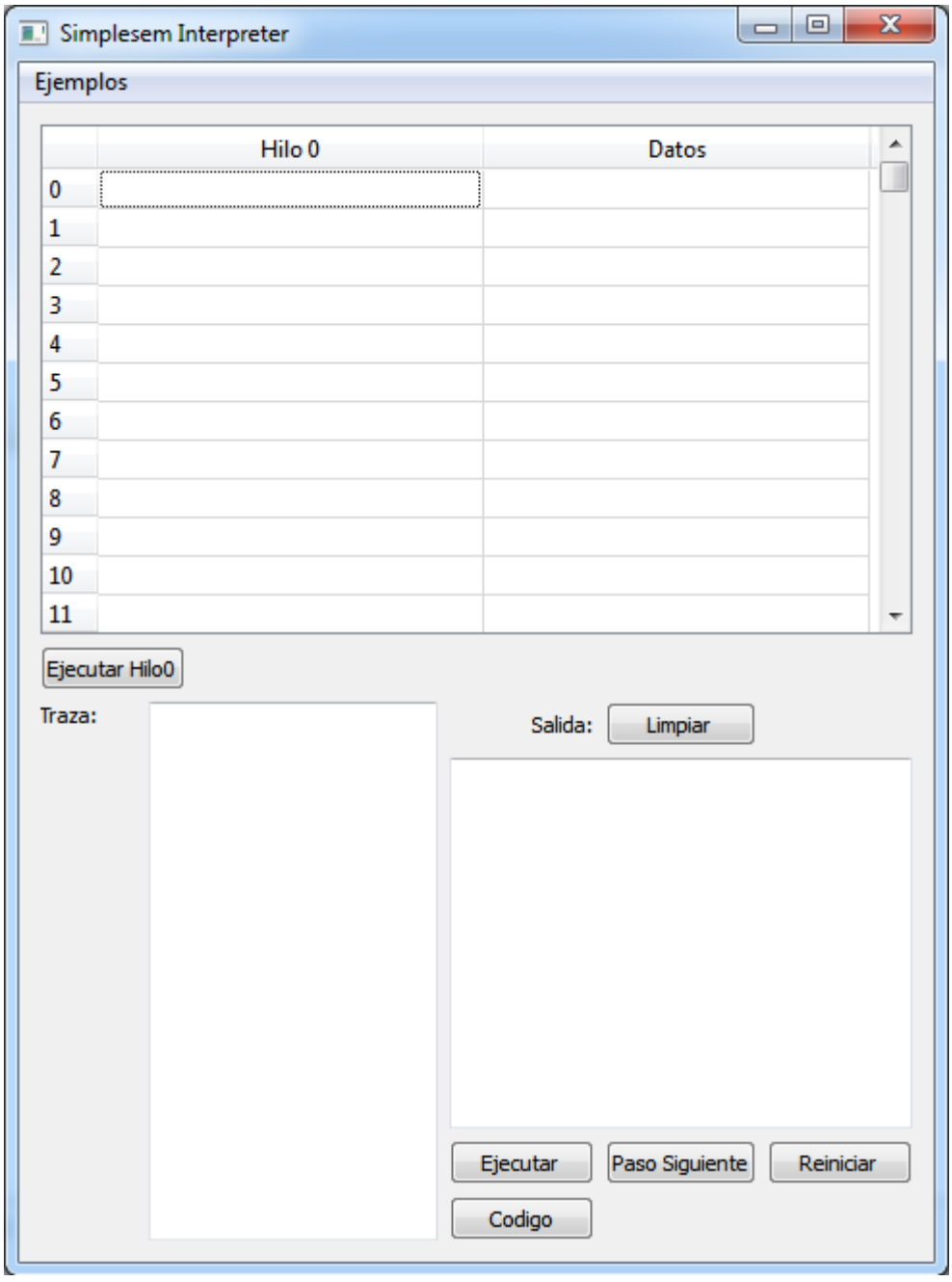


Figura 3.5: Interfaz gráfica del intérprete de Windows

La parte superior muestra el estado de la memoria. Las columnas tituladas "Hilo X" muestran las instrucciones que se encuentran en la memoria de código de cada hilo. La última columna (titulada "Datos") muestra el estado de la memoria de datos. Durante el tiempo de vida de cada

hilo, existirá un botón para ejecutar una única instrucción de un hilo en particular. En la Figura 3.5 sólo puede observarse un hilo, por lo que hay un único botón para ejecutar una instrucción de ese hilo.

En la sección inferior derecha pueden verse los botones de control de la aplicación. El botón "Ejecutar" permite ejecutar el programa hasta que se encuentre una instrucción `halt`. El botón "Paso Siguiente" ejecuta una instrucción de cada hilo, simulando una ejecución en paralelo. El botón "Reiniciar" permite volver la ejecución de un programa a su estado inicial (memoria de datos vacía, sólo existe el hilo 0, y su IP apunta a la instrucción 0). Finalmente, el botón "Código" o "Traza" permitirá intercambiar la vista para mostrar el código fuente o la traza de instrucciones ejecutadas, en el cuadro de texto inferior.

Para la captura de la salida en pantalla, se generó un cuadro de texto que cumple la función de consola, donde se imprime la salida de las instrucciones `write` del programa. El botón "Limpiar" permite borrar todo el contenido del cuadro de texto de la consola. Si bien aquí se presenta el diseño final de la interfaz gráfica, se realizó una primera aproximación a cómo sería esta para imaginar cómo debiera interactuar el usuario al utilizar la herramienta.

Aunque ya se habían planteado requerimientos para el desarrollo, se consideró en todo momento la posibilidad de ampliar las funcionalidades de la aplicación, por lo que ponernos del lado del usuario en una etapa temprana del desarrollo permitió identificar necesidades que debían ser implementadas. Una de ellas fue la necesidad de brindar la posibilidad de escribir código fuente en lenguaje C3P y compilarlo en la misma aplicación, ya que en un principio se planteó el desarrollo de un intérprete sin un editor integrado.

3.4.3. Casos de Uso

Definidas las funcionalidades que debía brindar la herramienta, se plantearon los siguientes casos de uso para la aplicación:

Titulo	***Escribir Código en lenguaje C3P***
Objetivo	*Dar la posibilidad al usuario de escribir un programa en lenguaje C3P*
Contexto	*Aplicación iniciada*
Recursos	*Aplicación, Cuadro de Texto*
Actores	*Usuario, Controlador de la IGU*
Episodios	*- Controlador de la IGU muestra a Usuario un cuadro de texto.* *- Usuario escribe el código fuente en lenguaje C3P dentro del cuadro de texto.*

Figura 3.6: Caso de Uso N°1

Titulo	***Compilar Código Fuente***
Objetivo	*Compilar el código escrito por el usuario para obtener las instrucciones Simplesem correspondientes.*
Contexto	*Código fuente correctamente escrito por el usuario en el cuadro de texto brindado para tal fin.*
Recursos	*Compilador, Código Fuente, Botón de Compilación*
Actores	*Usuario, Controlador de la IGU, Compilador*
Episodios	*- Controlador de la IGU muestra un cuadro de texto para escribir el código fuente de la aplicación, y un botón para iniciar la compilación.* *- Usuario finaliza de escribir el código fuente.* *- Usuario presiona el botón para iniciar la compilación.* *- Controlador de la IGU toma el código fuente escrito y lo guarda en un archivo.* *- Controlador de la IGU indica al Compilador que abra dicho archivo y compile el código fuente que allí se encuentra.* *- Compilador genera las instrucciones Simplesem y las guarda en otro archivo.* *- Controlador de la IGU toma las instrucciones del archivo y las muestra en la interfaz gráfica.*

Figura 3.7: Caso de Uso N°2

Titulo	**Ejecutar Instrucciones**
Objetivo	Ejecutar las instrucciones Simplesem que conforman el programa.
Contexto	Código fuente en lenguaje C3P compilado, con las instrucciones Simplesem cargadas en la memoria de código.
Recursos	Instrucciones Simplesem
Actores	Usuario, Ejecutor de Instrucciones, Controlador de la IGU
Episodios	- Controlador de la IGU muestra todas las instrucciones en la memoria de código. - Usuario presiona un botón para ejecutar una instrucción de un hilo específico, o para ejecutar instrucciones de todos los hilos en paralelo. - Controlador de la IGU toma la siguiente instrucción a ejecutar, y la envía al Ejecutor de Instrucciones. - Ejecutor de Instrucciones procesa la instrucción, modificando la memoria de datos de ser necesario.

Figura 3.8: Caso de Uso Nº3

Titulo	**Reproducir Ejemplos**
Objetivo	Brindar un conjunto de ejemplos que abarquen todas las características del lenguaje
Contexto	Aplicación Iniciada.
Recursos	Menú de ejemplos, Interfaz gráfica
Actores	Usuario, Controlador de la IGU
Episodios	- Controlador de la IGU muestra un menú con el título "Ejemplos". - Usuario abre el menú "Ejemplos" y selecciona uno de los ejemplos integrados. - Controlador de la IGU escribe el código fuente del ejemplo en Lenguaje C3P en la interfaz gráfica. - Usuario presiona el botón de compilación en la interfaz gráfica. - Controlador de la IGU completa la memoria de código. - Usuario comienza a controlar la ejecución del Código de ejemplo.

