Compiladores

TEORÍA E IMPLEMENTACIÓN

JACINTO RUIZ CATALÁN

libros R

COMPILADORES

Teoría e implementación

Jacinto Ruiz Catalán

COMPILADORES. Teoría e implementación

Jacinto Ruiz Catalán

ISBN: 978-84-937008-9-8

EAN: 9788493700898

RC Libros

Calle Mar Mediterráneo, 2

Parque Empresarial Inbisa, N-6 – P.I. Las Fronteras

28830 SAN FERNANDO DE HENARES, Madrid

Teléfono: +34 91 677 57 22

Fax: +34 91 677 57 22

Correo electrónico: info@rclibros.es

Internet: www.rclibros.es

Diseño de colección, preimpresión y cubierta: Grupo RC

Impresión y encuadernación: Gráficas Deva, S.L.

Depósito Legal: M-

Impreso en España

14-13 12 11 10 (03)

A la memoria de mi padre, Jacinto

A la memoria de Paqui

A Teresa, mi esposa. A mi hijo Alberto

A mi madre, María

A mis hermanos: Pepi, Manolo y Paco

ÍNDICE

Parte III. Implementación de C-0

Capítulo 13. Especificación de C-0

13.1 Introducción

13.2 Tokens

13.3 Constantes

13.4 Operadores y delimitadores

13.5 Identificadores y palabras reservadas

13.6 Tipos de datos

13.7 Sentencias de control de flujo

13.8 Instrucciones de entrada-salida

13.9 Declaración de variables

13.10 Programa principal

13.11 Sentencia if-then-else

13.12 Sentencia while

13.13 Ejemplo de programa válido

Capítulo 14. Análisis léxico, sintáctico y semántico de C-0

14.1 Análisis léxico

14.2 Análisis sintáctico

14.3 Análisis semántico

Capítulo 15. Generación de código intermedio de C-0

15.1 Introducción

15.2 Código de tres direcciones

15.3 Espacio de direcciones

AGRADECIMIENTOS

En primer lugar, quisiera agradecer la labor de mis profesores de la Universidad Nacional de Educación a Distancia por haberme dado la oportunidad de conocer el mundo de los compiladores. Sin sus indicaciones no podría haber adquirido los conocimientos suficientes para poder escribir este libro.

Agradezco también a los profesores del Departamento de Informática de la Universidad de Oviedo por sus excelentes manuales que ponen en internet a disposición del que los quiera estudiar.

Asimismo, agradezco al profesor Sergio Gálvez, de la Universidad de Málaga y tutor de la UNED, el haber puesto su libro, gratuitamente, en su página web. También le agradezco las horas que dedicó a explicarme aspectos desconocidos para mí sobre compiladores cuando fui alumno suyo.

Un agradecimiento especial a todos aquellos profesores, estudiantes y estudiosos que ponen sus artículos, resúmenes, manuales y cualquier otra información a disposición de la comunidad de internet.

Agradezco a Federico Javier Álvarez, el creador de Ens2001, el haber puesto a disposición de todos esta magnífica herramienta.

Y por último, y no menos importante, agradezco a mi mujer, Teresa y a mi hijo, Alberto, la oportunidad que me han dado para que haga realidad un sueño. Ya sólo me queda plantar un árbol.

PRÓLOGO

El presente libro pretende ser un manual de ayuda para estudiantes y estudiosos de procesadores de lenguajes y/o compiladores.

La teoría de compiladores es un mundo apasionante dentro de la informática. Pero a la vez complejo. El desarrollo de un compilador para un lenguaje medianamente potente es una tarea dura y costosa, tanto en tiempo como en recursos.

Pero al tiempo de ser una tarea costosa, puede ser muy gratificante, al implicar campos básicos de la informática como son la teoría de autómatas, de lenguajes, estructura y arquitectura de computadores, lenguajes de programación y algunos otros más.

Al construir un compilador, el desarrollador debe adentrarse en aspectos específicos de lo que es un computador y de lo que es un lenguaje de programación. Esto le da una visión profunda de aspectos clave del mundo de la informática.

Este libro pretende ocupar un espacio que está poco tratado, sobre todo en español. Se trata de la construcción de un compilador paso a paso, desde la especificación del lenguaje hasta la generación del código final (generalmente, un ejecutable). Hay muchos libros que tratan la teoría y algunos ejemplos más o menos complejos de compiladores. Existen también libros que desarrollan pequeños compiladores pero hasta ciertas fases, sin llegar a completar todo el proceso de desarrollo. Lo que pretendemos con este libro es dar las bases teóricas suficientes para poder abordar la construcción de un compilador completo, y luego implementarlo.

El libro consta de 5 partes, un prólogo y el índice.

En la parte I se analizan los aspectos teóricos de los procesadores de lenguajes y/o compiladores. Ocupa casi la mitad del libro. En estos ocho capítulos se desgranan las fases en las que se distribuye el proceso de creación de un compilador.

El capítulo 1 es una introducción, el capítulo 2 trata del análisis léxico, el 3 del sintáctico, el 4 del análisis sintáctico descendente, el 5 del ascendente, el 6 de la tabla de tipos y de símbolos, el 7 del análisis semántico y el 8 de la generación de código intermedio y final.

En las siguientes tres partes se desarrollan completamente sendos compiladores o traductores. Cada parte es más completa y compleja que la anterior.

En la parte II se desarrolla la creación de un traductor para un lenguaje de lógica de proposiciones. El lenguaje se llama **L-0**. En esta parte, el capítulo 9 trata la especificación del lenguaje, en el 10 se realiza el análisis léxico, en el 11 el análisis sintáctico y en el 12 el semántico y la generación de código.

En la parte III se desarrolla un compilador para un subconjunto de C. Le

llamamos **C-0** y es bastante simple. Pero es necesario su estudio para poder abarcar la parte IV, en la que se construye un compilador para C más complejo (**C-1**). Se deja para el lector la construcción de un compilador aún más complejo, que podríamos llamarle **C-2**.

Dentro de la parte III, el capítulo 13 trata la especificación del lenguaje, el 14 el análisis léxico, sintáctico y semántico, el 15 la generación de código intermedio y el 16 la generación de código final.

La parte IV, en la que se desarrolla un compilador para C más complejo, **C-1**, consta del capítulo 17 en el que se trata la especificación del lenguaje, el capítulo 18 para el análisis léxico y sintáctico, el 19 para el análisis semántico y el 20 para la generación de código.

La parte V consta de dos apéndices y la bibliografía. El apéndice A explica las herramientas que se han utilizado en el desarrollo de los compiladores y el apéndice B explica las instrucciones de código intermedio y final en Ens2001 para el lenguaje **C-1**. Esta parte concluye con la bibliografía.

Este libro puede servir de material para un curso de un semestre sobre compiladores o para un curso de dos semestres. Si se utiliza para un semestre, que suele ser lo más normal para una Ingeniería Técnica en Informática, se puede estudiar la parte I (capítulos 1 a 7) y luego implementar el compilador **L-0** de la parte II (capítulos 9, 10, 11 y 12) y el compilador **C-0** de la parte III (sólo capítulos 13 y 14). Por último, el apéndice A.

El resto del libro se podría incluir en un curso de dos semestres, que suele ser lo habitual en un segundo ciclo de Ingeniería Informática.

- Curso de 1 semestre à Ingeniería Técnica en Informática à Capítulos 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14 y apéndice A.
- Curso de 2 semestres à Ingeniería Informática à Capítulos del 1 al 20 y apéndices A y B.

Espero que este libro sea de interés del lector y que pueda sacar provecho a su lectura. Tanto como he sacado yo al escribirlo.

Nota: El código fuente de los compiladores se encuentra en la página web: www.rclibros.es en la sección Zona de archivos, también lo puede solicitar al autor en su dirección de correo electrónico jacinruiz@hotmail.com.

Jacinto Ruiz Catalán

Ingeniero en Informática

Ingeniero Técnico en Informática de Sistemas

Baena, Octubre de 2009

Parte I. Teoría

1. Introducción
2. Análisis léxico
3. Análisis sintáctico
4. Análisis sintáctico descendente
5. Análisis sintáctico ascendente
6. Tabla de tipos y de símbolos
7. Análisis semántico
8. Generación de código intermedio y final

CAPÍTULO 1

INTRODUCCIÓN

1.1 Definición de compilador

Un compilador es un tipo especial de traductor en el que el lenguaje fuente es un lenguaje de alto nivel y el lenguaje objeto es de bajo nivel (figura 1.1).

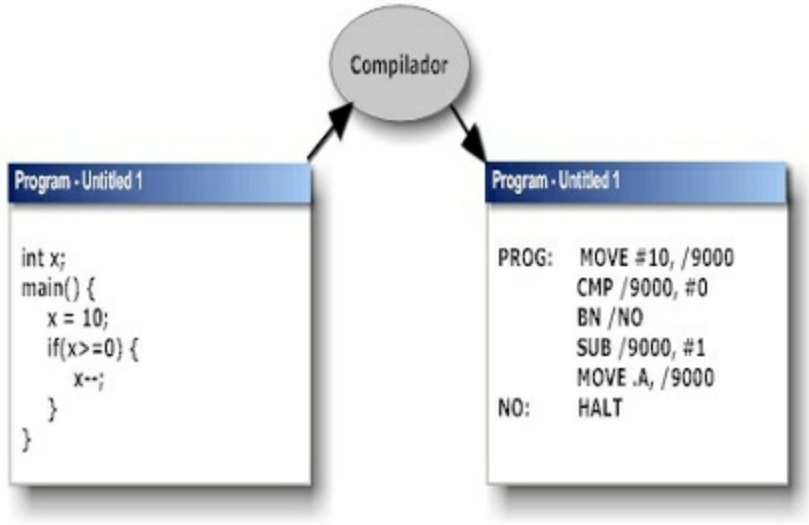


***Figura 1.1**. Esquema de un compilador*

Un traductor es un programa que convierte el texto escrito en un lenguaje en texto escrito en otro lenguaje (figura 1.2).

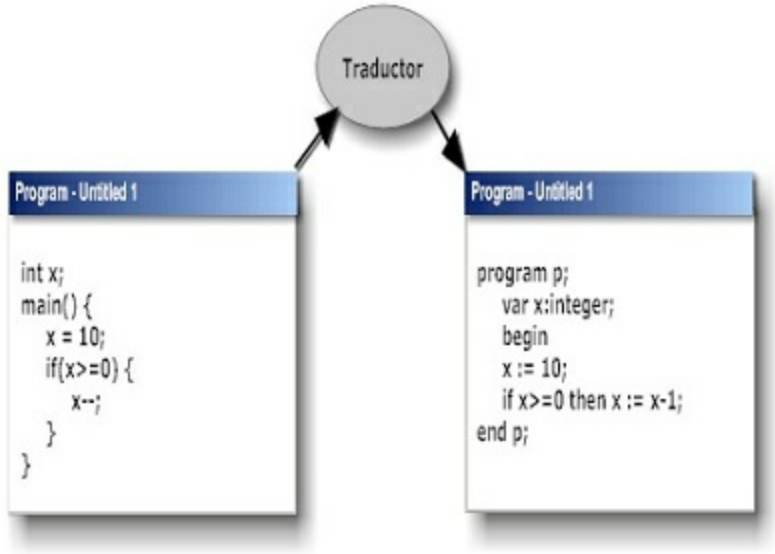


***Figura 1.2**. Esquema de un traductor*

Un ensamblador es un compilador donde el lenguaje fuente es un lenguaje ensamblador y el lenguaje objeto es el código de la máquina.

La diferencia entre compilador e intérprete es que el compilador analiza todo el programa fuente, crea el programa objeto y luego permite su ejecución (sólo del

programa objeto obtenido) y el intérprete lee sentencia por sentencia el programa fuente, la convierte en código objeto y la ejecuta. Por lo tanto, es fácil comprender que tras compilar un programa, su ejecución es mucho más rápida que la ejecución de un programa interpretado.

Uno de los motivos de la existencia de programas interpretados es que hay algunos lenguajes de programación que permiten añadir sentencias durante la ejecución, cosa que no se podría hacer si fueran compilados. Algunos ejemplos son las versiones más antiguas de Basic y actualmente el lenguaje Phyton.

El compilador es asistido por otros programas para realizar su tarea, por ejemplo, se utiliza un preprocesador para añadir ficheros, ejecutar macros, eliminar comentarios, etcétera.

El enlazador se encarga de añadir al programa objeto obtenido, las partes de las librerías necesarias.

El depurador permite al programador ver paso a paso lo que ocurre durante la ejecución del programa.

Hay compiladores que no generan código máquina sino un programa en ensamblador, por lo que habrá que utilizar un programa ensamblador para generar el código máquina.

1.2 Estructura de un compilador

Un compilador es un programa complejo que consta de una serie de pasos, generalmente entrelazados, y que como resultado convierte un programa en un lenguaje de alto nivel en otro de bajo nivel (generalmente código máquina o lenguaje ensamblador).

Los pasos o fases de la compilación están actualmente bien definidos y en cierta medida sistematizados, aunque no están faltos de dificultad. Esta aumenta conforme se incrementa la riqueza del lenguaje a compilar.

Las fases del proceso de compilación son las siguientes (figura 1.3):

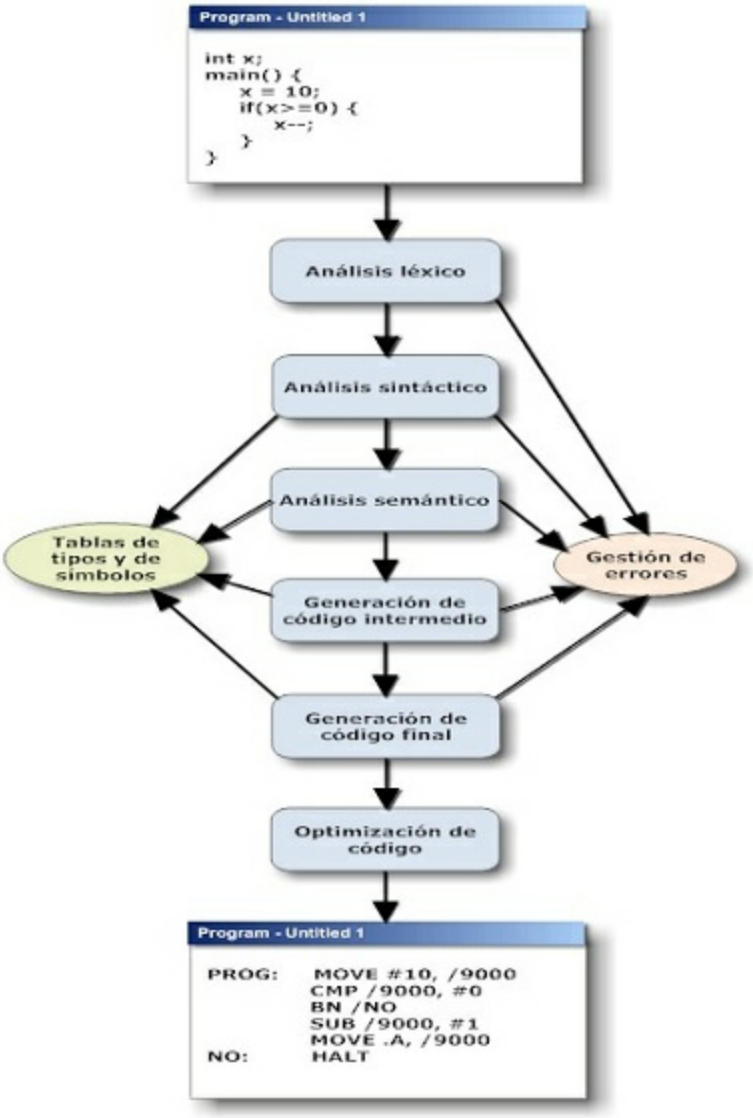


***Figura 1.3**. Fases del proceso de compilación*

1.2.1 Análisis léxico

Esta fase consiste en leer el texto del código fuente carácter a carácter e ir generando los *tokens* (caracteres relacionados entre sí). Estos *tokens* constituyen la entrada para el siguiente proceso de análisis (análisis sintáctico). El agrupamiento de caracteres en *tokens* depende del lenguaje que vayamos a compilar; es decir, un lenguaje generalmente agrupará caracteres en *tokens* diferentes de otro lenguaje.

Los *tokens* pueden ser de dos tipos; cadenas específicas como palabras reservadas, puntos y comas, etc., y no específicas, como identificadores, constantes y etiquetas. La diferencia entre ambos tipos de *tokens* radica en si ya son conocidos previamente o no. Por ejemplo, la palabra reservada *while* es conocida previamente en un lenguaje que la utilice, pero el nombre de una variable no es conocido anteriormente ya que es el programador el que le da nombre en cada programa.

Por lo tanto, y resumiendo, ***el analizador léxico lee los caracteres que componen el***

texto del programa fuente y suministra tokens al analizador sintáctico.

El analizador léxico irá ignorando las partes no esenciales para la siguiente fase, como pueden ser los espacios en blanco, los comentarios, etc., es decir, realiza la función de preprocesador en cierta medida.

Los *tokens* se consideran entidades con dos partes: su tipo y su valor o lexema. En algunos casos, no hay tipo (como en las palabras reservadas). Esto quiere decir que por ejemplo, si tenemos una variable con nombre "x", su tipo es *identificador* y su lexema es *x*.

Por ejemplo, supongamos que tenemos que analizar un trozo de programa fuente escrito en lenguaje Java:

x = x + y – 3;

Los *tokens* suministrados al analizador sintáctico serían estos (el nombre que le demos a los tipos de *tokens* depende de nosotros):

- "x" : Tipo *identificador*, lexema *x*
- "=" : Lexema =
- "x" : Tipo *identificador*, lexema *x*
- "+" : Lexema +
- "y" : Tipo *identificador*, lexema *y*
- "-" : Lexema –
- "3" : Tipo *entero*, lexema *3*
- ";" : Lexema *;*

1.2.2 Análisis sintáctico

El analizador léxico tiene como entrada el código fuente en forma de una sucesión de caracteres. El analizador sintáctico tiene como entrada los lexemas que le suministra el analizador léxico y su función es comprobar que están ordenados de forma correcta (dependiendo del lenguaje que queramos procesar). Los dos analizadores suelen trabajar unidos e incluso el léxico suele ser una subrutina del sintáctico.

Al analizador sintáctico se le suele llamar *párser*. El *párser* genera de manera teórica un árbol sintáctico. Este árbol se puede ver como una estructura jerárquica que para su construcción utiliza reglas recursivas. La estructuración de este árbol hace posible diferenciar entre aplicar unos operadores antes de otros en la evaluación de expresiones. Es decir, si tenemos esta expresión en Java:

$x = x * y - 2;$

El valor de x dependerá de si aplicamos antes el operador producto que el operador suma. Una manera adecuada de saber qué operador aplicamos antes es elegir qué árbol sintáctico generar de los dos posibles.

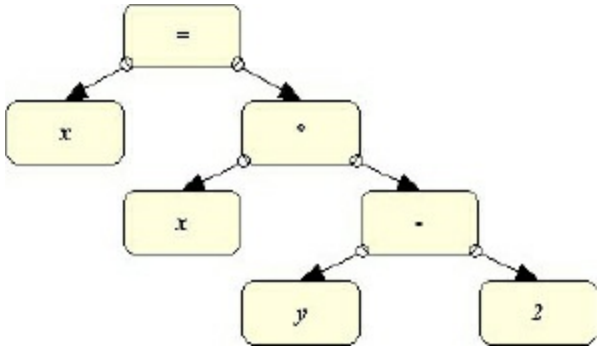


***Figura 1.4**. Arbol sintáctico*

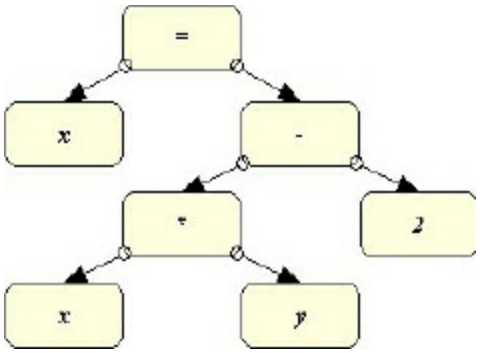


***Figura 1.5**. Arbol sintáctico*

En resumen, *la tarea del analizador sintáctico es procesar los lexemas que le suministra el analizador léxico, comprobar que están bien ordenados, y si no lo están, generar los informes de error correspondientes. Si la ordenación es correcta, se generará un árbol sintáctico teórico.*

1.2.3 Análisis semántico

Esta fase toma el árbol sintáctico teórico de la anterior fase y hace una serie de comprobaciones antes de obtener un árbol semántico teórico.

Esta fase es quizás la más compleja. Hay que revisar que los operadores trabajan sobre tipos compatibles, si los operadores obtienen como resultado elementos con tipos adecuados, si las llamadas a subprogramas tienen los parámetros adecuados tanto en número como en tipo, etc.

Esta fase debe preparar el terreno para atajar las fases de generación de código y debe lanzar los mensajes de error que encuentre. En resumen, su tarea es revisar el significado de lo que se va leyendo para ver si tiene sentido.

Esta fase, las anteriores y las siguientes se detallarán más adelante, en el desarrollo de los otros capítulos.

1.2.4 Generación de código intermedio

El código intermedio (CI) es la representación en un lenguaje sencillo (incluso inventado por el creador del compilador) de la secuencia real de las operaciones que se harán como resultado de las fases anteriores.

Se trata de representar de una manera formalizada las operaciones a llevar a cabo en un lenguaje más cercano a la máquina final, aunque no a una máquina concreta sino a una máquina abstracta. Es decir, no consiste en generar código ensamblador para una máquina basada en un procesador M68000, por ejemplo, sino en generar código que podría luego implementarse en cualquier máquina. Este lenguaje intermedio debe de ser lo más sencillo posible y, a la vez, lo más parecido a la manera de funcionar de la máquina final.

Hay dos ventajas clave por las que se debe utilizar la generación de código intermedio:

- Permite crear compiladores para diferentes máquinas con bastante menos esfuerzo que realizar todo el proceso para cada una de ellas.
- Permite crear compiladores de diferentes lenguajes para una misma máquina sin tener que generar cada vez el código final (puesto que tenemos ya el código intermedio creado).

Se suele utilizar una forma normalizada de instrucciones para este lenguaje intermedio que se llama *código de tres direcciones*.

Este código tiene su origen en la necesidad de evaluar expresiones, aunque se utiliza para todo el proceso (modificando el significado inicial de dirección).

Veamos un ejemplo aclaratorio. Supongamos que tenemos una máquina teórica que consta de registros numerados del R1 en adelante que pueden contener valores de tipo entero. Pensemos que tenemos este código en Java:

x = x +1;

Sea que *x* representa un valor entero y que hay un método que nos convierte el lexema de *x* en su valor entero. Supongamos que 1 es un lexema que representa otro valor entero, en este caso el valor 1 y que tenemos un método para convertir el lexema en su valor entero. Supongamos también que tenemos un método para convertir un número entero en su lexema.

Los pasos para generar el CI de la sentencia anterior serían:

- R1 = valor(x)
- R2 = valor(1)
- R3 = R1 + R2
- x = lexema(R3)

Estas operaciones las podríamos representar con el siguiente código intermedio:

- CARGAR x null R1
- CARGAR 1 null R2
- SUMAR R1 R2 R3
- CARGAR R3 null x

Si revisamos lo que hemos hecho, veremos que hay dos tipos de instrucciones de CI, una para cargar y otra para sumar (en el capítulo en que explicamos más detalladamente la generación de CI veremos que hay bastantes más instrucciones). Pero las dos tienen un punto en común, constan de un identificador de instrucción y de tres parámetros, aunque alguno de ellos puede ser nulo.

Cualquier expresión puede ser representada por una o varias de estas instrucciones de tres direcciones. Las llamamos de tres direcciones porque en realidad se utilizan direcciones de memoria (llamadas temporales) y no registros.

El código de tres direcciones consta de un identificador de código, dos direcciones de operandos y una dirección de resultado de la operación.

Para realizar cualquier operación, es suficiente con generar diferentes instrucciones de código de este tipo, por lo que podemos tener un CI sencillo, versátil y útil.

Veremos un caso real para ver cómo funciona:

Supongamos que tenemos un programa en lenguaje C que consiste sólo en declarar la variable *x*. Cargarla con el valor 0 y luego sumarle 1. Supongamos que nuestro programa se va a alojar a partir de la dirección de memoria 0. Vamos a guardar los contenidos de las variables a partir de la dirección 9000 y vamos a utilizar las direcciones a partir de la 10000 como direcciones temporales.

El programa sería algo así:

```
int x

main() {

x = 0;

x = x + 1;

}
```

Se podría generar este código intermedio:

```
CARGAR 0 null 10000
```

CARGAR 10000 null 9000

CARGAR 9000 null 10001

CARGAR 1 null 10002

SUMAR 10001 10002 10003

CARGAR 10003 null 9000

Podemos ver que vamos utilizando direcciones temporales conforme las vamos necesitando para guardar resultados parciales.

Al final de todo, vemos que hemos recuperado el resultado parcial guardado en la dirección 10003 y lo hemos cargado en la dirección donde guardábamos el valor de la variable *x*. Por lo que al final, el valor de *x* es 1, que es el contenido de la dirección 9000.

La optimización de código intermedio consiste en el proceso de ir reutilizando estas direcciones temporales para que el consumo de memoria no se dispare. Además, consiste en optimizar el código generado para reducir el número de instrucciones necesarias para realizar las mismas operaciones.

1.2.5 Generación de código final

La generación de código final (CF) es un proceso más mecánico, ya que consiste en ir pasando las distintas instrucciones de CI (que suelen ser de pocos tipos diferentes) al lenguaje ensamblador de la máquina que se vaya a utilizar (más adelante, se puede ensamblar el código y obtener un ejecutable, pero este proceso ya es automático).

Dependiendo de la cantidad de memoria disponible o del número de registros disponibles, quizás sea necesario utilizar en vez de direcciones de memoria como temporales, registros, ya que así se reduce la memoria a usar. Esto es recomendable sobre todo para programas grandes porque casi todas las operaciones que se realizarán van a ser de cargas y almacenamientos de valores.

1.2.6 Tablas de símbolos y de tipos

Ponemos aquí esta sección, aunque no es una fase del proceso de compilación, porque es una parte importantísima de todo el proceso.

La tabla de símbolos es una estructura de datos auxiliar donde se va a guardar información sobre las variables declaradas, las funciones y procedimientos, sus nombres, sus parámetros y en generar cuanta información vaya a ser necesaria para realizar todo el proceso de compilación.

La tabla de tipos no es menos importante, ya que va a guardar información tanto sobre los tipos básicos como sobre los tipos definidos en el programa a compilar.

El compilador debe tener acceso a estas tablas a lo largo de todo el proceso de

compilación. Además, el número de accesos suele ser alto, por lo que es conveniente optimizar la estructura de estas tablas para que su manipulación sea lo menos costosa en tiempo y así reducir el tiempo de compilación (aunque no habrá variación en el tiempo de ejecución del programa compilado ya que el proceso de manipulación de las tablas de tipos y símbolos se hace en tiempo de compilación).

1.2.7 Manejo de errores

El manejo de errores es vital para el proceso ya que informa al programador dónde hay errores, de qué tipo son, etc. Cuanta más información sea capaz de dar al programador, mejor.

Por lo tanto, no hay que dejar atrás este aspecto del proceso de compilación.

La detección de errores no siempre es posible en la fase en que se producen, algunas veces se detectarán en fases posteriores y algunos de ellos no se podrán detectar ya que se producirán en el tiempo de ejecución (serán responsabilidad del programador que ha realizado el código fuente).

Generalmente, es en las fases de análisis sintáctico y semántico donde suelen aparecer y *es importante que el proceso no se pare al encontrar un error,* sino que continúe su proceso de análisis hasta el final y luego informe de todos los errores encontrados.

1.3 Fases del proceso de compilación

Aunque las fases del proceso son las descritas en el epígrafe anterior, estas pueden agruparse desde un punto de vista diferente. Se pueden agrupar en una fase inicial o *front-end,* y en una fase final o *back-end*. El *front-end* agrupa las fases dependientes del lenguaje fuente e independientes de la máquina final y el *back-end* las fases independientes del lenguaje fuente pero dependientes del código intermedio y de la máquina de destino.

Cada fase comienza con un código y termina con otro diferente. El *front-end* comienza con el código fuente y finaliza con el código intermedio. Y el *back-end* comienza con el código intermedio y acaba con el código final.

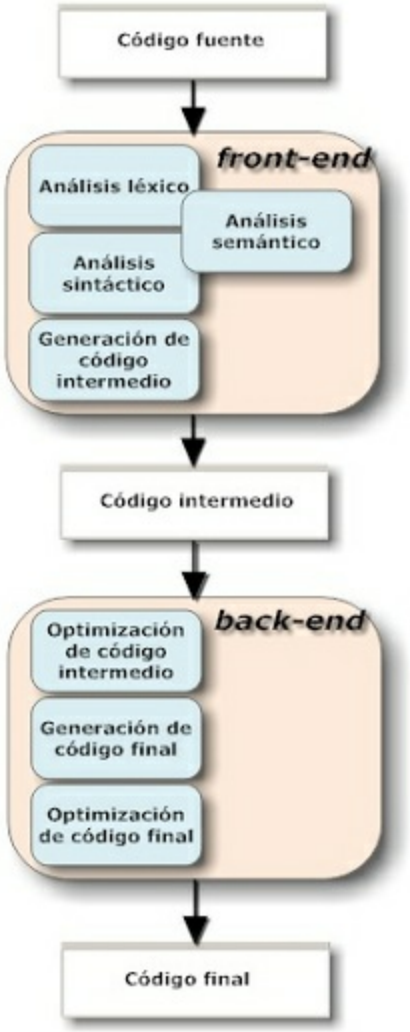


***Figura 1.6**. Front-end y back-end*

1.4 Herramientas y descripción del lenguaje

A la hora de hacer todo el proceso de confección de un compilador se suelen utilizar herramientas, que al no abarcar todas las fases, suelen combinarse. Aparte de las herramientas, se debe utilizar un lenguaje base para programar los pasos a seguir. Este lenguaje base debe ser compatible con las herramientas que utilicemos.

Por otra parte, antes de nada hay que saber para qué lenguaje vamos a hacer el compilador. Debemos definir su especificación léxica (los lexemas que utiliza), su especificación sintáctica (la ordenación válida de esos lexemas) y su especificación semántica (descripción del significado de cada lexema y las reglas que deben cumplirse).

También debemos saber si existe ya un código intermedio que podamos utilizar o hay que crear uno nuevo (este aspecto se suele hacer cuando llegamos a la fase correspondiente) y el código final que utilizaremos (también se suele hacer durante la fase de generación de código intermedio y generación de código final).

Otro aspecto que debemos tener en cuenta es el lenguaje base que vamos a utilizar, que dependerá tanto de nuestras preferencias como de las herramientas de ayuda que utilicemos.

CAPÍTULO 2

ANÁLISIS LÉXICO

2.1 Utilidad del análisis léxico

Las técnicas que vamos a describir en este capítulo se utilizan tanto para la construcción de compiladores como para otras áreas no tan relacionadas con este campo de la informática. Por ejemplo, se utilizan para interpretar el código HTML de una página web, para interpretar un archivo de tipo *.ini*, etcétera.

El analizador léxico va leyendo del fichero de entrada los caracteres secuencialmente y los va agrupando, según las directivas programadas, en *tokens* con un significado conocido por el programador. Los *tokens* serán la entrada del analizador sintáctico.

Podríamos pensar que el papel del analizador léxico podría ser asumido por el analizador sintáctico, pero el diferenciar estas dos tareas hace posible aplicar técnicas específicas para cada una de ellas.

El analizador léxico hace las funciones, a la vez, de preprocesador ya que se encarga de eliminar los caracteres innecesarios para el proceso de compilación. Por ejemplo, elimina los espacios en blanco que hay entre palabra y palabra (siempre que no sean necesarios, como por ejemplo dentro de las cadenas de texto).

Una ventaja importante es que permite obtener estructuras similares aun cuando los caracteres de entrada no lo sean. Por ejemplo, debe ser equivalente "6+2/4" a "6 +2 / 4". En ambos casos, el analizador léxico suministra al sintáctico los *tokens 6*, +, *2*, / y *4*.

Resumiendo, las funciones del analizador léxico son:

- *Agrupar caracteres según categorías establecidas por la especificación del lenguaje fuente.*
- *Rechazar texto con caracteres ilegales o agrupados según un criterio no especificado.*

2.2 Funcionamiento

La relación entre el analizador léxico y el sintáctico es una relación de maestro-esclavo. El sintáctico demanda al léxico que lea el siguiente lexema y el léxico lee los caracteres necesarios del fichero de entrada hasta que consigue completar un lexema, que será entregado a su maestro (el analizador sintáctico). Esta relación viene representada en la figura 2.1.

El funcionamiento del analizador léxico es:

- *Construir tokens* válidos a partir de la lectura carácter a carácter del fichero de entrada.
- *Pasar tokens* válidos al analizador sintáctico.
- *Gestionar el manejo del fichero de entrada.*
- *Ignorar los espacios en blanco, comentarios y demás caracteres o tokens innecesarios para el proceso de análisis.*
- *Avisar de los errores encontrados en esta fase.*
- *Llevar la cuenta del número de línea para incluirlo en el aviso de error (esto es imprescindible para que el usuario pueda encontrar y corregir el error).*
- *Hacer las funciones de preprocesador.*

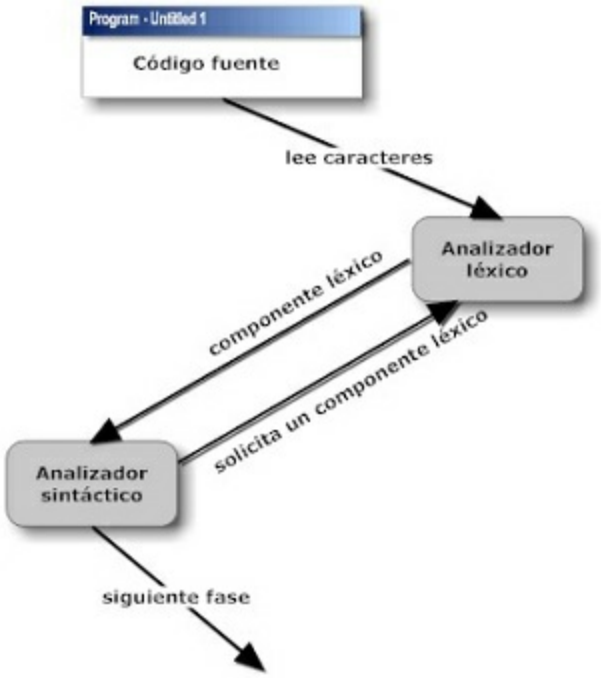


***Figura 2.1**. Conexión analizador léxico-sintáctico*

El analizador léxico va leyendo carácter a carácter el fichero de entrada y va guardando estos caracteres en un buffer. Cuando encuentra un carácter que no le sirve para construir un token válido, se para y envía los caracteres acumulados al analizador léxico y espera una nueva petición de lectura de este. Cuando recibe una nueva petición del analizador sintáctico, limpia el buffer y vuelve a leer el carácter donde paró la vez anterior (ya que ese no pertenecía al token que envió). Por ejemplo, supongamos que tenemos un fichero de entrada con el siguiente contenido:

```
int x;

main() {

}
```

El analizador léxico hará estas operaciones:

Entrada	Buffer	Acción
i	i	Leer otro carácter
n	in	Leer otro carácter
t	int	Leer otro carácter
Espacio en blanco	int	Enviar token y limpiar buffer
x	x	Leer otro carácter
;	x	Enviar token y limpiar buffer
;	;	Leer otro carácter
m	;	Enviar token y limpiar buffer
m	m	Leer otro carácter
a	ma	Leer otro carácter
i	mai	Leer otro carácter
n	main	Leer otro carácter
(	main	Enviar token y limpiar buffer
(	(	Leer otro carácter
)	(	Enviar token y limpiar buffer
)	)	Leer otro carácter
Espacio en blanco	)	Enviar token y limpiar buffer
{	{	Leer otro carácter
}	{	Enviar token y limpiar buffer
}	}	Leer otro carácter
Fin de fichero	}	Enviar token y finalizar proceso de análisis

Un aspecto importante es que el analizador léxico debe leer el token más largo posible. En caso de que no sea posible, retrocederá al menos largo que sea válido y después volverá a leer los que haya desechado para el siguiente token.

2.3 Términos utilizados

Hay unos términos que son utilizados en esta fase. Se trata de token, patrón, lexema y atributo. Los explicaremos a continuación.

1. **Patrón**. Es una representación lógica de una serie de agrupación de caracteres con unas características comunes. Por ejemplo, en Java el identificador de una variable

puede ser cualquier combinación de letras y números (y el símbolo de subrayado) que no comiencen con un número. El patrón sería la definición formal de esto que hemos dicho. Para describir formalmente esta definición, se utilizan las expresiones regulares.

Las expresiones regulares se utilizan también en teoría de autómatas y son una manera de ver las entradas válidas para un autómata.

Por ejemplo, para el ejemplo de los identificadores de variables en Java, serían:

Letra ::= (a-zA-Z)

Digito ::= (0-9)

Subrayado := (_)

Identificador ::= (Subrayado|Letra)(Subrayado|Letra|Dígito)*

1. **Lexema**. Es cada una de las combinaciones de caracteres que encajan en la definición de un patrón. Por ejemplo, las secuencias "variable1", "x", "y12" encajarían en el patrón de identificador de una variable en Java. Es decir, el patrón es la definición formal y el lexema es cada una de las secuencias que pueden encajar en esa definición.
2. **Token**. Es el nombre que se le va a dar a cada patrón definido. Este nombre se utilizará en los procesos de análisis siguientes en representación de todos los lexemas encontrados. Donde encaje en la gramática un token, encajará cada uno de los lexemas que represente. Luego, se podrá ver qué lexemas en concreto son los representados por un token. El token es como, por ejemplo, la palabra fruta y los lexemas son las frutas en concreto, como manzana, plátano, etc. Cuando hablemos de fruta, sabremos que nos estamos refiriendo a cualquier fruta.

1. **Atributo**. Cada tipo de token tiene una serie de características propias a su tipo que serán necesarias más adelante en las siguientes etapas del análisis. Cada una de estas características se denomina atributos del token. Cuando llegue el momento, sabremos cuáles son.

El analizador léxico no sólo pasa el lexema al sintáctico, sino también el token. Es decir, se le pasa una pareja (token, lexema).

2.4 Especificación del analizador léxico

Para poder comprender el funcionamiento de un analizador léxico, lo especificaremos como una máquina de estados (llamada *diagrama de transiciones*, DT). Es parecida a un autómata finito determinista (AFD) pero con las siguientes diferencias:

- El AFD sólo dice si la secuencia de caracteres pertenece al lenguaje o no y el DT debe leer la secuencia hasta completar un token y luego retornar ese token y dejar la entrada preparada para leer el siguiente token.
- En un DT cada secuencia no determinada es un error. En los AFD podía haber estados especiales de error o estados que englobaban secuencias no admitidas en otros estados.
- Los estados de aceptación de los DT deben ser terminales.
- En un DT, cuando se lea un carácter que no pertenezca a ninguna secuencia especificada, se debe ir a un estado especial terminal y volver el cursor de lectura de caracteres al carácter siguiente a la secuencia correcta leída.

2.5 Construcción de un analizador léxico

Hay algunas maneras de construir un analizador léxico, una de ellas es utilizar DT y otra es utilizar programas especiales que construyen analizadores a partir de unas pocas líneas de especificación. Es verdad que la segunda manera es más sencilla y es la que utilizaremos en la segunda parte de este libro, pero nos interesa saber cómo se hace con los DT ya que de esta manera aprenderemos cómo implementar un analizador léxico con la única herramienta de un lenguaje de programación.

2.5.1 Identificar las palabras reservadas

Una parte importante de un analizador léxico es poder diferenciar entre lo que es un identificador y una palabra reservada de un lenguaje, ya que el patrón de los identificadores, en general, engloba las palabras reservadas. Por lo tanto, ¿cómo sabemos si un identificador es una palabra reservada?

Responder a esta pregunta es crucial puesto que hay que pasar al analizador sintáctico el tipo de token a que pertenece cada lexema. Por ejemplo, si un analizador para Java encuentra el lexema *int*, debe pasar al analizador sintáctico que *int* no es un identificador sino una palabra reservada, aunque esté dentro del patrón de identificador.

Para responder a esta pregunta, podemos utilizar al menos dos métodos:

- Hacer una tabla con todas las palabras reservadas y consultarla para cada identificador y ver si está en la tabla o no.
- Implementar cada una de las palabras reservadas en el DT para diferenciarlas del resto de identificadores, lo que complicaría bastante su programación.

Adoptar la primera postura es adecuado cuando hay bastantes palabras

reservadas. Si hay pocas, cuesta poco trabajo crear un DT que implemente esta diferenciación (generalmente consiste en incluir algunas sentencias condicionales en la implementación del analizador léxico).

Si se adopta la solución de incluir una tabla de consulta, cada vez que se lee un lexema del tipo de identificador, se consulta la tabla donde previamente se han incluido todas las palabras reservadas y si está el lexema, se le cataloga de palabra reservada y si no lo está, entonces será un identificador. El acceso a la tabla debe ser lo suficientemente rápido como para que la búsqueda sea rápida, haya más o menos palabras reservadas.

2.5.2 Construir el diagrama de transiciones

La construcción del autómata que reconoce un lenguaje es un paso previo a la implementación del algoritmo de reconocimiento. A partir del autómata es relativamente sencillo llegar a la implementación. Un primer paso es representar el autómata en una tabla de transiciones.

Crearemos un ejemplo y a partir de él, haremos los pasos hasta justo antes de la implementación, que dependerá del lenguaje base que vayamos a utilizar y que básicamente consiste en un lector de caracteres de un fichero de entrada, una serie de sentencias *case* y una salida con un código de token.

Vamos a crear un analizador léxico para un lenguaje que reconoce números enteros sin signo, la suma, incremento y el producto. Por ejemplo, son válidos los siguientes lexemas: “01”, “32”, “+”, “++”, “*”.

Lo primero es definir los patrones o expresiones regulares. En nuestro ejemplo:

Entero ::= (“0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”)+

Suma ::= “+”

Producto ::= “*”

Incremento ::= “++”

El símbolo + significa que debe haber al menos uno de los números o más. Si se hubiera utilizado el símbolo *, significaría 0 o más. El símbolo | significa O lógico. Los caracteres van encerrados entre comillas dobles.

Una vez definidos los patrones, crearemos el autómata que los reconoce.

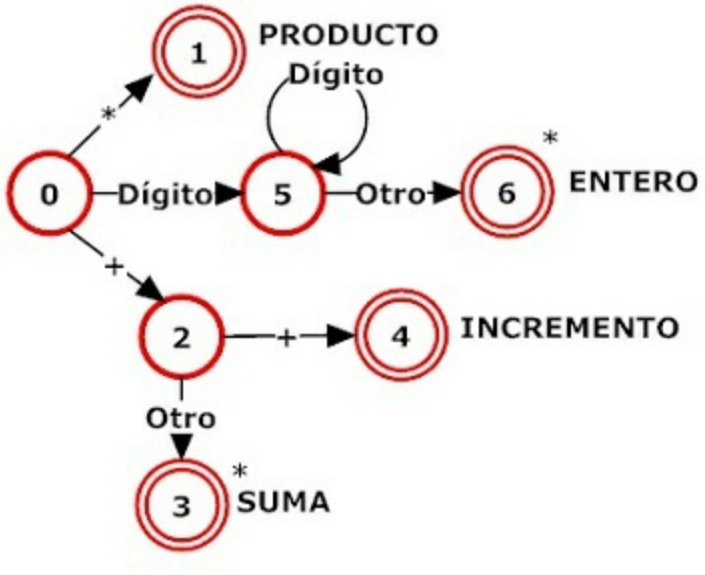


***Figura 2.2**. Autómata del ejemplo*

Los estados de aceptación están marcados con un círculo doble. En mayúsculas el nombre que le damos al token. Un asterisco en un estado de aceptación indica que el puntero que señala la lectura del siguiente carácter (para reconocer el siguiente token) debe retroceder una unidad (si hubiera más asteriscos, retrocedería tantas unidades como asteriscos). Tras la llegada a un estado de aceptación, se le pasaría el token al analizador sintáctico y se esperaría una nueva petición de este para comenzar otra vez en el estado 0 del autómata.

Una vez que tenemos el autómata y comprobamos que funciona correctamente, podemos ya crear la tabla de transiciones, que es una manera más cercana a la implementación de representar el autómata.

La tabla de transiciones consta de tantas filas como estados del autómata. En cuanto al número de columnas, tiene una para numerar el estado, tantas como distintas posibles entradas (se pueden unir si tienen origen y fin en el mismo estado todas ellas), otra para señalar el token que se reconoce y una última para numerar los retrocesos que hay que hacer al reconocer el lexema.

Para el ejemplo de la figura 2, tendríamos una tabla así:

	Entradas	Entradas	Entradas	Entradas		
Estado	**Digito**	**+**	*****	**Otro**	**Token**	**Retroceso**
0	5	2	1	Error	-	-
1	-	-	-	-	PRODUCTO	0
2	3	4	3	3	-	-
3	-	-	-	-	SUMA	1
4	-	-	-	-	INCREMENTO	0
5	5	6	6	6	6	6
6	-	-	-	-	ENTERO	1

Una vez que tenemos la tabla, la implementación obtiene cada estado buscando el estado que hay en la fila correspondiente al estado actual y la entrada actual. Este proceso continúa hasta llegar a un estado de aceptación o uno de error. Si es de

aceptación, devolverá el token junto con los caracteres acumulados hasta el momento. Si hay un retroceso en la fila, se retrocederá el cursor de selección de entrada tantas unidades como se indique en el retroceso. Se borra el buffer y se comienza en el estado 0. Si se ha llegado a un estado de error, se lanzará el error correspondiente.

Supongamos que tenemos esta entrada:

25+5*13+33++5**/5

El autómata efectuaría estos pasos:

Estado=0, Entrada=2, Estado=5, Entrada=5, Estado=5, Entrada=+, Estado=6, Token=ENTERO , Lexema=25, Retroceso=1, Estado=0, Entrada=+, Estado=2, Entrada=5, Estado=3, Token=SUMA, Lexema=+, Retroceso=1, Estado=0, Entrada=5, Estado=5, Entrada=*, Estado=6, Token=Entero, Lexema=5, Retroceso=1, Estado=0, Entrada=*, Estado=1, Token=PRODUCTO, Lexema=*, Retroceso=0, Estado=0, Entrada=1, Estado=5, Entrada=3, Estado=5, Entrada=+, Estado=6, Token=ENTERO, Lexema=13, Retroceso=1, Estado=0, Entrada=+, Estado=2, Entrada=3, Estado=3, Token=SUMA, Lexema=+, Retroceso=1, Estado=0, Entrada=3, Estado=5, Entrada=3, Estado=5, Entrada=+, Estado=6, Token=ENTERO, Lexema=33, Retroceso=1, Estado=0, Entrada=+, Estado=2, Entrada=+, Estado=4, Token=INCREMENTO, Lexema=++, Retroceso=0, Estado=0, Entrada=5, Estado=5, Entrada=*, Estado=6, Token=ENTERO, Lexema=5, Retroceso=1, Estado=0, Entrada=*, Estado=1, Token=PRODUCTO, Lexema=*, Retroceso=0, Estado=0, Entrada=*, Estado=1, Token=PRODUCTO, Lexema=*, Retroceso=0, Estado=0, Entrada=/, Estado=Error

Hay dos casos en que hay que decidir entre más de un token, uno es en el caso de que haya dos lexemas, uno más largo que otro, y que comiencen igual, entonces se debe decidir por el token correspondiente al lexema más largo. El otro caso es en el que el mismo lexema puede asociar a más de un token, entonces se elegirá el primero que se encuentre en la especificación.

2.6 Ejercicios resueltos

Ejercicio 2.1

Diseñar un DT para reconocer los siguientes componentes léxicos:

LETRAS: cualquier secuencia de una o más letras.

ENTERO: cualquier secuencia de uno o más números (si tiene más de un número, no deberá comenzar por 0).

ASIGNAR: la secuencia **=.**

SUMAR: la secuencia **+.**

RESTAR: la secuencia **–.**

IMPRIMIR: la palabra reservada **print.**

La solución sería:

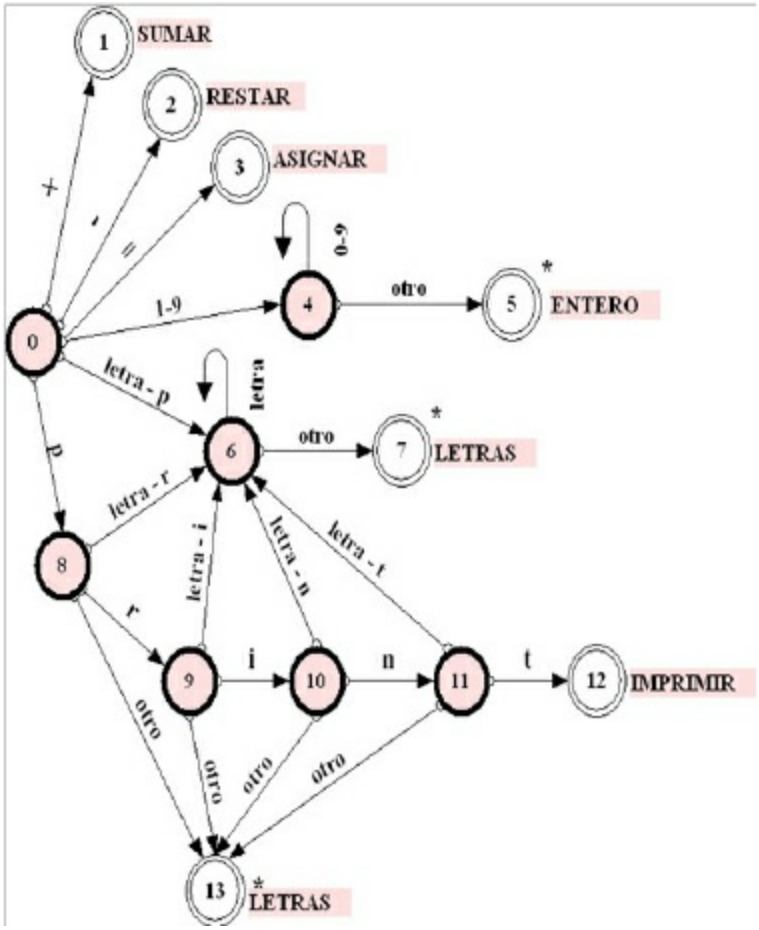


***Figura 2.3**. Autómata del ejercicio 2.1*

En realidad podríamos haber prescindido del estado 13 y remitir las flechas que llegan a él al estado 7, pero por claridad en el gráfico hemos creado ese estado.

Ahora, confeccionaremos la tabla de transiciones:

	-	**E**	**n**	**t**	**r**	**a**	**d**	**a**	**s**	**-**			
Estado	**+**	**-**	**=**	**0**	**-9**	**p**	**r**	**i**	**n**	**t**	**Otras**	**->letras**	**Token**	**Retroceso**
0	1	2	3	Error	4	8	6	6	6	6	6		-	-
1	-	-	-	-	-	-	-	-	-	-	-		SUMAR	0
2	-	-	-	-	-	-	-	-	-	-	-		RESTAR	0
3	-	-	-	-	-	-	-	-	-	-	-		ASIGNAR	0
4	5	5	5	4	4	5	5	5	5	5	5		-	-
5	-	-	-	-	-	-	-	-	-	-	-		ENTERO	1
6	7	7	7	7	7	6	6	6	6	6	6		-	-
7	-	-	-	-	-	-	-	-	-	-	-		LETRAS	1
8	13	13	13	13	13	6	9	6	6	6	6		-	-
9	13	13	13	13	13	6	6	10	6	6	6		-	-
10	13	13	13	13	13	6	6	6	11	6	6		-	-
11	13	13	13	13	13	6	6	6	6	12	6		-	-
12	-	-	-	-	-	-	-	-	-	-	-		IMPRIMIR	0
13	-	-	-	-	-	-	-	-	-	-	-		LETRAS	1

Ejercicio 2.2

Para el lenguaje generado por la expresión regular **(a|b)*abb**, crear el DT, generar la tabla de transiciones y un programa en Java para implementar la tabla.

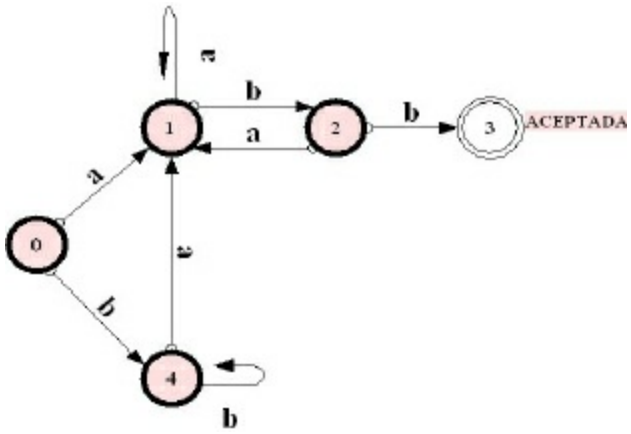


***Figura 2.4**. Autómata del ejercicio 2.2*

La tabla de transiciones es:

Estado	Entrada a	Entrada b	Token	Retroceso
0	1	4	-	-
1	1	2	-	-
2	1	3	-	-
3	-	-	ACEPTADA	0
4	1	4	-	-

El programa en Java sería este:

```
class Lexer {
static final int ERROR = -1;
static final int ACEPTAR = 0;
char obtenerCaracter() {
//Aqui se debe leer un caracter del fichero de
//entrada o del teclado
}
int estado_0() {
char c = obtenerCaracter();
switch(c) {
case 'a': return estado_1() ;
case 'b': return estado_4() ;
default: return ERROR;
}
}
int estado_1() {
char c = obtenerCaracter();
```

```
switch(c) {
case ‘a’: return estado_1();
case ‘b’: return estado_2();
default: return ERROR;
}
}
int estado_2() {
char c = obtenerCaracter();
switch(c) {
case ‘a’: return estado_1();
case ‘b’: return estado_3();
default: return ERROR;
}
}
int estado_3() {
return ACEPTAR;
}
int estado_4() {
char c = obtenerCaracter();
switch(c) {
case ‘a’: return estado_1();
case ‘b’: return estado_4();
default: return ERROR;
}
}
public static void main(String[] args) {
Lexer lexer = new Lexer();
System.out.println(“Resultado:”+lexer.estado_0());
}
}
```

CAPÍTULO 3

ANÁLISIS SINTÁCTICO

3.1 Funciones del analizador sintáctico

La función principal del analizador sintáctico (AS o *párser*) es comprobar que los tokens que le suministra el analizador léxico (AL o *léxer*) van ordenados según la especificación de la gramática del lenguaje a compilar. Y si no es así, dar los mensajes de error adecuados, pero continuar funcionando sin detenerse, hasta que se llegue al final del fichero de entrada. Es esencial que el proceso de análisis no se detenga al primer error encontrado, ya que así podrá informar al usuario en un solo informe de todos los errores generados.

Aparte de esta función principal, el *párser* es la unidad que guía todo el proceso, o casi todo, de la compilación. Esto es así porque por un lado va solicitando al *léxer* los tokens y al mismo tiempo va dirigiendo el proceso de análisis semántico y generación de código intermedio. Por lo que muchas veces se le llama al proceso de análisis semántico y generación de código intermedio, traducción dirigida por la sintaxis.

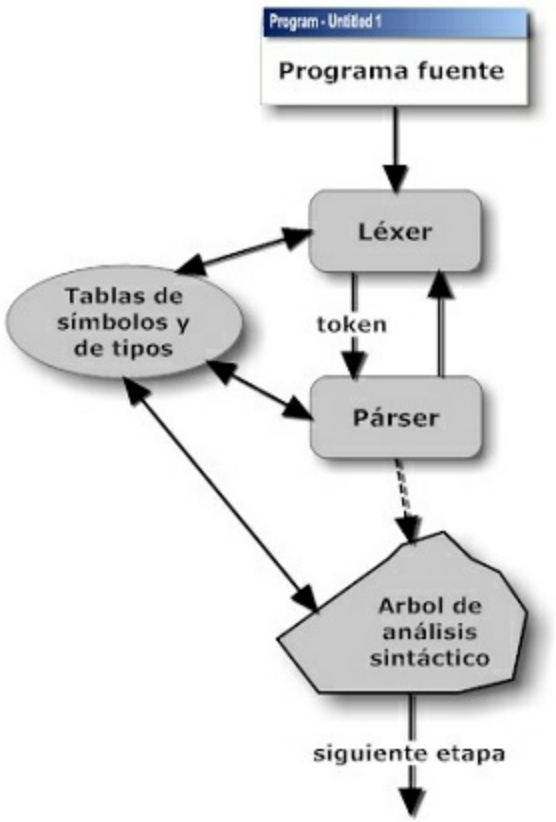


***Figura 3.1**. Funcionamiento del párser*

Generalmente, los analizadores sintácticos obtienen un árbol teórico que permite expresar el orden de los lexemas según van apareciendo. Ese árbol debe ser el modelo de donde salga el análisis semántico. Pero lo normal es que si utilizamos el método de la

traducción dirigida por la sintaxis no lleguemos ni siquiera a plantearnos la generación del árbol ya que el párser realizará las acciones semánticas e incorporará los métodos para realizar la generación de código intermedio y avisará de errores y su recuperación. Es decir, el analizador léxico hará las funciones de las dos siguientes etapas (analizador semántico y generación de código intermedio).

Pero si damos la oportunidad a la creación del árbol de análisis sintáctico, recorriéndolo es posible crear una representación intermedia del programa fuente, ya sea en forma de árbol sintáctico abstracto o en forma de programa en un lenguaje intermedio.

Para generar párser, podemos utilizar dos técnicas, o bien lo hacemos a mano (es difícil aunque eficiente en su funcionamiento) o mediante herramientas que lo generan automáticamente (es menos eficiente pero más fácil de implementar).

Para que un AS funcione, debemos especificar el lenguaje que debe poder leer. Para especificar este lenguaje, debemos representarlo con unas reglas únicas y bien formadas de manera que el párser funcione de una manera bien definida. Es decir, el lenguaje debe ser formal (tener unas reglas bien definidas). A estas reglas se le llama gramática. Por lo tanto, el primer paso para poder implementar un AS es definir la gramática que debe ser capaz de analizar.

3.2 Diseño de gramáticas

Una gramática independiente del contexto se define por tuplas de 4 elementos, terminales, no terminales, reglas de producción y axioma inicial. En adelante, representaremos los no terminales por palabras que comiencen con mayúscula y los terminales con negrita. También representaremos en las reglas de producción la flecha por ::= (cuando implementemos un compilador con las herramientas nos daremos cuenta de que suelen utilizar este formato de representación).

Pongamos un ejemplo de gramática e identificaremos cada uno de los elementos.

Sea la gramática:

E ::= *E* **+** *T* | *T*

T ::= *T* * *F* | *F*

F ::= ***id*** | *F* | ***(*** *E* ***)***

Esta gramática reconoce expresiones aritméticas con los operadores de suma y producto. Las palabras E, T y F son los no terminales. La palabra ***id*** es un terminal. Los operadores son también no terminales (ya que ninguno de ellos está en la parte izquierda de ninguna regla). Es decir, **+**, *, **(** y **)** son terminales. Vemos también que hay tres reglas y el axioma inicial es el antecedente de la primera regla de producción.

Una regla de producción puede considerarse como equivalente a una regla de reescritura, donde el no terminal de la izquierda de la regla es sustituido por la

pseudocadena de la parte derecha de la regla. Una pseudocadena es cualquier secuencia de terminales y/o no terminales.

Dependiendo de por dónde comencemos a reescribir en la pseudocadena, tendremos una derivación por la izquierda (si comenzamos por la izquierda) o por la derecha (si comenzamos por la derecha).

Si queremos construir una cadena de tokens que sean generados por una gramática concreta, podremos hacerlo aplicando las reglas de la gramática según vayan concordando con los tokens.

Por ejemplo, supongamos que tenemos la siguiente gramática:

E ::= E **+** *E | E * E |* ***num*** *|* ***id*** *|* ***(*** *E* ***)***

Intentaremos hacer todas las posibles derivaciones partiendo de la ristra de tokens:

*id1 + id2 * id3*

Derivando por la izquierda:

*E ::= E * E ::= E + E * E ::= id1 + E * E ::= id1 + id2 * E ::= id1 + id2 * id3*

Derivando por la derecha:

*E ::= E * E ::= E * id3 ::= E + E * id3 ::= E + id2 * id3 ::= id1 + id2 * id3*

Vemos que en ambos casos, aunque se reconoce la ristra de tokens, el orden a la hora de aplicar las reglas depende de la derivación que hagamos. A partir de estas derivaciones se pueden construir sus árboles sintácticos. Pero hay casos en que cada posible derivación dará lugar a un árbol sintáctico diferente. Esto significa que la gramática es ambigua.

Para construir el árbol sintáctico del reconocimiento de una ristra de tokens por una gramática se comienza poniendo en el nodo raíz el axioma inicial. Cada nodo interno es un no terminal de la gramática y los nodos externos u hojas son los terminales.

Veamos un ejemplo. Supongamos que tenemos la gramática anterior y queremos procesar esta ristra de tokens:

id1 + id2 + id3

Hay dos posibles árboles sintácticos:

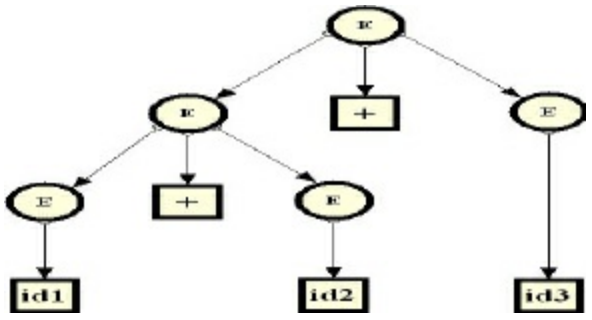


Figura 3.2*. Derivación por la izquierda*

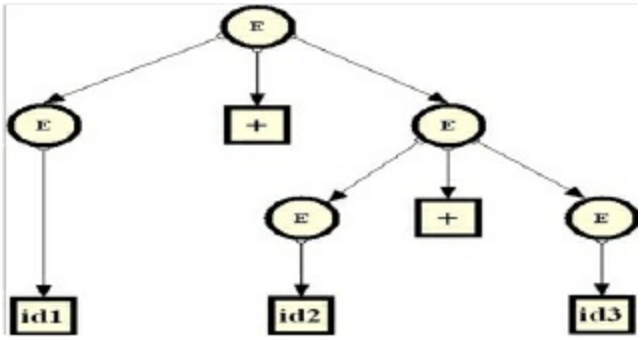


***Figura 3.3**. Derivación por la derecha*

Por lo tanto, deduciremos que la gramática es ambigua. Para la implementación de esta gramática, es necesario evitar esta ambigüedad.

El orden en que vamos creando las ramas y las hojas nos da la idea del orden en que se irán procesando las reglas. Por lo tanto, tenemos un mecanismo secuencial de procesamiento. Sólo las ristras de tokens correctas harán posible crear un árbol sintáctico, por lo que este mecanismo lo podemos utilizar para saber si una ristra de tokens es válida para una gramática dada (sólo con saber si es posible obtener su árbol sintáctico).

Nos hemos encontrado con que la ambigüedad puede ser un problema a la hora de implementar un párser, pero hay más. Intentaremos modificar las gramáticas para evitar estos problemas.

3.3 Dificultades para la creación de gramáticas

3.3.1 La recursividad

Si pensamos en que un lenguaje de programación permite crear infinito número de programas, podremos pensar que harían falta infinitas reglas en la especificación sintáctica. Pero hay un concepto que permite reconocer infinitos lenguajes con un número finito de reglas. Este concepto es la recursividad.

La recursividad se expresa por medio de una o más reglas no recursivas, que son la base, y una o más reglas que son recursivas y que permiten hacer crecer la estructura del lenguaje aplicándose a sí mismas una y otra vez.

Con un ejemplo, lo entenderemos mejor:

Supongamos que queremos expresar la estructura de un número entero compuesto por su signo seguido por un número indeterminado de números entre el 0 y el 9. Lo podríamos expresar con estas reglas:

Entero ::= Signo

Entero ::= Entero Digito

Donde Dígito representa cualquiera de los números del 0 al 9. Mediante esas dos reglas podemos representar la estructura de cualquier número entero sea de la longitud

que sea.

Una gramática se llama recursiva si es de la forma:

$A ::= a\ A\ b$

Donde A es un no terminal y a y b son terminales o no terminales. Al terminal A le llamamos terminal recursivo.

Si no existe el término a, se trata de una recursividad por la izquierda y si no existe b es una recursividad por la derecha.

3.3.2 La ambigüedad

Cuando una gramática contiene una cadena para la que hay más de un árbol de análisis sintáctico se dice que es ambigua. Debido a que una gramática de estas características permite que a partir del mismo código fuente se puedan obtener diferentes códigos intermedios, no es válida para construir un compilador (habría que ayudar con otras técnicas más complicadas).

Pero ¿cómo sabemos si una gramática, que suele poder generar lenguajes con infinitas cadenas posibles es ambigua o no? Tendríamos que probar todas las cadenas posibles y ver si admiten más de un árbol de análisis sintáctico. Como esto, evidentemente no es posible, podemos servirnos de unas técnicas que nos aseguran que si una gramática cumple ciertas reglas, sabemos con seguridad que es ambigua, y por tanto no podemos construir un compilador con ella. Pero en algunos casos, podemos hacer una serie de modificaciones en la gramática que la convierta en no ambigua.

Si una gramática tiene alguna de estas características, podremos afirmar que es ambigua:

- Gramáticas con ciclos:

  $S ::= A$
  $S ::= \boldsymbol{a}$
  $A :: = S$

- Gramáticas con alguna regla de la forma:

  $E ::= E \ldots\ldots E$

- Gramáticas con unas reglas que ofrezcan caminos alternativos entre dos puntos. Por ejemplo:

  $S ::= B$
  $S ::= C$

$B ::= C$

- Producciones recursivas en las que las variables no recursivas de la producción puedan derivar a la cadena vacía. Por ejemplo:

$S ::= A\ B\ S$

$S ::= \boldsymbol{s}$

$A ::= \boldsymbol{a} \mid \boldsymbol{€}$

$B ::= \boldsymbol{b} \mid \boldsymbol{€}$

- Símbolos no terminales que puedan derivar a la cadena vacía y a la misma cadena de terminales, y que aparezcan juntas en la parte derecha de una regla o en alguna forma sentencial. Por ejemplo:

$A ::= A\ B$

$A ::= \boldsymbol{a} \mid \boldsymbol{€}$

$B ::= \boldsymbol{b} \mid \boldsymbol{a} \mid \boldsymbol{€}$

3.3.3 La asociatividad

La asociatividad es un concepto que aparece cuando se operan tres o más operandos. La asociatividad de un operador es por la izquierda si cuando aparecen tres o más operandos se evalúan de izquierda a derecha. Si es de derecha a izquierda, la asociatividad es por la derecha.

Por ejemplo, si tenemos "6/3/2", si el operador "/" tiene asociatividad por la izquierda, primero se opera "6/3" y el resultado se opera con "2". En este caso, sería 1. Si fuera asociativo por la izquierda, sería primero "3/2" y luego "6" operaría con el resultado. En este caso, sería 4.

La manera de reflejar la asociatividad de un operador en una gramática es poniendo recursividad del mismo lado que el operador en la regla sintáctica donde interviene dicho operador.

3.3.4 La precedencia

La precedencia de un operador indica el orden en que se aplicará respecto a los demás operadores en caso de poder aplicar más de uno. Es decir, si en una regla podemos aplicar más de un operador, comenzaremos aplicando el de más precedencia y terminaremos por aplicar el de menor precedencia.

La manera de reflejar la precedencia en una gramática es utilizar para cada operador una variable en la gramática y situarla más cerca del símbolo inicial cuanto menor sea la precedencia.

3.3.5 La parentización

Para incluir paréntesis a la hora de evaluar expresiones en una gramática, se añade una variable que produzca expresiones entre paréntesis. Los operandos se ponen a la mayor distancia posible del símbolo inicial (porque tiene la mayor precedencia).

3.4 Análisis sintáctico lineal

Hay varios algoritmos de análisis sintáctico (incluso para gramáticas ambiguas), pero su coste computacional es elevado (del orden de n3). Por lo que debemos modificar un poco nuestra gramática (si es necesario) para que podamos utilizar un algoritmo de menor coste computacional (de coste lineal). Si conseguimos eliminar la ambigüedad, podremos utilizar dos estrategias:

- **Análisis descendente**: partimos de la raíz del árbol de análisis sintáctico y vamos aplicando reglas por la izquierda para obtener una derivación por la izquierda del símbolo inicial. Para saber la regla a aplicar, vamos leyendo tokens de la entrada. De esta manera, construimos el árbol de análisis sintáctico. Recorriendo el árbol en profundidad, de izquierda a derecha, tendremos en las hojas los tokens ordenados.
- **Análisis ascendente**: partimos de la cadena de entrada y vamos construyendo el árbol a partir de las hojas para llegar a la raíz. Si recorremos el árbol generado como en el caso anterior, encontraríamos los tokens ordenados.

Para los dos métodos es suficiente con recorrer la cadena de entrada una sola vez.

Por lo tanto, *con una sola pasada, existen dos métodos para crear un árbol de análisis sintáctico, siempre que la gramática no sea ambigua. Pero no todas las gramáticas se dejan analizar por estos dos métodos.*

Las gramáticas del tipo LL(k) se pueden analizar en tiempo lineal por el método de análisis descendente y las de tipo LR(k) lo hacen con el análisis ascendente.

La primera L significa que la secuencia de tokens se analiza de izquierda a derecha. La segunda letra (L o R) significa que la derivación es por la izquierda o por la derecha. El número k significa el número de símbolos de entrada que es necesario conocer en cada momento para poder realizar el análisis.

Como el estudio de estas dos técnicas de análisis es bastante amplio, lo haremos en capítulos aparte.

3.5 Diagramas de sintaxis

Los diagramas de sintaxis son grafos dirigidos donde los elementos no terminales de la gramática aparecen como rectángulos y los terminales como círculos o elipses.

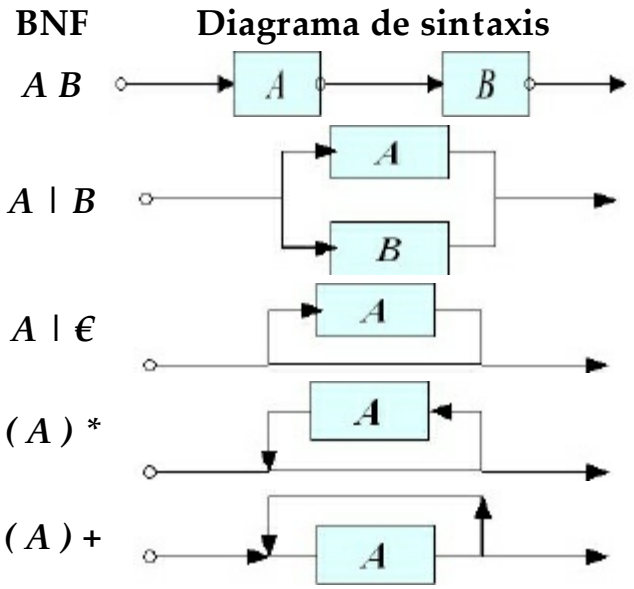
Todo diagrama de sintaxis se supone que tiene un origen y un destino, aunque no se dibujan (se supone que el origen está a la izquierda y el destino a la derecha).

Sean ***a*** y ***b*** dos terminales o no terminales. Un arco entre ***a*** y ***b*** significa que a ***a*** le puede seguir ***b***. Por lo que una manera de representar todas las sentencias válidas es ver los diferentes caminos que hay entre ***a*** y ***b***.

Para ver si una sentencia es válida sintácticamente, no tenemos más que comprobar si puede ser representada por un camino válido.

Cualquier diagrama en notación BNF tiene su correspondiente diagrama de sintaxis y encontrarlo es sencillo; al contrario, no lo es tanto.

Veamos la correspondencia entre la notación BNF y los diagramas de sintaxis.



Una vez que tenemos los diagramas de sintaxis de nuestra gramática, podemos hacer un programa para el análisis sintáctico, de manera que iremos pidiendo al léxer los tokens e iremos viendo si se puede recorrer la secuencia de diagramas correctamente, y si no es así, generar el error correspondiente.

Veamos algunos ejemplos.

Supongamos que queremos analizar si una determinada ristra de tokens es correcta sintácticamente. Supongamos que nuestra gramática especifica que una secuencia sea uno o más grupos de sentencias seguido de punto y coma, es decir:

Secuencia ::= (Sentencia ";")+

Su diagrama de sintaxis sería:

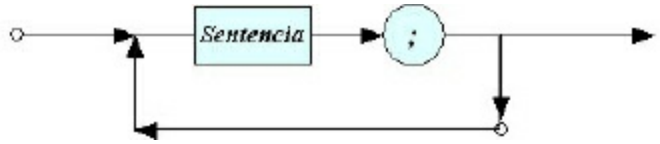


Figura 3.4*. Diagrama de sintaxis de Secuencia*

El programa que podría realizar el análisis sintáctico sería:

```
void secuencia() {
```

```
do {
sentencia();
while(token ¡= PUNTOCOMA) {
errorSintactico();
token = getToken();
}
token = getToken();
} while(token ¡= EOF);
}
```

En el programa anterior hemos supuesto que hay una variable global de la clase *Token* que guarda la información del token actual. Hemos supuesto que hay un método para leer el siguiente token (para solicitar al léxer que lea el siguiente token). Hemos supuesto que hay un método para registrar un error sintáctico y hay dos terminales, *PUNTOCOMA* y *EOF*.

El método *sentencia* lo crearíamos igual que este método, pero para representar la composición de las sentencias.

Para continuar con el ejemplo, supongamos que disponemos de un único tipo de sentencia, que sea la asignación y que se representara así:

Sentencia ::= Identificador "=" Numero

Su diagrama de sintaxis sería:

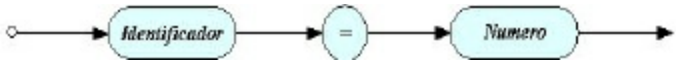


***Figura 3.5**. Diagrama de sintaxis de Sentencia*

Donde todos los elementos son terminales.

Y el programa para hacer el análisis sintáctico sería:

```
void sentencia() {
if(token == Identificador) {
token = getToken();
} else if(token == IGUAL) {
token = getToken();
} else if(token == Numero) {
token = getToken();
```

```
} else errorSintactico();
}
```

3.6 Ejercicios resueltos

Ejercicio 3.1

Diseñar una gramática no ambigua para el lenguaje de las expresiones que se pueden construir con ***true*** y ***false*** y los operadores booleanos ***or, and, not*** y los paréntesis. La precedencia de mayor a menor es ***not and or***. Los dos últimos son asociativos por la derecha.

La solución sería:

La idea inicial sería algo así:

E ::= *E* ***and*** *E*

E ::= *E* ***or*** *E*

E ::= ***not*** *E*

E ::= ***true***

E ::= ***false***

E ::= ***(*** *E* ***)***

Pero claramente es ambigua y además no tiene en cuenta la precedencia ni la asociatividad. Aunque sí está bien situada la regla de los paréntesis al final de todas.

Para mantener la precedencia, ponemos primero las reglas con menor precedencia y al final las de mayor precedencia.

E ::= *E* ***or*** *E*

E ::= *E* ***and*** *E*

E ::= ***not*** *E*

E ::= ***true***

E ::= ***false***

E ::= ***(*** *E* ***)***

Ahora, veremos cómo implementar la asociatividad. La asociatividad debe de ser del mismo sentido que la recursividad en la regla donde está el operador. Por lo tanto:

E ::= *T* ***or*** *E*

E ::= *T*

T ::= F ***and*** *T*

T ::= F

F ::= ***not*** *F*

F ::= ***true***

F ::= ***false***

F ::= ***(E)***

Ejercicio 3.2

Construir una gramática no ambigua que reconozca todas las declaraciones posibles de variables de los siguientes tipos: *int*, *String*, *boolean* y *double* en Java. Por ejemplo:

```
int x,y;
String cadena;
double a;
boolean b;
```

En principio, podría ser algo así;

S :: = S E

S ::= E

E ::= T F ***ptocoma***

T ::= ***int*** | ***String*** | ***double*** | ***boolean***

F ::= F ***id coma*** | ***id***

En principio, parece que no es una gramática ambigua.