Figura 3.9: Caso de Uso Nº4

Titulo	**Reiniciar ejecución**
Objetivo	Brindar un mecanismo que permita reiniciar la ejecución.
Contexto	Aplicación iniciada.
Recursos	Hilos, Interfaz Gráfica
Actores	Usuario, Controlador de la IGU
Episodios	- Controlador de la IGU muestra un botón con la leyenda "Reiniciar". - Usuario pulsa el botón Reiniciar. - Controlador de la IGU limpia la columna de la memoria de código, destruye todos los hilos y coloca el hilo principal apuntando a la primera instrucción.

Figura 3.10: Caso de Uso Nº5

Titulo	***Seguir la traza de instrucciones***
Objetivo	*Permitir que el usuario tenga acceso a la traza de instrucciones ejecutadas.*
Contexto	*Aplicación iniciada, alguna instrucción Simplesem ejecutada.*
Recursos	*Interfaz Gráfica*
Actores	*Usuario, Controlador de la IGU*
Episodios	*- Controlador de la IGU muestra un botón que permite que el usuario intercambie la vista entre la traza y el código.* *- Usuario presiona el botón.* *- El cuadro de texto que mostraba el código fuente muestra ahora la traza de instrucciones, o viceversa.*

Figura 3.11: Caso de Uso Nº6

Debemos notar en estos casos de uso que el compilador, que debiera ser un objeto, estará representado por un programa distinto, escrito en lenguaje Perl y ejecutado desde la aplicación con la interfaz gráfica cuando fuere necesario.

3.4.4. Diagrama de Clases

La clase controladora cumple tres funciones fundamentales:

- Administra los cambios de estado de los elementos de la interfaz gráfica, como los botones, o el contenido de las celdas
- Toma las instrucciones de las celdas que representan la memoria de datos, y pide su ejecución a otro objeto, el cual le devuelve el resultado. Para ello hace uso de dos clases más. Una representa el modelo de la aplicación, que en este caso será la información a cerca de los hilos (su IP actual, un identificador, y un indicador de la última instrucción ejecutada). La otra clase implementa el ejecutor de las instrucciones, que tiene como objetivo procesar adecuadamente cualquier instrucción que se le solicite.
- Carga un conjunto de ejemplos integrados a la aplicación. Dichos ejemplos se encuentran almacenados en otro objeto.

Para la comunicación entre la interfaz gráfica y las funcionalidades de la aplicación, se utiliza el mecanismo de "Signals and Slots" brindado por Qt. La Figura 3.12 muestra el diagrama de clases del intérprete. Allí se ve que el objeto MainWindow representará al controlador.

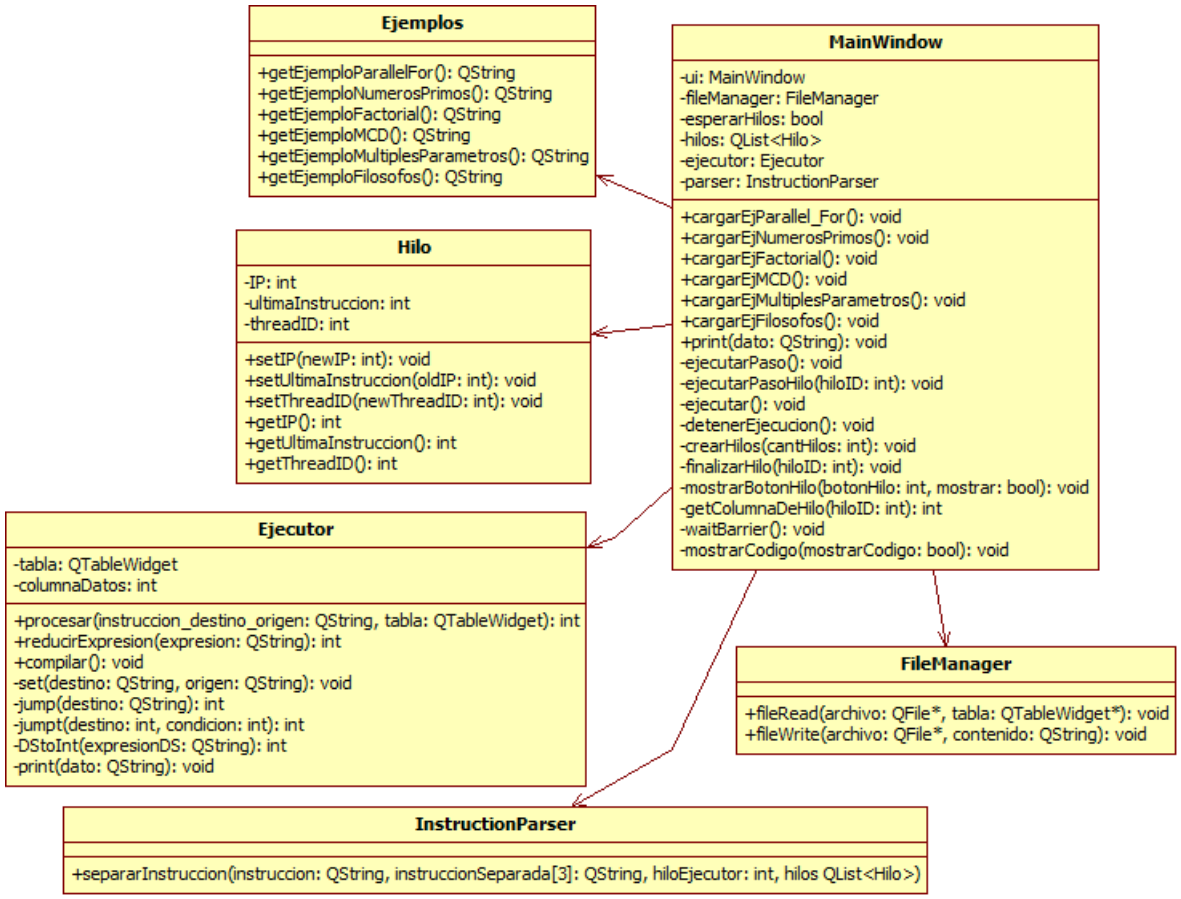


Figura 3.12: Diagrama de clases del intérprete Simplesem

Para simplificar el diagrama de clases, no se agregaron los SLOTS de cada clase, ya que los mismos no pertenecen a la interfaz pública. Sin embargo la función que cumplen es de suma importancia, por lo que los explicaremos a continuación.

- **on_botonEjecutar_clicked()**: Este método se llama cuando se presiona el botón "Ejecutar". Ejecuta todas las instrucciones hasta encontrar una instrucción halt (finalización del programa).
- **on_botonPaso_clicked()**: método llamado cuando se presiona el botón "Paso Siguiente", lo cual provoca que se ejecute una instrucción de cada hilo.
- **on_portada_clicked()**: al iniciar la aplicación, se muestra sólo una pantalla con un botón que contiene la leyenda "Extensión del modelo Simplesem a un Lenguaje Paralelo". Al presionar dicho botón, se ejecuta

el slot `on_portada_clicked`, que prepara la interfaz gráfica en un estado inicial.

- **on_botonReiniciar_clicked()**: método llamado cuando se presiona el botón "Reiniciar". Para reiniciar la ejecución del programa Simplesem, debemos limpiar la memoria de datos, finalizar todos los hilos que había en ejecución y colocar el IP del hilo principal en la posición 0.

- **on_botonHiloX_clicked()**: cada vez que se presiona un botón "Ejecutar HiloX" se llama al método correspondiente que realiza la acción de procesar una instrucción del hilo cuyo botón fue presionado.

- **on_botonCodigo_clicked()**: en la interfaz gráfica existe un botón que permite intercambiar el cuadro de texto de la esquina inferior izquierda para poder escribir código fuente y compilarlo, o para ver la traza de instrucciones ejecutadas. Al presionar dicho botón, se ejecuta este método cuya función es intercambiar el cuadro de texto mostrado, y modificar la leyenda del botón.

- **on_botonCompilar_clicked()**: este método se ejecuta al presionar el botón compilar, el cual no aparecerá si se está mostrando la traza de instrucciones ejecutadas. El método toma todo el texto que está escrito en el cuadro de texto del código fuente, y lo compila. Si el programa está correctamente escrito, se generan las instrucciones Simplesem correspondientes. En caso de que haya algún error de sintaxis, en la memoria de código aparecerán tres instrucciones. Las dos primeras prepararán la memoria de datos para la ejecución, y la tercera será una instrucción ARRAYhalt, que indica que no se produjeron instrucciones al realizar la compilación.