Ejercicio 3.3

Crear los diagramas de sintaxis y el programa para esta gramática:

Programa ::= Declaraciones Sentencias

Declaraciones ::= (Decl ";") +

Decl ::= ***Entero Identificado****r*

*Sentencias ::= (Asignacion "****;****") +*

*Asignación ::= ParteIzquierda "****=****" ParteDerecha*

ParteIzquierda ::= ***Identificador***

ParteDerecha ::= Expresion

*Expresion ::= (Expresión "**+**" Expresión)* | *(Expresión "**-**" Expresión)*

*Expresión ::= (**Identificador** | **Numero**)*

Lo primero que podemos hacer es simplificar un poco la gramática. Por ejemplo, los no terminales *ParteIzquierda* y *ParteDerecha* son prescindibles. Nos quedaría:

Programa ::= Declaraciones Sentencias

Declaraciones ::= (Decl ";") +

*Decl ::= **Entero Identificador***

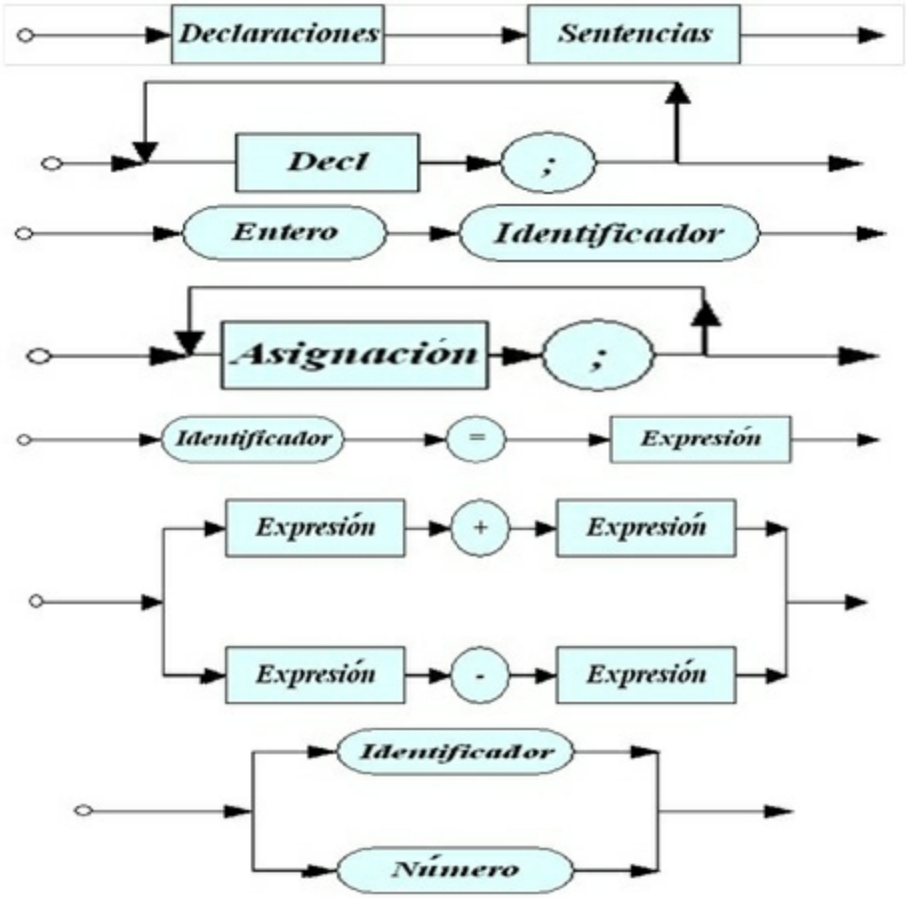
*Sentencias ::= (Asignacion "**;**") +*

*Asignación ::= **Identificador** "**=**" Expresion*

*Expresion ::= (Expresión "**+**" Expresión)* | *(Expresión "**-**" Expresión)*

*Expresión ::= (**Identificador** | **Numero**)*

Los diagramas de sintaxis serían:



Ahora vamos a programar estos diagramas.

```
void programa() {
declaraciones();
sentencias();
}
void declaraciones() {
do {
decl();
while(token ¡= PUNTOCOMA) {
errorSintactico();
token = getToken();
}
token = getToken();
} while(token == Entero);
}
void decl() {
if(token == Entero) {
token = getToken();
if(token == Identificador) {
token = getToken();
} else errorSintactico();
} else errorSintactico();
```

```
}
void sentencias() {
do {
asignación();
while(token ¡= PUNTOCOMA) {
errorSintactico();
token = getToken();
}
token = getToken();
} while(token == Identificador);
}
void asignacion() {
if(token == Identificador) {
token = getToken();
if(token == IGUAL) {
token = getToken();
expression();
} else errorSintactico();
} else errorSintactico();
}
void expresion() {
If(token == Identificador || token == Numero) {
token = getToken();
if(token == MAS || Token == MENOS) {
token = getToken();
expresion();
}
} else errorSintactico;
}
```

CAPÍTULO 4

ANÁLISIS SINTÁCTICO DESCENDENTE

4.1 Introducción

El análisis sintáctico descendente (ASD) consiste en ir haciendo derivaciones a la izquierda, partiendo del axioma inicial, hasta obtener la secuencia de derivaciones que reconoce a la sentencia. En realidad, se trata de aplicar un método de búsqueda en un árbol. El método es de búsqueda en profundidad.

Tal y como se dijo en el capítulo anterior, se va construyendo un árbol sintáctico en el que la raíz es el axioma inicial y los nodos son todas las derivaciones posibles para poder procesar una determinada sentencia. Pero podemos construir un árbol universal que englobe todas las posibles sentencias del lenguaje (o más bien de la gramática).

El método de ASD intenta encontrar en el árbol universal la sentencia a reconocer en cada momento. Pero es posible que esta búsqueda en profundidad se haga interminable porque haya ramas infinitas. Para evitar este inconveniente, se debe establecer un criterio para terminar la búsqueda por esa rama y establecerla por otra.

Veamos un ejemplo. Sea la gramática:

Regla 1 à E ::= T **+** *E*

Regla 2 à E ::= T

Regla 3 à T ::= F – T

Regla 4 à T ::= F

Regla 5 à F ::= ***id***

Regla 6 à F::= ***num***

Regla 7 à F ::= ***(E)***

Esta gramática reconoce expresiones aritméticas y no es ambigua. Vamos a construir su árbol universal.

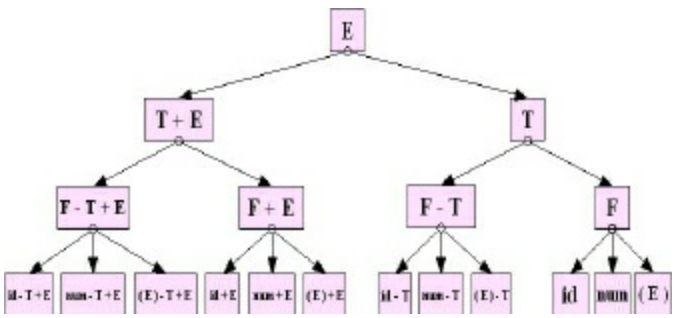


Figura 4.1*. Arbol universal para una gramática que reconoce expresiones*

En el árbol sólo hemos representado las derivaciones por la izquierda y los cuatro primeros niveles del árbol.

Si esta gramática fuera recursiva por la izquierda, claramente el proceso de

búsqueda podría no tener fin, por lo que debe evitarse la recursión por la izquierda.

El ASD consistiría en realizar la búsqueda de cada sentencia de la gramática y ver si existe en el árbol. Si existe, es que la sentencia es válida sintácticamente hablando, y si no existe es que es inválida.

Pero hay varias maneras de recorrer este árbol. Una de ellas es el *análisis con retroceso,* que consiste en recorrer el árbol de izquierda a derecha y de arriba abajo de manera que para la sentencia a analizar, la comparamos con cada nodo del árbol que vamos recorriendo y en el caso de que los terminales a la izquierda del primer no terminal no coincidan, terminaremos la búsqueda por esa rama y volveremos hacia atrás para buscar otra rama. Si todos los terminales a la izquierda del primer no terminal coinciden, seguiremos hacia abajo en el árbol. Si llegamos al momento en que todos son terminales y no queda ningún no terminal en un nodo, sabremos que la gramática reconoce la sentencia.

El problema del *análisis con retroceso*, aparte de requerir gramáticas no recursivas por la izquierda, es su ineficiencia.

Un ejemplo. Supongamos que queremos saber si la gramática anterior reconoce la sentencia "x – 65 + z".

Aplicamos la regla 1 à $E :: T + E$
Aplicamos la regla 3 à $E ::= F - T + E$
Aplicamos la regla 5 à $E ::= \mathbf{id} - T + E$
Aplicamos la regla 3 à $E ::= \mathbf{id} - F - T + E$
Aplicamos la regla 5 à $E ::= \mathbf{id} - \mathbf{id} - T + E$ *(falla)*
Aplicamos la regla 6 à $E ::= \mathbf{id} - \mathbf{num} - T + E$ *(falla)*
Aplicamos la regla 7 à $E ::= \mathbf{id} - \mathbf{(}\ E\ \mathbf{)} - T + E$ *(falla)*
Aplicamos la regla 4 à $E ::= \mathbf{id} - F + E$
Aplicamos la regla 5 à $E ::= \mathbf{id} - \mathbf{id} + E$ *(falla)*
Aplicamos la regla 6 à $E ::= \mathbf{id} - \mathbf{num} + E$
Aplicamos la regla 1 à $E ::= \mathbf{id} - \mathbf{num} + T + E$
Aplicamos la regla 3 à $E ::= \mathbf{id} - \mathbf{num} + F - T + E$
Aplicamos la regla 5 à $E ::= \mathbf{id} - \mathbf{num} + \mathbf{id} - T + E$ *(falla)*
Aplicamos la regla 6 à $E ::= \mathbf{id} - \mathbf{num} + \mathbf{num} - T + E$ *(falla)*
Aplicamos la regla 7 à $E ::= \mathbf{id} - \mathbf{num} + \mathbf{(}\ E\ \mathbf{)} - T + E$ *(falla)*
Aplicamos la regla 4 à $E ::= \mathbf{id} - \mathbf{num} + F + E$
Aplicamos la regla 5 à $E ::= \mathbf{id} - \mathbf{num} + \mathbf{id} + E$ *(falla)*
Aplicamos la regla 6 à $E ::= \mathbf{id} - \mathbf{num} + \mathbf{num} + E$ *(falla)*
Aplicamos la regla 7 à $E ::= \mathbf{id} - \mathbf{num} + \mathbf{(}\ E\ \mathbf{)} + E$ *(falla)*
Aplicamos la regla 2 à $E ::= \mathbf{id} - \mathbf{num} + T$
Aplicamos la regla 3 à $E ::= \mathbf{id} - \mathbf{num} + F - T$

Aplicamos la regla 5 à E ::= ***id – num + id*** *– T (falla)*
Aplicamos la regla 6 à E ::= ***id – num + num*** *– T (falla)*
Aplicamos la regla 7 à E ::= ***id – num + (*** *E* ***)*** *– T (falla)*
Aplicamos la regla 4 à E ::= ***id – num +*** *F*
Aplicamos la regla 5 à E ::= ***id – num + id*** *(sentencia encontrada)*

4.2 Analizadores sintácticos predictivos

Hemos visto que los *analizadores sintácticos con retroceso* son ineficientes. Pero hay maneras de aumentar la eficiencia. Una de ellas es siendo capaces de saber qué regla aplicar del conjunto de reglas aplicables en cada caso. Es decir, si tenemos un token y una serie de reglas a aplicar, debemos poder elegir una de ellas mirando si el primer token coincide con el que tenemos seleccionado.

Por ejemplo, en el ejemplo del epígrafe anterior, debíamos probar con cada una de las reglas 5, 6 y 7 para el no terminal *F*, pero si pudiéramos saber que el token actual es ***num***, directamente aplicaríamos la regla 6 ya que las otras no podrían ser válidas. Si tuviéramos una gramática en que en cada caso supiéramos con exactitud qué regla aplicar sabiendo sólo el primer token de la regla, podríamos utilizar un nuevo tipo de analizador sintáctico descendente, el *analizador sintáctico descendente predictivo* (ASDP).

A las gramáticas que cumplen estos requisitos, se les llama LL(1). Aunque su número es reducido, hay técnicas para convertir gramáticas que no sean LL(1) en LL(1).

Los ASDP son mucho menos costosos computacionalmente hablando que los ASD, por lo que se utilizan a la hora de implementar compiladores.

Para implementar un ASDP, se utiliza el concepto de *conjunto de predicción.* Se trata de relacionar cada regla de la gramática con todos los posibles terminales a que se puede acceder aplicando dicha regla.

Por ejemplo, si tenemos la siguiente gramática:

Regla 1 à A ::= ***a*** *B* ***c***

Regla 2 à A ::= ***x*** *C*

Regla 3 à A ::= B

Regla 4 à B ::= ***b*** *A*

Regla 5 à C ::= ***c***

Supongamos que tenemos en la entrada la siguiente sentencia “babxcc”. Veamos si es reconocible por el analizador.

¿Podemos aplicar la regla 1, la 2 o la 3? Sabemos que mediante la regla 3, seguida de la 4, podemos acceder al terminal ***b***, por lo que el terminal ***b*** pertenece al conjunto de predicción de la regla 3.

Luego, aplicamos la regla 3 à A ::= B

*Aplicamos la regla 4 à A ::= **b** A*

Ahora, nos encontramos con el token ***a***. ¿Se puede acceder a él desde *A*? Como se llega a él aplicando la regla 1, pertenece al conjunto de predicción de *la* regla 1 y, por lo tanto, es accesible.

*Aplicamos la regla 1 à A ::= **b a** B **c***

Ahora, tenemos el token ***b***. ¿Se accede a él mediante *B*? La respuesta es sí, ya que pertenece al conjunto de predicción de la regla 4.

*Aplicamos la regla 4 à A ::= **b a b** A **c***

Nos encontramos con ***x.*** ¿Es accesible desde A? Sí, ya que pertenece al conjunto de predicción de la regla 2.

*Aplicamos la regla 2 à A ::= **b a b x** C **c***

Tenemos ***c***. ¿Es accesible desde C? Como pertenece al conjunto de predicción de la regla 5, sí lo es.

*Aplicamos la regla 5 à A ::= **b a b x c c***

Hay algoritmos para obtener estos conjuntos de predicción.

4.3 Conjuntos de predicción y gramáticas LL(1)

Como vimos en el epígrafe anterior, debemos conocer el *conjunto de predicción* para poder implementar un ASDP. Para obtener este conjunto, debemos conocer, para cada regla, los primeros símbolos que puede generar su parte derecha. Si la parte derecha genera la cadena vacía, deberemos obtener los símbolos que aparecen a continuación de la parte izquierda de la regla en una forma sentencial derivable del símbolo inicial.

Para obtener el *conjunto de predicción*, primero debemos obtener dos conjuntos, el *conjunto de primeros* y el *conjunto de siguientes*.

4.3.1 Conjunto de primeros

Definimos la función PRIM como una función que se aplica a cadenas de símbolos, terminales o no terminales, de una gramática y devuelve un conjunto que contiene terminales de la gramática o la cadena vacía (€).

$$PRIM(\alpha) : (T \cup N)^* \rightarrow (T \cup \{€\})$$

Donde T es el conjunto de terminales de la gramática, N es el conjunto de no terminales de la gramática y a es un terminal o no terminal. U es el operador unión.

PRIM(α) es el conjunto de terminales o € que pueden aparecer iniciando las cadenas que pueden derivar de α.

Los casos para calcular a son:

- Si α es €, entonces PRIM(α) = {€} à Regla1
- Si no, sea α = a1 a2 a3 ...an donde al menos a1 no es € y donde ai es un terminal o un no terminal
- Si a1 es un terminal, entonces PRIM(α) = a1 à Regla2
- Si a1 es un no terminal, calculamos PRIM(a1). PRIM(a1) es el resultado de unir todos los PRIM de todas las partes derechas de las producciones de a1 en la gramática.
- Si € está incluido en PRIM(a1) y a1 no es el último símbolo de α, entonces,

PRIM(a) = (PRIM(a1) – {€}) U PRIM(a2….an) à Regla3

- En caso contrario, PRIM(α) = PRIM(a1) à Regla4

Veremos algunos ejemplos para comprenderlo mejor.

Sea la siguiente gramática:

Programa ::= Cabecera Declaracion Cuerpo | *Programa* ***fin***

Cabecera ::= ***inicio*** | ***€***

Declaracion ::= ***variable*** | ***€***

Cuerpo ::= ***sentencia*** | *Declaracion* ***finDeclaracion*** | ***€***

Vamos a calcular la función PRIM para cada uno de los no terminales.

PRIM(*Programa*) = (PRIM(*Cabecera Declaracion Cuerpo*) – {**€**}) U PRIM(*Programa* **fin**)

PRIM(*Cabecera Declaracion Cuerpo*) = (PRIM(*Cabecera*) – {**€**}) U PRIM(*Declaracion Cuerpo*)

Resultado 1 à PRIM(*Cabecera*) = {***inicio***,**€**}

PRIM(*Cabecera Declaracion Cuerpo*) = {***inicio***} U PRIM(*Declaracion Cuerpo*)

PRIM(*Declaracion Cuerpo*) = (PRIM(*Declaracion*) – {**€**}) U PRIM(*Cuerpo*)

Resultado 2 à PRIM(*Declaracion*) = {***variable***,**€**}

PRIM(*Declaracion Cuerpo*) = {**variable**} U PRIM(*Cuerpo*)

PRIM(*Cuerpo*) = {***sentencia***} U PRIM(*Declaracion* ***finDeclaracio****n*) U {**€**}

PRIM(*Declaracion* ***finDeclaracion***) = (PRIM(*Declaracion*) – {€}) U PRIM(***finDeclaracion***) = {***variable***} U {***finDeclaracion***}

PRIM(*Declaracion* ***finDeclaracion***) = {***variable***,***finDeclaracion***}

Resultado 3 à PRIM(*Cuerpo*) = {***sentencia***,***variable***,***finDeclaracion***,**€**}

PRIM(*Declaracion Cuerpo*) = {***sentencia***,***variable***,***finDeclaracion***,**€**}

PRIM(*Cabecera Declaracion Cuerpo)* = {***inicio***,***sentencia***,***variable***,***finDeclaracion***,**€**}

PRIM(*Programa fin*) = (PRIM(*Programa*) – {**€**}) U PRIM(***fin***) = (PRIM(*Programa*) – {**€**}) U {***fin***}

Como PRIM(*Cuerpo*) contiene {**€**} entonces:

PRIM(*Programa*) = {***inicio***,***sentencia***,***variable***,***finDeclaracion***,**€**} U {***fin***}

Si PRIM(*Cuerpo*) no contuviera {**€**}, entonces PRIM(*Programa*) sería:

({***inicio***,***sentencia***,***variable***,***finDeclaracion***,**€**} – {**€**}) U {***fin***}

Resultado 4 à PRIM(Programa) = {inicio,sentencia,variable,finDeclaracion,€,fin}

Hay una cosa a destacar. Cuando nos encontremos PRIM(*A*) = PRIM(*A*) U PRIM(*B*) y aún no hayamos calculado PRIM(*A*), eso equivale a PRIM(*A*) = PRIM(*B*).

Resumiendo, los conjuntos primeros son:

PRIM(*Programa*) = {***inicio***,***sentencia***,***variable***,***finDeclaracion***,**€**,***fin***}

PRIM(Cabecera) = {**inicio**,**€**}

PRIM(Declaracion) = {**variable**,**€**}

PRIM(Cuerpo) = {**sentencia**,**variable**,**finDeclaracion**,**€**}

Veamos otro ejemplo. Sea la gramática:

A ::= B C D

B ::= ***a*** *C* ***b***

B ::= **€**

C ::= ***c*** *A* ***d***

C ::= ***e*** *B* ***f***

C ::= ***g*** *D* ***h***

C ::= **€**

D ::= ***i***

Calcular los conjuntos primeros.

PRIM(*A*) = PRIM(*B C D*)

PRIM(*B C D*) = (PRIM(*B*) – {**€**}) U PRIM(*C D*)

PRIM(*C D*) = (PRIM(*C*) -{**€**}) U PRIM(*D*)

PRIM(*D*) = {***i***}

PRIM(*C*) = {***c***,***e***,***g***,**€**}

PRIM(*B*) = {***a***,**€**}

PRIM(*A*) = {***a***,***c***,***e***,***g***,***i***}

Vemos que PRIM(*A*) no contiene **€** porque PRIM(*D*) no lo contiene.

4.3.2 Conjunto de siguientes

Cada no terminal de la gramática tiene un conjunto llamado de siguientes. Este conjunto lo componen tanto terminales como el símbolo de final de la cadena de entrada ($). Son el conjunto de los elementos terminales o $ que pueden aparecer a continuación del no terminal en alguna forma sentencial derivada del símbolo inicial.

Para calcular el conjunto de siguientes, se realizan estos pasos. Sea *A* el no terminal del que vamos a calcular su conjunto de siguientes:

1. En principio SIG(*A*) = {}
2. Si *A* es el símbolo inicial, entonces SIG(*A*) = SIG(*A*) U {***$***}
3. Para cada regla de la forma *B* ::= α *A* β , entonces SIG(*A*) = SIG(*A*) U (PRIM(β) – {***€***})
4. Para cada regla de la forma *B* ::= α *A* o de la forma *B* ::= α *A* β en que ***€*** esté en PRIM(β) , entonces SIG(*A*) = SIG(*A*) U SIG(*B*)
5. Se repiten los pasos 3 y 4 hasta que no se puedan añadir más símbolos a SIG(*A*)

Como siempre, α y β pueden ser terminales o no terminales.

Veamos un ejemplo. Sea la gramática:

Programa ::= Cabecera Declaracion Cuerpo | *Programa* ***fin***

Cabecera ::= ***inicio*** | ***€***

Declaracion ::= ***variable*** | ***€***

Cuerpo ::= ***sentencia*** | *Declaracion* ***finDeclaracion*** | ***€***

Vamos a calcular los conjuntos siguientes de cada uno de los no terminales.

SIG(*Programa*) = {}

SIG(*Programa*) = SIG(*Programa*) U {***$***} = {***$***}

SIG(*Programa*) = SIG(*Programa*) U (PRIM(***fin***) – {***€***}) = {***$***,***fin***}

SIG(*Cabecera*) = {}

SIG(*Cabecera*) = SIG(*Cabecera*) U (PRIM(*Declaracion Cuerpo*) – {***€***})

Sabemos por el epígrafe anterior que,

PRIM(*Declaracion Cuerpo*) = {***sentencia***,***variable***,***finDeclaracion***,***€***}

SIG(Cabecera) = {sentencia,variable,finDeclaracion}

Como PRIM(*Declaracion Cuerpo*) contiene ***€***, entonces

SIG(*Cabecera*) = SIG(*Cabecera*) U SIG(*Programa*) = {***sentencia***,***variable***,***finDeclaracion***,***$***,***fin***}

SIG(Declaracion) = {}

SIG(Declaracion) = SIG(Declaracion) U (PRIM(Cuerpo) – {€}) = {**sentencia**,**variable**,**finDeclaracion**}

Como PRIM(*Cuerpo*) contiene €, entonces

SIG(*Declaracion*) = SIG(*Declaracion*) U SIG(*Programa*) = {***sentencia***,***variable***,***finDeclaracion***,***$***,***fin***}

SIG(Declaracion) = SIG(Declaracion) U (PRIM(**finDeclaracion**) – {€}) = {**sentencia**,**variable**,**finDeclaracion**,**$**,**fin**}

SIG(Cuerpo) = {}

SIG(Cuerpo) = SIG(Cuerpo) U SIG(Programa) = {**$**,**fin**}

4.3.3 Conjunto de predicción y gramáticas LL(1)

Ya estamos en condiciones de calcular el *conjunto de predicción* de cada regla de la gramática. Llamaremos PRED a la función que devuelve el *conjunto de predicción* de una regla. El *conjunto de predicción* puede contener cualquier terminal de la gramática y el símbolo de fin de cadena (*$*) pero nunca €.

Cuando el ASDP va a derivar un no terminal, mira el símbolo de entrada y lo busca en los conjuntos de predicción de todas las reglas de ese no terminal y sólo cuando esté en una sola de ellas es posible hacer una derivación por la izquierda con este método. Es decir, que todos los *conjuntos de predicción* de cada no terminal (reglas en que la parte izquierda sea ese no terminal) deben ser disjuntos entre sí para que se pueda utilizar un ASDP.

Para calcular el *conjunto de predicción* de una regla, se utiliza esta fórmula:

Si € está en PRIM(α), entonces PRED(*A* ::= α) = (PRIM(α) – {€}) U SIG(*A*)

Si € no está en PRIM(α), entonces PRED(*A* ::= α) = PRIM(α)

Como siempre, α puede ser un terminal o un no terminal.

Veamos si se puede construir un ASDP a partir de la gramática siguiente:

Programa ::= *Cabecera Declaracion Cuerpo* | *Programa* ***fin***

Cabecera ::= ***inicio*** | €

Declaracion ::= ***variable*** | €

Cuerpo ::= ***sentencia*** | *Declaracion* ***finDeclaracion*** | €

Como:

PRIM(*Cabecera Declaracion Cuerpo*) = {***inicio***,***sentencia***,***variable***,***finDeclaracion***,€}

Entonces:

PRED(*Programa* ::= *Cabecera Declaracion Cuerpo*) = (PRIM(*Cabecera Declaracion*

Cuerpo) – {**€**}) U SIG(*Programa*) = {***inicio,sentencia,variable,finDeclaracion***} U {***$,fin***} = {***inicio,sentencia,variable,finDeclaracion,$,fin***}

Como:

PRIM(*Programa **fin***) = {***inicio,sentencia,variable,finDeclaracion,fin***}

Entonces:

PRED(*Programa* ::= *Programa **fin***) = PRIM(*Programa **fin***) = {***inicio,sentencia,variable,finDeclaracion,fin***}

Y podemos apreciar que ambos *conjuntos de predicción* no son disjuntos, por lo que no es posible construir un ASDP.

De todas formas, como aprendizaje, seguiremos calculando los demás *conjuntos de predicción*.

PRED(*Cabecera* ::= ***inicio***) = {***inicio***}

PRED(Cabecera ::= **€**) = SIG(Cabecera) = {**sentencia**,**variable**,**finDeclaracion**,**$**,**fin**}

PRED(Declaracion ::= **variable**) = {**variable**}

PRED(Declaracion ::= **€**) = SIG(Declaracion) = {**sentencia**,**variable**,**finDeclaracion**,**$**,**fin**}

PRED(Cuerpo ::= **sentencia**) = {**sentencia**}

PRED(Cuerpo ::= Declaracion **finDeclaracion**) = {**variable**,**finDeclaracion**}

PRED(Cuerpo ::= **€**) = SIG(Cuerpo) = {**$**,**fin**}

La condición para que se pueda construir un ASDP a partir de una gramática nos dice si la gramática es del tipo LL(1). Es decir, toda gramática del tipo LL(1) permite construir un ASDP sobre ella, y viceversa.

El análisis de este tipo de gramáticas se caracteriza por:

Gramáticas LL(1) procesables con ASDP's

Condición LL(1):

Para todas las producciones de la gramática para un mismo no terminal A:

A ::= a1 | a2 | … | an

Se debe cumplir:

Para todo i,j distintos à (PRED(A ::= ai) and PRED(A ::= aj)) = {}

La secuencia de tokens se analiza de izquierda a derecha

Se deriva siempre el no terminal más a la izquierda

Con ver un solo token de la entrada se puede saber qué regla de producción hay que seguir

Veamos un ejemplo más. Hay que decidir si la siguiente gramática es de tipo LL(1).

S ::= ***if*** *C* ***then*** *P*

C ::= *C* ***or*** *D*

C ::= *D*
C ::= **(** *C* **)**
D ::= *D* ***and*** *E*
D ::= *E*
E ::= ***id***
P ::= *P* **+** *Q*
P ::= *Q*
Q ::= *Q* * *R*
Q ::= *R*
R ::= ***num***

Para ver si una gramática es de tipo LL(1), debemos comprobar si es posible analizarla con un ASDP. Debemos calcular sus conjuntos de predicción y ver si son disjuntos para cada grupo de conjuntos con el mismo no terminal en la parte izquierda de las reglas.

PRIM(*S*) = { ***if*** }
PRIM(*C*) = { **(** , ***id*** }
PRIM(*D*) = { ***id*** }
PRIM(*E*) = { ***id*** }
PRIM(*P*) = { ***num*** }
PRIM(*Q*) = { ***num*** }
PRIM(*R*) = { ***num*** }
SIG(*S*) = { **$** }
SIG(*C*) = PRIM(***then*** *P*) – {€} = { ***then*** }
SIG(*C*) = SIG(*C*) U (PRIM(***or*** *D*) – {€}) = { ***then*** , ***or*** }
SIG(*C*) = SIG(*C*) U (PRIM(**)**) – {€}) = { ***then*** , ***or*** , **)** }
SIG(*D*) = SIG(*C*) = { ***then*** }
SIG(*D*) = SIG(*D*) U (PRIM(***and*** *E*) – {€}) = { ***then*** , ***and*** }
SIG(*E*) = SIG(*D*) = { ***then*** , ***and*** }
SIG(*P*) = SIG(*S*) = { **$** }
SIG(*P*) = SIG(*P*) U (PRIM(+ *Q*) – {€}) = { **$** , **+** }
SIG(*Q*) = SIG(*P*) = { **$** , **+** }
SIG(*Q*) = SIG(*Q*) U (PRIM(* *R*) – {€}) = { **$** , **+** , * }
SIG(*R*) = SIG(*Q*) = { **$** , **+** , * }
PRED(*S* ::= ***if*** *C* ***then*** *P*) = PRIM(***if*** *C* ***then*** *P*) = { ***if*** }
PRED(*C* ::= *C* ***or*** *D*) = PRIM(*C* ***or*** *D*) = { **(** , ***id*** }
PRED(*C* ::= *D*) = PRIM(*D*) = { ***id*** }

Ya podemos saber que esta gramática no es LL(1) ya que el terminal ***id*** aparece en

los *conjuntos de predicción* de dos reglas que tienen a la izquierda el mismo no terminal.

4.4 Conversión a gramáticas LL(1)

La técnica anteriormente utilizada es infalible para asegurar si una gramática es LL(1) o no lo es. Pero se trata de una tarea que lleva su tiempo. Hay métodos que permiten saber sin realizar cálculos si una gramática no es directamente LL(1).

Con seguridad, sabemos que *una gramática no es LL(1) si*:

1. *Es una gramática ambigua.*
2. *Tiene factores comunes por la izquierda.*
3. *Es recursiva por la izquierda.*

Pero el caso contrario no tiene por qué ser cierto. Es decir, si una gramática no es ambigua no nos permite decir que es LL(1). Si una gramática no tiene factores comunes por la izquierda, no nos indica con certeza que sea LL(¡). Si una gramática no es recursiva por la izquierda, no nos asegura que sea LL(1).

Hay métodos para eliminar la ambigüedad, los factores comunes por la izquierda y la recursión por la izquierda. Pero aun así, es posible que la gramática obtenida sea LL(1), pero también es posible que no lo sea.

Ambigua o factores comunes por la izquierda o recursiva por la izquierda à no es LL(1)

No ambigua y sin factores comunes por la izquierda y no recursiva por la izquierda ¿ à ? es LL(1*)*

Veamos los métodos que podemos utilizar para salvar estas tres dificultades y obtener una gramática equivalente pero que a lo mejor sí es LL(1).

Para eliminar la ambigüedad, no hay ningún método establecido, por lo que lo mejor es replantear la gramática de nuevo.

4.4.1 Eliminación de la factorización por la izquierda

Si dos o más producciones de un mismo símbolo comienzan igual, no se sabe cuál de ellas elegir si se desconocen los primeros símbolos que son iguales. Por lo que habrá que evitar que esos símbolos sean iguales.

Por ejemplo:
*A ::= **a** B **c** D*
*A ::= **a** B **c** E*
*A ::= **a** B **c** F*

Para saber qué producción aplicar, debemos saber los tres primeros símbolos de

cada regla. Vamos a transformar la gramática para evitar esta situación.

Se soluciona de la siguiente manera.

Cambiar la regla del tipo:

A ::= α b1 | α b2 | . | α bn

Por estas reglas:

$A ::= \alpha\ A'$

$A' ::= b1 \mid b2 \mid \ldots \mid bn$

Por ejemplo:

$A ::= \boldsymbol{a}\ B\ \boldsymbol{c}\ D\ \boldsymbol{e} \mid C\ D$
$A ::= \boldsymbol{a}\ B\ \boldsymbol{c}\ D\ F$
$A ::= \boldsymbol{a}\ B\ \boldsymbol{c}\ D\ \boldsymbol{f}$
$C ::= \boldsymbol{k}$
$D ::= \boldsymbol{h}$
$F ::= \boldsymbol{d}$

Se convertiría en:

$A ::= \boldsymbol{a}\ B\ \boldsymbol{c}\ D\ A' \mid C\ D$
$A' ::= \boldsymbol{e} \mid F \mid \boldsymbol{f}$
$C ::= \boldsymbol{k}$
$D ::= \boldsymbol{h}$
$F ::= \boldsymbol{d}$

Pero como se dijo antes, esto no asegura que la nueva gramática sea LL(1).

4.4.2 Eliminación de la recursividad por la izquierda

Un analizador sintáctico descendente no podría nunca analizar este tipo de gramáticas porque se pondría a expandir el árbol por la izquierda indefinidamente.

El problema se soluciona de esta manera.

Cambiar la regla del tipo:

$A ::= A\ \alpha \mid b$

Por estas reglas:

$A ::= \beta\ A'$

$A' ::= \alpha\ A'$

$A' ::= €$

Por ejemplo:

E ::= *E* **+** *T* | *E* – *T* | *T* | *F*
T ::= *T* * *F* | *T* / *F* | *F*
F ::= ***num*** | ***(*** *E* ***)***

Se convertiría en:
E ::= *T E′* | *F E′*
E′ ::= **+** *T E′* | **-** *T E′*
E′ ::= **€**
T ::= *F T′*
T′ ::= * *T′* | / *T′*
T′ ::= **€**
F ::= ***num*** | ***(*** *E* ***)***

Pero hay un tipo de recursividad, que es indirecta y que no se resuelve por el método anterior. Por ejemplo:
S ::= *A* ***a*** | ***b***
A ::= *A* ***c*** | *S* ***d*** | **€**

Hay un algoritmo para eliminar esta recursividad, siempre que no haya reglas de la forma A :: A o de la forma A ::= € y que sean las únicas producciones del terminal a la izquierda.

Veamos el algoritmo para eliminar la recursividad indirecta:

1. Ordenar los no terminales según A1 , A2 , … , An
2. for(int i=1;i<=n;i++) {

```
for(int j=1;j<i;j++) {
sustituir(Ai ::= Aj χ , Ai ::= d1 χ | d2 χ | … | dk χ);
elimiarRecursividadIzquierda(Ai);
}
}
```

Donde Aj ::= d1 | d2 | … | dk son las producciones actuales de Aj

Por ejemplo. Eliminar la recursividad indirecta de:
S ::= *A* ***a*** | ***b***
A ::= *A* ***c*** | *S* ***d*** | **€**
A1 = *S*
A2 = *A*
i = 1, no se hace nada
i = 2 , j = 1 , sustituir (*A* ::= *S* ***d*** , *A* ::= *A* ***a d*** | ***b d***)
Nos queda:

$A ::= A\ \boldsymbol{a} \mid \boldsymbol{b}$

$A ::= A\ \boldsymbol{c} \mid A\ \boldsymbol{a}\ \boldsymbol{d} \mid \boldsymbol{b}\ \boldsymbol{d} \mid \boldsymbol{€}$

Eliminamos la recursividad por la izquierda:

$S :: = A\ \boldsymbol{a} \mid \boldsymbol{b}$

$A ::= \boldsymbol{b}\ \boldsymbol{d}\ A' \mid A'$

$A' ::= \boldsymbol{c}\ A' \mid \boldsymbol{a}\ \boldsymbol{d}\ A' \mid \boldsymbol{€}$

4.5 Analizadores sintácticos descendentes recursivos (ASDR)

Son un tipo especial de ASDP y por tanto sólo pueden analizar gramáticas LL(1). Se trata de implementar tantas funciones recursivas como no terminales de la gramática. Para cada no terminal se crea una función recursiva.

En la sección 3.5 vimos cómo construir un analizador sintáctico con diagramas de sintaxis. A partir de estos diagramas de sintaxis, y conociendo el siguiente token a analizar, es posible implementar un analizador sintáctico. En concreto, se trata de un analizador sintáctico descendente recursivo.

Para poder implementar un ASDR, es necesario que el lenguaje de implementación admita la recursión.

El proceso a seguir es crear los diagramas de sintaxis conforme a la gramática y a partir de ellos, realizar la implementación en el lenguaje elegido.

4.6 Implementación de ASDP's

Hemos visto en la sección anterior, que si conocemos el siguiente token a analizar, podemos construir un analizador sintáctico predictivo mediante diagramas de sintaxis y un lenguaje que admita la recursión. Pero hay otra manera de construir este tipo de analizadores: se trata de implementarlos a partir de una tabla.

Vamos a aprender en este apartado cómo construir esta tabla y a partir de ella implementar el analizador.

En vez de hacer llamadas recursivas, utilizaremos una pila de símbolos. Cuando tengamos el siguiente token, miraremos en la tabla, que se llama *tabla de análisis*, para saber qué producción toca invocar. Una vez construida la tabla, procederemos a implementar un algoritmo que se encargará de todo el proceso de análisis.

4.6.1 Construcción de la tabla de análisis

La tabla consta de filas, una por cada no terminal de la gramática, y columnas, una por cada terminal de la gramática y el símbolo de fin de fichero (*$*).

Para llenar las diferentes celdas, primero calcularemos los conjuntos de predicción de todas las reglas.

Para cada fila de la tabla buscaremos todas las reglas en las que el no terminal de la fila aparezca a la izquierda de la regla. Le llamaremos a este conjunto de reglas, *reglas de fila*. Tomamos cada regla del conjunto de *reglas de fila* y para cada uno de los elementos de su *conjunto de predicción*, ponemos la regla en la columna del terminal de su *conjunto de predicción* (y en la fila de su conjunto de *reglas de fila*). En las celdas que se queden vacías pondremos una señal para indicar que hay un error sintáctico (por ejemplo, la palabra *error*).

Con un ejemplo lo entenderemos mejor. Supongamos que tenemos estas dos reglas:

E ::= *T S*

T ::= **+** | **-**

S ::= ***num*** | **€**

PRED(E ::= T S) = { **+** , **-** } à Tabla[Fila1,Col1] y Tabla[Fila1,Col2]

PRED(T ::= **+**) = { **+** } à Tabla[Fila2,Col1]

PRED(T ::= **-**) = { **-** } à Tabla[Fila2,Col2]

PRED(S ::= **num**) = { **num** } à Tabla[Fila3,Col3]

PRED(S ::= **€**) = { **$** } à Tabla[Fila3,Col4]

	Col1	Col2	Col3	Col4
	+	**-**	***num***	***$***
Fila1	*E* *E ::= T S*	*E ::= T S*	*error*	*error*
Fila2	*T* *T ::=* **+**	*T ::=* **-**	*error*	*error*
Fila3	*S* *error*	*error*	*S ::=* ***num***	*S ::=* **€**

Las celdas con el fondo en blanco sólo son ilustrativas para entender el ejemplo y no se ponen en la tabla.

La tabla que hemos construido representa el autómata finito de pila de la gramática, por lo que es capaz de analizar cualquier sentencia y ver si es reconocida por la gramática correspondiente.

Una vez que tenemos la tabla construida, podemos saber cuándo habrá un error sintáctico. Hay dos casos posibles, uno es cuando intentemos emparejar un terminal (el siguiente token) con el símbolo de la cima de la pila. Si el símbolo de la cima es un terminal y no es el que queremos emparejar, habrá un error sintáctico.

También habrá un error sintáctico cuando intentemos acceder a una celda de la tabla en la que aparezca la palabra *error* (o la que hayamos elegido en su caso).

Estos analizadores sólo sirven para gramáticas LL(1). Si al construir la *tabla de análisis* aparece más de una regla en una celda, eso demostrará que la gramática no es LL(1) y, por tanto, no puede ser analizada por un ASDP.

La principal ventaja de este método respecto al recursivo es que el recursivo se construye a mano y este método es automatizable. Es decir, la *tabla de análisis* se puede

construir de manera automática y si por ejemplo la gramática cambia, es fácil adaptar la tabla a esos cambios.

Para ver el proceso de análisis con este tipo de tablas, utilizaremos una tabla manual (no forma parte del proceso de creación del ASDP) que iremos rellenando conforme vayamos avanzando en el análisis. Esta tabla consta de tres columnas, una para representar la pila de símbolos, otra para señalar los tokens que quedan en la entrada sin analizar y la última para indicar la regla a aplicar o la acción a realizar.

4.6.2 Algoritmo de análisis

El algoritmo de análisis sería este en Java (supondremos que tenemos implementadas las clases y métodos que vamos a utilizar).

```
class AnalisisASDP {
PilaSimbolos pila;
TablaAnalisis tabla;
void llenarTablaAnalisis() {
//Aquí se llenaria la tabla de analisis
}
void analizar() {
Símbolo A,a;
pila.push(Símbolo.símbolo$);
pila.push(Símbolo.simboloInicial);
while(A ¡= Símbolo.simbolo$) {
A = pila.getTope();
a = analex();//Siguiente símbolo de
//preanalisis
if(A.esTerminal() ||
A == Símbolo.simbolo$) {
if(A == a) {
pila.pop(A);
a = analex();
} else {
ErrorSintactico();
}
} else {
if(tabla(A,a) = regla(A ::= c1 c2
… ck)) {
pila.pop(A);
for(int i=k;i>0;i--) {
pila.push(ci);
}
} else {
ErrorSintactico;
}
}
}
}
public static void main(String[] args) {
AnalisisASDP analisis = new AnalisisASDP();
```

```
analisis.pila = new PilaSimbolos();
analisis.llenarTablaAnalisis();
analisis.analizar();
}
}
```

Para ver el desarrollo del proceso de análisis, utilizaremos la *tabla de análisis* y la tabla manual que hemos indicado en la sección anterior.

Veamos un ejemplo. Sea la gramática:
$S ::= A\ B$
$A ::= \boldsymbol{a} \mid \boldsymbol{€}$
$B ::= \boldsymbol{b}\ C\ \boldsymbol{d}$
$C ::= \boldsymbol{c} \mid \boldsymbol{€}$

Debemos calcular los *conjuntos de predicción* de cada regla.
PRIM(S) = { $\boldsymbol{a}$, $\boldsymbol{b}$ }
PRIM(A) = { $\boldsymbol{a}$, $\boldsymbol{€}$ }
PRIM(B) = { $\boldsymbol{b}$ }
PRIM(C) = { $\boldsymbol{c}$, $\boldsymbol{€}$ }
SIG(S) = { $\boldsymbol{\$}$ }
SIG(A) = { $\boldsymbol{b}$ }
SIG(B) = { $\boldsymbol{\$}$ }
SIG(C) = { $\boldsymbol{d}$ }
PRED($S ::= A\ B$) = { $\boldsymbol{a}$, $\boldsymbol{b}$ }
PRED($A ::= \boldsymbol{a}$) = { $\boldsymbol{a}$ }
PRED($A ::= \boldsymbol{€}$) = { $\boldsymbol{b}$ }
PRED($B ::= \boldsymbol{b}\ C\ \boldsymbol{d}$) = { $\boldsymbol{b}$ }
PRED($C ::= \boldsymbol{c}$) = { $\boldsymbol{c}$ }
PRED($C ::= \boldsymbol{€}$) { $\boldsymbol{d}$ }

Vemos que esta gramática es LL(1). Vamos a construir la *tabla de análisis*.

	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{\$}$
S	$S ::= A\ B$	$S ::= A\ B$	*error*	*error*	*error*
A	$A := \boldsymbol{a}$	$A ::= \boldsymbol{€}$	*error*	*error*	*error*
B	*error*	$B ::= \boldsymbol{b}\ C\ \boldsymbol{d}$	*error*	*error*	*error*
C	*error*	*error*	$C ::= \boldsymbol{c}$	$C ::= \boldsymbol{€}$	*error*

Ahora queremos saber si se aceptará la siguiente entrada: ***a b a d***

Construyamos la tabla manual.

Pila de símbolos	**Entrada**	**Regla o acción**
$\boldsymbol{\$}\ S$	$\underline{\boldsymbol{a}}\ \boldsymbol{b}\ \boldsymbol{c}\ \boldsymbol{d}\ \boldsymbol{\$}$	$S ::= A\ B$

$ B A	***a b c d $***	*A ::=* ***a***
$ B ***a***	***a b c d $***	Emparejar(***a***)
$ B	***b c d $***	*B ::=* ***b*** *C* ***d***
$ ***d*** *C* ***b***	***b c d $***	Emparejar(***b***)
$ ***d*** *C*	***c d $***	*C ::=* ***c***
$ ***d c***	***c d $***	Emparejar(***c***)
$ ***d***	***d $***	Emparejar(***d***)
$	***$***	Aceptar

Vemos que la entrada ha sido correctamente aceptada por la gramática.

Ahora vamos a ver si la siguiente entrada es aceptada por la gramática: ***a b d***

Pila de símbolos	**Entrada**	**Regla o acción**
$ S	***a b d $***	*S ::= A B*
$ B A	***a b d $***	*A ::=* ***a***
$ B ***a***	***a b d $***	Emparejar(***a***)
$ B	***b d $***	*B ::=* ***b*** *C* ***d***
$ ***d*** *C* ***b***	***b d $***	Emparejar(***b***)
$ ***d*** *C*	***d $***	*C ::=* **€**
$ ***d***	***d $***	Emparejar(***d***)
$	***$***	Aceptar

También es aceptada. Por último, probaremos con la siguiente: ***a b c***

Pila de símbolos	**Entrada**	**Regla o acción**
$ S	***a b c $***	*S ::= A B*
$ B A	***a b c $***	*A ::=* ***a***
$ B ***a***	***a b c $***	Emparejar(***a***)
$ B	***b c $***	*B ::=* ***b*** *C* ***d***
$ ***d*** *C* ***b***	***b c $***	Emparejar(***b***)
$ ***d*** *C*	***c $***	*C ::=* ***c***
$ ***d c***	***c $***	Emparejar(**c**)
$ ***d***	***$***	Error (***d*** <> *$*)

Cuando se detecte este error o el error de que se intente aplicar una regla y la celda de la tabla esté marcada como *error*, se debe sacar un mensaje con el tipo de error y el token causante al usuario. Si se estudia bien la implementación del proceso, es posible seguir analizando otros tokens e ir generando los errores que se encuentren. Pero hay veces en que un error provoca que todo el resto del análisis esté ya viciado, por lo que aparecerán infinidad de errores. Si esto es así, es mejor detener el análisis al encontrar el primer error.

4.7 Ejercicios resueltos

Ejercicio 4.1

Sea la siguiente gramática:

S ::= *A* | ***a***

A ::= ***b*** ***(*** *E* ***)*** *S L*

L ::= ***c*** *S* | ***€***

E ::= ***0*** | ***1***

Comprobar si es LL(1) y si lo es, construir su *tabla de análisis* y verificar si la entrada siguiente es analizada correctamente: ***a b (0) a c a***

PRIM(*S*) = { ***b*** , ***a*** }

PRIM(*A*) = { ***b*** }

PRIM(*L*) = { ***c*** , ***€*** }

PRIM(*E*) = { *0* , *1* }

SIG(*E*) = { ***)*** }

SIG(*S*) = { ***$*** , ***c*** }

SIG(*A*) = { ***$*** , ***c*** }

SIG(*L*) = { ***$*** , ***c*** }

PRED(*S* ::= *A*) = { ***b*** }

PRED(*S* ::= ***a***) = { ***a*** }

PRED(*A* ::= ***b*** ***(*** *E* ***)*** *S L*) = { ***b*** }

PRED(*L* ::= ***c*** *S*) = { ***c*** }

PRED(*L* ::= ***€***) = { ***$*** , ***c*** }

PRED(*E* ::= ***0***) = { ***0*** }

PRED(*E* ::= ***1***) = { ***1*** }

Vemos que para L hay dos *conjuntos de predicción* con el mismo terminal, por lo que la gramática no es LL(1).

Claramente podríamos haber notado que como la gramática tiene ciclos, no es LL(1).

Ejercicio 4.2

Hacer lo mismo que en el caso anterior y si no es LL(1) hacer las modificaciones pertinentes (si se puede) para convertirla en LL(1).

S ::= *S* ***or*** *Q*

S ::= *Q*

Q ::= *Q R*

Q ::= *R*

R ::= *F* ***and***

R ::= ***x***

R ::= ***y***

F ::= ***z***

Claramente no puede ser LL(1) porque es recursiva por la izquierda (*S* ::= *S* ***or*** *Q* y *Q* ::= *Q R*). Eliminamos esta recursividad.

S ::= *Q S′*

S′ ::= ***or*** *Q S′*

S′ ::= **€**

Q ::= *R Q′*

Q′ ::= *R Q′*

Q′ ::= **€**

R ::= *F* ***and***

R ::= ***x***

R ::= ***y***

F ::= ***z***

Veamos ahora si es LL(1).

PRIM(*S*) = { ***z*** , ***x*** , ***y*** }

PRIM(*S′*) = { *or* , **€** }

PRIM(*Q*) = { ***z*** , ***x*** , ***y*** }

PRIM(*Q′*) = { ***z*** , ***x*** , ***y*** , **€** }

PRIM(*R*) = { ***z*** , ***x*** , ***y*** }

PRIM(F) = { ***z*** }

SIG(*S*) = { **$** }

SIG(*S′*) = { **$** }

SIG(*Q*) = { **or** , **$** }

SIG(*Q′*) = { *or* , **$** }

SIG(*R*) = { ***z*** , ***x*** , ***y*** , ***or*** , ***$*** }

SIG(*F*) = { ***and*** }

PRED(*S ::= Q S′*) = { ***z*** , ***x*** , ***y*** }

PRED(*S′ ::=* ***or*** *Q S′*) = { ***or*** }

PRED(*S′ ::=* ***€***) = { ***$*** }

PRED(*Q ::= R Q′*) = { ***z*** , ***x*** , ***y*** }

PRED(*Q′ ::= R Q′*) = { ***z*** , ***x*** , ***y*** }

PRED(*Q′ ::=* ***€***) = { ***or*** , ***$*** }

PRED(*R ::= F* ***and***) = { ***z*** }

PRED(*R ::=* ***x***) = { **x** }

PRED(*R ::=* ***y***) = { ***y*** }

PRED(*F ::=* ***z***) = { ***z*** }

Y vemos que es LL(1). Construyamos su *tabla de análisis*.

	or	***and***	***x***	***y***	***z***	***$***
S	*error*	*error*	*S ::= Q S′*	*S ::= Q S′*	*S ::= Q S′*	*error*
S′	*S′ ::=* ***or*** *Q S′*	*error*	*error*	*error*	*error*	*S′ ::=* ***€***
Q	*error*	*error*	*Q ::= R Q′*	*Q ::= R Q′*	*Q ::= R Q′*	*error*
Q′	*Q′ ::=* ***€***	*error*	*Q′ ::= R Q′*	*Q′ ::= R Q′*	*Q′ ::= R Q′*	*Q′ ::=* ***€***
R	*error*	*error*	*R ::=* ***x***	*R ::=* ***y***	*R ::= F* ***and***	*error*
F	*error*	*error*	*error*	*error*	*F ::=* ***z***	*error*

Ahora vamos a probar si la siguiente entrada es analizada correctamente: ***z and or y x***

Pila de símbolos	**Entrada**	**Regla o acción**
$ *S*	***z and or y x $***	*S ::= Q S′*
$ *S′ Q*	***z and or y x $***	*Q ::= R Q′*
$ *S′ Q′ R*	***z and or y x $***	*R ::= F* ***and***
$ *S′ Q′* ***and*** *F*	***z and or y x $***	*F ::= z*
$ *S′ Q′* ***and z***	***z and or y x $***	Emparejar(***z***)
$ *S′ Q′* ***and***	***and or y x $***	Emparejar (***and***)
$ *S′ Q′*	***or y x $***	*Q′ ::=* ***€***

$ *S′*	***or y x $***	*S′ ::=* ***or*** *Q S′*
$ *S′ Q* ***or***	***or y x $***	Emparejar (***or***)
$ *S′ Q*	***y x $***	*Q ::= R Q′*
$ *S′ Q′ R*	***y x $***	*R ::=* ***y***
$ *S′ Q′* ***y***	***y x $***	Emparejar (***y***)
$ *S′ Q′*	***x $***	*Q′ ::= R Q′*
$ *S′ Q′ R*	***x $***	*R ::=* ***x***
$ *S′ Q′* ***x***	***x $***	Emparejar (***x***)
$ *S′ Q′*	***$***	*Q′ ::=* ***€***
$ *S′*	***$***	*S′ ::=* ***€***
$	***$***	Aceptar

CAPÍTULO 5

ANÁLISIS SINTÁCTICO ASCENDENTE

5.1 Introducción

Como ya se comentó en el capítulo 3, el análisis sintáctico ascendente consiste en ir construyendo la inversa de la derivación por la derecha, a partir de una cadena de entrada, de manera que se parte de las hojas del árbol de análisis y se llega a la raíz.

Este método en sí no es ni mejor ni peor, en principio, que el análisis sintáctico descendente. Pero se puede demostrar que el número de gramáticas que es posible analizar con este método es mayor que las analizables por el otro. Por lo tanto, si una gramática se nos resiste por métodos descendentes, podemos intentarlo por los ascendentes.

Lo mismo que había gramáticas LL(1), las hay LR(1), pero estas últimas forman un conjunto más amplio.

Hay métodos de análisis para gramáticas lineales por la derecha del tipo LR(1), pero el más utilizado, sobre todo por su sencillez, es el método SLR.

Hay dos ventajas que hacen que sea adecuado utilizar métodos ascendentes en vez de descendentes:

- Reconocen muchos más lenguajes que las gramáticas LL y en general, reconocen casi todos los lenguajes más comúnmente empleados en programación.
- Permiten localizar errores sintácticos muy precisamente.

Un inconveniente es que construir las tablas de análisis necesarias a mano es una tarea larga y compleja, por lo que se hace necesaria la utilización de herramientas de ayuda. Estas herramientas, incluso pueden informar al usuario de posibles problemas con la gramática, como por ejemplo cuando hay ambigüedad, de manera que facilitan al programador su tarea de diseño de la gramática.

La mayoría de los métodos o algoritmos para aplicar sobre gramáticas LR emplean lo que se llama un algoritmo por desplazamiento y reducción. Se utiliza una pila y una tabla de análisis, igual que en los métodos descendentes, pero con funcionamiento diferente.

Aparte del método SLR, hay varios métodos más para construir las tablas necesarias para el análisis LR, pero estos métodos, a pesar de ser más potentes (permiten analizar más número de gramáticas), son más complejos de implementar. De menos a más potencia (y de menos a más complejidad para su implementación) están los métodos:

SLR à LALR à LR canónico

Había ciertas condiciones para que una gramática fuera analizable por un ASDP, y la única condición para que lo sean mediante SLR es que sea posible construir la tabla de análisis sin problemas.

En lo sucesivo vamos a utilizar el algoritmo de desplazamiento y reducción para construir las tablas de análisis sintáctico.

5.2 Algoritmo de desplazamiento y reducción

Este algoritmo emplea cuatro acciones básicas:

- ACEPTAR: Se acepta la cadena.
- RECHAZAR: Se rechaza la cadena.
- REDUCIR: Se sustituye en la pila de símbolos la parte de la derecha de una producción por su parte izquierda.
- DESPLAZAR: Se lleva el símbolo de entrada a la pila y se pide el siguiente símbolo al analizador léxico.

Se utiliza una pila de estados y una tabla de análisis. En función de la cima de la pila y del estado actual, se pasa a otro estado o se realiza una acción (de las cuatro descritas anteriormente). A la acción de ir a un estado le vamos a llamar método GOTO.

Cada estado significa que se ha reconocido un elemento de la parte derecha de una regla.

El algoritmo de análisis es independiente de la gramática pero la tabla de análisis se debe construir para cada gramática.

En Java, el algoritmo sería:

```
boolean algoritmoDesplazarReducir() {
boolean retorno = false;
//Se supone que tenemos una pila de estados (desde
//el estdo 0)
apilar(0);
a = analex();//a es el siguiente token de la entrada
while(true) {
if(ACCION(cimaPila,a) == DESPLAZAR(n) {
apilar(n);
a = analex();
} else {
if(ACCION(cimaPilaPila,a)==REDUCIR(n){
for(int i=1;i<=longitudParteDerecha(n);i++){
desapilar();
}
A = parteIzquierdaRegla(n);
```

```
apilar(GOTO(cimaPila,A));
} else {
if(ACCION(cimaPila,a) == ACEPTAR) {
retorno = true;
break;
} else {
retorno = false;
break;
}
}
}
}
return retorno;
}
```

Describiremos más pormenorizadamente las acciones.

5.2.1 Acción ACEPTAR

Significa que el análisis ha finalizado con éxito.

5.2.2 Acción RECHAZAR

Significa que se ha producido un error. Por ejemplo que no existe ninguna acción de desplazar o reducir para un símbolo terminal de la entrada.

5.2.3 Método GOTO

Su formato es GOTO(*e*,*A*), donde *e* es un estado y *A* un no terminal (el de la parte izquierda de una regla). El estado es el de la pila de estados. Cuando se hace GOTO se debe apilar el estado indicado por esta acción. Por ejemplo, si tenemos GOTO(*2*,*A*) y en la tabla la fila correspondiente al estado *2* y la columna correspondiente al no terminal *A* está el estado *3*, significaría que deberíamos apilar el estado *3*.

5.2.4 Acción REDUCIR

Consiste en deshacer una regla. Su formato es REDUCIR(*n*), donde *n* es el número de regla (primero se deberían numerar todas las reglas de la gramática). Para reducir una regla, se desapilan tantos estados como símbolos tenga la parte derecha de la regla y luego se llama al método GOTO(*e*,*A*) donde *e* es el estado que hay en la cima de la pila (tras desapilar los estados correspondientes) y *A* es la parte izquierda de la regla.

5.2.5 Acción DESPLAZAR

Consiste en desplazar el puntero de entrada y apilar un estado. Su formato es DESPLAZAR(*n*), donde *n* es el estado a apilar.

5.2.6 Ejemplo de aplicación del algoritmo de desplazamiento y reducción

Supongamos que tenemos la siguiente gramática:

Regla 1 à E ::= E + *T*

Regla 2 à E ::= T

*Regla 3 à T ::= T * F*
Regla 4 à T ::= F
Regla 5 à F ::= (E)
Regla 6 à F ::= id

Supongamos que ya hemos construido la tabla de análisis:

Estado	*id*	+	*	(	)	$	*E*	*T*	*F*
0	D(5)			D(4)			1	2	3
1		D(6)				ACEPTAR			
2		R(2)	D(7)		R(2)	R(2)			
3		R(4)	R(4)		R(4)	R(4)			
4	D(5)			D(4)			8	2	3
5		R(6)	R(6)		R(6)	R(6)			
6	D(5)			D(4)				9	3
7	D(5)			D(4)					10
8		D(6)			D(11)				
9		R(1)	D(7)		R(1)	R(1)			
10		R(3)	R(3)		R(3)	R(3)			
11		R(5)	R(5)		R(5)	R(5)			

***Tabla 5.1**. Tabla de análisis*

Supongamos que tenemos la siguiente cadena de entrada: ***(id + id) * id***

Vamos a construir una tabla manual para ver lo que va ocurriendo conforme se procesa la entrada. Inicialmente, se carga la pila con el estado inicial (en este caso, le llamaremos 0).

0	***(id + id) * id $***	D(4)
0 4	***+ id) * id $***	D(5)
0 4 5	***+ id) * id $***	R(6) (*F ::=* ***id***)
0 4 3	***+ id) * id $***	R(4) (*T ::= F*)
0 4 2	***+ id) * id $***	R(2) (*E ::= T*)
0 4 8	***+ id) * id $***	D(6)
0 4 8 6	***id) * id $***	D(5)
0 4 8 6 5	***) * id $***	R(6) (*F ::=* ***id***)
0 4 8 6 3	***) * id $***	R(4) (*T ::= F*)
0 4 8 6 9	***) * id $***	R(2) (*E ::= T*)
0 4 8 6 9	***) * id $***	R(1) (*E ::= E* + *T*)
0 4 8	***) * id $***	D(11)
0 4 8 11	**** id $***	R(5) (*F ::=* ***(E)***)
0 3	**** id $***	R(4) (*T ::= F*)

0 2	*** id $**	D(7)
0 2 7	***id $***	D(7)
0 2 7 5	***$***	R(6) (*F ::=* ***id***)
0 2 7 10	***$***	R(3) (*T ::= T * F*)
0 2	***$***	R(2) (*E ::= T*)
0 1	***$***	ACEPTAR

Tabla 5.2*. Tabla manual para la entrada* ***(id + id) * id***

5.3 Construcción de tablas de análisis sintáctico SLR

De los tres métodos indicados en los apartados anteriores, el más sencillo para construir tablas de análisis sintáctico es el SLR, aunque es el menos potente (ya que permite analizar un menor número de gramáticas).

Vamos a ver en esta sección cómo se utiliza este método para construir la tabla de análisis sintáctico SLR para una gramática. Ya dijimos anteriormente que si somos capaces de construir la tabla sin problemas, significará que la gramática es analizable por el método SLR y, por tanto, será una gramática SLR.

Definiremos una serie de conceptos para estudiar el proceso.

5.3.1 Elemento

Un elemento se obtiene situando un punto ● en cualquier posición de la parte derecha de una regla. Los símbolos a la izquierda del punto han sido ya reconocidos y los de la derecha aún no.

Por ejemplo, de la siguiente gramática se pueden obtener los siguientes elementos:

$$S ::= T\ F$$
$$T ::= \boldsymbol{€}$$
$$F ::= \boldsymbol{id}\ G$$
$$G ::= \boldsymbol{num} \mid \boldsymbol{€}$$

↓

$$S ::= \bullet\ T\ F$$
$$S ::= T \bullet F$$
$$S ::= T\ F \bullet$$
$$T ::= \bullet$$
$$F ::= \bullet\ id\ G$$
$$F ::= id \bullet G$$
$$F ::= id\ G \bullet$$
$$G ::= \bullet\ num$$
$$G ::= num \bullet$$
$$G ::= \bullet$$

5.3.2 Cierre o clausura

Si I es un conjunto de elementos de una gramática, la operación clausura es el conjunto de elementos construido a partir de I por estas reglas:

- 1. Todo elemento de I se añade a *clausura(I)*
- 2. Si $A ::= \alpha \bullet B \beta$ está en *clausura(I)* y $B ::= \chi$ es una producción, añádase el elemento $B ::= \bullet \chi$ a *clausura(I)* si no estaba ya
- 3. Repítase la regla 2 hasta que no se puedan añadir más elementos a *clausura(I)*

Los términos α, β y χ pueden ser terminales o no terminales.

Veamos un ejemplo. Sea la gramática:
S ::= E
E ::= E **+** *T*
E ::= T
T ::= T * *F*
T ::= F
F ::= ***(*** *E* ***)***
F ::= ***id***

Supongamos que inicialmente I = { *S ::= ● E* }. Vamos a calcular *clausura(I).*

clausura(I) = clausura({ S ::= ● E })

Aplicando la regla 1, *clausura(I) = { S ::= ● E }*

Como { *S ::= ● E* } está en *clausura(I)* y *E ::= E* **+** *T* es una producción, aplicamos la regla 2 y añadimos a *clausura(I)* el elemento *E ::= ● E* **+** *T*. Entonces, tenemos que *clausura(I) = { S ::= ● E , E ::= ● E* **+** *T }.*

Aplicamos también la regla 2 para la producción *E ::= T* , y nos queda *clausura(I) = { S ::= ● E , E ::= ● E* **+** *T , E ::= ● T }.*

Ahora, tomamos *E ::= ● T* y obtenemos que *clausura(I) = { S ::= ● E , E ::= ● E* **+** *T , E ::= ● T , T ::= ● T* * *F , T ::= ● F }.*

Tomamos *T ::= ● F* y obtenemos que *clausura(I) = { S ::= ● E , E ::= ● E* **+** *T , E ::= ● T , T ::= ● T* * *F , T ::= ● F , F ::= ●* ***(*** *E* ***)*** *, F ::= ●* ***id*** *}.*

Ya no hay más producciones que no estén repetidas y que podamos añadir, por lo que se ha terminado el proceso.

5.3.3 Operación ir_a

Su formato es *ir_a(I,X)* donde I es un conjunto de elementos y *X* es un símbolo de la gramática.

ir_a(I,X) es la clausura del conjunto de todos los elementos { A ::= α *X* ● β } tales que { *A* ::= α ● *X* β } esté en *I*.

Donde *X*, α y β son terminales o no terminales.

Un ejemplo. Sea la siguiente gramática:

S ::= *E*
E ::= *E* + *T*
E ::= *T*
T ::= *T* * *F*
T ::= *F*
F ::= ***(E)***
F ::= ***id***
Sea *I = { S ::= E* ● *, E ::= E* ● *+ T }*, entonces *ir_a(I,+)* sería:
E ::= *E* + ● *T*
T ::= ● *T* * *F*
T ::= ● *F*
F ::= ● ***(E)***
F ::= ● ***id***

5.3.4 Construcción de la colección canónica de conjuntos de elementos

Llamemos *C* a la colección canónica de conjuntos de elementos de una gramática dada. Este es un paso previo a la construcción de la tabla de análisis.

Para calcular *C*, debemos seguir estos pasos:

- 1. Ampliar la gramática, añadiendo la regla *X* ::= *S* donde *S* es el símbolo inicial
- 2. Calcular *I0 = clausura({ X* ::= ● *S })* y añadirlo a *C*
- 3. Para cada conjunto de elementos *Ii* de *C* y para cada símbolo *A* (terminal o no terminal) para el que exista en *Ii* un elemento del tipo *B* ::= α ● *A* β donde la marca no esté al final, añadir *ir_a(Ii,A)* a *C* (si no estaba ya antes)
- 4. Repetir 3 hasta que no se puedan añadir nuevos conjuntos a *C*

Vamos a desarrollar todo el proceso para una gramática concreta. Sea la siguiente gramática:

E ::= *E* + *T*
E ::= *T*

T ::= *T* * *F*

T ::= *F*

F ::= ***(*** *E* ***)***

F ::= ***id***

Se añade la regla *X* ::= *E* . Nos quedaría la gramática:

X ::= *E*

E ::= *E* + *T*

E ::= *T*

T ::= *T* * *F*

T ::= *F*

F ::= ***(*** *E* ***)***

F ::= ***id***

I0 = clausura({ *X* ::= ● *E* }) = { *X* ::= ● *E* , *E* ::= ● *E* + *T* , *E* ::= ● *T* , *T* ::= ● *T* * *F* , *T* ::= ● *F* , *F* ::= ● ***(*** *E* ***)*** , *F* ::= ● ***id*** }

Como *X* ::= ● *E* y *E* ::= ● *E* + *T* están en I0, calculamos ir_a(I0,*E*).

I1 = ir_a(I0,*E*) = clausura({ *X* ::= *E* ● , *E* ::= *E* ● + *T* }) = { *X* ::= *E* ● , *E* ::= *E* ● + *T* }

Como *E* ::= ● *T* y *T* ::= ● *T* * *F* están en I0, calculamos ir_a(I0,*T*).

I2 = ir_a(I0,*T*) = clausura({ *E* ::= *T* ● , *T* ::= *T* ● * *F* }) = { *E* ::= *T* ● , *T* ::= *T* ● * *F* }

Como *T* ::= ● *F* está en I0, calculamos ir_a(I0,*F*).

I3 = ir_a(I0,*F*) = clausura({ *T* ::= *F* ● }) = { *T* ::= *F* ● }

Como *F* ::= ● ***(*** *E* ***)*** está en I0, calculamos ir_a(I0,***(***).

I4 = ir_a(I0,***(***) = clausura({ *F* ::= ***(*** ● *E* ***)*** }) = { *F* ::= ***(*** ● *E* ***)*** , *E* ::= ● *E* + *T* , *E* ::= ● *T* , *T* ::= ● *T* * *F* , *T* ::= ● *F* , *F* ::= ● ***(*** *E* ***)*** , *F* ::= ● ***id*** }

Como *F* ::= ● ***id*** está en I0, calculamos ir_a(I0,***id***).

I5 = ir_a(I0,***id***) = clausura({ *F* ::= ***id*** ● }) = { *F* ::= ***id*** ● }

Como *E* ::= *E* ● + *T* está en I1, calculamos ir_a(I1,+).

I6 = ir_a(I1,+) = clausura({ *E* ::= *E* + ● *T* }) = { *E* ::= *E* + ● *T* , *T* ::= ● *T* * *F* , *T* ::= ● *F* , *F* ::= ● ***(*** *E* ***)*** , *F* ::= ● ***id*** }

Como *T* ::= *T* ● * *F* está en I2, calculamos ir_a(I2,*).

I7 = ir_a(I2,*) = clausura({ *T* ::= *T* * ● *F* }) = { *T* ::= *T* * ● *F* , *F* ::= ● ***(*** *E* ***)*** , *F* ::= ● ***id*** }

Como *F* ::= ***(*** ● *E* ***)*** y *E* ::= ● *E* + *T* están en I4, calculamos ir_a(I4,*E*).

I8 = ir_a(I4,*E*) = clausura({ *F* ::= ***(*** *E* ● ***)*** , *E* ::= *E* ● + *T* }) = { *F* ::= ***(*** *E* ● ***)*** , *E* ::= *E* ● + *T* }

Como *E* ::= ● *T* y *T* ::= ● *T* * *F* están en I4, calculamos ir_a(I4,*T*).

I9 = ir_a(I4,*T*) = clausura({ *E* ::= *T* ● , *T* ::= *T* ● * *F* }) = { *E* ::= *T* ● , *T* ::= *T* ● * *F* } = I2

Reutilizamos I9.

Como $T ::= \bullet F$ está en I4, calculamos ir_a(I4,*F*).

I9 = ir_a(I4,F) = clausura({ $T ::= F \bullet$ }) = I3. Reutilizamos I9.

Como $F ::= \bullet$ ***(*** E ***)*** está en I4, calculamos ir_a(I4,***(***).

I9 = ir_a(I4,***(***) = clausura({ $F ::=$ ***(*** $\bullet E$ ***)*** }) = I4 . Reutilizamos I9.

Como $F ::= \bullet$ ***id*** está en I4, calculamos ir_a(I4,***id***).

I9 = ir_a(I4,***id***) = clausura({ $F ::=$ ***id*** $\bullet$ }) = I5. Reutilizamos I9.

Como $E ::= E + \bullet T$ y $T ::= \bullet T * F$ están en I6, calculamos ir_a(I6,*T*).

I9 = ir_a(I6,*T*) = clausura({ $E ::= E + T \bullet$, $T ::= T \bullet * F$ }) = { $E ::= E + T \bullet$, $T ::= T \bullet * F$ }

Como $T ::= \bullet F$ está en I6, calculamos ir_a(I6,*F*).

I10 = ir_a(I6,*F*) = clausura({ $T ::= F \bullet$ }) = I3. Reutilizamos I10.

Como $F ::= \bullet$ ***(*** E ***)*** está en I6, calculamos ir_a(I6,***(***).

I10 = ir_a(I6,***(***) = clausura({ $F ::=$ ***(*** $\bullet E$ ***)*** }) = { $F ::=$ ***(*** $\bullet E$ ***)*** , $E ::= \bullet E + T$, $E ::= \bullet T$, $T ::= \bullet T * F$, $T ::= \bullet F$, $F ::= \bullet$ ***(*** E ***)*** , $F ::= \bullet$ ***id*** } = I4. Reutilizamos I10.

Como $F ::= \bullet$ ***id*** está en I6, calculamos ir_a(I6,***id***).

I10 = ir_a(I6,***id***) = clausura({ $F ::=$ ***id*** $\bullet$ }) = I5. Reutilizamos I10.

Como $T ::= T * \bullet F$ está en I7, calculamos ir_a(I7,*F*).

I10 = ir_a(I7,F) = clausura({ $T ::= T * F \bullet$ }).

Como $F ::= \bullet$ ***(*** E ***)*** está en I7, calculamos ir_a(I7,***(***).

I11 = ir_a(I7,***(***) = clausura({ $F ::=$ ***(*** $\bullet E$ ***)*** }) = { $F ::=$ ***(*** $\bullet E$ ***)*** , $E ::= \bullet E + T$, $E ::= \bullet T$, $T ::= \bullet T * F$, $T ::= \bullet F$, $F ::= \bullet$ ***(*** E ***)*** , $F ::= \bullet$ ***id*** } = I4. Reutilizamos I11.

Como $F ::= \bullet$ ***id*** está en I7, calculamos ir_a(I7,***id***).

I11 = ir_a(I7,***id***) = clausura({ $F ::=$ ***id*** $\bullet$ }) = I5. Reutilizamos I11.

Como $F ::=$ ***(*** $E \bullet$ ***)*** está en I8, calculamos ir_a(I8,***)***).

I11 = ir_a(I8,***)***) = clausura({ $F ::=$ ***(*** E ***)*** $\bullet$ }) = { $F ::=$ ***(*** E ***)*** $\bullet$ }.

Como $E ::= E \bullet + T$ está en I8, calculamos ir_a(I8,+).

I12 = ir_a(I8,+) = clausura({ $E ::= E + \bullet T$ }) = { $E ::= E + \bullet T$, $T ::= \bullet T * F$, $T ::= \bullet F$, $F ::= \bullet$ ***(*** E ***)*** , $F ::= \bullet$ ***id*** } = I6. Reutilizamos I12.

Como $T ::= T \bullet * F$ está en I9, calculamos ir_a(I9,*).

I12 = ir_a(I9,*) = clausura({ $T ::= T * \bullet F$ }) = { $T ::= T * \bullet F$, $F ::= \bullet$ ***(*** E ***)*** , $F ::= \bullet$ ***id*** } = I7. Reutilizamos I12.

Y ya no se pueden añadir más conjuntos a C, por lo que se ha terminado el proceso.

C = { I0 , I1 , I2 , I3 , I4 , I5 , I6 , I7 , I8 , I9 , I10 , I11 }

5.3.5 Construcción de un autómata a partir de la colección canónica

Partiendo de los resultados obtenidos a la hora de construir la colección canónica de la gramática, podemos construir un autómata que reconozca todos los prefijos viables. No hay más que crear tantos estados como conjuntos de elementos de que conste la colección canónica.

Se crearán tantas transiciones como métodos *ir_a* hayamos utilizado. De manera que si Ii = ir_a(Ij,A), la transición asociada será Ij −−A−−> Ii.

Por ejemplo, para el caso del ejemplo anterior, el autómata será:

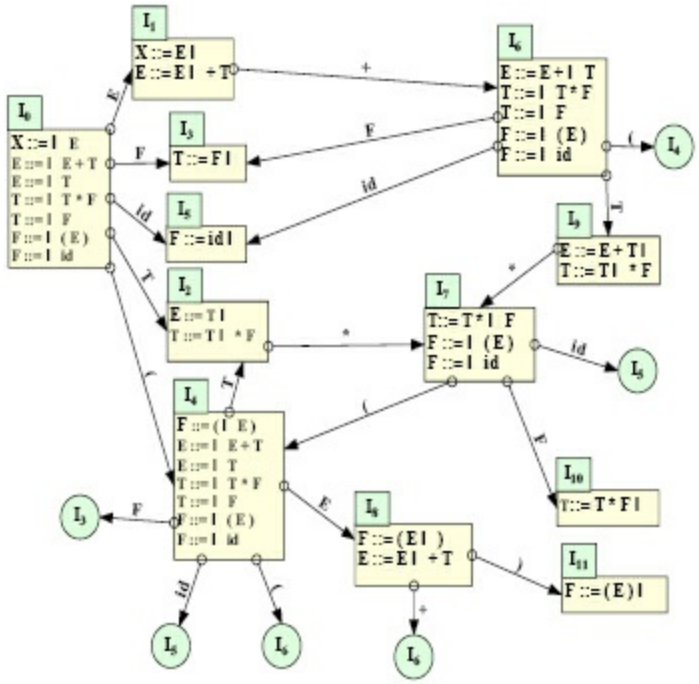


***Figura 5.1**. Autómata reconocedor de prefijos viables*

Como podemos ver en la figura 1, cada estado, identificado por Ii, contiene la lista de elementos de su conjunto I y un enlace con otros estados etiquetado con el terminal o no terminal que se debe encontrar para realizar la transición de estado. Para poder ver mejor la figura, cuando un estado conecta con otro estado lejano, en vez de poner una flecha que llegue a él, se pone una flecha que llegue a un círculo donde se indica el estado de destino de la flecha.