- **on_botonLimpiar_clicked()**: al presionar el botón "Limpiar" se ejecuta este método, que tiene por objetivo borrar todo el contenido del cuadro de texto que contiene la salida en pantalla.

- **cargarEjX()**: en el menú "Ejemplos", pueden seleccionarse algunos ejemplos integrados con la aplicación. Cada vez que se selecciona un ejemplo X, se ejecuta el método que carga dicho ejemplo en el entorno de desarrollo. Esta tarea consiste en mostrar el cuadro de texto que contiene el código fuente, y escribir todo el código correspondiente al ejemplo en dicho cuadro.

- **print(QString dato)**: este es el único ejemplo de un slot que recibe un valor como parámetro. Para ello, la señal conectada a dicho slot debe

enviar un objeto del mismo tipo (QString) que será recibido por el slot en cuestión. En este caso la señal no es emitida por la interfaz gráfica, sino por el objeto `ejecutor` que posee el controlador. La acción de imprimir un texto en la interfaz corresponde al controlador, pero el texto que deseamos imprimir lo conoceremos en el ejecutor. Para ello, el ejecutor emite una `señal` al controlador, indicando el texto que deberá imprimir. El controlador captura dicha señal, y la imprime en la interfaz, eliminando de esta manera la necesidad de conectar el ejecutor con la interfaz gráfica.

3.5. Pruebas con ejemplos en lenguaje C3P

Al finalizar el desarrollo del compilador, se realizaron pruebas para verificar que todas las reglas eran reconocidas correctamente, y que generaban las instrucciones Simplesem adecuadas. Para tal fin, se desarrolló un paquete de ejemplos, que abarca todas las reglas y las distintas alternativas que las componen. A través de estos ejemplos se obtuvieron todas las instrucciones Simplesem soportadas por la herramienta.

Las instrucciones obtenidas luego de compilar cada ejemplo fueron tomadas por el intérprete para realizar las pruebas en el mismo. El objetivo de estos ejemplos era lograr la correcta compilación y ejecución de las instrucciones Simplesem de cada programa. Al desarrollar estos casos surgieron ciertos problemas, de los cuales algunos requirieron ampliar la propuesta de extensión. Entre los principales problemas se destacan:

- Las llamadas a funciones sólo funcionaban correctamente si se encontraban dentro de una expresión compuesta, o dentro de una llamada a write, por ejemplo write(F()).
- Las funciones no podían recibir más de un parámetro.
- Existían situaciones que no estaban contempladas en el diseño del lenguaje de programación, como la necesidad de una instrucción atómica que realizara las funciones de una instrucción wait().
- Errores en el cálculo de los destinos de algunas instrucciones de salto.
- En el diseño del lenguaje, no se había contemplado la posibilidad de identificar de alguna manera al hilo que se estaba ejecutando.
- El intérprete no manejaba correctamente los botones de control al finalizar un programa, o al finalizar un hilo.

A continuación se presentan los programas en lenguaje C3P con su equivalente Simplesem, y una breve explicación de los mismos. Todos los ejemplos se encuentran integrados en la herramienta desarrollada, en el menú "Ejemplos", de manera tal que el lector pueda realizar la ejecución paso a paso utilizando el intérprete Simplesem.

3.5.1. Cálculo del factorial de un número

En este ejemplo se toma un número a través del teclado, y luego se calcula el factorial de dicho número. Existe un manejo de error para la entrada del usuario: si el número ingresado es negativo, la salida en pantalla será 0. El Código 3.11 muestra el programa escrito en lenguaje C3P, y la Figura 3.13 muestra cómo se organiza la memoria de la máquina Simplesem para este caso.

Código 3.11: Ejemplo del cálculo del factorial de un número en lenguaje C3P

```
1  var: n;
2  fact()
3  var: loc;
4  {
5      if (n > 1){
6          loc = n;
7          n = n - 1;
8          loc = loc * fact();
9          return loc;
10     }
11     else {
12         return 1;
13     }
14 }
15
16 program{
17     read(n);
18     if (n >= 0){
19         write(fact());
20     }
21     else{
22         write(0);
23     }
24 }
```

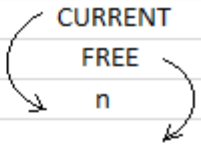
	Hilo 0	Datos
0	set 0, 0	
1	set 1, 3	
2	set 2, read	
3	jumpt 12, D[2] < 0	
4	set 1, D[1]+1	
5	set D[1], ip + 4	
6	set D[1]+1, D[0]	
7	set 0,D[1]	
8	set 1,D[1] + 3	
9	jump 14	
10	set write, D[D[1]-1]	
11	jump 13	
12	set write, 0	
13	halt	
14	jumpt 26, D[2] <= 1	
15	set D[0]+ 2, D[2]	
16	set 2, D[2] - 1	
17	set 1, D[1]+1	
18	set D[1], ip + 4	
19	set D[1]+1, D[0]	
20	set 0,D[1]	
21	set 1,D[1] + 3	
22	jump 14	
23	set D[0]+ 2, D[D[0]+2] * D[D[1]-1]	
24	set D[0]-1, D[D[0]+2]	
25	jump 27	
26	set D[0]-1, 1	
27	set 1,D[0]	
28	set 0, D[D[0]+1]	
29	jump D[D[1]]	

Figura 3.13: Cálculo del factorial de un número en lenguaje Simplesem

En este ejemplo, existe una variable global (n) que almacenará el valor del número del cual queremos calcular el factorial. Para esta variable, se reserva la celda 2 de la memoria de datos, por lo tanto la instrucción `read(n)` se representa en lenguaje Simplesem con la instrucción `set 2,`

`read`.

Luego se implementa la función `fact()`, cuyas instrucciones equivalentes en lenguaje Simplesem se ubican desde la celda 14 a la 29 de la memoria de código. Como dijimos anteriormente, las instrucciones correspondientes a la declaración de una función se colocan después de las instrucciones del programa principal, evitando de esta forma realizar un salto ni bien comienza la ejecución, para posicionarnos en la primera instrucción del programa principal.

Desde la instrucción de la celda 2 hasta la de la celda 13 representan el programa principal. La condición del if en el programa principal indica que si `n` es mayor o igual a 0, deberá ejecutarse el bloque if, y en caso contrario el bloque else. Con mayor frecuencia, los programadores colocan en el bloque `if` las instrucciones que se ejecutarán más a menudo, y en el bloque else alguna condición particular. En el modelo Simplesem el costo de realizar un salto condicional o incondicional es el mismo, ya que no se utiliza un pipeline para ejecutar las instrucciones. Sin embargo todos los procesadores reales de uso frecuente en el diseño de computadoras, utilizan un pipeline, en el que realizar un salto condicional provoca una pérdida de ciclos, debido a la necesidad de evaluar la condición del salto y, en caso de que este deba ser tomado, calcular la dirección de destino. Por todo esto, para perder la menor cantidad de ciclos posibles, suele invertirse la condición del salto (en este caso para `n >= 0` el procesador evalúa la condición `n <0`, con la instrucción de la celda 3), de manera tal que la mayor parte de las veces se ejecute el código que está a continuación de la instrucción de salto (correspondiente al bloque `if`). Nótese que la instrucción de la celda 11 (correspondiente al final del bloque `if`) también debe ser un salto, para no ejecutar la instrucción del bloque `else` (celda 12), pero esta vez se trata de un salto incondicional, cuya penalidad es menor.

Al realizar la llamada a una función, se definió un conjunto de instrucciones Simplesem que debían ser ejecutadas. Estas instrucciones se ubican desde la celda 4 a la celda 9 de la memoria de código, y realizan la preparación del registro de activación de la función, la modificación de los punteros FREE y CURRENT, y el salto a las instrucciones de la función, que comienzan desde la celda 14. La primera instrucción realiza la evaluación del salto.

Nuevamente, notamos que se invierte la condición, como dijimos, para minimizar la penalización por los saltos. En las instrucciones del bloque `if` analizaremos dos en particular. La instrucción de la celda 23 corresponde

a la línea `loc = loc * fact();` del código en lenguaje C3P. Debemos notar que cuando se ejecuta esta instrucción, es porque se ha vuelto de la función `fact()`, y se tiene el valor de retorno en la posición `D[1]-1` de la memoria de datos (la celda anterior al registro de activación de la función que devolvió ese valor). El valor de esta celda es escrito cuando se vuelve de la función `fact()`. La instrucción de la celda 24 representa la línea `return loc;` del bloque if, mientras que la instrucción de la celda 26 representa el return del bloque else.

3.5.2. Máximo Común Divisor

Con este ejemplo se pretende probar la compilación de dos ciclos while anidados. El programa presentado en el Código 3.12 tiene como objetivo encontrar el máximo común divisor de dos números ingresados por teclado. Para ello se implementa el algoritmo de Euclides, que se describe en tres etapas reiteradas.

- Dados dos segmentos AB y CD (con AB > CD), restamos CD de AB tantas veces como sea posible. Si no hay residuo, entonces CD es la máxima medida común.
- Si se obtiene un residuo EA, éste es menor que CD y podemos repetir el proceso: restamos EA tantas veces como sea posible de CD. Si al final no queda un residuo, EA es la medida común. En caso contrario obtenemos un nuevo residuo FC menor a EA.
- El proceso se repite hasta que en algún momento no se obtiene residuo. Entonces el último residuo obtenido es la mayor medida común.

Código 3.12: Ejemplo del cálculo del MCD entre dos números en lenguaje C3P

```
1  var: numMay, numMin, resto;
2  program {
3      resto = 1;
4      read(numMay);
5      read(numMin);
6      while(resto <> 0){
7          while(numMay >= 0){
8              numMay = numMay - numMin;
9          }
10         numMay = numMay + numMin;
```

```
11          resto = numMay;
12          numMay = numMin;
13          numMin = resto;
14          }
15          write(numMay);
16  }
```

La Figura 3.14 muestra la memoria de código y datos del modelo simplesem.

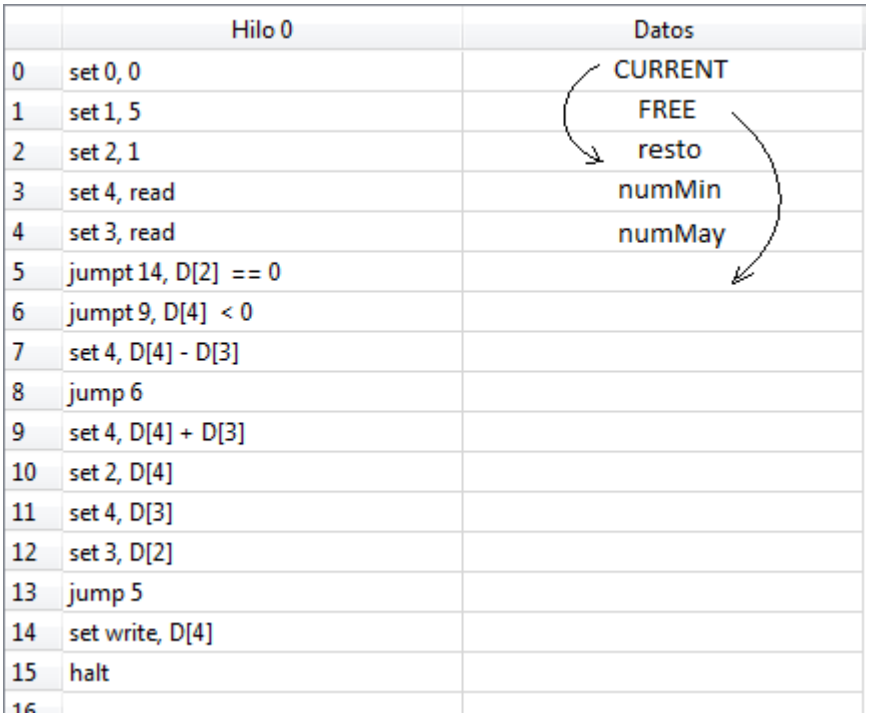


Figura 3.14: Cálculo del MCD de dos números en lenguaje Simplesem

En el algoritmo se utilizan dos ciclos `while`, uno que finaliza cuando `resto == 0`, y otro que finaliza cuando `numMay <0`. Las instrucciones que representan el bucle más externo van desde la celda 5 a la 13, mientras que las que representan el bucle interno van desde la 6 a la 8. Nuevamente podemos ver que en las instrucciones a bajo nivel (Simplesem), se invierte la condición para evitar tomar la mayor cantidad de saltos condicionales posibles, ya que en estos la penalidad es mayor. A diferencia de los bloques `if` que vimos en el ejemplo anterior, el salto incondicional del fin de cada iteración, tiene como destino el comienzo del bucle.

3.5.3. Números primos

Para probar el funcionamiento de los ciclos `for` y el manejo de arreglos, se utilizó un ejemplo de un programa que permite listar todos los números primos entre 0 y un número ingresado por el usuario, menor a 200 (que será el tamaño del arreglo). El Código 3.13 muestra el programa escrito en lenguaje C3P.

Código 3.13: Ejemplo del cálculo números primos en lenguaje C3P

```
1  var: esPrimo range 1..200,j, i, max;
2  program{
3      read(max);
4      max--;
5      for(j=0;j < max; j++){
6          esPrimo[j] = 1;
7      }
8
9      for(j=2;j < max; j++){
10         if(esPrimo[j]==1){
11             for(i=2*j;i < max; i=i+j){
12                 esPrimo[i]=0;
13             }
14         }
15     }
16     for(j=0;j < max; j++){
17         if(esPrimo[j]==1){
18             write(j);
19         }
20     }
21 }
```

En este programa se declaran tres variables (j, i y max) y un arreglo de 200 elementos (esPrimo). La variable `max` representa el número hasta el cual se quieren encontrar los números primos. El primer ciclo for se utiliza para indicar que, en principio, todos los números son primos, el segundo ciclo for se utiliza para determinar qué números son realmente primos, y el tercero permite imprimir los números primos en pantalla. A su vez, dentro del segundo ciclo hay otro en el cual la inicialización de la variable `i` depende del valor que tenga `j`, y a su vez el incremento no es unitario. La Figura 3.15 muestra las instrucciones Simplesem que representan al programa del ejemplo, y cuáles son las posiciones de memoria reservadas para cada variable.

	Hilo 0	Datos
0	set 0, 0	CURRENT
1	set 1, 205	FREE
2	set 2, read	max
3	set 2, D[2]-1	i
4	set 4, 0	j
5	jumpt 9, D[4] >= D[2]	esPrimo[0]
6	set 5 + D[4], 1	esPrimo[1]
7	set 4, D[4]+1	...
8	jump 5	...
9	set 4, 2	esPrimo[199]
10	jumpt 19, D[4] >= D[2]	
11	jumpt 17, D[5 + D[4]] != 1	
12	set 3, 2 * D[4]	
13	jumpt 17, D[3] >= D[2]	
14	set 5 + D[3], 0	
15	set 3, D[3] + D[4]	
16	jump 13	
17	set 4, D[4]+1	
18	jump 10	
19	set 4, 0	
20	jumpt 25, D[4] >= D[2]	
21	jumpt 23, D[5 + D[4]] != 1	
22	set write, D[4]	
23	set 4, D[4]+1	
24	jump 20	
25	halt	

Figura 3.15: Cálculo de números primos en lenguaje Simplesem

Debido a que un ciclo for puede ser representado como un ciclo while, agregando antes la inicialización de la variable de control y al final de cada ciclo el incremento de la misma, las instrucciones Simplesem que representan a un ciclo for serán las mismas que las del ciclo while sólo que se agregará una instrucción set, para inicializar la variable de control, y antes de cada salto al comienzo del ciclo, otra instrucción set para incrementar dicha variable.

El primer ciclo for está representado por las instrucciones desde la celda 4 hasta la celda 8. La instrucción `set 4,0` inicializa la variable j con valor

0. En la celda 5 se encuentra la evaluación de la condición, y en la celda 7 el incremento de la variable de control. Para continuar con la siguiente iteración se realiza un salto a la instrucción de la celda 5, que es la de evaluación de la condición. Esta estructura se repite para los otros tres ciclos for. Con este ejemplo se probaron las distintas alternativas con las que un ciclo for podía ser construido.

3.5.4. Suma de dos arrays en paralelo

En esta sección veremos un ejemplo de ejecución de 4 hilos en paralelo. El Código 3.14 muestra un programa que tiene como objetivo sumar todos los elementos de dos arreglos en paralelo.

Código 3.14: Suma paralela del contenido de dos arreglos en lenguaje C3P

```
1  var: a range 1..8, b range 1..8, suma, i range 1..4, j
       , sem;
2  program{
3          suma = 0;
4      for(j=0;j <= 7; j++){
5          a[j] = 1 + j;
6          b[j] = 2 + j;
7      }
8      sem = 1;
9      par_for 4 (i = 0;i <= 7;i++){
10         wait(sem);
11         suma = suma + a[i];
12         suma = suma + b[i];
13         notify(sem);
14     }
15     end_par_for;
16     write(suma);
17 }
```

Aquí vemos cómo los cuatro hilos en paralelo suman los elementos del array, y los guardan en la variable suma. Para evitar problemas de concurrencia, se utiliza un semáforo mutex que permite que sólo un hilo escriba la variable `suma`. Si bien carece de sentido que todo el bloque pueda ser ejecutado por un hilo a la vez, este ejemplo permite ver cómo se generan las instrucciones para la ejecución de múltiples hilos en paralelo. Debemos notar que para utilizar la variable `i` como variable independiente de cada hilo, debemos declarar un arreglo de nombre `i`, y de tantos elementos como

hilos queremos crear.