5.3.6 Construcción de la tabla de análisis a partir de un autómata

Una vez que tenemos el autómata reconocedor de prefijos viables, es posible construir la tabla de análisis.

Nos fijamos en el autómata y ponemos en la tabla de análisis lo siguiente:

- Por cada transición etiquetada con un terminal, se pone la acción

correspondiente al código del estado de destino. Por ejemplo, si hay una transición del estado 1 al 2 etiquetado con el terminal ***a***, en la celda de la fila del estado 1 y en la columna del terminal ***a*** se pone un desplazamiento D(2).

- Por cada transición etiquetada con un no terminal, se pone un *goto* correspondiente al estado de destino. Por ejemplo, si tenemos una transición del estado 1 al 2 etiquetada con el no terminal *A*, en la celda de la fila del estado 1 y la columna del no terminal *A* se pone el estado 2.
- Se calcula el conjunto de SIGUIENTES de cada no terminal de la gramática. Para cada estado, se toma cada uno de los elementos en los que el punto esté al final. Si está la regla que añadimos al comienzo con el punto al final, pondremos ACEPTAR en la fila del estado y la columna del terminal ***$***. Si no es esta regla, se pondrá un REDUCIR en la fila del estado y la columna del no terminal que esté en SIGUIENTES (para cada uno de los elementos que haya en SIGUIENTES) del no terminal de la parte izquierda del elemento. La regla a reducir es la correspondiente al elemento pero sin el punto al final.

Por ejemplo, para el caso del autómata anterior. Vamos a hacer su tabla de análisis.

Regla 0 à X ::= E
Regla 1 à E ::= E **+** *T*
Regla 2 à E ::= T
Regla 3 à T ::= T * *F*
Regla 4 à T ::= F
Regla 5 à F ::= ***(*** *E* ***)***
Regla 6 à F ::= ***id***

Calcularemos primero los SIGUIENTES de cada uno de los no terminales.
PRIM(*E*) = { ***(*** , ***id*** }
PRIM(*T*) = { ***(*** , ***id*** }
PRIM(*F*) = { ***(*** , ***id*** }
SIG(*E*) = { ***$*** , **+** , ***)*** }
SIG(*T*) = { ***$*** , **+** , ***)*** , * }
SIG(*F*) = { ***$*** , **+** , ***)*** , * }

Ahora, vamos a tomar uno por uno los estados y veremos cómo se irá llenando la tabla.

Para el estado I0, tenemos que salen 5 flechas y además no contiene ningún

elemento con un punto al final. Por lo tanto, para cada flecha etiquetada con un terminal, ponemos un desplazamiento en su columna y si es un no terminal, ponemos una transición a un estado en su columna.

La tabla quedaría así:

Estado	***id***	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	D(5)			D(4)			1	2	3

Para el estado I1, hay una transición etiquetada con un terminal y además hay un elemento con un punto al final, por lo que debemos añadir un desplazamiento y como el elemento sin el punto al final es la regla que añadimos al principio, debemos poner ACEPTAR en la columna correspondiente.

Estado	***id***	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	D(5)			D(4)			1	2	3
1		D(6)				ACEPTAR			

Para el estado I2, tenemos también una transición etiquetada con un terminal y además un elemento con un punto al final. Para la transición ponemos desplazar al tratarse de una etiqueta de terminal y para el elemento del punto al final, tomamos la regla quitándole el punto al final. El número de regla es el 2. Ahora, tomamos el no terminal de la parte izquierda de la regla y en cada una de las columnas de los no terminales del conjunto de los SIGUIENTES del no terminal, ponemos un REDUCE con el número de la regla, que es el 2.

Estado	***id***	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	D(5)			D(4)			1	2	3
1		D(6)				ACEPTAR			
2		R(2)	D(7)		R(2)	R(2)			

El proceso seguiría así hasta completar todos los estados. La tabla resultante es la Tabla 1.

Por último, comentar que los huecos en blanco significan que si llegamos a esa celda en la tabla significará que hay un error.

Podemos gestionar la salida de errores codificando cada uno de ellos con un número perteneciente a la celda donde se produce el error. Si la gramática es muy grande, esta tarea podrá ser tediosa (ya que seguramente habrá bastantes celdas de error) y se puede generar un mismo error para todas las celdas vacías de una misma fila. Pero toda la gestión de errores suele ser implementada según los gustos del programador.

5.3.7 Conflictos en las tablas SLR

Ya se comentó en capítulos anteriores que si es posible construir la tabla SLR para una gramática, entonces esa gramática es analizable por un SLR.

Hay dos posibles conflictos a la hora de construir una tabla SLR:

- **Desplazamiento-reducción**: se produce cuando en una misma celda podemos poner un desplazamiento y una reducción. Esto significa que la gramática no es SLR. Pero este conflicto se puede resolver eligiendo una de las dos opciones en función de la gramática que tengamos. Esta tarea es delicada, por lo que hay que tener cuidado de elegir la opción adecuada para que el analizador reconozca el lenguaje descrito por la gramática y no otro.
- **Reducción-reducción**: aparece cuando en una misma celda es posible reducir de dos maneras diferentes. Esto también implica que la gramática no es SLR. Su solución puede ser elegir una de las dos reducciones, teniendo cuidado de que el analizador reconozca bien el lenguaje definido por la gramática. También podemos optar por modificar la gramática para que esto no ocurra.

5.4 Organigrama de las gramáticas

Como punto final a los diferentes tipos de análisis sintáctico, vamos a resumir de una manera general las gramáticas, sus relaciones y sus características.

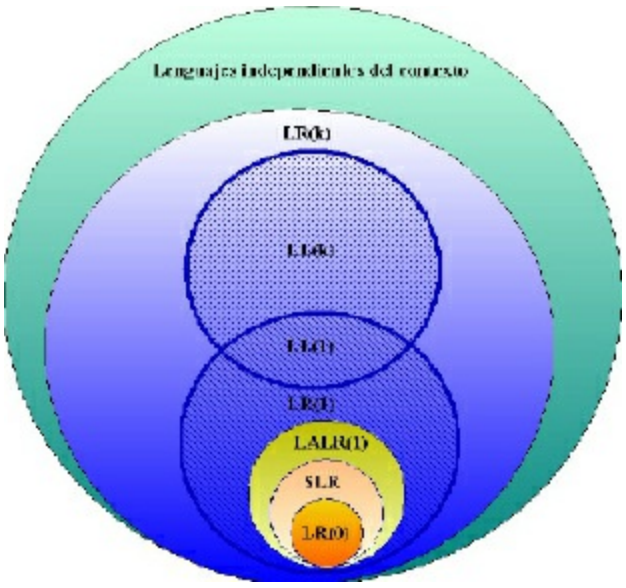


***Figura 5.2**. Organigrama de las gramáticas independientes del contexto*

Una aclaración a la figura 5.2. Se ha simplificado un poco para tener una visión mejor de las relaciones entre las gramáticas, pero es posible que haya un solapamiento

entre algunas gramáticas LL(1) y LALR(1) (y por ende, de las SLR y LR(0)). Es decir, que puede haber gramáticas que sean LL(1) y también LALR(1), o por ejemplo, LL(1) y SLR, etc.

Las gramáticas ambiguas no pueden ser LR(1), por lo tanto, tampoco pueden ser LL(1), ni LALR(1), ni SLR ni LR(0). Pero lo contrario no tiene por qué ser cierto: es decir, que una gramática no sea ambigua no quiere decir que sea LR(1) o LL(1) o LALR(1) o SLR o LR(0).

El hecho de que una gramática genere un lenguaje LL(1) no tiene por qué implicar que esa gramática sea LL(1), puede ser una que englobe a LL(1), por ejemplo LR(1).

Como en capítulos anteriores hemos hablado más abundantemente de gramáticas y analizadores LL(1) y SLR, veremos un resumen de ellas.

Una gramática es LL(1) si y sólo si los conjuntos de predicción de las reglas de cada variable son disjuntos entre sí.

Una gramática es SLR si es posible construir una tabla SLR para ella.

Si una gramática tiene recursividad por la izquierda, entonces no es LL(1).

Si una gramática tiene factores comunes por la izquierda, entonces no es LL(1).

Si una gramática es ambigua, entonces no es ni LL(1) ni SLR.

Si una gramática no es ambigua, puede que sea LL(1) o SLR., o puede que no lo sea.

Si una gramática es LL(1) o SLR, entonces no es ambigua.

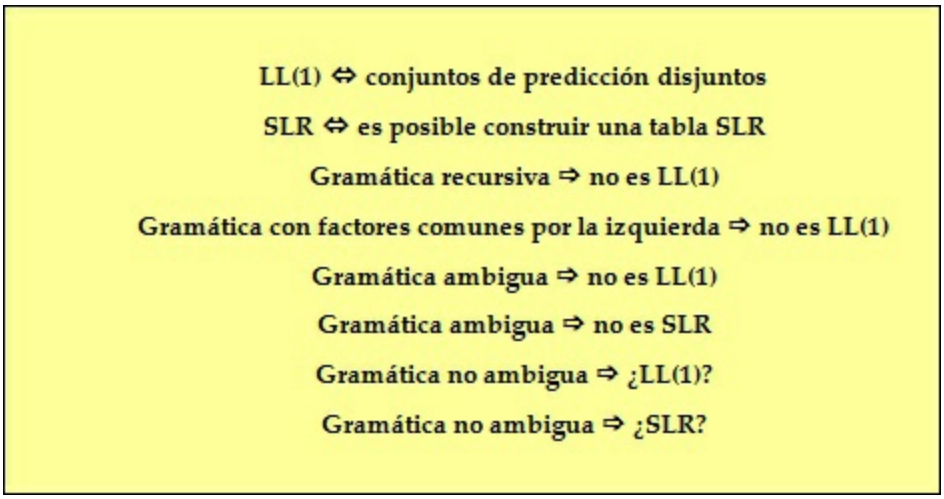


***Figura 5.3**. Características de las gramáticas LL(1) y SLR*

5.5 Ejercicios resueltos

Ejercicio 5.1

Para la siguiente gramática:
S ::= ***id*** *X* | ***id*** *Y* ***end***
X ::= ***otro*** | ***€***
Y ::= ***begin*** *X* ***end*** | ***€***

Construir el autómata y la tabla de análisis SLR.

Primero, añadimos la regla inicial y las numeramos.

Regla 0 à P ::= S

Regla 1 à S ::= ***id*** *X*

Regla 2 à S ::= ***id*** *Y* ***end***

Regla 3 à X ::= ***otro***

Regla 4 à X ::= ***€***

Regla 5 à Y ::= ***begin*** *X* ***end***

Regla 6 à Y ::= ***€***

Calculamos los diferentes estados.

I0 = clausura({ *P ::= ● S* }) = { *P ::= ● S* , *S ::= ●* ***id*** *X* , *S ::= ●* ***id*** *Y* ***end*** }

I1 = ir_a(I0,*S*) = clausura({ *P ::= S ●* }) = { *P ::= S ●* }

I2 = ir_a(I0,***id***) = clausura({ *S ::=* ***id*** *● X* , *S ::=* ***id*** *● Y* ***end*** }) = { *S ::=* ***id*** *● X* , *S ::=* ***id*** *● Y* ***end*** , *X ::= ●* ***otro*** , *X ::= ●* , *Y ::= ●* ***begin*** *X* ***end*** , *Y ::= ●* }

I3 = ir_a(I2,***otro***) = clausura({ *X ::=* ***otro*** *●* }) = { *X ::=* ***otro*** *●* }

I4 = ir_a(I2,*X*) = clausura({ *S ::=* ***id*** *X ●* }) = { *S ::=* ***id*** *X ●* }

I5 = ir_a(I2,*Y*) = clausura({ *S ::=* ***id*** *Y ●* ***end*** }) = { *S ::=* ***id*** *Y ●* ***end*** }

I6 = ir_a(I5,***end***) = clausura({ *S ::=* ***id*** *Y* ***end*** *●* }) = { *S ::=* ***id*** *Y* ***end*** *●* }

I7 = ir_a(I2,***begin***) = clausura({ *Y ::=* ***begin*** *● X* ***end*** }) = { *Y ::=* ***begin*** *● X* ***end*** , *X ::= ●* ***otro*** , *X ::= ●* }

I8 = ir_a(I7,*X*) = clausura({ *Y ::=* ***begin*** *X ●* ***end*** }) = { *Y ::=* ***begin*** *X ●* ***end*** }

I9 = ir_a(I7,***otro***) = clausura({ *X ::=* ***otro*** *●* }) = I3

I9 = ir_a(I8,***end***) = clausura({ *Y ::=* ***begin*** *X* ***end*** *●* }) = { *Y ::=* ***begin*** *X* ***end*** *●* }

Ahora, calculamos los conjuntos de SIGUIENTES.

PRIM(*S*) = { ***id*** }

PRIM(*X*) = { ***otro*** , ***€*** }

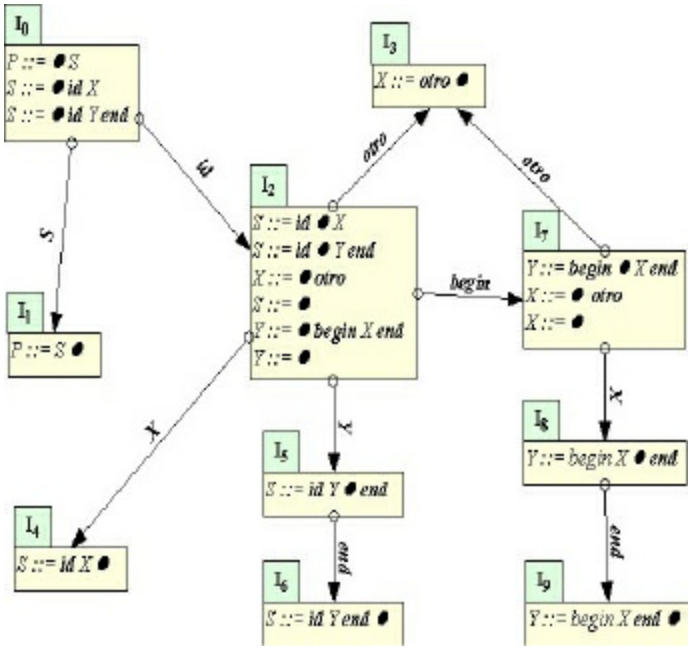
PRIM(*Y*) = { ***begin*** , ***€*** }

SIG(*S*) = { ***$*** }

SIG(*X*) = { ***$*** , ***end*** }

SIG(*Y*) = { ***end*** }

Dibujamos el autómata.



Y ahora, la tabla de análisis.

Estado	*id*	*end*	*begin*	*otro*	*$*	*S*	*X*	*Y*
0	D(2)					1		
1					ACEPTAR			
2		R(4)/R(6)	D(7)	D(3)	R(4)		4	5
3		R(3)			R(3)			
4					R(1)			
5		D(6)						
6					R(2)			
7		R(4)		D(3)	R(4)		8	
8		D(9)						
9		R(5)						

Vemos que hay un conflicto en el estado 2 con el terminal ***end***. Habría que modificar la gramática o elegir una de las dos reducciones, teniendo cuidado de que el lenguaje reconocido sea el que queremos.

Ejercicio 5.2

Para la siguiente gramática:

D ::= ***var*** *V* ***dospuntos*** *T* ***puntocoma***

V ::= ***id coma*** *V*

V ::= ***id***

T ::= ***integer***

T ::= ***boolean***

Construir el autómata y la tabla SLR. Analizar la cadena ***var id coma id dospuntos integer puntocoma***

Primero, añadimos la regla inicial y las numeramos:

Regla 0 à X ::= D

*Regla 1 à D ::= **var** V **dospuntos** T **puntocoma***

*Regla 2 à V ::= **id coma** V*

*Regla 3 à V ::= **id***

*Regla 4 à T ::= **integer***

*Regla 5 à T ::= **boolean***

I0 = clausura({ *X ::= ● D* }) = { *X ::= ● D* , *D ::= ● **var** V **dospuntos** T **puntocoma*** }

I1 = ir_a(I0,D) = clausura({ *X ::= D ●* }) = { *X ::= D ●* }

I2 = ir_a(I0,***var***) = clausura({ *D ::= **var** ● V **dospuntos** T **puntocoma*** }) = { *D ::= **var** ● V **dospuntos** T **puntocoma*** , *V ::= ● **id coma*** V , *V ::= ● **id*** }

I3 = ir_a(I2,*V*) = clausura({ *D ::= **var** V ● **dospuntos** T **puntocoma*** }) = { *D ::= **var** V ● **dospuntos** T **puntocoma*** }

I4 = ir_a(I2,***id***) = clausura({ *V ::= **id** ● **coma** V* , *V ::= **id** ●* }) = { *V ::= **id** ● **coma** V* , *V ::= **id** ●* }

I5 = ir_a(I3,***dospuntos***) = clausura({ *D ::= **var** V **dospuntos** ● T **puntocoma*** }) = { *D ::= **var** V **dospuntos** ● T **puntocoma*** , *T ::= ● **integer*** , *T ::= ● **boolean*** }

I6 = ir_a(I4,***coma***) = clausura({ *V ::= **id coma** ● V* }) = { *V ::= **id coma** ● V* , *V ::= ● **id coma*** V , *V ::= ● **id*** }

I7 = ir_a(I5,*T*) = clausura({ *D ::= **var** V **dospuntos** T ● **puntocoma*** }) = { *D ::= **var** V **dospuntos** T ● **puntocoma*** }

I8 = ir_a(I5,***integer***) = clausura({ *T ::= **integer** ●* }) = { *T ::= **integer** ●* }

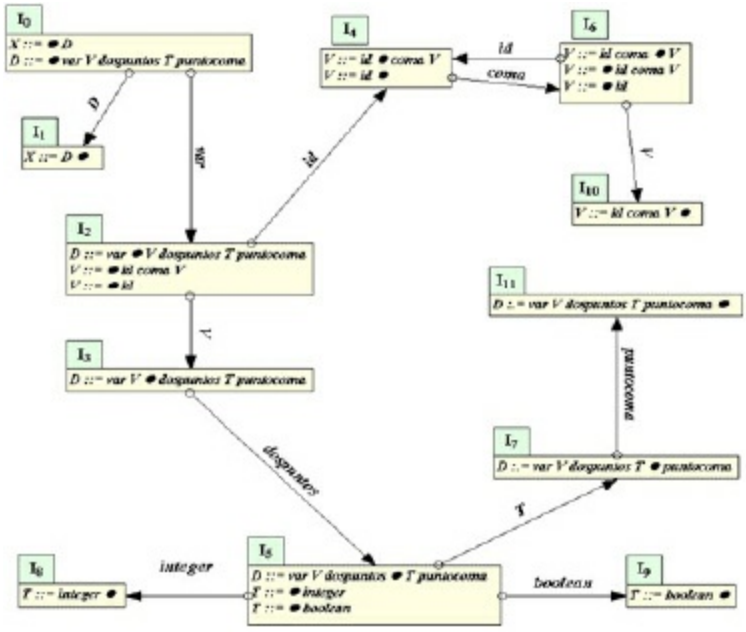
I9 = ir_a(I5,***boolean***) = clausura({ *T ::= **boolean** ●* }) = { *T ::= **boolean** ●* }

I10 = ir_a(I6,*V*) = clausura({ *V ::= **id coma** V ●* }) = { *V ::= **id coma** V ●* }

I11 = ir_a(I6,***id***) = clausura({ *V ::= **id** ● **coma** V* , *V ::= **id** ●* }) = I4

I11 = ir_a(I7,***puntocoma***) = clausura({ *D ::= **var** V **dospuntos** T **puntocoma*** ● }) = { *D ::= **var** V **dospuntos** T **puntocoma*** ● }

Dibujamos el autómata:



Calculamos los conjuntos de SIGUIENTES:

PRIM(*D*) = { ***var*** }

PRIM(*V*) = { ***id*** }

PRIMT(*T*) = { ***integer , boolean*** }

SIG(*D*) = { ***$*** }

SIG(*V*) = { ***dospuntos*** }

SIG(*T*) = { ***puntocoma*** }

Y ahora creamos la tabla SLR:

Estado	*var*	*dospuntos*	*puntocoma*	*id*	*coma*	*integer*	*boolean*	*$*	*D*	*V*	*T*
0	D(2)								1		
1								ACEPTAR			
2				D(4)					3		
3		D(5)									
4		R(3)			D(6)						
5						D(8)	D(9)			7	
6				D(4)					10		
7			D(11)								
8			R(4)								
9			R(5)								
10		R(2)									
11								R(1)			

Ahora, veremos lo que ocurre al procesar la cadena de entrada:

0	***var id coma id dospuntos integer puntocoma $***	D(2)
0 2	***id coma id dospuntos integer puntocoma $***	D(4)
0 2 4	***coma id dospuntos integer puntocoma $***	D(6)
0 2 4 6	***id dospuntos integer puntocoma $***	D(4)
0 2 4 6 4	***dospuntos integer puntocoma $***	R(3) (*V ::= **id***)
0 2 4 6 10	***dospuntos integer puntocoma $***	

		R(2) (*V* ::= ***id coma*** *V*)
0 2 3	***dospuntos integer puntocoma $***	D(5)
0 2 3 5	***integer puntocoma $***	D(8)
0 2 3 5 8	***puntocoma $***	R(4) (*T* ::= ***integer***)
0 2 3 5 7	***puntocoma $***	D(11)
0 2 3 5 7 11	***$***	R(1) (*D* ::= ***var*** *V* ***dospuntos*** *T* ***puntocoma***)
0 1	***$***	ACEPTAR

CAPÍTULO 6

TABLA DE TIPOS Y DE SÍMBOLOS

6.1 Introducción

Normalmente, los lenguajes de programación mantienen tipos primitivos de datos y permiten crear nuevos tipos a partir de los primitivos. También es generalizado el empleo de variables que pertenecen a alguno de los tipos primitivos o definidos por el usuario. Aparte de las variables, suelen utilizarse subprogramas. Tanto las variables como los subprogramas son generalmente creados por el programador y se les suele llamar símbolos.

Por lo tanto, debemos mantener básicamente dos estructuras de información: referentes a los tipos del lenguaje y a los símbolos del mismo.

Hay muchas maneras de mantener esa información, pero lo más utilizado es mediante dos tablas, una para los tipos y otra para los símbolos (aunque hay veces que se utilizan pilas de tablas si tenemos ámbitos de ejecución).

Tanto la tabla o tablas de tipos como la o las de símbolos deben estar accesibles durante todo el proceso de creación del compilador ya que las diferentes fases harán uso de estas tablas (en especial las fases de análisis sintáctico y semántico).

6.2 La tabla de tipos

Una de las tareas del análisis semántico, como veremos en el capítulo siguiente, es comprobar que los tipos de las diferentes variables, funciones o parámetros sean los adecuados conforme a la estructura del compilador que se esté creando. Por lo que una gestión de los tipos es esencial para hacer las comprobaciones semánticas necesarias.

Generalmente, los lenguajes de programación más utilizados mantienen una serie de tipos primitivos o predefinidos de datos y además permiten que el programador cree sus propios tipos de datos, basados en los primitivos.

Por lo tanto, la tabla o tablas de tipos deben inicializarse con los tipos primitivos del lenguaje del que vayamos a crear el compilador.

Hablamos de tabla o tablas porque hay varias posibilidades de implementar la tabla de tipos. Una de ellas es mantener una tabla de tipos por cada ámbito de ejecución.

Los lenguajes más utilizados permiten definir ámbitos de ejecución y dentro de estos ámbitos definir nuevos tipos de datos. Por lo que es adecuado crear una tabla de

tipos para cada ámbito.

Por ejemplo, C permite definir nuevos tipos dentro del código de subprogramas. Como esos tipos sólo son accesibles dentro del ámbito del subprograma, los tipos definidos en él se suelen eliminar ya que no son accesibles fuera del ámbito del subprograma. Es decir, dentro de un ámbito, se crea una tabla de tipos (y de símbolos) para los tipos (y los símbolos) que sólo son accesibles para ese ámbito (y los ámbitos que estén englobados bajo ese ámbito) y cuando se cierra el ámbito, se debe destruir esa tabla de tipos (y de símbolos) ya que no es posible acceder a ella desde fuera del ámbito.

Cuando hay ámbitos anidados, desde dentro de un ámbito se puede acceder sólo a su ámbito superior.

Con un ejemplo lo entenderemos mejor. Supongamos que tenemos un programa en Pascal (que permite definir subprogramas dentro de otros subprogramas):

```
program p;
type tipo0=array[0..3] of boolean;
procedure pp;
type tipo1=array[0..5] of integer;
begin
end;
begin
end.
```

Aquí tenemos dos ámbitos, uno es el del programa principal, y otro el del procedimiento.

Desde dentro del procedimiento se puede acceder a las variables que sean del tipo del programa principal pero desde el programa principal no se puede acceder a los tipos de dentro del procedimiento. Por lo tanto, cuando se cierra el procedimiento, la información de sus tipos (y variables) no es accesible, por lo que se debe eliminar. Desde el programa principal sólo es accesible el procedimiento, sus parámetros (si los hay) y el tipo que devuelve (si fuera una función). Toda la información añadida es local al procedimiento y, por tanto, no es accesible desde fuera de él.

Como en el proceso de compilación serán utilizadas varias tablas de tipos (generalmente), es muy común utilizar una pila de tablas de tipos e ir apilando y desapilando tablas según entramos o salimos de los ámbitos. Pero también es posible utilizar una sola tabla de tipos con información sobre el ámbito al que pertenece e ir eliminando entradas o añadiéndolas según salimos o entramos en los ámbitos (esto mismo se puede hacer para la tabla de símbolos).

Debido a esto, se suele utilizar una tabla de tipos para cada ámbito y luego destruirla cuando se sale del ámbito (lo mismo ocurriría con la tabla de símbolos).

Como el acceso a la tabla de tipos suele ser muy repetido, se deben utilizar estructuras eficientes a la hora de implementarla.

Dependiendo del diseño del compilador, es posible que se permita redefinir tipos o símbolos con el mismo nombre dentro de ámbitos diferentes (en algunos casos solapando los mismos tipos o símbolos de ámbitos que los engloban). Pero esto depende del diseñador del compilador. Cuando se permite redefinir tipos o símbolos, a la hora de buscar un tipo o símbolo para por ejemplo ver si el tipo devuelto por una función es compatible con un tipo concreto, se debe buscar el tipo o símbolo partiendo del ámbito más interno (el más cercano) y así sucesivamente hasta el ámbito más global o externo. De esta manera, nos aseguramos de que si se permite declarar tipos o símbolos con el mismo nombre pero en ámbitos que engloban unos a otros, encontraremos primero el tipo o símbolo más cercano (que es el que necesitamos).

Para implementar las tablas de tipos y símbolos, se suelen emplear tablas *hash*, que son las que tienen un tiempo de acceso más reducido (ya que habrá que realizar muchas lecturas a lo largo del proceso de compilación).

6.2.1 Implementación de la tabla de tipos

Generalmente, las tablas de tipos contienen información del nombre del tipo, el tamaño, el padre si se trata de un tipo compuesto y alguna información más dependiendo del tipo de compilador de que se trate.

Como en una misma tabla de tipos no se debe repetir el mismo tipo, la tabla suele estar ordenada por el nombre del tipo.

Si el compilador no admite ámbitos anidados, con una sola tabla de tipos es suficiente. Pero si el compilador admite definición de tipos en ámbitos anidados, es necesario gestionar o bien una tabla con información sobre el ámbito, o bien una pila de tablas de tipos (una entrada para cada ámbito). Es tarea del programador decidir cuál de las dos opciones es más recomendable.

Para el caso de implementar una sola tabla de tipos, habrá que añadir a cada entrada de la tabla un campo que indique el ámbito al que pertenece el tipo. En este caso, se pueden repetir nombres de tipos pero en diferentes ámbitos.

Los campos mínimos necesarios que debe guardar una tabla de tipos son:

- **Nombre**: es el nombre o identificador del tipo. Por ejemplo, en Java podría ser ***int***.
- **TipoBase**: se utiliza para tipos compuestos. Por ejemplo, si declaramos en Java un tipo ***String[] tipo***, el tipo base sería ***String***.
- **Padre**: es el tipo en el caso de declarar registros. Por ejemplo, en Pascal

```
type reg = record
x:integer;
y:boolean;
```

```
end;
```

En este caso, el padre del campo ***x*** sería el tipo ***reg***.

- **Dimensión**: esta información hace referencia al número de elementos de un tipo básico que están contenidos en un tipo compuesto. Por ejemplo, para los tipos básicos, la dimensión sería 1, pero para un vector de 10 enteros, la dimensión sería 10.
- **Mínimo**: este campo se utiliza para el caso de la definición de vectores. En C no hay problemas porque el índice mínimo sería 0, pero por ejemplo en Pascal, el índice mínimo lo define el programador.
- **Máximo**: lo mismo que en el caso anterior pero para el índice máximo. En algunos lenguajes como C, el índice mayor sería la dimensión del vector menos 1, pero en otros como Pascal, lo define el programador.
- **Ambito**: si vamos a utilizar una sola tabla para la gestión de tipos en diferentes ámbitos, pondremos aquí el ámbito en el que está definido el tipo. Lo normal es que comencemos con 0 y vayamos subiendo o bajando dependiendo de que entremos en un nuevo ámbito o salgamos de uno. Ya se ha comentado que no debe haber un mismo tipo en un mismo ámbito. También se ha comentado que cuando se sale de un ámbito se deben eliminar todos los tipos declarados en él.

Es habitual utilizar un campo adicional que contenga un código de tipo (en vez de hacer siempre referencia al nombre del tipo, podemos hacer referencia al código del tipo).

Otro aspecto a tener en cuenta a la hora de implementar la tabla de tipos es que debemos saber si el lenguaje es sensible a mayúsculas o no. Es decir, si por ejemplo ***integer*** es equivalente a ***Integer***. Esto hay que tenerlo en cuenta a la hora de decidir si un tipo está ya incluido en la tabla o no lo está.

6.2.2 Implementación de una tabla de tipos única

Vamos a implementar una tabla de tipos con este sistema y teniendo en cuenta que el lenguaje es sensible a las mayúsculas. Además, debemos señalar que los tipos básicos o predefinidos del lenguaje se suelen añadir al crearse la tabla de tipos.

La tabla de tipos que vamos a implementar puede servir para, por ejemplo, un subconjunto del lenguaje Java. Vamos a suponer que los tipos primitivos son ***int*** y ***boolean***.

```
class Tipo {
int cod;
```

```
String nombre;
int tipoBase;
int padre;
int dimension;
int minimo;
int maximo;
int ambito;
Tipo() {
cod = -1;
nombre = “”;
tipoBase = -1;
padre = -1;
dimension = -1;
minimo = -1;
maximo = -1;
ambito = -1;
}
void setCod(int c) {
cod = c;
}


void setNombre(String n) {
nombre = n;
}
void setTipoBase(int t) {
tipoBase = t;
}
void setPadre(int p) {
padre = p;
}
void setDimension(int d) {
dimension = d;
}
void setMinimo(int m) {
minimo = m;
}
void setMaximo(int m) {
maximo = m;
}
void setAmbito(int a) {
ambito = a;
}
int getCod() {
return cod;
}
String getNombre() {
return nombre;
}
int getTipobase() {
return tipoBase;
```

```
}
int getPadre() {
return padre;
}
int getDimension() {
return dimension;
}
int getMinimo() {
return minimo;
}
int getMaximo() {
return maximo;
}
int getAmbito() {
return ambito;
}
}
```

Utilizaremos el valor -1 cuando un atributo no tenga valor asignado.

Ahora implementaremos la clase para la gestión de la tabla. Utilizaremos por claridad la estructura *Vector* de Java, aunque en la implementación de un compilador comercial se debería utilizar una estructura más rápida. Al menos debe tener los siguientes métodos (aparte de otros que dependerán de la implementación concreta).

```
import java.util.Vector;
class TablaTipos {
private Vector tabla;
TablaTipos() {
Tabla = new Vector();
Tipo tipo = new Tipo();
tipo.setCod(0);
tipo.setNombre("int");
tipo.setDimension(1);
tipo.setAmbito(0);
addTipo(tipo);
tipo.setCod(1);
tipo.setNombre("boolean");
tipo.setDimension(1);
tipo.setAmbito(0);
addTipo(tipo);
}
//Inserta un elemento al final de la tabla
void addTipo(Tipo tipo) {
tabla.addElement(tipo);
}
//Comprueba si existe el nombre de un tipo en
//un ambito
Boolean existeTipoEnAmbito(String nombreTipo,
int ambito) {
//Como los ambitos iran incrementandose, se
```

```
//busca desde el
//final hacia el principio
//hasta que el ambito sea menor que el que
//buscamos
Tipo tipo;
boolean retorno = false;
for(int i=tabla.size()-1;i>=0;i--) {
tipo = getFinal();
if(tipo.getAmbito()>ambito) {
break;

} else {

if(tipo.getNombre().equals(nombreTipo)) {

retorno = true;

break;

}

}

return retorno;

}

}

//Devuelve el ultimo elemento de la tabla

Tipo getFinal() {

return (Tipo)tabla.lastElement();

}

//Comprueba si existe un tipo en toda la tabla

boolean existeTipo(String nombreTipo) {

boolean retorno = false;

Tipo tipo;

for(int i=tabla.size()-1;i>=0;i--) {
```

```
tipo = getTipo(i);

if(tipo.getNombre().equals(nombreTipo)) {

retorno = true;

break;

}

}

return retorno;
}
//Devuelve el tipo con un numero de orden
//concreto. Es el mismo que su codigo
Tipo getTipo(int i) {
return (Tipo)tabla.elementAt(i);
}
//Elimina el ultimo elemento de la tabla
void delFinal() {
tabla.removeElementAt(tabla.size()-1);
}
//Pone un tipo en una posicion de la tabla
void setTipo(Tipo tipo,int pos) {
tabla.setElementAt(tipo,pos);
}
//Devuelve el tipo correspondiente a un nombre
Tipo getTipo(String nombre) {
Tipo retorno;
Tipo tipo;
for(int i=tabla.size()-1;i>=0;i--) {
tipo = getTipo(i);
if(tipo.getNombre.equals(nombre)) {

retorno = tipo;
break;
}
}
return retorno;
}
//Devuelve el codigo de un tipo por su nombre
int getCodigoTipo(String nombre) {
Tipo tipo = getTipo(nombre);
return tipo.getCod();
}
//Elimina todos los tipos del ultimo ambito
void delUltimoAmbito() {
int tamano = tabla.size();
Tipo tipo = getFinal();
int ambitoActual = tipo.getAmbito();
```

```
for(int i=0;i<tamano;i++) {
tipo = getFinal();
if(tipo.getAmbito()>=ambitoActual) {
delFinal();
} else {
break;
}
}
}
```

Como ya se ha comentado, esto es sólo una parte de los métodos que se van a utilizar. Conforme se avance en la implementación, será aconsejable utilizar algunos métodos más.

Ahora, pondremos un ejemplo de utilización para ver cómo funciona. Supongamos un subconjunto del lenguaje Pascal. Supondremos que se permite la definición de tipos de registro y de vector, aparte de los dos tipos básicos ***integer*** y ***boolean***. Es decir, se permiten estas definiciones, por ejemplo:

```
type vector=array[0..10] of integer;
type registro=record
x:integer;
y:boolean;
end;
```

Supongamos que tenemos el siguiente programa y veremos cómo va cambiando el contenido de la tabla de tipos conforme se avanza en el análisis del programa.

```
1: program p;
2: type vector=array[0..9] of integer;
3: type registro=record
4: x:integer;
5: y:boolean;
6: end;
7: procedure p1;
8: type vector=array[0..8] of boolean;
9: type r1=record
10: y:integer;
11: z:integer;
12: end;
13: begin
14: end;
15: begin
16: end.
```

Programa 1

Vamos a suponer que este subconjunto de Pascal es sensible a las mayúsculas (aunque realmente Pascal no lo es).

Comencemos procesando el programa. Inicialmente, se añaden los tipos básicos a

la tabla de tipos.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0

Ahora, procesamos la línea 1, luego la 2.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0

Procesamos la línea 3.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0
3	registro	-1	-1	-1	-1	-1	0

Procesamos las líneas 4 y 5.

0	integer	-1	-1	1	-1	-1	0
1	boolean	1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0
3	registro	-1	-1	2	-1	-1	0
4	x	0	3	1	-1	-1	0
5	y	1	3	1	-1	-1	0

Una aclaración. Como vimos, hemos incluido los dos campos del registro en la tabla de tipos. Esto es necesario ya que cuando declaremos una variable de este tipo, deberemos acceder a cada campo de la variable por el nombre que tiene el campo en la tabla de tipos. Además, hemos calculado la dimensión del tipo 3 (registro) sumando las dimensiones de sus hijos (sus campos). Esto se ha hecho a posteriori, ya que en el momento de procesar la línea 3 no se ha procesado aún la 4 ni la 5. Más adelante, se comentará cómo calcular la dimensión de los tipos complejos.

Procesemos las líneas 6, 7 y 8. Tenemos la siguiente tabla.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0
3	registro	-1	-1	2	-1	-1	0
4	x	0	3	1	-1	-1	0
5	y	1	3	1	-1	-1	0
6	vector	1	-1	9	0	8	1

Vemos que como estamos en un ámbito más interno que el anterior, se ha incrementado el número del ámbito. También vemos que hemos definido un tipo con el mismo nombre de otro tipo anterior. Esto es posible ya que está en otro ámbito. Si queremos buscar el tipo de una variable, comenzaremos por el final, por lo que el que utilizaremos será el del ámbito 1.

Procesamos la línea 9.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0
3	registro	-1	-1	2	-1	-1	0
4	x	0	3	1	-1	-1	0
5	y	1	3	1	-1	-1	0
6	vector	1	-1	9	0	8	1
7	r1	-1	-1	-1	-1	-1	1

Ahora, procesamos las líneas 10 y 11.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0
3	registro	-1	-1	2	-1	-1	0
4	x	0	3	1	-1	-1	0
5	y	1	3	1	-1	-1	0
6	vector	1	-1	9	0	8	1
7	r1	-1	-1	2	-1	-1	1
8	y	0	7	1	-1	-1	1
9	z	0	7	1	-1	-1	1

Vemos que se ha modificado la dimensión del registro al procesar los dos campos. Procesemos ahora las líneas 12 y 13. La tabla no cambia. Pero al procesar la 14, terminamos la zona del procedimiento, eliminamos el ámbito y, por tanto, los tipos que están en él. La tabla quedaría así.