La variable que se encuentra en la definición del ciclo `par_for` será interpretada como la variable independiente, por lo que cada vez que se encuentre una referencia a dicha variable, se formará la dirección efectiva de esa variable como la base del arreglo, más un offset obtenido del identificador del hilo que ejecuta la instrucción. A su vez, cada hilo una vez creado entrará en un ciclo for convencional, cuyos límites estarán determinados por los valores de la variable `i` que deban procesar. Así, el hilo 0 ejecutará un ciclo for con `i` desde 0 hasta 1, el hilo 1 otro ciclo for con `i` de 2 a 3, etc. La Figura 3.16 muestra las instrucciones Simplesem en la memoria de código y la manera en que se asignan las celdas de la memoria de datos.

	Hilo 0	Datos
0	set 0, 0	
1	set 1, 25	
2	set 3, 0	
3	jumpt 8, D[3] > 7	
4	set 17 + D[3], 1 + D[3]	
5	set 9 + D[3], 2 + D[3]	
6	set 3, D[3]+1	
7	jump 3	
8	set 2, 1	
9	processors 4	
10	parallel_for 8, 0	
11	set 4 + numHilo, 0 + numHilo*2	
12	jumpt 19, D[4+numHilo] > 2*(nu...	
13	wait 2	
14	set 8, D[8] + D[17+D[4+numHilo]]	
15	set 8, D[8] + D[9+D[4+numHilo]]	
16	set 2, D[2]+1	
17	set 4 + numHilo, D[4+numHilo]+1	
18	jump 12	
19	end_par_for	
20	set write, D[8]	
21	halt	
22		

Figura 3.16: Suma paralela del contenido de dos arreglos en lenguaje Simplesem

La parte importante de este ejemplo es la definición del bloque de instrucciones que se ejecutará en paralelo, y qué instrucciones se utilizan para preparar el contexto de la máquina Simplesem para crear los hilos y comenzar su ejecución.

La primera instrucción relacionada a la primitiva `par_for` se encuentra en la celda 9. La instrucción `processors 4` indica al intérprete que deberá crear 4 hilos, inicializando su puntero de instrucción con el valor 10 (es decir que la primera instrucción que todos los hilos van a ejecutar es la primera del bloque `par_for`). La instrucción de la celda 10 no tiene efecto en sí. Sólo se utiliza para indicar al programador cuál será el rango que procesarán todos los hilos. En este caso, se procesarán 8 valores de `i` (8 - 0 = 8), por lo que cada hilo tomará dos de ellos.

Desde la instrucción de la celda 11 hasta la celda 18 podemos ver la misma estructura de un ciclo for convencional. Si bien en este caso todos los hilos ejecutarán las mismas instrucciones, debemos notar que se utiliza el indicador `numHilo` para determinar a qué variable independiente se hará referencia.

La instrucción de la celda 11 inicializa el valor de `i` para el ciclo for de cada hilo. La base del arreglo que contiene todas las variables independientes de cada hilo, se encuentra en la posición de memoria 4, por lo que `D[4 + numHilo]` representa la variable `i` de cada hilo. Si observamos el operando derecho de esta instrucción, `0 + numHilo * 2`, 0 representa el primer elemento que debe ser procesado, y el número 2 es la cantidad de valores que procesará cada hilo. En la instrucción de la celda 12 ocurre algo similar al generar la condición del salto. Los valores que tendrá la variable `i` para cada caso son:

- `hilo 0`: la variable i comienza en 0, y se producirá el salto cuando i sea mayor que 1.
- `hilo 1`: la variable i comienza en 2, y se producirá el salto cuando i sea mayor que 3.
- `hilo 2`: la variable i comienza en 4, y se producirá el salto cuando i sea mayor que 5.
- `hilo 3`: la variable i comienza en 6, y se producirá el salto cuando i sea mayor que 7.

Luego se ejecutarán todas las instrucciones correspondientes al bloque `par_for`, donde nuevamente las instrucciones producen resultados

distintos para cada hilo. Finalmente, en la celda 19 aparece la instrucción `end_par_for`, que indica que un hilo debe ser finalizado, excepto que sea el hilo principal (hilo 0) en cuyo caso deberá esperar que los demás hilos finalicen para continuar con su ejecución.

Los ejemplos aquí presentados pueden ser vistos en el intérprete Simplesem, en cualquiera de sus distribuciones, donde podrán ser ejecutados paso a paso para un análisis minucioso. Estos ejemplos abarcan todas las reglas de la gramática, y sus distintas alternativas, por lo que el criterio que se tuvo para considerar las pruebas satisfactorias, fue que los programas pudieran ser compilados y ejecutados correctamente, obteniendo el resultado esperado.

3.5.5. Problema de los Filósofos

Por último mostraremos un ejemplo clásico de concurrencia: el problema de la cena de los filósofos. El enunciado del problema es el siguiente. Cuatro filósofos se encuentran alrededor de una mesa cenando y pensando. Cada filósofo tiene un plato de comida y un cubierto a la izquierda de su plato. Para comer son necesarios dos cubiertos, y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo agarra un cubierto y el otro está ocupado, se quedará esperando para tomar el otro y empezar a comer.

Si dos filósofos adyacentes intentan tomar el mismo cubierto a una vez, se produce una condición de carrera: ambos compiten por el mismo recurso, y solo uno de ellos lo podrá adquirir. Esto provoca que si todos los filósofos toman el cubierto que está a su derecha al mismo tiempo, todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Este bloqueo mutuo se denomina interbloqueo o deadlock. El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

La solución planteada aquí consiste en dos condiciones: todos los filósofos toman primero el cubierto de la izquierda, excepto uno que tomará primero el de su derecha, y los filósofos liberan los cubiertos en orden inverso al que lo tomaron. La Figura 3.17 muestra el estado inicial del problema, en el cual ningún filósofo se encuentra comiendo, y todos los recursos (cubiertos) están disponibles.

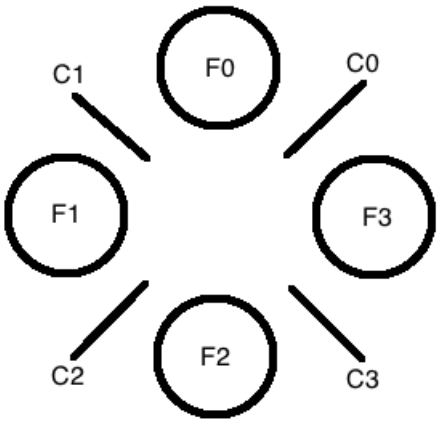


Figura 3.17: Problema de la cena de Filósofos

Como ya dijimos, si todos los filósofos toman el cubierto de su derecha, podría llegarse a un estado de interbloqueo. Si F3 es el filósofo que toma los cubiertos en distinto orden, la siguiente secuencia muestra la necesidad de la segunda condición (que los recursos se liberen en orden inverso al que fueron tomados).

- *F0* toma *C0*.
- *F1* toma *C1*.
- *F2* toma *C2* y luego *C3*, ya que *F3* no puede tomar *C3*.
- *F2* come y libera *C2*, pero no *C3*.
- *F1* toma *C2*, y como ya tenía *C1*, come y luego libera *C1*, pero no *C2*.
- *F0* toma *C1*, come y libera *C0*, pero no *C1*.
- *F3* toma *C0*, y ahora cada filósofo tiene un cubierto, por lo que llegan a un estado de interbloqueo

Si cada filósofo libera primero el último cubierto que tomó, este estado es inalcanzable.

Veremos a continuación el programa en lenguaje C3P que permite implementar la solución al problema de los filósofos, en el cual cada hilo representa a un filósofo.

Código 3.15: Problema de la cena de los filósofos en lenguaje C3P

```
1  var: tenedor range 1..4, filosofo range 1..4, j;
2
3  program{
4  //Problema de los filosofos implementado con semaforos
       .
5
6      //Inicializacion de los semaforos (tenedores).
7      for(j=0;j <= 3; j++){
8          tenedor[j] = 1;
9      }
10
11     //Cada filosofo toma el tenedor que esta a su
           izquierda y a su derecha.
12     //Estados:
13     //    x1 = Filosofo x Pensando
14     //    x0 = Filosofo x Comiendo
15     par_for 4 (filosofo = 0;filosofo <= 3; filosofo++)
           {
16         while(1 > 0){
17             if(filosofo == 3){
18                 //Pensando
19                 write(filosofo*10+1);
20
21                 //Toma los recursos
22                 wait(tenedor[0]);
23                 wait(tenedor[3]);
24
25                 //Comiendo
26                 write(filosofo*10+0);
27
28                 //Libera los recursos
29                 notify(tenedor[3]);
30                 notify(tenedor[0]);
31             }
32             else{
33                 //Pensando
34                 write(filosofo*10+1);
35
36                 //Toma los recursos
```

```
37                  wait(tenedor[filosofo]);
38                  wait(tenedor[filosofo+1]);
39
40                  //Comiendo
41                  write(filosofo*10+0);
42
43                  //Libera los recursos
44                  notify(tenedor[filosofo+1]);
45                  notify(tenedor[filosofo]);
46              }
47          }
48      }
49      end_par_for;
50  }
```

Como se ve en este ejemplo, el filósofo 3 será el que tome primero el cubierto de su derecha, y luego el de su izquierda (bloque *if*), mientras que los demás tomarán primero el de la izquierda y luego el de su derecha (bloque *else*). Además, todos liberarán primero el último cubierto que tomaron.

Cada filósofo puede estar en dos estados posibles: Pensando o comiendo. Para identificar estos dos estados, se imprimirá primero un número con el que se identifica al filósofo, y luego otro número que identifica su estado. (1 = Pensando, 0 = Comiendo). Debido a que por simplicidad el lenguaje solo utiliza valores enteros, y el valor 01 es equivalente al valor 1, para el filósofo 0, sólo se imprimirá el número 1 (si su estado es "Pensando") o 0 (si su estado es "Comiendo"). La Figura 3.18 muestra las instrucciones Simplesem generadas para este ejemplo.

En la figura se observa el estado de la memoria de instrucciones y de la memoria de datos, una vez inicializados los cuatro hilos que representan a los filósofos.

En las posiciones 7 a 11 de la memoria de datos puede verse el estado de los cubiertos, donde "1" representa un cubierto disponible, y "0" representa un cubierto ocupado.

	Hilo 0	Hilo 1	Hilo 2	Hilo 3	Datos
0	set 0, 0	set 0, 0	set 0, 0	set 0, 0	0
1	set 1, 11	set 1, 11	set 1, 11	set 1, 11	11
2	set 2, 0	set 2, 0	set 2, 0	set 2, 0	4
3	jumpt 7, D[2] > 3	jumpt 7, D[2] > 3	jumpt 7, D[2] > 3	jumpt 7, D[2] > 3	
4	set 7 + D[2], 1	set 7 + D[2], 1	set 7 + D[2], 1	set 7 + D[2], 1	
5	set 2, D[2]+1	set 2, D[2]+1	set 2, D[2]+1	set 2, D[2]+1	
6	jump 3	jump 3	jump 3	jump 3	
7	processors 4	processors 4	processors 4	processors 4	1
8	parallel_for 4, 0	parallel_for 4, 0	parallel_for 4, 0	parallel_for 4, 0	1
9	set 3 + numHilo, 0 + numH...	set 3 + numHilo, 0 + numH...	set 3 + numHilo, 0 + numH...	set 3 + numHilo, 0 + numH...	1
10	jumpt 29, D[3+numHilo] >...	jumpt 29, D[3+numHilo] >...	jumpt 29, D[3+numHilo] >...	jumpt 29, D[3+numHilo] >...	1
11	jumpt 27, 1 <= 0	jumpt 27, 1 <= 0	jumpt 27, 1 <= 0	jumpt 27, 1 <= 0	
12	jumpt 20, D[3+numHilo] !...	jumpt 20, D[3+numHilo] !...	jumpt 20, D[3+numHilo] !...	jumpt 20, D[3+numHilo] !...	
13	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	
14	wait 7+0	wait 7+0	wait 7+0	wait 7+0	
15	wait 7+3	wait 7+3	wait 7+3	wait 7+3	
16	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	
17	set 7+3, D[7+3]+1	set 7+3, D[7+3]+1	set 7+3, D[7+3]+1	set 7+3, D[7+3]+1	
18	set 7+0, D[7+0]+1	set 7+0, D[7+0]+1	set 7+0, D[7+0]+1	set 7+0, D[7+0]+1	
19	jump 26	jump 26	jump 26	jump 26	
20	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	
21	wait 7+D[3+numHilo]	wait 7+D[3+numHilo]	wait 7+D[3+numHilo]	wait 7+D[3+numHilo]	
22	wait 7+D[3+numHilo] + 1	wait 7+D[3+numHilo] + 1	wait 7+D[3+numHilo] + 1	wait 7+D[3+numHilo] + 1	
23	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	set write, D[3+numHilo] * 1...	
24	set 7+D[3+numHilo] + 1, ...	set 7+D[3+numHilo] + 1, ...	set 7+D[3+numHilo] + 1, ...	set 7+D[3+numHilo] + 1, ...	
25	set 7+D[3+numHilo], D[7...	set 7+D[3+numHilo], D[7...	set 7+D[3+numHilo], D[7...	set 7+D[3+numHilo], D[7...	
26	jump 11	jump 11	jump 11	jump 11	
27	set 3 + numHilo, D[3+num...	set 3 + numHilo, D[3+num...	set 3 + numHilo, D[3+num...	set 3 + numHilo, D[3+num...	
28	jump 10	jump 10	jump 10	jump 10	
29	end_par_for	end_par_for	end_par_for	end_par_for	
30	halt	halt	halt	halt	

Figura 3.18: Problema de la cena de Filósofos - Instrucciones Simplesem

Desde la posición 13 a la 18 de la memoria de cada hilo, se observan las instrucciones correspondientes al filósofo 3, que será el que tome primero el cubierto de su derecha (*C0*) y luego el de su izquierda (*C3*). Desde la celda 20 a la 25 se observan las instrucciones que ejecutarán los filósofos 0, 1 y 2. El conjunto de instrucciones de cada filósofo consiste en imprimir el estado "Pensando", luego tomar los dos cubiertos, imprimir el estado "Comiendo" y finalmente liberar los recursos.

Si comenzamos la ejecución del programa, veremos que el estado de los recursos (cubiertos) alterará entre los valores 0 y 1 de acuerdo a si están disponibles o no. Los filósofos tomarán los recursos de a uno a la vez, y lo que evitará llegar a un estado de interbloqueo será el orden en que los recursos son tomados y liberados por cada filósofo.

3.6. Conclusión

A lo largo de la carrera, al abordar el estudio los sistemas de cómputo, notamos la manera en que los alumnos comprendían su funcionamiento. La utilización de gráficos y celdas que representaran los recursos de la PC, como la memoria o el procesador, eran fundamentales para entenderlo, pero resultaba complicado para el alumno tomar notas debido a que estas no son dinámicas, y obtenían un esquema sólo del estado final del sistema. Surgió entonces la propuesta de incorporar una interfaz gráfica para facilitar el aprendizaje.

En este trabajo se propuso desarrollar una herramienta con fines educativos que permitiera transmitir el impacto que un lenguaje de alto nivel tiene sobre las instrucciones que finalmente ejecuta el procesador. Esta tarea a menudo resulta complicada de explicar, o requiere realizar un conjunto de gráficos confusos para el alumno, dificultando de esta manera el proceso de aprendizaje. Esta herramienta posibilita que los alumnos escriban un código fuente tan simple como una asignación, o tan complejo como el problema de los filósofos, y analicen paso a paso la manera en que se ejecutan las instrucciones Simplesem. El resultado obtenido al finalizar este proyecto satisface las expectativas con que se comenzó, brindando a la comunidad educativa una herramienta multiplataforma que cumpla las funciones con las que fue creada.

Esperamos que esta herramienta sea de utilidad al momento de abordar:

- La manera en que se define la gramática de un lenguaje de programación.
- La manera en que un texto es procesado para determinar si pertenece a un lenguaje de programación.
- Las distintas etapas del proceso de compilación de un programa.
- Las herramientas utilizadas para realizar el proceso de compilación (toolchain).
- La manera de representar la semántica de un lenguaje con instrucciones de bajo nivel.
- La ejecución en el sistema de computación de las instrucciones de bajo nivel.

3.7. Trabajo a futuro

A partir de aquí existen una gran cantidad de mejoras que podrían hacerse a la herramienta. Se detalla aquí una lista de los aspectos que podrían mejorarse en las distintas partes de la herramienta:

- **Definición del lenguaje**: pueden introducirse funcionalidades nuevas al definir nuevas reglas en la gramática del lenguaje C3P. La gramática propuesta contiene lo esencial de un lenguaje de programación, pero podrían incorporarse características como la definición de más de una función, o la existencia de tipos de datos. La definición del lenguaje es la base de todo el desarrollo, ya que será la que determine qué podrá hacerse, y qué no.

- **Compilador**: puede aumentarse la robustez del compilador al generar mensajes que indiquen al programador cuáles son los errores detectados, por ejemplo la falta de un punto y coma, o la utilización de una variable no declarada. Si se incorporó alguna nueva regla en la gramática del lenguaje, el compilador debería ser modificado para soportar dicha regla.

- **Intérprete**: puede agregarse la posibilidad de guardar archivos de código fuente, para que el usuario no deba volver a escribir el código cada vez que desee utilizar la herramienta. También puede agregarse un remarcado de la sintaxis, para marcar palabras claves, o bloques en el programa que el usuario está escribiendo. Por último, pueden incorporarse más ejemplos a los ya existentes.

Los aspectos a mejorar son ilimitados. Pueden tomarse ideas de los lenguajes de programación existentes, y agregar soporte a las funcionalidades que esos lenguajes poseen. Una refactorización del código fuente también es una buena posibilidad para aumentar la comprensión del mismo por parte de los desarrolladores.