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	10	0	9	0
3	registro	-1	-1	2	-1	-1	0
4	x	0	3	1	-1	-1	0
5	y	1	3	1	-1	-1	0

Finalmente, se procesan las líneas 15 y 16. Al procesar la línea 16 (la última línea del programa), ya no necesitamos la tabla de tipos y por lo tanto la podemos eliminar.

6.2.3 Implementación de una pila de tablas de tipos

En esta implementación, vamos a utilizar la clase *Tipo* pero con un campo menos, el del ámbito. En vez de tener una sola tabla de tipos, tendremos una por cada ámbito. De manera que cuando entremos en un ámbito añadiremos a la pila de tablas una nueva tabla de tipos y al salir del ámbito la eliminaremos.

La búsqueda de tipos cambia un poco respecto a la implementación anterior ya que ahora no está todo en la misma tabla. Además, debemos implementar una pila de tablas.

En este caso, los tipos básicos sólo se incluirán en la primera tabla de tipos de la

pila y no en el resto.

```
import java.util.Vector;
class TablaTipos {
private Vector tabla;
TablaTipos() {
tabla = new Vector();
}
//Inserta los tipos basicos
void addTiposBasicos() {
Tipo tipo = new Tipo();
tipo.setCod(0);
tipo.setNombre("int");
tipo.setDimension(1);
tipo.setAmbito(0);
addTipo(tipo);
tipo.setCod(1);
tipo.setNombre("boolean");
tipo.setDimension(1);
tipo.setAmbito(0);
addTipo(tipo);

}

//Inserta un elemento al final de la tabla

void addTipo(Tipo tipo) {

tabla.addElement(tipo);

}
//Comprueba si existe el nombre de un tipo
boolean existeTipo(String nombreTipo) {
Tipo tipo;
boolean retorno = false;
for(int i=tabla.size()-1;i>=0;i--) {
tipo = getFinal();
if(tipo.getNombre().equals(nombreTipo)) {
retorno = true;
break;
}
return retorno;
}
}

//Devuelve el ultimo elemento de la tabla

Tipo getFinal() {

return (Tipo)tabla.lastElement();
```

```
}

//Devuelve el tipo con un numero de orden concreto. Es el mismo que

//su codigo

Tipo getTipo(int i) {

return (Tipo)tabla.elementAt(i);

}

//Elimina el ultimo elemento de la tabla

void delFinal() {

tabla.removeElementAt(tabla.size()-1);

}

//Pone un tipo en una posicion de la tabla

void setTipo(Tipo tipo,int pos) {

tabla.setElementAt(tipo,pos);

}

//Devuelve el tipo correspondiente a un nombre

Tipo getTipo(String nombre) {

Tipo retorno;

Tipo tipo;

for(int i=tabla.size()-1;i>=0;i--) {

tipo = getTipo(i);

if(tipo.getNombre.equals(nombre)) {

retorno = tipo;

break;

}
```

```
}

return retorno;

}

//Devuelve el codigo de un tipo por su nombre

int getCodigoTipo(String nombre) {

Tipo tipo = getTipo(nombre);

return tipo.getCod();

}

}
```

En esta nueva implementación de la tabla de tipos hay unos cambios respecto a la anterior. En concreto, se elimina la búsqueda por diversos ámbitos (se hará en la pila de tablas).

Tampoco se utiliza la eliminación de los tipos de un ámbito al cerrarlo, ya que se hará en la pila de tablas (se eliminará completamente la tabla correspondiente al ámbito).

Ahora toca implementar la pila de tablas.

```
import java.util.Vector;

class PilaTablasTipos {

private Vector pila;

PilaTablasTipos() {

TablaTipos tablaTipos = new TablaTipos();

tablaTipos.addTiposBasicos();

addTablaTipos(tablaTipos);

}

//Inserta un elemento en la cima de la pila

void addTablaTipos(TablaTipos tabla) {

pila.addElement(tabla);
```

```
}

//Elimina el elemento de la cima de la pila

void delCima() {

pila.removeElementAt(pila.size()-1);

}

//Extrae, pero sin eliminarlo, el elemento de la cima de la pila

TablaTipos getTablaTipos() {

return (TablaTipos)pila.iterator();

}

//Extrae, pero sin eliminarlo, el elemento de la pila situado en una

//posicion

TablaTipos getTablaTipos(int pos) {

return (TablaTipos)pila.elementAt(pos);

}

//Busca el nombre de un tipo en toda la pila de tablas de tipos

boolean existeTipo(String nombre) {

TablaTipos tabla;

boolean retorno = false;

for(int i=pila.size()-1;i>=0;i--) {

tabla = getTablaTipos(i);

if(tabla.existeTipo(nombre)) {

retorno = true;

break;

}
```

```
        }

        return retorno;

        }

        }
```

Es posible que necesitemos más métodos, pero eso lo veremos en la implementación.

Veremos cómo es la pila de tablas cuando se procesa el **Programa1**.

Al final de la línea 6 tendremos en la pila una sola tabla de tipos con este contenido.

Tabla 0

0	nteger	-1	1	1	1	1
1	boolean	-1	-1	1	-1	-1
2	vector	0	-1	10	0	9
3	registro	-1	-1	2	-1	-1
4	x	0	3	1	-1	-1
5	y	1	3	1	-1	-1

Cuando procesemos la línea 12, tendremos dos tablas en la pila:

Tabla 1

6	vector	1	-1	9	0	8
7	r1	-1	-1	2	-1	-1
8	y	0	7	1	-1	-1
9	z	0	7	1	-1	-1

Tabla 0

0	integer	-1	-1	1	-1	-1
1	boolean	-1	1	1	1	1
2	vector	0	-1	10	0	9
3	registro	-1	-1	2	-1	-1
4	x	0	3	1	-1	-1
5	y	1	3	1	-1	-1

Vemos que los códigos de la tabla 1 continúan con los de la tabla anterior y eso es necesario para que se pueda hacer referencias entre tipos de diferentes tablas. Sólo cuando se elimine una tabla, se pueden retomar los códigos a partir de la tabla anterior.

Por último, al procesar la línea 14, eliminaremos la tabla 1 de la pila y nos quedaremos sólo con un elemento, la tabla 0.

6.2.4 Dimensión y acceso a los elementos de los tipos

En esta sección vamos a estudiar un aspecto importante en las tablas de tipos, el dimensionado de los tipos y el acceso a los distintos elementos.

Cuando calculamos la dimensión de un tipo, si este es básico o predefinido es sencillo ya que sabemos de antemano su dimensión. Pero si se trata de un tipo estructurado no sabemos su dimensión previamente, sino que dependerá de la dimensión de cada uno de sus componentes. Si hay varios grados de anidamiento en la estructuración, habrá que realizar cálculos recursivos para obtener la dimensión.

Cuando decimos que sabemos por anticipado la dimensión de los tipos predefinidos, en realidad, lo que queremos decir es que vamos a normalizar su dimensión (ya que la dimensión real en la máquina dependerá de la estructura de la máquina donde corra el compilador).

Por convención, vamos a suponer que los tipos enteros, cardinales (enteros positivos) y booleanos son de dimensión 1. En cambio, los tipos reales, como necesitan más espacio para ser almacenados en memoria, tendrán una dimensión mayor en la máquina final pero también supondremos que en teoría tendrán dimensión 1 (al implementar el código final haremos las conversiones necesarias para adaptarlos a la dimensión real). Es decir, vamos a suponer que todos los tipos básicos se pueden contener en una unidad de memoria, por lo que tendrán dimensión 1.

Para el caso de los vectores, su dimensión se calcula muy fácilmente, es el producto entre el número de elementos del vector por la dimensión de su tipo base. Por ejemplo, si tenemos un vector de 10 elementos de tipo entero, la dimensión del vector será 10 x 1 = 10.

Para los registros el cálculo es igual de sencillo. La dimensión de un registro será la suma de las dimensiones de cada uno de sus campos. Por ejemplo, si tenemos un registro con tres campos, uno de tipo entero y dos de tipo booleano, la dimensión será 1 + 1 + 1 = 3.

Pero hemos supuesto el caso más sencillo, el caso en que el tipo base de los elementos del tipo estructurado es un tipo predefinido y por tanto de dimensión 1. Ahora vamos a suponer este caso en Pascal:

```
type registro= record
x:integer;
y:boolean;
end;
type vector=array[0..9] of registro;
```

Aquí se complica algo la cosa. Para el registro, el cálculo de la dimensión es sencillo, es 2. Pero para el vector ya no es tan sencillo, debemos multiplicar el número de elementos por la dimensión de su tipo base, que es 2. Por lo tanto, la dimensión del tipo

vector es 10 x 2 = 20. Tampoco es tan complicado.

Pero aún se puede complicar más, supongamos este caso:

```
type vector=array[0..10] of boolean;
type registro1=record
x:integer;
y:vector;
s:boolean;
end;
type registro2=record
z:boolean;
k:registro1;
end;
```

En este caso, la dimensión del tipo *vector* es 11 x 1 = 11. La dimensión del tipo *registro1* es 1 + 11 + 1 = 13. Y la dimensión del tipo *registro2* es 1 + 13 = 14.

Vemos que las dimensiones se irán calculando a partir de las dimensiones de los tipos base.

Otro aspecto importante es el acceso a cada uno de los elementos de un tipo estructurado. Por ejemplo, supongamos que necesitamos acceder al elemento *i* de un vector. Para saber en qué posición está el elemento que buscamos, debemos calcular el tamaño de todos los elementos anteriores (la suma del tamaño de los elementos desde *0* a *i-1*) y sumarle 1 para obtener la posición del elemento buscado.

Vamos a calcular la posición dentro del tipo *vector* del elemento *vector[6].*

posicion(vector,6) = (dimension(vector,0) + dimension(vector,1) + dimension(vector,2) + dimension(vector,3) + dimension(vector,4) + dimension(vector,5)) + 1 = (1 + 1 + 1 + 1 +1) + 1 = 5 + 1 = 6

Calculemos ahora la posición del elemento *registro1.y[6].*

posicion(registro1,y[6]) = dimension(registro1,x) + posicion(vector,6) = 1 + 6 = 7

La posición del elemento *registro1.s* sería la siguiente:

posicion(registro1,s) = dimension(registro1,x) + dimension(registro1,y) + 1 = 1 + 11 + 1 = 13

La posición del elemento *registro2.k.y[6]* sería la siguiente:

posicion(registro2,k.y[6]) = dimension(registro2,z) + posicion(registro1,y[6]) = 1 + dimension(registro1,x) + posicion(vector,6) = 1 + 1 + 6 = 8

6.3 La tabla de símbolos

La tabla de símbolos tiene una estructura parecida a la de tipos, pero su sentido es diferente. Cuando hablamos de símbolos en un programa nos referimos a las constantes, las variables, los subprogramas y los parámetros de los subprogramas.

La tabla de símbolos mantiene generalmente una gestión de ámbitos, tal y como

ocurría con la tabla de tipos (dependiendo de si el lenguaje admite ámbitos anidados). Por lo tanto, hay también al menos dos implementaciones posibles para la tabla de símbolos, con una sola tabla o con una pila de tablas.

En cuanto a la información que hay que guardar en la tabla de símbolos, no es exactamente la misma que en la de tipos ya que el contenido de los símbolos en un programa concreto hay que guardarlo en algún lugar de la memoria del computador (los tipos no se guardan en memoria una vez que se ha finalizado la compilación del programa, sino que sólo se utilizan para comprobaciones semánticas). Es decir, en tiempo de compilación, se mantiene la tabla de tipos y de símbolos en memoria, pero en tiempo de ejecución, sólo son los símbolos los que se guardan en la memoria que ocupa el programa compilado.

Los símbolos ocupan memoria en el programa compilado ya que ellos son los contenedores de la información que se procesa. Generalmente, la manera de guardar la información de los símbolos es en la memoria del computador donde corre el programa compilado. Por lo tanto, los símbolos se guardan en direcciones de memoria. Es por ello que uno de los campos de cada símbolo en la tabla de símbolos es una dirección de memoria donde se guardará el valor del símbolo (ese valor puede cambiar a lo largo de la ejecución del programa, pero su dirección suele ser fija). Vamos a suponer que cada símbolo ocupará tantas direcciones de memoria consecutivas como indique su dimensión.

En cuanto a los subprogramas, son representados también por símbolos (el nombre del subprograma) pero no guardan su valor en una dirección de memoria (más adelante, en la generación de código intermedio y final veremos para qué sirve la información que mantenemos en la tabla de símbolos).

En principio hay una serie de campos que debemos incluir en la tabla de símbolos, pero más adelante, en las siguientes fases del proceso de creación del compilador, veremos que será necesario incluir más datos en la tabla de símbolos. Por ahora, nos centraremos en los estrictamente necesarios en esta fase.

Como hicimos en la tabla de tipos, es habitual poner un código único a cada símbolo para referirnos a él en lugar de con su nombre. Los campos básicos de la tabla de símbolos son:

- **Nombre**: indica el nombre del símbolo. Por ejemplo, el nombre de una variable, el nombre de una función, etc.
- **Categoría**: indica si es una variable, una función, un procedimiento, una constante o un parámetro. Es necesario ya que en función de la categoría, el significado de algunos campos será diferente.
- **Tipo**: es el tipo al que pertenece el símbolo (de entre los tipos de la tabla de tipos). En el caso de los subprogramas es el tipo que devuelve (si no devuelve ninguno, este campo será nulo o generalmente tendrá el valor

-1).

- **NumeroParametros**: este campo sólo tiene significado si se trata del símbolo de un subprograma. En otro caso se mantiene nulo o -1.
- **ListaDeParametros**: este campo es estructurado, es decir, que debe contener una lista con al menos los tipos de los parámetros de un subprograma (para poder comprobar si cuando se hace una llamada a un subprograma coinciden los tipos de los parámetros). Si el subprograma no tiene parámetros, es nulo y si el símbolo no es de un subprograma es nulo también.
- **Dirección**: indica la dirección en memoria (absoluta o relativa) donde estará almacenado el valor del símbolo (si tiene valor). Si el símbolo pertenece a un tipo estructurado, esta dirección guardará el contenido del primer elemento y las sucesivas los demás elementos. La información de direccionamiento se incluirá en fases posteriores del análisis. Ahora, lo importante es saber que este campo es necesario.
- Ámbito: este campo sólo es necesario si la implementación va a ser sólo con una tabla de símbolos. Si va a implementarse como una pila de tablas, este campo es innecesario.

En cuanto al funcionamiento de la tabla de símbolos y la de tipos es el mismo. Por lo tanto, vamos a obviar la implementación ya que sería muy parecida a la tabla de tipos.

En cuanto al acceso a los elementos de la tabla de símbolos, ya se ha explicado en la sección anterior.

Veremos un ejemplo de funcionamiento. Sea el siguiente programa en Pascal:

```
Línea 1 : program p;
Línea 2 : type vector =array[5 .. 10] of integer;
Línea 3 : var v : vector; x : integer;
Línea 4 : begin
Línea 5 : v[7] := 15;
Línea 6 : x := v[7];
Línea 7 : end.
```

Este es un ejemplo muy sencillo pero puede dar una idea clara de cómo funciona todo el entramado de tablas. Vamos a suponer que los valores de las variables se van a guardar a partir de la dirección de memoria 9000 y consecutivas. Vamos a suponer también que cada entero se puede guardar en una sola dirección de memoria. Supongamos que nuestro lenguaje admite dos tipos básicos o predefinidos, el entero y el booleano. Y por último, vamos a utilizar la implementación de una sola tabla para todos los ámbitos.

Procesamos la línea 1.

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0

Tabla de símbolos:

Procesamos la línea 2.

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	6	5	10	0

Tabla de símbolos:

Procesamos la línea 3.

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	6	5	10	0

Tabla de símbolos:

0	v	variable	2	-1	null	9000	0
1	x	variable	0	-1	null	9006	0

Vemos que al declarar una variable, ponemos su entrada en la tabla de símbolos. También comprobamos que la dirección asignada es la 9000 (para el primer elemento del vector, el resto irán a partir de la 9000). Para la variable *x*, su dirección es la 9006 ya que las 6 anteriores, de la 9000 a la 9005 están ocupadas por el vector.

Procesamos las líneas 4 y 5. El contenido de las dos tablas es el mismo ya que no se han definido tipos nuevos ni se han declarado símbolos nuevos. Pero el contenido de la memoria sí ha cambiado ya que se le ha asignado un valor al elemento *v[7]*. Para calcular la dirección donde hay que poner el valor 15, hacemos los cálculos que se explicaron en la sección anterior.

$$direccion(v,7) = direccion(v,0) + posición(v,7) = 9000 + (7 - 5) = 9002$$

Es decir, en la posición de memoria 9002 se guardará el valor 15.

Ahora procesamos la línea 6. Se llena la dirección donde está la variable *x* con el

valor que hay en la dirección donde está *v[7]*. Es decir, la dirección 9006 se llena con el valor que hay en la dirección 9002.

El problema del direccionamiento se estudiará con más detalle cuando se genere el código intermedio y final, ahora sólo decir que en la información sobre los símbolos debe haber un campo para la dirección.

6.4 Ejercicios resueltos

Ejercicio 6.1

Dado el siguiente programa en un sublenguaje de Pascal:

```
Línea 1 : program p ;
Línea 2 :
Línea 3 : type vector = array [0 .. 7 ] of integer ;
Línea 4 : var x : integer ; v : vector ;
Línea 5 :
Línea 6 : function factorial( x : integer ) : integer;
Línea 7 : var y , z : integer ;
Línea 8 : begin
Línea 9 : y := 1 ;
Línea 10 : for z := 1 to x do y := y * z ;
Línea 11 : factorial := y
Línea 12 : end ;
Línea 13 :
Línea 14 : function suma( v : vector ; k : integer ) :vector;
Línea 15 : type vect = array [ 1 .. 8 ] of integer ;
Línea 16 : var x : integer ; v1 : vect ;
Línea 17 : begin
Línea 18 : for x := 0 to k do v1[ x + 1 ] := v[ x ] ;
Línea 19 : suma := v1
Línea 20 : end ;
Línea 21 :
Línea 22 : begin
Línea 23 : x := 5 ;
Línea 24 : x := factorial( x ) ;
Línea 25 : for x := 0 to 7 do v[ x ] := x ;
Línea 26 : v := suma( v , x )
Línea 27 : end .
```

Suponemos que este sublenguaje tiene dos tipos predefinidos: ***integer*** y ***boolean***. Suponemos que es sensible a las mayúsculas.

Sin tener en cuenta las direcciones y mediante la implementación de sólo dos tablas, una para los tipos y otra para los símbolos, mostramos el estado de ambas tablas tras procesar las líneas 4, 7, 12, 16, 20 y 26.

Inicialmente, el contenido de las tablas es:

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0

Tabla de símbolos:

Tras procesar la línea 4:

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	8	0	7	0

Tabla de símbolos:

0	x	variable	0	-1	null	-1	0
1	v	variable	2	-1	null	-1	0

Tras procesar la línea 7:

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	8	0	7	0

Tabla de símbolos:

0	x	variable	0	-1	null	-1	0
1	v	variable	2	-1	null	-1	0
2	factorial	funcion	0	1	[0]	-1	0
3	x	parametro	0	-1	null	-1	1
4	y	variable	0	-1	null	-1	1
5	z	variable	0	-1	null	-1	1

Vemos que en la tabla de símbolos, la función está en el ámbito 0 pero los parámetros y las variables declarados dentro de ella están en un ámbito superior. Esto es así porque fuera de la función, sólo ella es accesible pero no sus variables locales ni los parámetros concretos (sólo es necesario saber su número y el tipo de cada uno de ellos). La lista de parámetros de la función guarda los tipos de sus parámetros pero no los nombres, ya que son irrelevantes para quien llame a la función.

Tras procesar la línea 12:

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	8	0	7	0

Tabla de símbolos:

0	x	variable	0	-1	null	-1	0
1	v	variable	2	-1	null	-1	0
2	factorial	funcion	0	1	[0]	-1	0

Vemos que al terminar la zona de código de la función eliminamos todo su ámbito menos la entrada de la propia función.

Tras procesar la línea 16:

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	8	0	7	0
3	vect	0	-1	8	1	8	1

Tabla de símbolos:

0	x	variable	0	-1	null	-1	0
1	v	variable	2	-1	null	-1	0
2	factorial	funcion	0	1	[0]	-1	0
3	suma	funcion	2	2	[2,0]	-1	0
4	v	parametro	2	-1	null	-1	1
5	k	parametro	0	-1	null	-1	1
6	x	variable	0	-1	null	-1	1
7	v1	variable	3	-1	null	-1	1

Tras procesar la línea 20:

Tabla de tipos:

0	integer	-1	-1	1	-1	-1	0
1	boolean	-1	-1	1	-1	-1	0
2	vector	0	-1	8	0	7	0

Tabla de símbolos:

0	x	variable	0	-1	null	-1	0

1	v	variable	2	-1	null	-1	0
2	factorial	funcion	0	1	[0]	-1	0
3	suma	funcion	2	2	[2,0]	-1	0

Vemos que se han eliminado tanto en la tabla de tipos como en la de símbolos todas las entradas del ámbito 1.

Tras procesar la línea 26, las tablas quedan igual ya que no se han declarado ni tipos ni símbolos nuevos.

CAPÍTULO 7

ANÁLISIS SEMÁNTICO

7.1 Introducción

En las fases anteriores se han desarrollado las técnicas adecuadas para comprobar que una serie de palabras y símbolos están bien construidos respecto a las reglas impuestas por un lenguaje concreto. Pero el hecho de que estén bien construidos u ordenados no quiere decir que su significado sea el deseado por el creador del compilador. Queda una fase muy importante, que es comprobar que la serie de símbolos tiene un sentido respecto a lo que el creador del compilador desea. Es decir, es la hora de comprobar que en el programa hay una cohesión y un sentido.

Esta fase de análisis trata de verificar que los tipos que intervienen en las expresiones sean compatibles entre sí, que los parámetros que se le pasan a los subprogramas sean los adecuados tanto en número como en tipo, que las funciones devuelven valores adecuados en cuanto al tipo, etc.

Por ejemplo, no tendría sentido que intentáramos multiplicar un número entero con un vector de caracteres, o no tendría sentido que intentáramos llamar a una función de un parámetro pero no le pasáramos ninguno.

Generalmente, en la construcción de un compilador hay una relación muy intensa entre el análisis semántico y la siguiente fase (la generación de código intermedio y final). Además, el análisis sintáctico guía el proceso de análisis semántico, por lo que se habla de *traducción dirigida por la sintaxis*. Esto es así porque el analizador sintáctico es el que invoca los procesos de análisis semántico y generación de código al tiempo que realiza sus propias tareas de análisis sintáctico. Es decir, hay una superposición de tareas de análisis sintáctico, semántico y generación de código.

Hay que destacar que las tareas de análisis léxico y sintáctico están bastante sistematizadas, pero el análisis semántico se escapa en muchos puntos a la sistematización y es esencial que el ser humano dote de significado a su gramática para poder implementar el análisis semántico sobre ella.

Para poder dotar de significado a los símbolos, hay que asignarles cierta información que le llamamos *atributos*. Para manipular la información portada por los atributos de los símbolos, es decir, para poder realizar el análisis semántico, es necesario que las reglas de la gramática puedan realizar *acciones semánticas* y manipular los atributos. Estas acciones y esta manipulación se realizan incluyendo código de un lenguaje de programación determinado.

En la práctica, las fases de análisis sintáctico, semántico y generación de código intermedio se hacen todo junto y a la vez, por lo que se suele utilizar una misma herramienta para todo.

7.2 Atributos y acciones semánticas

En la sección anterior se ha visto que para realizar el análisis semántico se utilizan los conceptos de *atributo* y de *acción semántica*.

Los *atributos* son información personalizada (semántica) de los símbolos. Por lo que cada tipo de símbolo puede tener *atributos* diferentes. Esta información viaja a través del árbol de análisis sintáctico adjunta a cada símbolo. Las *acciones semánticas* se encargan de manipular el contenido de los atributos para verificar que existe un significado correcto en la relación de los símbolos entre sí. Cuando hablamos de símbolos nos referimos tanto a terminales como a no terminales.

La estructura de datos formada por los símbolos y sus atributos es parecida a una estructura de registro. Cada tipo de símbolo es un tipo de registro y cada atributo es un campo del registro. A la vez, cada atributo puede ser de un tipo concreto.

Por ejemplo, supongamos que tenemos el símbolo de una variable de tipo entero. Posiblemente, necesitaremos un atributo para guardar información sobre la dirección en memoria en donde se guardará el valor que tome la variable, también necesitaremos saber el tipo de la misma y alguna información más. Por ahora, supongamos que sólo necesitamos saber, aparte del nombre de la variable, su dirección en memoria y su tipo. Supongamos que vamos a referenciar su tipo por el nombre del tipo. Entonces, en Pascal, si tenemos estas reglas:

DeclVar ::= ***var nombreVar dospuntos*** *TipoVar* ***puntocoma***

TipoVar ::= ***entero*** | ***booleano***

Donde todos los símbolos son terminales excepto *DeclVar* y *TipoVar*, que son no terminales.

var.*lexema*, ***nombreVar***.*lexema*, ***nombreVar***.*direccion*, ***nombreVar***.*nombreTipo*, ***dospuntos***.*lexema*,, ***puntocoma***.*lexema*, ***entero***.*lexema*, ***booleano***.*lexema*

Los lexemas de ***entero*** y ***booleano*** serían “integer” y “boolean”, respectivamente.

Los diferentes atributos serían de diferentes tipos; por ejemplo, los lexemas podrían ser de tipo cadena, la dirección de tipo entero y el nombre del tipo de una variable de tipo cadena.

Hay dos tipos de terminales, los que están definidos en el análisis léxico como fijos y los que siguen un patrón definido, por lo que representan a una variedad de posibles lexemas. En el caso del ejemplo, hay un terminal que no tiene por qué tener un lexema fijo. Se trata del terminal ***nombreVar***, que representa el nombre de la variable concreta (acorde con el patrón establecido en el análisis léxico).

Para poder llenar la tabla de símbolos con la información necesaria de cada

símbolo, es preciso que los diferentes atributos tengan sus valores establecidos. Algunos valores vienen dados por la fase de análisis léxico, como son el contenido de los lexemas, pero hay otros valores que se deben calcular en el proceso de *traducción dirigida por la sintaxis*. Una vez que conocemos los valores de los *atributos*, ya podemos realizar el análisis semántico.

Pero para calcular los valores de los *atributos*, es preciso incluir las llamadas *acciones semánticas*, que no son más que trozos de código en un lenguaje de programación concreto, que manipulan los símbolos y sus *atributos*.

Por ejemplo, en el caso anterior, debemos incluir en la tabla de símbolos la dirección de la variable declarada y además su tipo. La dirección la podemos calcular a partir de la información que hay en la tabla de símbolos y de información global que utilicemos, pero el tipo sólo lo podemos obtener de la propia gramática.

Para pasar la información del tipo del no terminal *TipoVar* al tributo *nombreTipo* del terminal ***nombreVar***, utilizamos *acciones semánticas* inmersas en las reglas. Las *acciones semánticas* se suelen contener entre llaves (o llaves y dos puntos) para separarlas de la gramática. A grandes rasgos tendríamos:

```
DeclVar ::= var nombreVar dospuntos TipoVar puntocoma

{:

nombreVar.nombreTipo=TipoVar.nombreTipo;

:}

TipoVar ::= entero {:

TipoVar.nombreTipo=entero.lexema;

:}

| booleano {:

TipoVar.nombreTipo=booleano.lexema;

:}
```

Vemos que hemos utilizado un *atributo* para el no terminal *TipoVar*. Este *atributo* se encarga de guardar la información necesaria para poder ser utilizada por el terminal ***nombreVar***.

Reglas para la utilización de *atributos* en *acciones semánticas*:

1. Las *acciones semánticas* en las que intervengan *atributos* de la parte izquierda de una regla se pondrán al final de la regla.
2. Sólo se podrán incluir *acciones semánticas* en que intervengan *atributos* de un símbolo detrás de la aparición del símbolo en la regla.

En el ejemplo anterior hemos visto que se ha pasado información entre una regla y otra. Para entender esto, podemos asemejar un no terminal con una función de un lenguaje de programación. Esa función se encarga de leer de la entrada una serie de lexemas y luego devolver esa serie de lexemas a quien la ha llamado y, además, adjuntarle alguna información en forma de atributos. En el caso anterior, cuando se procesa la primera regla y se llega al no terminal *TipoVar*, es como si se llamara al procesamiento de la segunda regla. En la segunda regla, se hace una serie de reconocimiento de lexemas y se devuelve cierta información, que en este caso es el nombre del tipo de la variable. Esa información se devuelve en el *atributo* del no terminal *TipoVar* y es utilizada después para llenar el *atributo nombreTipo* del terminal ***nombreVar***. Por eso es necesario que se utilice un *atributo* sólo después de que aparezca en una regla.

Dependiendo de la herramienta que utilicemos, los no terminales y los terminales pueden tener todos sus atributos en una clase (por ejemplo, en Java). De manera que el terminal o no terminal es un objeto que pertenece a una clase. De esta manera, al devolver información en el caso anterior, podríamos devolver no sólo un atributo sino un objeto completo (así se podría pasar mucha más información entre reglas).

Por ejemplo, para el caso anterior, y utilizando Java, implementaremos más detalladamente las acciones y los tipos que utilizaríamos.

Supongamos definida la siguiente clase:

```
class Variable {

String lexema;

String tipoVariable;

Variable(String lex) {

lexema = lex;

tipoVariable = "";

}

void setTipo(String tipo) {

tipoVariable = tipo;
```

```
}

String getTipo() {

return tipoVariable;

}

}
```

Ahora, supongamos que usamos una herramienta que nos permite utilizar código java y reglas de la gramática. Tendríamos algo parecido a:

```
Analizador{

Terminales{

String var;

String nombreVar;

String dospuntos;

String puntocoma;

String entero;

String booleano;

}

NoTerminales{

String DeclVar;

String TipoVar;

}

}
```

Vemos que delante del nombre del terminal o no terminal ponemos el tipo al que pertenece. En principio, todos son de tipo cadena y sólo contendrán un *atributo* que es el nombre del lexema.

Ahora, podríamos incluir las reglas y las *acciones semánticas*.

```
DeclVar ::= var nombreVar dospuntos TipoVar puntocoma

{:

Variable variable = new Variable(nombreVar);

variable.setTipo(TipoVar);

//TipoVar es una cadena “integer” o “boolean”

//si_existe_simbolo_en_tabla_simbolos(nombreVar)→

//errorSemantico;

//si_no → insertar_en_tabla_simbolos(variable);

//Aquí no se devuelve nada, pero habría gramáticas en que sí

// hiciera falta

:}

TipoVar ::= entero

{:

return entero; //Devuelve la cadena “integer”

:}

| booleano

{:

return booleano; //Devuelve la cadena “boolean”

:}
```

Se ha puesto un ejemplo muy simplificado y sin utilizar la sintaxis de ninguna herramienta de ayuda concreta. En la parte de implementación se pondrán casos más complejos y completos.

Vemos que se aprovechan las *acciones semánticas* tanto para verificar la semántica como para manipular las tablas. En el siguiente capítulo veremos que también se aprovecharán para generar el código intermedio.

7.3 Tipos de atributos

Cuando realizamos una *traducción dirigida por la sintaxis*, en realidad estamos construyendo un árbol de análisis sintáctico y dotando a cada nodo de una serie de características, llamadas *atributos*. El proceso de análisis consiste en recorrer la cadena de componentes léxicos de entrada e ir construyendo su árbol de análisis sintáctico y después se recorre el árbol y se van ejecutando las *acciones semánticas* correspondientes.

A los árboles de análisis sintáctico en los que en sus nodos se guardan atributos se les suele llamar árboles adornados y a las gramáticas que se organizan de esta manera, *gramáticas atribuidas*.

Los atributos utilizados pueden ser de dos tipos:

- ***Atributos sintetizados***: se calculan a partir de los valores de los atributos de sus nodos hijos en el árbol adornado.
- ***Atributos heredados***: se calculan a partir de los atributos de los nodos hermanos o padres del árbol adornado.

Los atributos heredados se suelen utilizar para pasar información entre diferentes reglas de la gramática.

Cada nodo del árbol representa una instancia del símbolo gramatical de que se trata y, por lo tanto, tiene valores propios para sus atributos. Es decir, dos árboles que son iguales en cuanto a su estructura, no tienen por qué tener los mismos valores en los atributos de sus nodos.

Generalmente, los terminales no tienen atributos heredados sino sólo sintetizados ya que sus valores vienen dados por etapas anteriores del análisis y no cambian.

Veamos un ejemplo de utilización de atributos tanto sintetizados como heredados.

Sea la siguiente gramática en la que se especifica la declaración de variables en un lenguaje como por ejemplo Java:

DeclVar ::= TipoVar ListaVar ***puntocoma***

TipoVar ::= ***entero*** | ***real***

ListaVar ::= ListaVar ***coma nombreVar*** | ***nombreVar***

Hay que hacer unas puntualizaciones. El lexema del terminal ***entero*** es “int” y el del terminal ***real*** es “float”. Para ver mejor que dos no terminales son el mismo símbolo pero con atributos diferentes, vamos a diferenciarlos en el nombre, pero sin perder el significado. Es decir, utilizaremos la siguiente regla:

ListaVar ::= ListaVar1 ***coma nombreVar*** | ***nombreVar***

Utilizaremos una nueva manera de pasar los valores de los atributos. En vez de devolverlos como resultado de una función, utilizaremos la asignación de valores. Para ello, utilizaremos los siguientes atributos:

nombreVar*.lexema,* ***nombreVar****.tipo, ListaVar.tipo, TipoVar.tipo,* ***puntocoma****.lexema,* ***coma****.lexema,* ***entero****.lexema,* ***real****.lexema*

Introduciremos las acciones semánticas necesarias.

```
DeclVar ::= TipoVar ListaVar puntocoma

{:

ListaVar.tipo = TipoVar.tipo;

:}

TipoVar ::= entero

{:

TipoVar.tipo = entero.lexema;

:}

| real

{:

TipoVar.tipo = real.lexema;

:}

ListaVar ::= ListaVar1 coma nombreVar

{:

ListaVar1.tipo = ListaVar.tipo;

InsertaSimbolo(nombreVar.lexema,ListaVar.tipo);

:}

| nombreVar
```

```
{:

InsertaSimbolo(nombreVar.lexema,ListaVar.tipo);

:}
```

Hemos utilizado una función para asignar un tipo a un símbolo en la tabla de símbolos (la misma función es la que introduce el símbolo en la tabla). Si el símbolo ya está en la tabla de símbolos, la función deberá lanzar un error semántico.

Vemos que el atributo *tipo* de *ListaVar* es heredado. El resto son sintetizados.

7.4 Notaciones para la especificación de un traductor

Existen dos formas de asociar acciones semánticas con reglas de producción. En la sección anterior ya hemos visto ejemplos de alguna de ellas, pero en este apartado se especificarán las dos más significativas.

7.4.1 Definición dirigida por sintaxis (DDS)

Consiste en asociar una *acción semántica* a una regla de producción pero sin indicar cuándo se debe ejecutar dicha acción.

Las gramáticas con atributos son un ejemplo restringido de DDS. En ellas, los atributos son calculados a partir de otros atributos, pero sin que intervenga o se modifique nada externo a la gramática.

En una DDS se hace lo mismo que en las gramáticas con atributos pero además se puede manipular información externa a la gramática (esta manipulación suele ser necesaria para implementar compiladores reales). De todas formas, no se indica el orden de ejecución de las *acciones semánticas*.

Se suele representar en una tabla con dos columnas y tantas filas como reglas de la gramática. En la primera columna se pone la regla de la gramática y en la segunda las *acciones semánticas* asociadas a dicha regla.

Por ejemplo, supongamos que tenemos una gramática que nos permita calcular la suma de dos números e introducirla en una variable, pero esos números pueden ser de tipo entero o de tipo real. No se permite sumar dos números de un tipo diferente. Si la sintaxis es de tipo Java, las reglas podrían ser:

Asignación ::= ***nombreVar igual*** *Suma*

Suma ::= ***nombreVar mas nombreVar1 puntocoma***

Hemos supuesto que ya teníamos las reglas necesarias para declarar las variables e insertarlas en la tabla de símbolos. Las acciones semánticas asociadas podrían ser:

```
                                    if(Suma.tipo == nombreVar.tipo) {
```

Asignación ::= ***nombreVar igual*** *Suma*	nombreVar.valor = Suma.valor; } else Error_semantico;
Suma ::= ***nombreVar mas nombreVar1 puntocoma***	if(nombreVar.tipo == nombreVar1.tipo) { Suma.tipo = nombreVar.tipo; Suma.valor = nombreVar.valor + nombreVar1.valor; } else Error_semantico;

Un caso particular de gramática con atributos es una GAI (*gramática con atributos por la izquierda*) o gramática L-atribuida. En este tipo de gramáticas toda la información necesaria para su manipulación por las *acciones semánticas* está disponible en el momento de la ejecución de cada regla.

Una gramática DDS es GAI si para cada regla de la forma A ::= B1 B2 B3 … Bn cada atributo heredado de Bj (con $1 \leq j \leq n$) depende sólo de:

1. *Los atributos heredados o sintetizados de los símbolos B1 B2 B3 … Bj-1*
2. *Los atributos heredados de A*

Hay un ejemplo muy esclarecedor de una gramática que no es GAI. Vimos en la sección anterior una gramática para declarar variables en Java. Era de la forma:

DeclVar ::= TipoVar ListaVar ***puntocoma***

TipoVar ::= ***entero*** | ***real***

ListaVar ::= ListaVar1 ***coma nombreVar*** | ***nombreVar***

Esta gramática sí es GAI. Pero ahora veamos la misma declaración pero para Pascal:

DeclVar ::= ***var*** *ListaVar* ***dospuntos*** *TipoVar* ***puntocoma***

TipoVar ::= ***entero*** | ***real***

ListaVar ::= ListaVar1 ***coma nombreVar*** | ***nombreVar***

La diferencia entre ambas es que en la primera, cuando encontramos el nombre de una variable ya sabemos su tipo y podemos asignárselo, pero en la segunda, no sabemos el tipo de la variable hasta que no hemos leído toda la lista de variables. Vemos que *ListaVar* depende de *TipoVar* pero está antes en la regla por lo que no es GAI y por lo tanto no es suficiente con una sola pasada y sin utilizar información externa (ya que un DDS no puede utilizarla) realizar un análisis sintáctico. Si pudiéramos utilizar información externa, sí sería posible realizar dicho análisis (se utilizaría una estructura global para guardar las diferentes variables y al final se le asignaría el tipo a todas a la

vez).

Es posible modificar una DDS para que sea GAI. Por ejemplo, en el segundo caso:

DeclVar ::= ***var*** *ListaVar* ***puntocoma***

ListaVar ::= ***nombreVar coma*** *ListaVar1*

ListaVar ::= ***nombreVar dospuntos*** *TipoVar*

TipoVar ::= ***entero***

TipoVar ::= ***real***

7.4.2 Esquema de traducción (ETDS)

Un esquema de traducción es una gramática atribuida en la que hay intercalados en el lado derecho de las reglas de producción, fragmentos de código en un lenguaje de programación, que implementan acciones semánticas. Un ETDS es un DDS en que se da un orden en la ejecución de las *acciones semánticas*. Como ya se ha comentado antes, las *acciones semánticas* se sitúan a la derecha de los símbolos a los que se refieren y entre llaves. Esta regla de situar las *acciones semánticas* después de los símbolos que utilizan da un orden en su ejecución.

Una característica fundamental de un ETDS es que la traducción pueda realizarse de una sola pasada. Por lo tanto, un ETDS no permite herencia de los atributos desde la derecha hacia la izquierda.

Por todo ello, un ETDS es un GAI en que se insertan acciones semánticas.

Los ETDS se utilizan a menudo para convertir un formato de un lenguaje en el formato de otro lenguaje.

Si tenemos una gramática y queremos que sea un ETDS, deberemos decidir los atributos necesarios y asignarlos a los símbolos de la gramática. Luego, debemos insertar las acciones semánticas necesarias. En este segundo paso debemos tener en cuenta que se deben cumplir las restricciones de un ETDS, es decir:

1. Si todos los atributos son sintetizados, pondremos las acciones semánticas después de los atributos implicados. Lo mejor es situarlas al final de la regla de producción. Para cada atributo sintetizado, siempre hay que calcularlo después de que hayan tomado valor todos los demás atributos que intervienen en el cálculo.
2. Si hay atributos heredados:

- Un atributo heredado A.h debe calcularse antes de que aparezca el símbolo A.
- Un atributo sintetizado A.s no debe usarse antes de que aparezca el símbolo A.

1. Una acción semántica no debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.

Veamos un ejemplo de los pasos necesarios para traducir un trozo de código de un lenguaje de programación en el código correspondiente y equivalente en otro lenguaje de programación. Supongamos que queremos convertir la declaración de variables en C en declaración de variables en Pascal. Supongamos que sólo tenemos un tipo de dato predefinido, el tipo entero.

Por ejemplo, pasaremos de:

int x,y;

A esto:

var x,y:integer;

La gramática para la declaración de variables en C sería:

DeclVar ::= TipoVar ListaVar ***puntocoma***

TipoVar ::= ***entero***

ListaVar ::= ListaVar ***coma nombreVar***

ListaVar ::= ***nombreVar***

Como vamos a traducir, debemos tener un atributo en cada no terminal que se encargue de almacenar la traducción. En vez de incluir las reglas de producción y las acciones semánticas en una tabla vamos a ponerlo todo junto ya que es como se hace a la hora de implementarlo en la realidad.

Una manera de hacerlo sería:

```
DeclVar ::= TipoVar ListaVar puntocoma

{:

DeclVar.trad="var"+ListaVar.trad+":"+
```

```
TipoVar.trad+puntocoma.lexema;

:}

TipoVar ::= entero

{:

TipoVar.trad = "integer";

:}

ListaVar ::= ListaVar1 coma nombreVar

{:

ListaVar.trad = ListaVar1.trad + coma.lexema

+ nombreVar.lexema;

:}

ListaVar ::= nombreVar

{:

ListaVar.trad = nombreVar.lexema;

:}
```

Hemos situado todas las *acciones semánticas* al final de las reglas de producción. Veremos que es válido hacerlo así.

TipoVar.trad es sintetizado ya que es de un no terminal a la izquierda de una regla y se calcula a partir de atributos a la derecha de la regla.

ListaVar.trad es sintetizado ya que está también a la izquierda en las reglas en que se calcula.

Claramente, ***nombreVar***.*lexema*, ***coma***.*lexema* y ***puntocoma***.*lexema* son también sintetizados ya que pertenecen a terminales.

DeclVar.trad también es sintetizado porque el no terminal está a la izquierda de la regla y se calcula a partir de los de la derecha de la regla.

Por lo tanto, como todos los atributos son sintetizados, sabemos que se cumplen las propiedades de ETDS y además podemos poner todas las *acciones semánticas* al final

de las reglas.

7.5 Comprobaciones semánticas

Una vez que ya hemos dotado a una gramática de sus atributos y de las acciones semánticas oportunas. Podemos darle la capacidad de detectar errores semánticos.

La detección de errores semánticos depende mucho de las características del lenguaje del que vayamos a realizar el compilador y de la implementación elegida. Debido a esto, no vamos a explicar todas las posibles características de todos los lenguajes que existan (lo cual es imposible), sino que pondremos como ejemplo de lenguajes los lenguajes imperativos más comunes. Por ejemplo, Pascal, C o Java.

Estos lenguajes constan de declaración de variables, tipos predefinidos, tipos estructurados, subprogramas, etc.

Nos centraremos en comprobar la validez semántica a la hora de operar con variables de un determinado tipo, en si los parámetros (si los hay) que se utilizan en un subprograma son adecuados a la definición del subprograma, si los tipos devueltos por una función son los adecuados, si se utiliza una variable antes de ser inicializada, etc.

Aparte de estas comprobaciones, el programador tendrá que hacer otras dependiendo del lenguaje del que vaya a construir el compilador. En concreto, tendrá que impedir, o avisar del mal uso, de los elementos que existen en la definición del lenguaje.

Lo más común es realizar las comprobaciones semánticas, avisar de los errores que vayan ocurriendo e intentar que el análisis continúe para así procesar todo el programa y darle al programador una lista de todos los errores que aparezcan. Esto a veces no es posible ya que un error semántico puede provocar una larga lista de otros errores, por lo que en estos casos, quizás sea más adecuado parar el análisis y avisar del error para que sea modificado antes de continuar.

7.6 Ejercicios resueltos

Ejercicio 7.1

Construir un ETDS para traducir declaraciones de variables en Modula 2 a C. La gramática en Modula 2 sería:

S ::= ***VAR id :*** *T* ***;***

T ::= ***ARRAY [num .. num] OF*** *T*

T ::= ***REAL*** | ***INTEGER*** | ***CHAR***

Por ejemplo, se debería traducir:

VAR x:INTEGER;

VAR y:ARRAY[1..5] OF REAL;

VAR z:ARRAY[0..3] OF ARRAY[0..6] OF CHAR;

En:

int x;

float y[4];

char z[3][6];

Un aspecto importante, los elementos de los vectores en Modula 2 pueden comenzar con un índice arbitrario pero en C todos comienzan por el índice 0.

Vamos a suponer que el lexema del terminal ***num*** es un número natural.

Vamos a utilizar como lenguaje de apoyo Java.

Introduciremos las acciones semánticas oportunas:

```
S ::= VAR id : T ;
{:
if(T.array == null) {
S.trad = T.tipo + id.lexema + ";";
} else {
S.trad = T.tipo + id.lexema + T.array + ";";
}
:}
T ::= ARRAY [ num .. num1 ] OF T1
{:
T.tipo = T1.tipo;
int valor = (new Integer(num.lexema)).intValue;
int valor1 = (new Integer(num1.lexema)).intValue;
int indice = valor1 - valor;
if(T1.array == null) {
T.array = "[" + String.valueOf(indice) + "]";
} else {
T.array = "[" + String.valueOf(indice) + "]" + T1.array;
}
:}
T ::= REAL
{:
T.array = null;
```

```
T.tipo = "float";
:}
| INTEGER
{:
T.array = null;
T.tipo = "int";
:}
| CHAR
{:
T.array = null;
T.tipo = "char";
:}
```

Claramente, todos los atributos son sintetizados. Por lo que se trata de un ETDS.

Ejercicio 7.2

Realizar un esquema de traducción para comprobar los tipos en las expresiones de un sublenguaje del tipo Pascal. Las reglas para las expresiones son:

E ::= E1 **+** *E2* **;**

*E ::= E1 * E2 ;*

E ::= ***(E1)***

E ::= ***num*** **|** ***id***

Vamos a suponer que ***num*** es del tipo entero (*integer*) e ***id*** es el identificador de una variable que está en la tabla de símbolos (junto a la información de su tipo).

La solución sería:

```
E ::= E1 + E2 ;

{:

if(E1.tipo == E2.tipo) {

E.tipo = E1.tipo;

}

else ErrorSemantico("Tipos incompatibles en la suma");

:}
```

```
E ::= E1 * E2 ;

{:

if(E1.tipo == E2.tipo) {

E.tipo = E1.tipo;

} else

ErrorSemantico("Tipos incompatibles en el producto");

:}

E ::= ( E1 )

{:

E.tipo = E1.tipo;

:}

E ::= num

{:

E.tipo = "integer";

:}

E ::= id

{:

E.tipo = id.tipo;

:}
```

Ejercicio 7.3

Ahora, vamos a utilizar la información del ejercicio anterior y comprobaremos que las sentencias son válidas semánticamente hablando.

S ::= ***id :=*** *E*

S ::= ***if*** *E* ***then*** *S1*

*S ::= **while** E **do** S1*

*S ::= S1 **;** S2*

La solución sería:

```
S ::= id := E
{:
if(id.tipo == E.tipo) {
S.tipo = null;
}
else ErrorSemantico(Tipos incompatibles en la asignación”);
:}
S ::= if E then S1
{:
if(E.tipo == “boolean”) {
S1.tipo = S.tipo;
}
else ErrorSemantico(“La condicion del if no es del tipo logico”);
:}
S ::= while E do S1
{:
if(E.tipo == “boolean”) {
S1.tipo = S.tipo;
}
else ErrorSemantico(“La condicion del while no es del tipo logico”);
```

```
:}

S ::= S1 ; S2

{:

if(S1.tipo == null && S2.tipo == null) {

S.tipo = null;

} else ErrorSEmantico(“Sentencias con un tipo”);

:}
```

CAPÍTULO 8

GENERACIÓN DE CÓDIGO INTERMEDIO Y FINAL

8.1 Introducción

En esta fase de la creación de un compilador se genera el código final que será o bien ejecutado por la máquina o bien reutilizado si se trata de un traductor. Pero antes de obtener el código final, y no necesariamente, se suele crear una versión de más alto nivel que sirve para dar la posibilidad de que el mismo código pueda servir para distintas máquinas.

Es decir, se suele crear un código intermedio entre el código del lenguaje que queremos compilar y el lenguaje máquina de la máquina de destino. Esto se hace así para que con un pequeño esfuerzo adicional sea posible crear un compilador que pueda ser utilizado para máquinas diferentes.

El código intermedio es un código que no concreta las características de la máquina final pero que organiza el código objeto (el código del programa a compilar) para que sea el mismo o casi el mismo, independientemente del lenguaje.

Figura 8.1

Dos características que hacen beneficioso crear un lenguaje intermedio entre el lenguaje fuente y el ensamblador son:

1. Permite ahorrarse esfuerzo a la hora de crear varios compiladores para la misma máquina, ya que cada lenguaje a compilar crearía el mismo código intermedio.
2. Se ahorra esfuerzo a la hora de crear un compilador para varias máquinas diferentes, ya que el mismo código intermedio sirve para todas las

máquinas.

Vistas las ventajas de la creación de un código intermedio, tendremos que definir cómo será dicho lenguaje para que se adapte a lenguajes diferentes de alto nivel y lenguajes distintos de bajo nivel. En la realidad, es posible que haya que hacer pequeños cambios en el código intermedio para diferentes máquinas, pero esos cambios suelen ser mínimos.

La generación de código intermedio, en adelante CI, es una tarea íntimamente ligada a la comprobación de tipos, símbolos y al análisis semántico, por lo que se suele hacer paralelamente. Por esto, todo el proceso de análisis semántico y la generación de CI están guiados por la sintaxis.

Si en vez de hacer un compilador lo que queremos es hacer un traductor, no suele ser necesario generar código intermedio y luego código final, sino que directamente generamos código final en el lenguaje de destino de la traducción.

8.2 Tipos de código intermedio

Aunque hay más tipos de CI, los más utilizados son dos: el código de tres direcciones y el código de máquina virtual de pila.

Hay veces en que la generación de código intermedio es la última fase del proceso de creación de un compilador. El resultado es un programa en CI que luego será interpretado o compilado en una máquina final.

Aunque este sistema en que se interpretaba un lenguaje en vez de compilarlo quedó en desuso, con la entrada de Java se ha vuelto a utilizar la interpretación. Un programa compilado en Java no consiste en código máquina sino los llamados *bytecodes*, es decir, código intermedio. Este código intermedio vale para casi cualquier máquina y es un programa especial, dependiente de la máquina concreta, la máquina virtual de Java JVM, el que se encarga de interpretar los *bytecodes*.

8.2.1 Código de tres direcciones

Se trata de un formato de representación en el que todas las operaciones se representan por cuádruplas, o registros con cuatro campos, en las que la primera de ellas es el tipo de operación, las dos siguientes son dos direcciones y la última una dirección de resultado.

Este tipo de representación tiene su origen en la posibilidad de representar las expresiones aritméticas por medio de una operación, dos operandos y un resultado. Aunque hay otras manipulaciones de código que no son expresiones, se pueden acoplar a este formato.

Se habla de direcciones porque son las direcciones de memoria donde finalmente se guarda la información en una máquina real. Por ejemplo, si tenemos que una variable x guarda su valor en la dirección 9000 y otra variable y la guarda en la dirección 9002 y

queremos hacer la asignación *x:=x+y* en Pascal, el código ensamblador en un lenguaje estándar sería algo así:

```
ADD     (9000)      (9002)      temporal
MOVE    (temporal)              9000
```

Donde (*d*) hace referencia al contenido de la dirección de memoria *d*.

Vemos que aparece un nuevo concepto, el de temporal. Una temporal es una dirección de memoria que nos sirve para guardar resultados parciales en las operaciones. Generalmente, cada operación dejará su resultado en una dirección temporal y de ahí se extraerá para su utilización por la siguiente operación. Aunque podríamos haber no utilizado una temporal en este caso y haber puesto directamente la dirección de la variable x (la 9000). Pero esto no suele ser posible siempre ya que a veces hay operaciones intermedias antes de asignar el valor a una variable. Por ejemplo: x:=x+y-2.

```
ADD     (9000)      (9002)      temporal1
SUB     (temporal1) 2           temporal2
MOVE    (temporal2)             9000
```

Vemos que aunque se le llama código de tres direcciones, en realidad hay casos en que no se trata de direcciones sino de otros valores cuya semántica dependerá del empleo que le demos en cada tipo de instrucción (por ejemplo, el segundo operando de la instrucción SUB no es una dirección sino un valor inmediato).

8.2.2 Código de máquina virtual de pila

Consiste en utilizar una pila para guardar los resultados intermedios en la evaluación de expresiones y para otras funciones. Por ejemplo, para el caso anterior en que queríamos hacer x:=x+y-2, las instrucciones que deberíamos realizar serían:

Apilar(Contenido(9000)) → Se apila el contenido de la dirección 9000

Apilar(Contenido(9001)) → Se apila el contenido de la dirección 9001

Sumar → Se desapilan dos valores de la pila y se suman. El resultado se apila

Apilar(2) → Se apila el valor 2

Restar → Se desapilan dos valores de la pila y se restan. El resultado de apila

Asignar(9000) → Se desapila un valor y se guarda en la dirección 9000

Aunque el funcionamiento de este tipo de generación de código intermedio es tan adecuado como el sistema de tres direcciones, se puede comprobar con la experiencia

que es más difícil pasar del CI al código final en sistemas de pila. Por lo tanto, en los casos prácticos vamos a utilizar código de tres direcciones.

8.2.3 Operadores sobrecargados

Generalmente, cuando se quiere obtener el CI de una operación entre dos valores, se debe analizar que sean del mismo tipo y así generar su CI. Pero hay lenguajes en que se permite que se operen valores de distinto tipo. En este caso, aunque no se generará ningún error semántico si se intenta operar entre dos valores de tipos diferentes, se deberá decidir si el resultado es de un tipo u otro. De todas formas, el CI generado deberá hacer las conversiones de tipos adecuadas.

8.3 Código intermedio para expresiones

Vamos a utilizar en adelante el sistema de tres direcciones porque es el más utilizado en la actualidad. Iremos creando las diferentes instrucciones de CI conforme las vayamos necesitando. En cuanto a las direcciones, dejaremos su tratamiento para más adelante.

Las expresiones constan de uno o más valores (pueden ser números, variables, etc.) que se combinan mediante operadores para obtener como resultado otro valor. Dependiendo del lenguaje a compilar, deberán ser compatibles en tipos o serán convertidos. Esta comprobación de tipos se hace en el análisis semántico (al igual que la inserción y búsqueda de variables, constantes y subprogramas en la tabla de símbolos).

Pero ya se dijo anteriormente que el análisis semántico se hace paralelamente a la generación CI, por lo que en la realidad es simultáneo todo el proceso.

Generalmente, conforme se va generando CI, este se va escribiendo en un archivo de código intermedio y al final del proceso se procesará este archivo para generar el código final. Otra posibilidad es mantener una estructura de datos en memoria para guardar el CI y luego procesarla al final. Ambas técnicas son válidas.

Vamos a poner un ejemplo. Supongamos que tenemos un lenguaje de tipo Pascal y queremos ver el CI que genera la suma de dos operandos. Vamos a suponer que hay dos posibles tipos de datos para poder ser sumados, los enteros y los reales. Supondremos también que no se permite sumar tipos diferentes. Se permite el uso de variables.

Vamos a utilizar lo que en secciones anteriores llamamos direcciones temporales. Son direcciones donde se guardan valores parciales. Debemos tener un contador de direcciones temporales que iremos incrementando conforme vayamos necesitando nuevas temporales. La localización de las direcciones temporales debe cuidarse para que no se utilicen valores ya utilizados por variables o por el código. Vamos a suponer en este ejemplo que la primera dirección temporal va a estar situada en la dirección de memoria 10000. Por lo tanto, el contador valdrá 10000.

Suponemos que ya se ha procesado el código correspondiente a la declaración de

las variables y están ya guardadas en la tabla de símbolos. Las posibles acciones semánticas que tendríamos que realizar para generar el CI de la suma serían:

```
E ::= E1 + T

{:

if(E1.tipo == T.tipo) {

E.tipo = E1.tipo;

E.dir = temporal;

if(E.tipo == REAL) {

insertarCI("SUMAR_REAL",E1.dir,T.dir, E.dir);

} else {

insertarCI("SUMAR_ENTERO",E1.dir,T.dir, E.dir);

}

temporal++;

} else {

ErrorSemantico("Tipos diferentes");

}

:}

E ::= T

{:

E.tipo = T.tipo;

E.dir = T.dir;

:}

T ::= entero

{:
```

```
T.tipo = ENTERO;

T.dir = temporal;

insertarCI("CARGAR_ENTERO",entero.lexema,null,T.dir);

temporal++;

:}

T ::= real

{:

T.tipo = REAL;

T.dir = temporal;

insertarCI("CARGAR_REAL",real.lexema,null,T.dir);

temporal++;

:}

T ::= id

{:

if(existe_en_tabla(id.lexema) == false) {

ErrorSemantico("La variable no esta declarada");

} else {

T.tipo = id.tipo;

T.dir = direccion_en_tabla(id.lexema);

}

:}
```

Vemos que hemos creado dos tipos de instrucciones de código intermedio, una que se encarga de cargar valores enteros o reales en una dirección de memoria, y otra que se encarga de sumar los contenidos reales o enteros de dos direcciones de memoria.

Una de las instrucciones no necesita nada más que un operando (la de cargar una

dirección con un valor), por lo que el otro operando será nulo.

Se ha comprobado que los tipos coinciden y que la variable está en la tabla de símbolos. Al final, se guarda en el atributo *E.dir* la dirección donde se contendrá el resultado.

Veremos qué código final se generaría para la máquina ENS2001 (esta máquina es una máquina virtual que procesa ensamblador. Su funcionamiento se estudiará en un apéndice).

Si tuviéramos esta expresión por ejemplo:

x := x +1;

Vamos a suponer que la variable *x* se guarda en la dirección 9000 y que a partir de esta dirección se pueden utilizar las temporales. Es decir, la variable global *temporal* vale 9001.

La dirección de la variable *x* se incluye en su entrada de la tabla de símbolos al ser declarada la variable.

Se generarían estas instrucciones de CI:

"CARGAR_ENTERO", "1", null, "9001"

"SUMAR_ENTERO", "9000", "9001", "9002"

La primera de ellas, en el código ensamblador de ENS2001, podría ser (habría varias maneras de hacerlo):

MOVE #1, /9001

La segunda sería:

ADD /9001, /9002

MOVE .A, /9003

Para la suma, se hace necesaria la utilización del registro acumulador que es el que guarda los resultados de las operaciones aritméticas.

Se aprecia que cuando utilizamos valores concretos en vez de variables, un paso previo debe ser generar el CI para guardar el valor en una dirección temporal. Para las variables, utilizamos directamente su dirección asignada en la tabla de símbolos.

Tal y como se ha hecho con la suma, se hace con los demás operadores aritméticos.

Un caso especial es el de la división ya que en algunos lenguajes está sobrecargada. En Pascal, por ejemplo, se utiliza el operador "*div*" para dividir enteros y el operador "/" se utiliza para dividir tanto enteros como reales aunque el resultado se convertirá a real. Estas características hay que tenerlas en cuenta a la hora del análisis semántico y a la hora de generar el CI correspondiente.

Para los operadores relaciones, hay algunas diferencias dependiendo del lenguaje. Por ejemplo, el resultado de una operación de relación en C sería un 1 si es verdadero y un 0 si es falso. En Pascal sería *true* o *false*.

Independientemente de cómo sea el compilador, el CI generado debe tener en cuenta que sólo se opera con números, por lo que lo más adecuado es convertir los valores booleanos a números (al estilo de C) en los condicionales. Según esto, *true* estaría representado por un 1 y *false* por un 0.

Hay varios operadores relacionales (mayor, menor, igual, etc.) que dependiendo del lenguaje tendrán una sintaxis diferente, aunque su significado es el mismo. Por ejemplo, sea en Pascal la evaluación de la condición *menor* en un bucle *while*.

Bucle ::= ***while*** *Condicion* ***do*** *BloqueSentencias* ***puntocoma***

Condicion ::= Expresión1 **<** *Expresion2*

Vamos a fijarnos sólo en que una condición es una expresión en la que se relacionan dos expresiones. Por lo que una condición tiene la misma estructura que una expresión. Se insertarían estas comprobaciones semánticas y este CI:

```
Condicion ::= Expresion1 < Expresion2

{:

if(Expresion1.tipo == Expresion2.tipo) {

Condicion.dir = temporal;

Condicion.tipo = BOOLEANO;

insertarCI("MENOR", Expresion1.dir,

Expresion2.dir, Condicion.dir);

temporal++;

} else {

ErrorSemantico("Tipos incompatibles");

:}
```

Hemos supuesto que sólo se pueden comparar elementos con tipos iguales (hay lenguajes en que esta condición no se exige).

Para los demás operadores relacionales, el CI sería parecido.

Lo que sí hay que destacar es que al pasar del CI al código final, una condición verdadera estaría representada por un 1 y una falsa por un 0.

En cuanto a los operadores lógicos, el funcionamiento es similar a los relacionales.

8.4 Código intermedio para asignaciones

Esta instrucción consiste en asignar el valor de la expresión de la derecha de la asignación a la variable a la izquierda de la misma.

Habría que realizar algunas comprobaciones semánticas, dependiendo del lenguaje. Por ejemplo, habría que comprobar que los tipos de la parte derecha y de la variable de la parte izquierda son iguales (o compatibles) o realizar la conversión de tipos adecuada, dependiendo de si el lenguaje lo permite. Otra comprobación sería ver si la variable de la parte izquierda está en la tabla de símbolos.

Un ejemplo es la regla de la asignación en C:

Asignación ::= ***id igual*** *Expresion* ***puntocoma***

Las acciones semánticas podrían ser:

```
Asignacion ::= id igual Expresión puntocoma

{:

if(existe_en_tabla(id.lexema) == false) {

ErrorSemantico("La variable no esta declarada");

} else {

if(get_tipo_en_tabla(id.lexema) == Expresion.tipo) {

insertarCI("COPIAR", Expresión.dir, null, direccion_en_tabla(id.lexema));

} else {

ErrorSemantico("Tipos diferentes");

}

}

:}
```

Veamos qué código en ENS2001 se generaría. Supongamos que *Expresión.dir* =

9005 y *direccion_en_tabla(id.lexema)* = 9000, entonces sería:

MOVE /9005, /9000

En todos estos casos, suponemos que las variables de todos los tipos ocupan una sola dirección de memoria (en realidad, hay tipos que ocupan más espacio que otros, pero para ENS2001 todos ocupan una dirección de memoria).

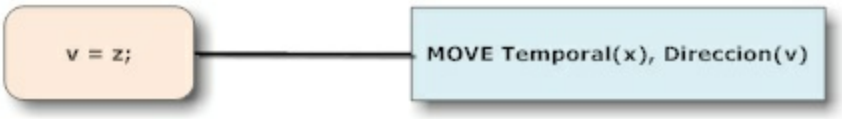


***Figura 8.2**. Esquema de una asignación*

8.5 Sentencias de entrada y salida

Estas sentencias se tratan de manera diferente para máquinas diferentes, por lo que el CI dependería de la máquina final.

Por lo general, se trataría de comprobar que en la sentencia de entrada la variable donde se guarda la entrada existe y es del tipo adecuado y para las sentencias de salida los datos de salida son los adecuados.

8.6 Sentencia condicional

Esta sentencia es parecida en todos los lenguajes imperativos, por lo que pondremos como ejemplo C. En C, una sentencia de este tipo sería:

SentenciaIf ::= ***if parentIzq*** *Condicion* ***parentDer llaveIzq*** *Bloque* ***llaveDer*** *SentenciaElse*

SentenciaElse ::= | ***else llaveIzq*** *Bloque* ***llaveDer***

C admite en la condición cualquier tipo (si el valor es cero es falso y si no, es verdadero), pero Pascal admite sólo tipos booleanos, por lo que dependiendo del lenguaje habría que comprobar los tipos en la condición.

Pero lo más importante es ver que si la condición es verdadera se ejecuta el primer bloque, y si es falsa pueden ocurrir dos cosas: si existe la sentencia ***else*** se ejecuta su bloque, y si no existe se termina sin hacer nada.

En ensamblador, para acceder a un punto de un programa se suele utilizar una etiqueta y un salto a ella. Esa es la técnica que vamos a utilizar para generar el CI de esta instrucción.

Insertamos las etiquetas necesarias en CI y la etiqueta del salto. Como se puede

saltar al segundo bloque o al final de la instrucción, habrá que insertar al menos dos etiquetas, una al comienzo de la sentencia ***else*** (si no existe, se pone al final) y otra al final. En C sería algo así:

```
SentenciaIf ::= if parentIzq Condicion parentDer llaveIzq
{:
insertarCI("SALTO_SI_FALSO",Condicion.dir,
"ETIQUETA_ELSE", null);
:}
Bloque llaveDer
{:
insertarCI("SALTO", "ETIQUETA_FIN", null, null);
insertarCI("ETIQUETA", "ETIQUETA_ELSE", null, null);
:}
SentenciaElse
{:
insertarCI("ETIQUETA", "ETIQUETA_FIN", null, null);
:}
SentenciaElse ::= | else llaveIzq Bloque llaveDer
```

En este código hay algunos aspectos a tener en cuenta. La instrucción en CI de saltar si es falso, debe generar el código necesario para comprobar si el contenido de la dirección de la condición es un 1 (verdadero) o un 0 (falso) para no saltar o saltar a la etiqueta señalada ("ETIQUETA_ELSE") que es la que pondremos al principio del bloque de la sentencia ***else***. Debemos insertar un salto incondicional al final del primer bloque ya que si la condición es verdadera, sólo se ejecuta el primer bloque. El salto incondicional debe ser al final de toda la instrucción, para lo cual pondremos la etiqueta correspondiente al final de la instrucción.

Otro aspecto a tener en cuenta es que habrá más sentencias de este tipo en todo el programa y, por lo tanto, las etiquetas estarán repetidas. Para evitar esto, se debe tener un

contador global de este tipo de etiquetas de manera que cada instrucción aumente el contador y se añada este número en la propia etiqueta para hacerla diferente a las otras. Por ejemplo, estas serían las etiquetas generadas para dos sentencias condicionales seguidas:

"SALTO_SI_FALSO", Condicion1.dir, "ETIQUETA_ELSE_1", null

"SALTO", "ETIQUETA_FIN_1", null, null

"ETIQUETA", "ETIQUETA_ELSE_1", null, null

"ETIQUETA", "ETIQUETA_FIN_1", null, null

"SALTO_SI_FALSO", Condicion2.dir, "ETIQUETA_ELSE_2", null

"SALTO", "ETIQUETA_FIN_2", null, null

"ETIQUETA", "ETIQUETA_ELSE_2", null, null

"ETIQUETA", "ETIQUETA_FIN_2", null, null

Vemos que en este caso el formato del CI ya no es de una instrucción y tres direcciones, sino que se utiliza otro formato.

En ENS2001, la instrucción de salto condicional ("SALTO_SI_FALSO" , 9000, "ETIQUETA_ELSE", null) sería:

CMP /9000, #0

BZ /ETIQUETA_ELSE

Par la instrucción ("SALTO", "ETIQUETA_FIN", null, null):

BR /ETIQUETA_FIN

Para la instrucción ("ETIQUETA", "ETIQUETA_ELSE", null, null):

ETIQUETA_ELSE: NOP

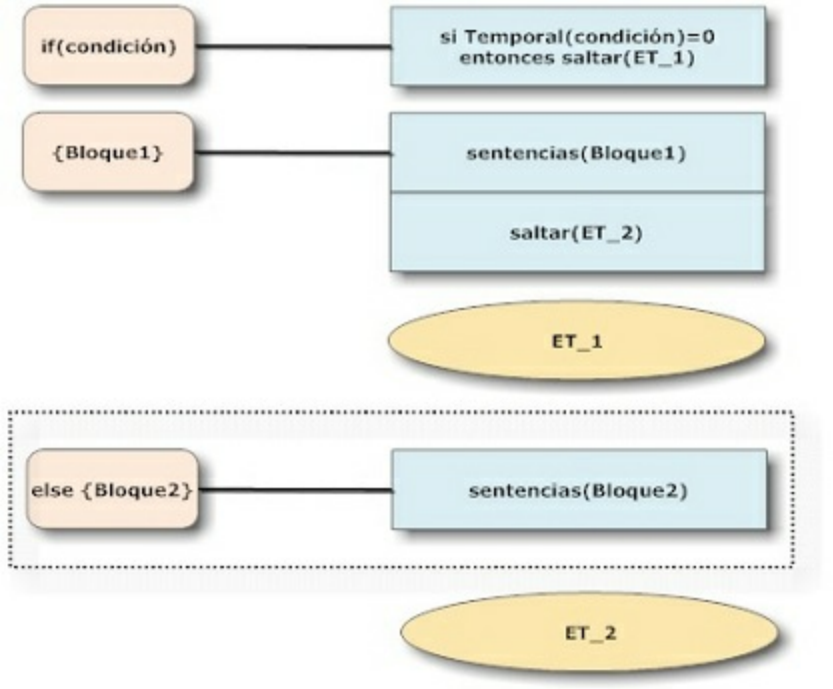


Figura 8.3**. Esquema de una sentencia **if-then-else

Otra consideración que hay que tener en cuenta es que hay sentencias de este tipo anidadas. Cuando se ponga la etiqueta que señale el ***else*** o el fin de la instrucción debe corresponder a su bloque. Para ello, es necesario mantener una estructura global de tipo pila para meter y sacar etiquetas según se entre o se salga de un bloque condicional.

Por ejemplo, si tenemos este código en C:

```
if(condicion1) {

if(condicion2) {

instrucciones1

} else {

instrucciones2

}

} else {

instrucciones3

}
```

Supongamos que el contador de condicionales vale 6. Al entrar en el primer ***if*** aumentamos el contador, que vale ahora 7. Al evaluar la condición 1 ponemos un salto a la etiqueta "ETIQUETA_ELSE_7". Ahora entramos en el segundo ***if,*** aumentamos el contador, que ahora vale 8. Al evaluar la condición 2 ponemos un salto a

“ETIQUETA_ELSE_8”. Tras generar el código de las instrucciones 1 ponemos un salto al final del código de su bloque. Como el contador vale 8, el salto sería a la etiqueta 8, es decir, a “ETIQUETA_FIN_8”. Después, ponemos la etiqueta “ETIQUETA_ELSE_8”. Ahora entramos en su ***else*** y al final ponemos la etiqueta del fin del bloque, es decir, “ETIQUETA_FIN_8”. Hasta ahora va todo bien. Pero ahora hemos finalizado el código del primer ***if*** y debemos insertar el salto al final, pero como el contador vale 8, el salto que insertaríamos es a la etiqueta “ETIQUETA_FIN_8” en vez de a “ETIQUETA_FIN_7” que es el número que le dimos al comienzo de su bloque. Para las demás referencias también las hacemos a 8 en vez de a 7. Por lo tanto, tenemos la necesidad de mantener dos estructuras, una que nos indique el número de sentencias ***if*** que van apareciendo, y que puede ser un contador que aumente siempre al entrar en un ***if*** y otra que tenga la forma de pila y que señale en cada momento el código de la instrucción ***if*** en la que estamos. En esta estructura de pila se irán metiendo el contador de ***if*** al entrar en un ***if*** y se irá sacando al salir. Entre medias, las etiquetas que se pongan llevarán el número del contador que haya en la cima de la pila.

Por ejemplo, para el programa anterior. Supongamos que inicialmente la pila está vacía y el contador de sentencias ***if*** vale 0.

Al leer el primer ***if***, aumentamos el contador, que vale ahora 1 y metemos un 1 en la pila. Después de evaluar su condición ponemos un salto condicional a la etiqueta “ETIQUETA_ELSE_1” donde 1 es el contenido de la cima de la pila. Entramos ahora en el segundo ***if*** y aumentamos el contador, que vale ahora 2. Metemos un 2 en la pila. Ponemos un salto condicional a “ETIQUETA_ELSE_2” donde 2 es el contenido de la cima de la pila.

Ahora generamos el CI de las instrucciones 1 y al final ponemos un salto a la etiqueta “ETIQUETA_FIN_2” donde 2 es la cima de la pila. Después, ponemos la etiqueta que señala el comienzo del ***else*** y que es “ETIQUETA_ELSE_2” donde 2 es la cima de la pila. Luego, generamos el CI de las instrucciones 2 y al final ponemos la etiqueta “ETIQUETA_FIN_2” donde 2 es la cima de la pila. Al salir del bloque correspondiente al segundo ***if***, desapilamos el elemento de la cima y, por tanto, en la pila sólo tenemos un 1. De ahora en adelante, las etiquetas y saltos que pongamos se referirán al primer if, tal y como deseábamos. El proceso continúa hasta llegar al final del primer ***if*** y desapilar el 1. De manera que el contador de ***if*** se queda con un valor de 2 y la pila está vacía. De esta manera, cuando nos encontremos con otro ***if***, su contador valdrá 3 e insertaremos en la pila un 3.

8.7 Iteración tipo while

Esta sentencia también es parecida en todos los lenguajes imperativos. Guarda mucha similitud con la sentencia condicional salvo estas dos diferencias:

1. No hay bloque ***else.***

2. Al final del bloque de sentencias hay un salto al comienzo de la condición.

Debido a estas dos diferencias, las operaciones a realizar están situadas de distinta forma. Se pone una etiqueta al principio de la instrucción. Si la condición es falsa se salta a otra etiqueta que se pone al final de la instrucción. Pero si la condición es verdadera se ejecuta el bloque de código. Al final del bloque se debe poner un salto incondicional a la etiqueta que señala el principio de la instrucción.

Lo mismo que ocurría con las sentencias ***if-then-else***, como habrá más de una de ese tipo, las etiquetas se tendrán que ir numerando para que no se repitan. Y si se permite el anidamiento, habrá que implementar una pila global.

Por ejemplo, en C sería algo así:

```
SentenciaWhile ::= {:

insertarCI("ETIQUETA", "ETIQUETA_INICIO", null, null);

:}

while parentIzq Condicion parentDer llaveIzq

{:

insertarCI("SALTO_SI_FALSO", Condicion.dir,

"ETIQUETA_FIN", null);

:}

Bloque llaveDer

{:

insertarCI("SALTO", "ETIQUETA_INICIO", null, null);

insertarCI("ETIQUETA","ETIQUETA_FIN", null, null);

:}
```

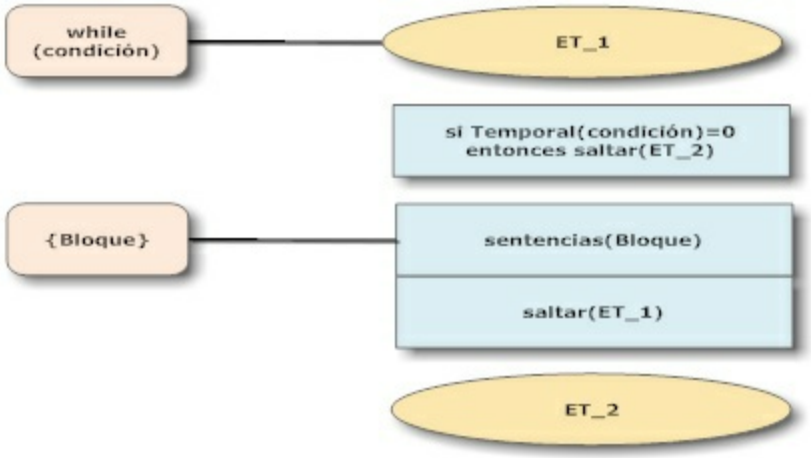


Figura 8.4**. Esquema de una sentencia **while

Generalmente, se permite salir de la iteración con una sentencia de ruptura (por ejemplo, ***break***). Para implementar esta sentencia, no hay más que provocar un salto a la etiqueta del final de la iteración.