Referencias

[Carloman76] Antonio Gonzales Carloman, *Lenguaje Matemático*, Publicaciones de la Universidad de Oviedo, 1976.

[Chomsky65] N. Chomsky, *Aspects of the Theory of Syntax*, Cambridge, MIT Press, 1965.

[Ghezzi96] Carlo Ghezzi y Mehdi Jazayeri, *Programming Language Concepts*, Tercera edición, 1996. Publicado el 23 de junio de 1997 por John Wiley & Sons. 448 Páginas. ISBN: 0471104264.

[Ruiz01] Edgar Ruiz L., *Lenguajes de Programación: Conceptos y Paradigmas*, Universidad Nacional Mayor de San Marcos. Lima, Perú. Versión digital disponible en http://sisbib.unmsm.edu.pe

[Goff01] Goff, Jeffrey, *Parse::RecDescent Tutorial*, Sitio oficial de Perl. 13 de Junio de 2001. Versión digital disponible en www.perl.com/pub/2001/06/13/recdecent.html

[Perl12] Sitio web oficial de Perl: www.perl.org Consultado en diciembre de 2012.

[CPAN12] Comprehensive Perl Archive Network: www.cpan.org Consultado en diciembre de 2012.

[LAST05] Lindner, Edgard, Molinari, Enrique y Matich, Graciela. *LaST: Language Study Tool*. Documento de Trabajo Número 143, Universidad de Belgrano. 2005. Disponible en la red: http://www.ub.edu.ar/investigaciones/dt_nuevos/143_lindner.pdf

[Blanchete08] Blanchete, Jasmin y Summerfield, Mark. *C++ GUI Programming with Qt 4*. 2da Edición. Publicado el 4 de Febrero de 2008. Ed Prentice Hall. 752 Páginas. ISBN: 0137143974.

[Litt01] Litt, Steve. *Perl Regular Expressions*. Disponble en: www.troubleshooters.com/codecorn/littperl/perlreg.htm

[Winzer98] Winzer, Alexander. *A SIMPLESEM Interpreter in Java*. Master's Thesis. Publicado el 2 de Febrero de 1998 en Vienna.

[Hernandez02] Hernandez-Campos, Felix. *Filminas de la materia COMP 144 Programming Language Concepts*, dictada en la universidad de North Carolina en Chapel Hill. Publicadas el 14 de Enero de 2002.

[Patterson06] Hennessy, John y Patterson, David A. *Computer Architecture: A Quantitative Approach*. 4ta Edición. Publicado el 27 de Septiembre de 2006 por Morgan Kaufmann. 704 Páginas. ISBN: 0123704901.

[QtSite] Sitio oficial de Qt:
http://www.developer.nokia.com/Develop/Qt/Tools/

[1] Laboratorio de Arquitectura de Computadoras. *Pautas para la elaboración de un informe*. Publicado el 2 de Julio de 2012 por el Laboratorio de Arquitectura de Computadoras de la Facultad de Ciencias Exactas, Físicas y Naturales de la Universidad Nacional de Córdoba.

Apéndice A

Gramática del lenguaje C3

Código A.1: Gramática del lenguaje C3

```
1  programm :  declaration fun_decla main
2
3  declaration : declare ";"
4              | ""
5
6  declare : "var:" declare_varlist
7
8  declare_varlist : declare_var "," declare_varlist
9                  | declare_var
10
11 declare_var : declare_array
12             | iden
13
14 declare_array : iden "range" range_dims
15
16 range_dims : range_value "," range_dims
17            | range_value
18
19 range_value : int".."int
20
21 fun_decla : iden "(" parameters ")" declaration
      statement_block
22           | ""
23
24 statement_block : "{" statements "}"
25
26 parameters : parameter "," parameters
27            | parameter
28
29 parameter : iden
30           | ""
31
32 statements : statement statements
33            | statement
34            | ""
35
36 statement : assign ";"
37           | "read" "(" iden ")" ";"
38           | "write" "(" expr ")" ";"
39           | iterate
40           | condition
41           | "return" expr ";"
```

```
42          | call ";"
43
44 main : "program" statement_block
45      | ""
46
47 assign : iden "=" expr
48        | var_inc
49        | var_dec
50
51 var_inc : iden "++"
52
53 var_dec : iden "--"
54
55 iterate : while_iteration
56         | for_iteration
57
58 while_iteration : "while" "(" bool_expr ")" statement_block
59
60 for_iteration : "for" for_definition statement_block
61
62 for_definition : "(" assign ";" bool_expr ";" assign ")"
63
64 condition : "if" "(" bool_expr ")" statement_block "else"
      statement_block
65          | "if" "(" bool_expr ")" statement_block
66
67 bool_expr :  expr comp expr
68
69 comp : "=="
70      | "<>"
71      | ">="
72      | "<="
73      | ">"
74      | "<"
75
76 call : iden "(" args ")"
77
78 args : expr "," args
79      | expr
80      | ""
81
82 exprs : expr exprs
83       | expr
```

```
84        | ""
85
86  expr : factor operator expr
87       | factor
88
89  factor : call
90         | iden
91         | int
92
93  int : sign digits
94
95  digits : /[0-9]+/
96
97  sign : "-"
98       | ""
99
100 iden : array_iden
101      | single_iden
102
103 single_iden : /[a-zA-Z]\\w*/
104
105 array_iden : single_iden array_pos
106
107 array_pos : "["array_dim_pos"]"
108
109 array_dim_pos : expr
110
111 letter_or_digits : /\\w*/
112
113 operator : "+"
114          | "-"
115          | "/"
116          | "*"
117
118 letter : /[a-zA-Z]/
119
120 comment : /\/\/.*/
```

Apéndice B

Gramática del lenguaje C3P.

Código B.1: Gramática del lenguaje C3P.

```
1  programm :  declaration fun_decla main
2
3  declaration : declare ";"
4              | ""
5
6  declare : "var:" declare_varlist
7
8  declare_varlist : declare_var "," declare_varlist
9                  | declare_var
10
11 declare_var : declare_array
12             | iden
13
14 declare_array : iden "range" range_dims
15
16 range_dims : range_value "," range_dims
17            | range_value
18
19 range_value : int".."int
20
21 fun_decla : iden "(" parameters ")" declaration
      statement_block
22           | ""
23
24 statement_block : "{" statements "}"
25
26 parameters : parameter "," parameters
27            | parameter
28
29 parameter : iden
30           | ""
31
32 statements : statement statements
33            | statement
34            | ""
35
36 statement : assign ";"
37           | "read" "(" iden ")" ";"
38           | "write" "(" expr ")" ";"
39           | iterate
40           | condition
41           | "return" expr ";"
```

```
42            | "wait" "(" iden ")" ";"
43            | "notify" "(" iden ")" ";"
44            | call ";"
45            | parallel_statement
46            | comment
47
48 main : "program" statement_block
49      | ""
50
51 assign : iden "=" expr
52        | var_inc
53        | var_dec
54
55 var_inc : iden "++"
56
57 var_dec : iden "--"
58
59 iterate : while_iteration
60         | for_iteration
61
62 while_iteration : "while" "(" bool_expr ")" statement_block
63
64 for_iteration : "for" for_definition statement_block
65
66 for_definition : "(" assign ";" bool_expr ";" assign ")"
67
68 condition : "if" "(" bool_expr ")" statement_block "else"
      statement_block
69          | "if" "(" bool_expr ")" statement_block
70
71 parallel_statement : parallel_for
72                    | "barrier" ";"
73
74 parallel_for : parallel_for_beginning statement_block
      parallel_for_end
75
76 parallel_for_beginning : "par_for" numbers for_definition
77
78 parallel_for_end : "end_par_for" ";"
79
80 bool_expr :  expr comp expr
81
82 comp : "=="
```

```
83       | "<>"
84       | ">="
85       | "<="
86       | ">"
87       | "<"
88
89  call : iden "(" args ")"
90
91  args : expr "," args
92       | expr
93       | ""
94
95  exprs : expr exprs
96        | expr
97        | ""
98
99  expr : factor operator expr
100      | factor
101
102 factor : call
103        | iden
104        | int
105
106 int : sign digits
107
108 digits : /[0-9]+/
109
110 sign : "-"
111      | ""
112
113 iden : array_iden
114      | single_iden
115
116 single_iden : /[a-zA-Z]\\w*/
117
118 array_iden : single_iden array_pos
119
120 array_pos : "["array_dim_pos"]"
121
122 array_dim_pos : expr
123
124 letter_or_digits : /\\w*/
125
```

```
126 operator : "+"
127          | "-"
128          | "/"
129          | "*"
130
131 letter : /[a-zA-Z]/
132
133 comment : /\/\/.*/
```

Apéndice C

Instalación

La siguiente sección conforma el manual de instalación que deberá leer todo aquel que desee utilizar la máquina Simplesem en su equipo. La herramienta fue probada bajo tres sistemas operativos: Windows 7, Linux Ubuntu 12.04 y Mac OS X Mountain Lion.

En el manual se indican cuáles son las herramientas necesarias para extender el desarrollo de la aplicación, ya sea para agregar funcionalidades o para corregir futuros errores. Además se incluye para cada sistema operativo, cuáles son las dependencias que deberán ser instaladas en caso de que sólo quiera utilizarse el programa con fines educativos.

La máquina Simplesem fue desarrollada en los lenguajes Perl y C++. El primero fue utilizado para la implementación del parser, el cual analiza el código fuente escrito en el lenguaje de alto nivel propuesto, y traduce las instrucciones de dicho lenguaje a instrucciones en lenguaje Simplesem. En esta etapa se utilizó además el módulo Parse::RecDescent, que incorpora un parser recursivo descendente al intérprete de Perl. Debido a que Perl es un lenguaje interpretado, se deberá instalar el intérprete de Perl, tanto para extenderse la aplicación, como para utilizarla sin realizar modificaciones. El módulo Parse::RecDescent está incluido en el paquete descargado, por lo que no será necesaria la instalación de este módulo.

Una vez obtenido el código Simplesem, se desarrolló un intérprete que permitiera analizar gráficamente cómo interactúa el código de los distintos hilos con la memoria de datos de la máquina virtual. Para ello se utilizó lenguaje C++ y la biblioteca de clases Qt. Si se desea extender el desarrollo presentado en este trabajo, recomendamos instalar el entorno de desarrollo Qt (Qt SDK) que puede ser descargado directamente desde la página del desarrollador[QtSite]. En caso de querer solo ejecutarse el intérprete, los requisitos varían de acuerdo al sistema operativo que se utilice.

La máquina Simplesem conforma un entorno de desarrollo que permitirá escribir código fuente en un lenguaje de alto nivel, compilarlo, y luego seguir paso a paso la ejecución de las instrucciones del programa, mostrando de un modo comprensible para el usuario la manera en que las instrucciones interactúan con la memoria de datos para lograr el propósito del programa.

Se indican a continuación los pasos necesarios para realizar la instalación de las herramientas que se utilizarán tanto para ejecutar la aplicación como para extender las características del intérprete en los distintos sistemas

operativos.

Existen tres herramientas utilizadas en el desarrollo de la aplicación, las cuales son:

- Intérprete de PERL
- Módulo Parse::RecDescent
- Qt SDK

Veremos cómo instalar cada una de estas herramientas en los distintos sistemas operativos.

C.1. Intérprete de Perl

El intérprete de Perl deberá estar instalado en el equipo donde la máquina Simplesem sea ejecutado. Para verificar si el intérprete de Perl está instalado, puede ejecutarse el comando `perl -v`. El módulo Parse::RecDescent, el cual es utilizado para generar el parser propiamente dicho, está incluido en el paquete publicado, junto con el código fuente de la máquina Simplesem, por lo que no será necesario instalarlo.

Las instrucciones para instalar el intérprete de Perl se colocan a continuación para distintos sistemas operativos donde fue probada la máquina Simplesem.

Linux Ubuntu 12.04: Por lo general las distribuciones de Linux incluyen el intérprete de Perl instalado. De no estar presente, el intérprete de PERL puede ser instalado con el siguiente comando:

```
sudo apt-get install perl
```

Finalmente, para verificar si el intérprete de perl fue correctamente instalado, se utiliza el siguiente comando:

```
perl -v
```

Mac OS X: Los sistemas operativos Mac OS X proveen el intérprete de Perl instalado por defecto. En caso de no estar presente, puede descargarse desde el sitio oficial de Perl: `www.perl.org`

Windows: Este sistema operativo no incorpora un intérprete de Perl instalado por defecto, por lo que el mismo deberá ser descargado desde el sitio `www.perl.org`. Se recomienda descargar el intérprete *Strawberry Perl*, con el cual fue probada la aplicación. En el sitio de descarga se brinda un ejecutable que permitirá instalar y configurar el intérprete de perl para ser utilizado desde la consola. Una vez instalado el intérprete, puede ejecutarse el comando `perl -v` para verificar la versión instalada.

C.2. Módulo Parse::RecDescent

Como se mencionó anteriormente, el módulo que implementa el parser (Parse::RecDescent) se brinda con el paquete distribuido.

Para que el script desarrollado en Perl detecte el módulo RecDescent, se deberá colocar, en el mismo directorio, el archivo `parser.pl` y la carpeta `Parse`, que contiene dentro el archivo `RecDescent.pm`. Una vez compilado el programa, deberá colocarse el archivo ejecutable en el mismo directorio donde se encuentra el archivo `parser.pl` y la carpeta `Parse`.

C.3. Qt SDK

Lo último que debemos instalar es el entorno de desarrollo de Qt. En el sitio [QtSite] se ofrece el entorno de desarrollo para Windows, Linux y Mac OS X. Se recomienda descargar el instalador offline, ya que dicha descarga es más rápida, y provee de un instalador desatendido. El archivo descargado es un ejecutable, por lo que en Linux deberá ser marcado como tal con el comando `chmod +x [nombre_del_archivo_descargado]`. Luego, al ejecutarlo, se instalará lo siguiente:

- **Qt Libraries:** Librerías Qt que fueron utilizadas para el desarrollo de la Máquina Simplesem.
- **Qt Creator:** Un entorno de desarrollo integrado para la gestión de proyectos, que permite utilizar tanto las librerias Qt como otras librerias externas.
- **Qt Tools:** Herramientas que simplifican el desarrollo. Entre ellas, la más importante es qmake, una herramienta que facilita la creación de un Makefile para linux.

Una vez instalado el SDK, podrá abrirse el proyecto y compilarse en cualquier plataforma. Para ello, debe eliminarse el archivo SimplesemInterpreter.pro.user en caso de que exista. Este archivo contiene información sobre un equipo en particular, que el compilador utilizará para realizar su trabajo. Al eliminar este archivo, una vez que se abra el proyecto se solicitará crearlo nuevamente, pero con la información del equipo local.

Luego deberá abrirse el archivo `SimplesemInterpreter.pro` con Qt Creator, y compilarlo. Para que la máquina Simplesem funcione correctamente, en la carpeta donde se encuentra el archivo de proyecto (.pro), deberá colocarse el archivo `parser.pl` y la carpeta `Parse` que contiene dentro el archivo `RecDescent.pm`. Si la ejecución de la máquina Simplesem se realiza directamente desde el ejecutable generado a partir de la compilación, el archivo `parser.pl` y la carpeta `Parse` deberán ser colocados junto con el archivo ejecutable.

Nota para Mac OS X: Al compilar el proyecto en Mac OS X se genera un paquete llamado `SimplesemInterpreter.app`. Para que la aplicación funcione correctamente, debemos colocar el archivo `parser.pl` y la carpeta `Parse` dentro de este paquete. Para ver el contenido del paquete, hacemos click derecho en él y seleccionamos la opción "Mostrar contenido del paquete". Luego ingresamos a la carpeta `Contents` y colocamos el archivo `parser.pl` y la carpeta `Parse` dentro de la carpeta `MacOS`

C.3.1. Instalación de las librerías para ejecución

Puede ocurrir que el usuario no desee modificar la máquina Simplesem, para lo cual no deberá instalarse el entorno de desarrollo de Qt, sino solo las librerías necesarias. A continuación se detalla cuál es el camino a seguir en cada sistema operativo si desea utilizarse la máquina Simplesem sin realizarle modificaciones.

Linux Ubuntu 12.04: El paquete provisto incluye el parser, el módulo Parse::RecDescent y el archivo ejecutable. En primer lugar se deberá marcar el archivo como ejecutable con el siguiente comando:

```
chmod +x SimplesemInterpreter
```

Luego, bastará con ejecutar el archivo `SimplesemInterpreter` para comenzar a utilizar la aplicación. Linux Ubuntu 12.04 trae por defecto las librerías en tiempo de ejecución de Qt, por lo que no debería ser necesario instalarlas, pero en caso de no disponer de dichas librerías, pueden instalarse con el siguiente comando:

```
sudo apt-get install qt-sdk
```

Este comando instalará todas las librerías necesarias para ejecutar la aplicación, como así tambien las herramientas necesarias para recompilar la aplicación si se realizaran modificaciones.

Mac OS X: El paquete provisto para Mac OS X es una imagen de disco que contiene dentro un paquete llamado `SimplesemInterpreter.app`. Este paquete contiene todo lo necesario para ejecutar la herramienta, por lo que para utilizarlo bastará con copiarlo a cualquier carpeta en la que se tengan permisos de escritura. Durante el proceso de compilación, la herramienta escribe archivos dentro del mismo paquete, por lo que dicho proceso no podrá ser llevado a cabo desde ninguna ubicación en la que no se tenga permiso de escritura.

Windows: En el paquete se brinda el parser, el módulo Parse::RecDescent, y el archivo ejecutable, junto con todas las librerías dinámicas (dll's) necesarias para ejecutar la aplicación, por lo que sólo bastará con abrir el archivo `SimplesemInterpreter.exe`.