Hay lenguajes que permiten la declaración de variables locales dentro de los ámbitos de este tipo de instrucciones (y de otras). Habrá que insertar estas variables locales en la tabla de símbolos con un nuevo ámbito (como si se tratase del ámbito de un subprograma). Al salir del bloque de estas instrucciones, habrá que eliminar estas variables locales de todo el ámbito.

8.8 Iteración tipo repeat-until y do-while

Dependiendo de unos lenguajes u otros, se utiliza una de las sentencias indicadas. Pero la característica común es que la condición de la iteración se evalúa al final del bloque, es decir que el bloque se ejecuta al menos una vez.

Habrá que numerar las etiquetas de este tipo para que no se repitan, tal y como se hizo para las sentencias anteriores. Si se permite el anidamiento, habrá que implementar una pila global.

Su funcionamiento es muy parecido a ***while***. Para C sería algo así:

```
SentenciaDo ::= {:
insertarCI("ETIQUETA", "ETIQUETA_INICIO", null, null);
:}
do llaveIzq Bloque llaveDer
while parentIzq Condicion parentDer
{:
insertarCI("SALTO_SI_FALSO", Condicion.dir,
"ETIQUETA_FIN", null);
insertarCI("SALTO", "ETIQUETA_INICIO", null, null);
```

```
insertarCI("ETIQUETA", "ETIQUETA_FIN", null, null);
:}
```

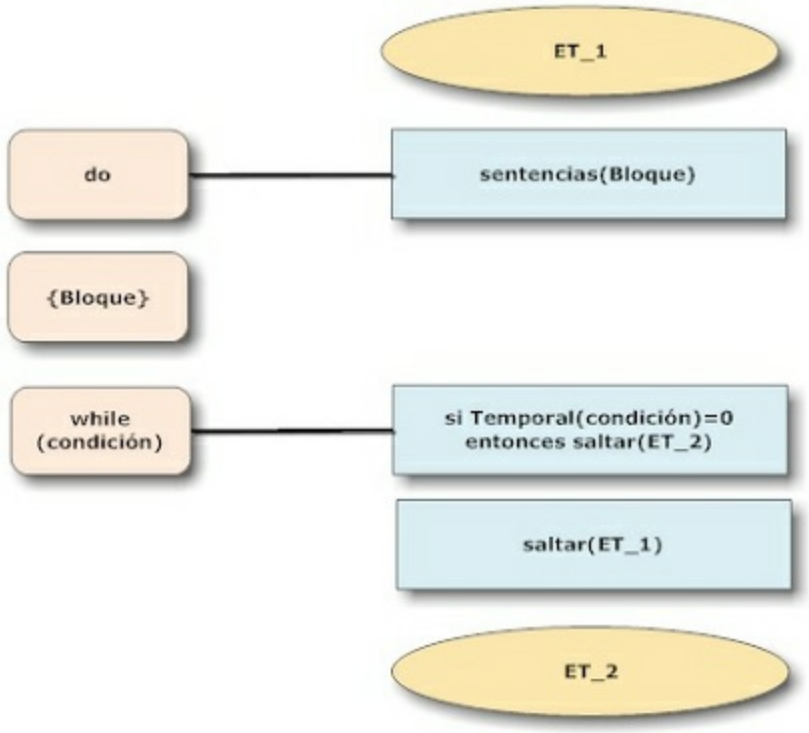


Figura 8.5**. Esquema de una sentencia **do-while** o **repeat-until

Normalmente, se permite salir de la iteración con una sentencia de ruptura (por ejemplo ***break***). Para implementar esta sentencia, no hay más que provocar un salto a la etiqueta del final de la iteración.

8.9 Iteración tipo for

Este tipo de iteración, aunque tiene similitudes en los lenguajes más comunes, también hay algunas diferencias. Por ejemplo, en Pascal sólo se permite utilizar variables de tipo entero para el contador de la iteración. Pero a la hora de generar el CI los lenguajes imperativos más comunes funcionan igual. Por ejemplo, para C. vamos a suponer que se utiliza una variable ya inicializada:

```
SentenciaFor ::= for parentIzq Asignacion puntocoma
{:
insertarCI("ETIQUETA", "ETIQUETA_INICIO", null, null);
:}
Condicion puntocoma Incremento parentDer
{:
```

```
insertarCI("SALTO_SI_FALSO", Condicion.dir,

"ETIQUETA_FIN", null);

:}

llaveIzq Bloque llaveDer

{:

insertarCI("SALTO", "ETIQUETA_INICIO", null, null);

insertarCI("ETIQUETA", ETIQUETA_FIN", null, null);

:}
```

Hay que hacer un par de aclaraciones respecto al código anterior. Una es que no se incluye aquí el CI generado por la asignación porque ya se ha incluido en su regla de producción correspondiente. Tampoco se incluye el del incremento (o decremento) ya que se trata de una expresión que crea su propio CI en sus reglas de producción.

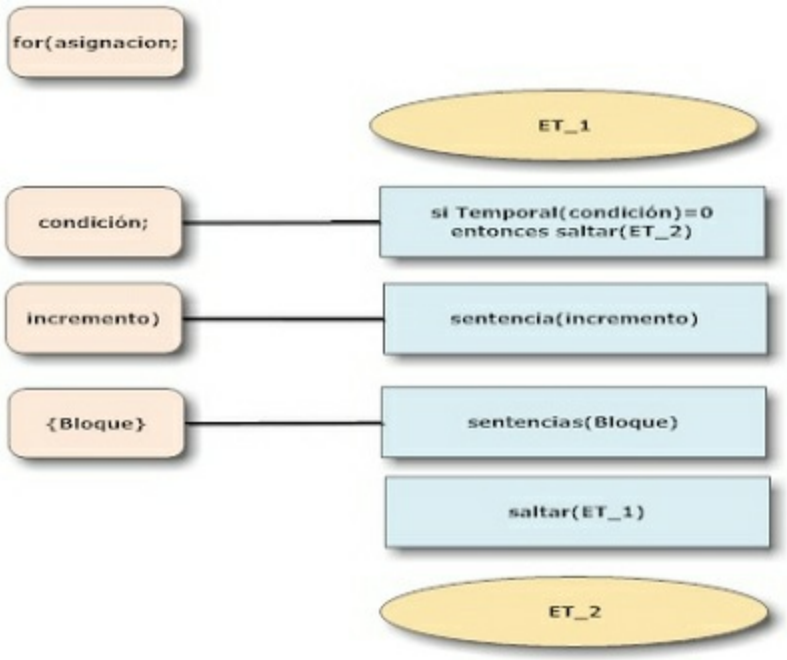


Figura 8.6. *Esquema de una sentencia* ***for***

Tal y como se hizo en sentencias anteriores, como habrá varias sentencias ***for***, habrá que numerarlas para que no se repitan las etiquetas. Si se permite el anidamiento, habrá que implementar una pila global.

Generalmente, se permite salir de la iteración con una sentencia de ruptura (por ejemplo, ***break***). Para implementar esta sentencia, no hay más que provocar un salto a la etiqueta del final de la iteración.

8.10 La selección

Esta estructura condicional es muy parecida en casi todos los lenguajes imperativos. Por ejemplo en C sería algo así:

```
switch(E) {

case 1: /*instrucciones*/

case 2: /*instruccuines*/

case 3: /* instrucciones*/

break;

case 4: /*instrucciones*/

default: /*instrucciones por defecto*/

}
```

Lo primero que habría que realizar es hacer las comprobaciones semánticas (por ejemplo, comprobar que E es del tipo adecuado y acorde con los valores de ***case***). Después, se deben incluir las acciones semánticas oportunas.

Hay varias maneras de atajar este tipo de sentencia, una de ellas es convertirla a varias sentencias ***if-then-else***. Otra manera es crear una tabla de pares *valor-etiqueta* y generar código que realice la búsqueda y que lance el CI correspondiente. Pero es el programador el que decide cómo implementar estas acciones. Hay que saber que se utilizan las mismas instrucciones de CI.

Antes de atajar la generación de CI para tipos estructurados, debemos tener en cuenta el reciclado de direcciones temporales. Aunque esta tarea no es obligatoria, sí es aconsejable para reducir el tamaño del entorno de trabajo de nuestros programas.

Lo más utilizado es reiniciar el contador de temporales tras cada instrucción.

8.11 Código intermedio para vectores

Ya se vio en capítulos anteriores la búsqueda de los elementos de un vector y las comprobaciones que había que hacer para ver si el acceso era adecuado (en cuanto al tipo y en cuanto a si el elemento al que se accedía estaba en el rango del vector). Ahora generaremos las acciones semánticas para acceder a los elementos.

Hay dos pasos para obtener la dirección de un elemento de un vector, el primero es calcular el orden en que está en la lista de elementos del vector y el segundo es acceder a la dirección de ese elemento.

Como el acceso a la posición de un elemento de un vector no suele estar

concretado en tiempo de compilación, no se puede comprobar directamente si tal elemento está o se intenta acceder a un elemento fuera del rango del vector, por lo que esta comprobación corre a cargo del usuario (el que utilice el compilador para compilar sus programas). También se puede comprobar el acceso dentro del rango del vector generando el código final correspondiente para esta tarea, aunque es complejo hacerlo (y, por tanto, dejar que el código final avise de accesos ilegales) y siempre se haría en tiempo de ejecución y no de compilación.

Una vez que nos desentendemos de comprobar si el acceso a un elemento está dentro del rango (supondremos que es así), debemos generar el CI para apuntar a dicho elemento. Internamente, los elementos de un vector estarán ordenados desde 0 en adelante (si no es así, haremos las conversiones oportunas). Por ejemplo, en C los elementos van desde el 0 hasta el rango del vector menos 1. Pero en Pascal se permite que el primer elemento no sea el 0, por lo que como internamente va a estar ordenado desde el 0, haremos la conversión adecuada. Por ejemplo, si tenemos esta declaración en Pascal:

```
program p;

type vector=array[4..9] of integer;

var v:vector;

begin

v[4]:=10;

end.
```

Aunque los elementos del vector son desde el 4 al 9, internamente están ordenados desde el 0 al 5. Es decir, el 4 es internamente el 0. Por lo tanto, cuando asignemos el valor 10 al elemento 4 estaremos asignándolo internamente al elemento 0.

En memoria, los elementos del vector están guardados uno detrás de otro y en la tabla de símbolos se indica la dirección del primero (del elemento 0). Por lo tanto, para acceder al elemento i tendremos que calcular su posición respecto al primer elemento, es decir, al elemento 0.

En C:

```
int[n] v;
```

Direccion(v[0]) = Direccion(v)

Direccion(v[i]) = Direccion(v) + i

Donde i va de 0 a n-1

En Pascal:

```
type vector=array[a0..an] of integer;
```

```
var v:vector;
```

Direccion(v[a0]) = Direccion(v)

Direccion(v[ai]) = Direccion(v) + ai – a0

Como el acceso a un elemento de un vector irá referido por una expresión, utilizaremos la dirección donde guarda el valor la expresión como valor que apunta al elemento del vector (en Pascal tendríamos que hacer las operaciones antes indicadas para obtener la verdadera dirección del elemento).

Supongamos que queremos utilizar un elemento de un vector en una expresión. El acceso a ese elemento del vector lo dará el resultado de otra expresión (la dirección de la expresión guarda el valor del elemento a acceder). En C sería algo así (no introduciremos comprobaciones semánticas para no complicar el código):

```
Expresion ::= id corcheteIzq Expresion1 corcheteDer puntocoma

{:

insertarCI("INCREMENTAR", id.dir, Expresión1.dir,

temporal);

Expresion.dir = temporal;

temporal++;

:}
```

Aparece una nueva instrucción de CI. Si suponemos que la dirección a partir de la que está el vector guardado en memoria (la dirección del primer elemento del vector está en su entrada de la tabla de símbolos) es la 9000 y la dirección donde está el índice es en la dirección 10000 (viene dada por Expresion1.dir) y el contador de temporales vale 10001. Su código en ENS2001 podría ser:

```
ADD #9000, /10000

MOVE .A, /10001
```

Podemos apreciar que en ningún caso estamos reciclando las temporales. En un compilador real habría que reciclar las temporales para ahorrar espacio, pero esta tarea es delicada y por eso no lo hacemos aquí.

En Pascal, el código para el acceso a un elemento de un vector sería:

```
Expresion ::= id corcheteIzq Expresion1 corcheteDer

puntocoma
```

```
{:

insertarCI("INCREMENTAR", id.dir - id.min,

Expresion1.dir, temporal);

Expresion.dir = temporal;

temporal++;

:}
```

Aunque en C no es necesario acceder al límite inferior del vector (ya que es 0), es necesario que esté en la tabla de símbolos al menos el límite superior para guardar espacio en memoria para todos los elementos (en Pascal habría que conocer ambos límites).

La obtención del CI para vectores de más de una dimensión es poco más compleja. Lo importante es saber que en memoria, los elementos del vector están ordenados secuencialmente. Por ejemplo, en C:

```
int[f][c] v;
```

El elemento v[i][j] estaría en la dirección:

Direccion(v[i][j]) = Direccion(v) + (i x (c + 1)) + (j + 1)

En todos los casos hemos supuesto que los elementos ocupan una sola dirección de memoria. Si no fuera así, habría que multiplicar cada posición por la dimensión que ocupe el tipo del elemento.

Para más dimensiones, el proceso sería similar.

En Pascal se haría teniendo en cuenta los cálculos para el caso de una sola dimensión.

Las comprobaciones semánticas deben tener en cuenta también que todas las variables de un tipo vector tienen las mismas dimensiones que el tipo.

8.12 Código intermedio para registros

La generación de CI para registros puede ser muy simple o compleja como los vectores. Todo depende de si el compilador que se quiere construir admite como campo un vector o no. Si admite como campo un vector, no se podría calcular en tiempo de compilación la posición de cada elemento del registro (habría que seguir el mecanismo explicado para vectores). Si no se admiten vectores como campos, el cálculo de la dirección de cada campo del registro se podría hacer en tiempo de compilación.

Como la estructura y el nombre de los campos de los registros están en la tabla de tipos, debemos acudir a ella para realizar las comprobaciones semánticas oportunas (ver

si el acceso de un campo de una variable registro es adecuado respecto al nombre y al tipo, etc.). También acudimos a ella para saber la dirección relativa de los elementos de cada campo respecto al primer elemento de la variable registro (que está en la tabla de símbolos).

Por ejemplo, si tenemos definido un registro en Pascal:

```
type registro=record

x:integer;

y:boolean

end;

var r:registro;
```

Si la dirección base de la variable *r* es 9000, la dirección del campo *x* será 9000 y la del campo *y* será 9001. Es decir:

Direccion(r.x) = Direccion(r)

Direccion(r.y) = Direccion(r) + Tamaño(x)

Si por ejemplo, *x* fuera del tipo de un registro de tamaño 10, la dirección del campo *y* sería Direccion(r) + Tamaño(x) = Direccion(r) + 10.

El tamaño o la dimensión de cada campo están en la tabla de tipos, por lo que su utilización es inmediata.

Por ejemplo, la utilización de un campo de un registro en una expresión sería algo así (si no se admite la definición de registros recursivos):

```
Expresion ::= id1 punto id2 puntocoma

{:

Expresión.dir = direccion_en_tabla(id1.lexema)+

direccion_relativa(id1.lexema,id2.lexema);

:}
```

La dirección relativa se tiene que calcular en la tabla de tipos. Esta dirección relativa es la distancia desde el primer elemento del registro en la tabla de tipos hasta el elemento señalado (si los elementos ocupan más o menos espacio en memoria habría que tenerlo en cuenta). Vemos que aquí no se genera CI, ya que se generará a partir de la utilización de la expresión en otras operaciones (ya que le hemos asignado ya a esta

expresión la dirección donde se guarda el valor del campo indicado).

8.13 Espacio de direcciones

Un aspecto muy importante a la hora de afrontar la creación de un compilador es organizar el espacio de direcciones, es decir, dónde va cada objeto dentro de la memoria física del computador de destino.

En principio, podría parecer que esto contradice la definición de lo que es el CI, que debe ser independiente de la máquina final. Pero en la práctica es aconsejable que no lo sea del todo (ya que eso simplificará mucho la implementación del compilador). Por eso vamos a generar CI tomando en cuenta la organización de la máquina final y, por lo tanto, de las funcionalidades de su lenguaje ensamblador.

La máquina final que vamos a utilizar es una máquina virtual generada por un programa que se llama ENS2001. Este programa hace las veces de una máquina física con una serie de características y un lenguaje ensamblador propio.

La descripción del ENS2001 viene detallada en un apéndice. Aquí sólo nos referiremos a la organización de la memoria que vamos a establecer para nuestro compilador.

La memoria del ENS2001 consta de 64K unidades de memoria de 16 bits cada una (los registros son todos de 16 bits). Comienza en la dirección 0 y termina en la 65535.

Por defecto, nuestros programas van a situarse a partir de la dirección 0. Las variables y las direcciones temporales no van a estar mezcladas con el código y, por lo tanto, se les asignará una dirección a partir de la cual estarán los datos.

ENS2001 admite también la utilización de una pila. La pila, por defecto, comienza en la dirección 65535 y va hacia abajo (se puede indicar que comience en otra dirección cualquiera y corra hacia arriba).

ENS2001 tiene una serie de registros especiales (contador de programa, contador de pila, etc.), una serie de registros de propósito general (del R0 al R9), un registro acumulador (A) y un par de registros índice (IX e IY).

Una cosa importante es que directamente sólo se puede operar con números enteros (para utilizar reales, habría que complicar algo el código). Cada entero ocupa una unidad de memoria, es decir, que los enteros son guardados en memoria en 16 bits. Por lo tanto, su rango es desde -32768 a 32767 si tienen signo y desde 0 a 65535 si no lo tienen.

Como hemos comentado, el código y los datos van aparte, por lo que debemos saber en qué dirección vamos a poner los datos. Como el proceso de compilación se hace de una sola pasada es imposible saber dónde va a terminar el código para poner los datos a continuación. Para solucionar este problema o bien se asigna una dirección lo suficientemente alta a los datos como para que el código nunca “pise” esa dirección (esto es posible para pequeños programas), o bien los datos se ponen como si se tratara de la pila (desde la última dirección hacia abajo) o bien se utiliza un índice para referirse a los

datos (el índice debe calcularse al final del análisis de todo el programa pero insertarse su código ensamblador al comienzo del programa ensamblado).

Nosotros vamos a optar en los ejemplos por un código que comience en la dirección 0, los datos de las variables globales y de las temporales del programa principal a partir de la última dirección del código y el resto de variables locales y temporales a partir de los datos globales. Es decir, va a haber tres zonas en la memoria, una para el código, otra para las variables y temporales globales y otra para las locales (las locales son todas las pertenecientes a los subprogramas).

Un programa en un lenguaje imperativo consta de una parte principal donde están definidas las variables globales, los tipos globales y el programa principal y una serie de subprogramas con sus propias variables locales, tipos locales, código local y en algunos lenguajes, sus propios subprogramas (por ejemplo, en Pascal).

***Figura 8.7**. Organización de la memoria*

Para acceder a las dos zonas de datos, disponemos de dos registros índices en ENS2001 (IX e IY). La carga de los valores de ambos se hará al final de la generación del código y se insertará al comienzo del mismo. Es decir,

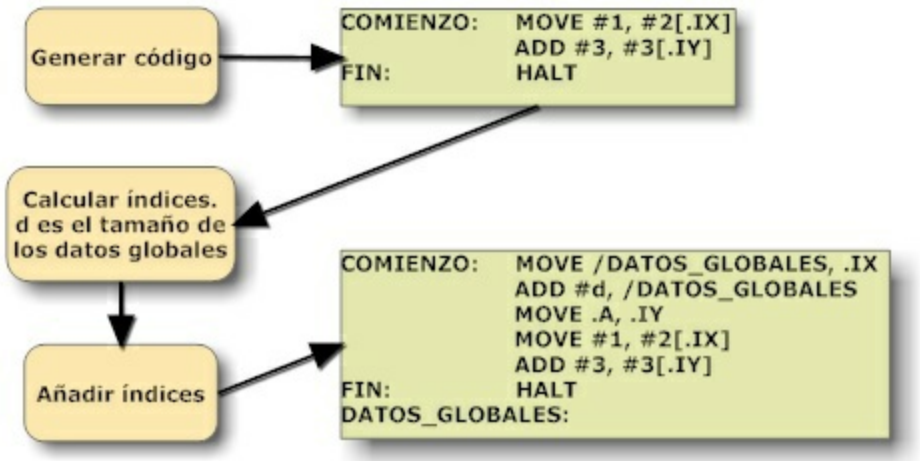


***Figura 8.8**. Inserción de los índices*

En la figura 8.8 vemos que primero se crea el código del programa suponiendo

que los registros índices tienen sus valores ya calculados y al final se inserta la etiqueta para los datos globales. Como ya se sabe cuánto ocupan los datos globales, el segundo registro índice se obtiene sumándole al primer registro índice el tamaño de los datos globales. Después, se insertan ambos al comienzo del todo del programa.

Otro aspecto a tener en cuenta es que generalmente un programa consta de un programa principal y una serie de subprogramas que generalmente se escriben antes del código del programa principal. Por lo tanto, el código generado de los subprogramas irá delante del código del programa principal. Como lo primero que se debe ejecutar es el programa principal, y este se encargará de llamar a los subprogramas, al comienzo del código en ensamblador deberemos insertar una o unas instrucciones (además de las de llenado de los índices) para pasar el control al programa principal. Ya que si no lo hiciéramos así, se ejecutaría primero el primer subprograma que apareciera escrito en el código fuente.

Por lo tanto, después de generar el CI para calcular los índices, se obtiene el CI para el salto al programa principal. Y estos dos códigos generados se insertan al comienzo del código del programa.

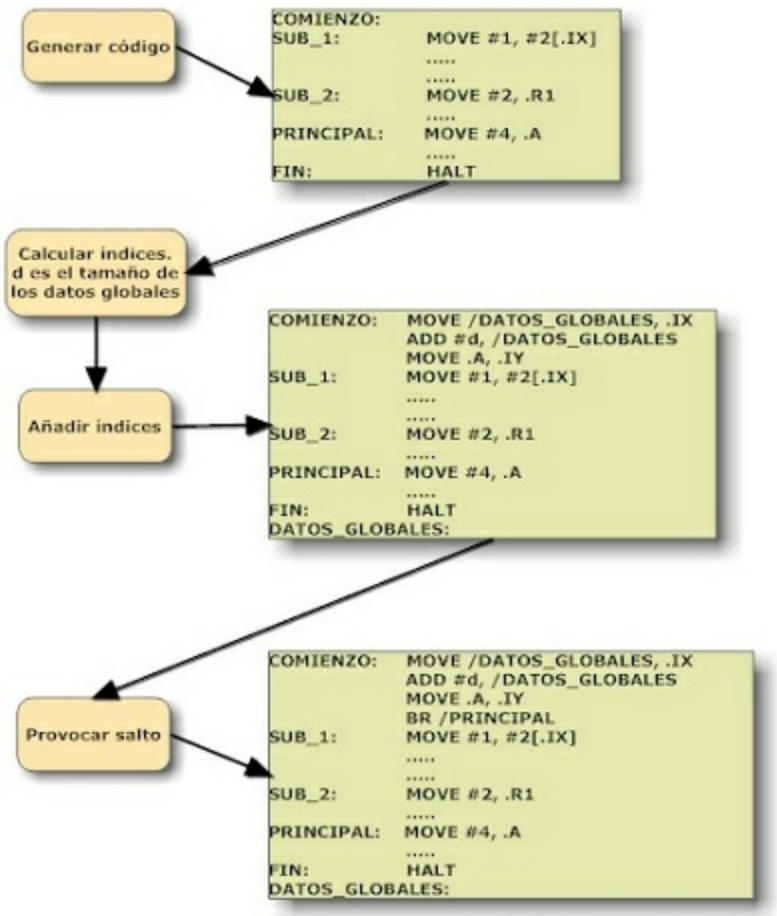


***Figura 8.9**. Salto al código del programa principal*

En entornos estáticos (aquellos en que no se permiten subprogramas con llamadas recursivas) es posible poner todas las variables globales, temporales globales y variables y temporales locales en una misma zona de direcciones, unas detrás de las otras. Pero como veremos cuando estudiemos la generación de código para

subprogramas, esta técnica no es posible y tendremos que utilizar la técnica de dos espacios diferentes para las globales y para las locales.

Aunque en este libro estudiaremos la organización de la memoria basándonos en la utilización de temporales localizadas secuencialmente, también es posible utilizar una pila para esta tarea.

En las siguientes secciones estudiaremos los subprogramas. La generación de código para ellos es más compleja que para las instrucciones que hemos visto hasta ahora.

8.14 Registro de activación (RA)

El RA es una estructura de datos en memoria donde se guarda información necesaria para el uso de subprogramas. En la máquina física, esta estructura de datos es una serie de direcciones de memoria, consecutivas, que guardan información necesaria para la utilización de subprogramas. Estos datos son los parámetros del subprograma, las variables locales, las direcciones temporales utilizadas en el código del subprograma, el valor devuelto (si es una función) y una serie de otros datos que más adelante detallaremos.

Cada subprograma debemos verlo como una serie de código, que se generará como el resto del código del programa y una serie de datos que estarán en una zona de la memoria reservada al subprograma (que es su RA). Generalmente, la zona del RA va aparte de la zona de las variables globales o temporales. El RA es una zona de datos propios de un subprograma.

Para almacenar el RA de los subprogramas hay varias formas de hacerlo; por ejemplo, en una pila, un montículo, etc. Nosotros almacenaremos los RA apilados en la memoria (si utilizar la pila de la máquina).

Por ejemplo, en la figura 8.10 podemos ver la organización de la memoria de un posible programa. Lo primero que se aloja en la memoria es el código del programa principal (incluido el de los subprogramas). Después, van los datos globales (variables globales, temporales, etc.). A continuación, se aloja la zona de los RA de cada subprograma (sus parámetros, sus variables locales, sus temporales, etc.). Cada RA de cada subprograma se pone a continuación del RA del subprograma anterior (veremos que cuando hay subprogramas recursivos, cada llamada generará un RA propio). Al final va la pila de la máquina.

Los RA no son estructuras estáticas, normalmente, sino que se irán generando conforme el código del programa encuentre una llamada a un subprograma. El hecho de encontrar una llamada a un subprograma lanza una serie de acciones enfocadas a crear el RA del mismo detrás del RA creado anteriormente (es decir, apilándolo sobre el RA anterior). Se trata de una especie de pila de RA. Tras la llamada al subprograma y la creación del RA, se va al código del subprograma y se ejecuta el mismo (que manipula los datos de su RA). Al final del código del subprograma, se devuelve el control y la

dirección en memoria donde se ha puesto el valor devuelto. Luego, se pasa el control a la línea siguiente de donde se invocó el subprograma (la dirección del valor devuelto será conocida por el invocante). Después, se podrá reutilizar la zona del RA ya que no será necesaria.

Decimos que los RA se apilan porque es posible que un subprograma invoque a otro subprograma y por eso sus RA se apilarán uno tras otro.

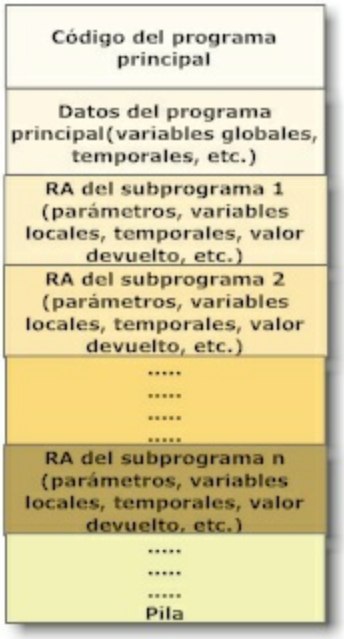


Figura 8.10

8.15 Secuencia de acciones en subprogramas no recursivos

Como se verá con más detalle en las siguientes secciones, la llamada a un subprograma requiere una serie de pasos anteriores a la ejecución del código del subprograma invocado. Por ejemplo, se debe reservar memoria para el RA, se debe guardar la dirección de retorno tras la llamada, se deben pasar los parámetros, etc.

En cambio, el código del subprograma debe situar el valor a devolver en el lugar adecuado (conocido por el invocante) y debe retornar el control al invocante.

Al mismo tiempo que se realizan estas acciones, se deben hacer una serie de comprobaciones semánticas como son:

- Reservar la utilización de paréntesis sólo para subprogramas y no para variables.
- Verificar la corrección en cuanto al número y tipo de argumentos.
- Verificar el tipo devuelto si existe.

Con el siguiente ejemplo, en C, veremos cómo es el proceso de llamada a subprogramas en un ambiente en el que no se permiten subprogramas recursivos. Sea el

programa siguiente:

```
int x;

int f1() {

return 10;

}

int f2(int a) {

return a+1;

}

main() {

x = f1();

x = f2(x);

x = f2(5);

}
```

Supongamos que las variables globales y las temporales las guardamos a partir de la dirección 9000 y los RA a partir de la dirección 10000 (ya vimos anteriormente que estos valores se calcularán al final del código del programa, pero ahora supondremos que están ya calculados).

Como estamos en un ambiente estático y cada RA de cada subprograma estará a continuación del RA del subprograma anterior, en tiempo de compilación asignaremos a cada subprograma la dirección a partir de la cual estará su RA (esta información debe guardarse en su entrada en la tabla de símbolos). De esta manera, antes llamar a un subprograma, se mira la dirección en la que debe alojarse su RA, se carga el registro IX con la dirección + 2 unidades (para el valor devuelto y para la dirección de retorno) y se procede a poner los parámetros y a llamar. Para este proceso, debemos tener un contador global que nos indique dónde se debe alojar el RA de cada subprograma a partir del espacio que ocupa el RA anterior (eso se hace mirando el contador de temporales locales del subprograma anterior).

Un posible código final sería:

```
Línea 1 → BR /MAIN
```

```
Línea 2 → F1: MOVE #10, #0[.IX]

Línea 3 → MOVE #0[.IX], #-2[.IX]

Línea 4 → MOVE #-1[.IX], .A

Línea 5 → MOVE .A, .PC

Línea 6 → F2: MOVE #0[.IX], #1[.IX]

Línea 7 → MOVE #1, #2[.IX]

Línea 8 → ADD #1[.IX], #2[.IX]

Línea 9 → MOVE .A, #3[.IX]

Línea 10 → MOVE #3[.IX], #-2[.IX]

Línea 11 → MOVE #-1[.IX], .A

Línea 12 → MOVE .A, .PC

Línea 13 →MAIN: MOVE #10002, .IX

Línea 14 → MOVE #L1_F1, #-1[.IX]

Línea 15 → BR /F1

Línea 16 →L1_F1: MOVE #-2[.IX], /9001

Línea 17 → MOVE /9001, /9000

Línea 18 → MOVE /9000, /9002

Línea 19 → MOVE #10005, .IX

Línea 20 → MOVE /9002, #0[.IX]

Línea 21 → MOVE #L1_F2, #-1[.IX]

Línea 22 → BR /F2
```

```
Línea 23 →L1_F2: MOVE #-2[.IX], /9003

Línea 24 → MOVE /9003, /9000

Línea 25 → MOVE #5, /9004

Línea 26 → MOVE #10005, .IX

Línea 27 → MOVE /9004, #0[.IX]

Línea 28 → MOVE #L2_F2, #-1[.IX]

Línea 29 → BR /F2

Línea 30 →L2_F2: MOVE #-2[.IX], /9005

Línea 31 → MOVE /9005, /9000

Línea 32 → HALT
```

Programa 1

A partir de este código final explicaremos el proceso de creación de CI para subprogramas y su código final.

Lo primero que destacamos es la estructura del RA de cada subprograma. Sería la indicada en la figura 8.11.

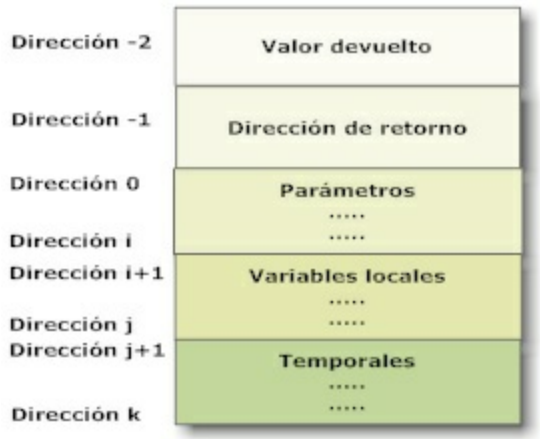


Figura 8.11*. Estructura de un RA de subprogramas no recursivos*

Las direcciones de los diferentes campos de un RA son relativas a un índice; en este caso, el índice está guardado en el registro IX.

Las variables globales y las temporales globales están a partir de la dirección absoluta 9000. Por ejemplo, la variable *x* está en la dirección 9000 y las temporales necesarias estarán a partir de la dirección 9001 y siguientes.

En la línea 1 se salta al código del programa principal.

La línea 2 es la primera del código de la función *f1*. Vemos que se ha señalado con una etiqueta con el nombre de la función. Por lo tanto, podemos emplear el nombre de la función, que está en la tabla de símbolos, para señalar esta etiqueta. Ya sabemos cómo insertar el CI para poner etiquetas.

Lo primero que destaca a partir de esta línea es que guardamos el valor 10 en la tercera temporal del RA (la primera está en #-2[.IX]). Es decir, nos hemos saltado dos direcciones, la primera es para situar el valor devuelto y la segunda es para indicar la dirección de retorno. En el caso de que el valor devuelto ocupe más de una dirección de memoria (por ejemplo, si es un vector de más de un elemento), reservaremos tantas direcciones para el valor devuelto como dimensión del mismo. Esta información la obtenemos de la entrada del subprograma en la tabla de símbolos (uno de los campos es el tipo devuelto y por tanto buscando en la tabla de tipos sabemos su dimensión).

En nuestro ejemplo, como se trata de un entero y ocupa una dirección, reservamos sólo una dirección.

Como no hay parámetros, no se reserva más espacio en el RA. Por lo tanto, la siguiente dirección temporal a utilizar será la señalada por #0[.IX].

En la línea 2 cargamos el valor 10 en una temporal.

En la línea 3 cargamos la dirección del valor a devolver con el valor obtenido de la temporal.

La regla de la gramática que hace esto sería:

```
SentenciaRetorno ::= return Expresion

{:

//Se comprueba que el tipo de la expresión es el mismo

//que el que devuelve la función

insertarCI("RETORNO", Expresión.dir, null, null);

:}
```

Si *Expresion.dir* vale 0 como en nuestro ejemplo, el código final para el retorno sería:

MOVE #0[.IX], #-2[.IX]→ Pone el valor de retorno

MOVE #-1[.IX], .A → Pone la dirección de retorno en el registro .A

MOVE .A, .PC → Devuelve el control a la dirección de retorno

Aunque parezca caprichoso que el registro IX apunte a dos direcciones después del comienzo del RA en vez de al comienzo del RA, no lo es. El código generado es más sencillo de leer. El RA siempre apuntará a la primera dirección local asignada a un parámetro, variable local o temporal.

En el caso de que en vez de una función se trate de un procedimiento, como no hay instrucción de retorno, se pone al final de todo el código del procedimiento la sentencia de retorno. En este caso, no es necesario poner el valor a devolver en la dirección #-2[.IX].

Esto se hace así para el caso de C, pero para Pascal es diferente. En Pascal, el valor a devolver se pone en una variable que tiene el mismo nombre que el subprograma. El contenido de esta variable es el que se pasa a la dirección relativa #-2[.IX]. Por lo tanto, cuando se genera el código de una función en Pascal, hay que introducir en la tabla de símbolos del ámbito de la función una nueva variable por defecto que tiene el mismo nombre que la función y el mismo tipo. Al final del código del subprograma, se copiará el contenido de la dirección señalada por esta variable en la dirección relativa #-2[.IX].

Una característica que no se ha comentado hasta ahora es que el contador de temporales de un subprograma debe comenzar desde 0. Por lo tanto, debe haber un contador para las temporales globales y otro para las de los subprogramas.

Como en C no se permite el anidamiento de subprogramas, no hay nada más que decir al respecto. Pero en Pascal, se permite declarar subprogramas dentro de otros, por lo que si se pone a cero el contador de temporales en un subprograma y dentro de él se declara otro subprograma, se pondría otra vez a 0 el contador, por lo que se perdería el del subprograma padre. Por lo tanto, habrá que implementar una pila de contadores de temporales de manera que cada subprograma maneje el suyo propio. Cuando se entra en un subprograma dentro de otro, se apila su contador de temporales y cuando se sale, se desapila.

Por ejemplo, sea en Pascal:

```
program p;

procedure p1;

procedure p2;

var x,z:integer;

begin

end;

var y:integer;
```

```
begin

end;

begin

end.
```

Debemos tener un contador de temporales global y una pila de contadores locales.

Cuando procesamos el código del procedimiento *p1*, insertamos su contador de temporales locales en la pila. Este contador vale 2 (reserva dos direcciones, la 0 y la 1, y apunta a la siguiente utilizable como temporal). Luego, entramos en el código de *p2* y apilamos su contador, que en este caso vale también 2. Más tarde, se declara la variable *x* que tendrá asignada la dirección 2. Después, la *z* que estará en 3. Se termina el código de *p2* y su contador de temporales valdrá lo que sea (dependiendo de las instrucciones que se hayan procesado). Se desapila y se recupera el contador de temporales de *p1*, que valía 2. Aparece la variable y que será asignada a la dirección 2 (la que indica su contador de temporales). Se termina el código de *p1* y se desapila el contador (la pila se queda vacía).

El tema de subprogramas locales lo explicaremos más adelante, en otra sección. Ahora nos centraremos sólo en subprogramas no recursivos.

Continuemos viendo el código del **Programa 1**.

En la línea 6 se inserta la etiqueta con el nombre del subprograma *f2*. En este caso, se reservan las dos primeras direcciones relativas a .IX, la #-2[.IX] y la #-1[.IX] para el valor devuelto y la dirección de retorno. Ahora, el contador de temporales es 0. Se asigna al parámetro *a* esta dirección en su entrada en la tabla de símbolos y el contador de temporales local se aumenta en una unidad. Se evalúa la expresión de retorno y el valor se copia en la dirección #-2[.IX]. Luego, se procede a insertar el código para la dirección de retorno.

Estas son las acciones que realizan los subprogramas llamados.

En relación con los parámetros, hay que comprobar que son los correctos en cuanto al número y al tipo (todo está en la tabla de símbolos).

Un aspecto más respecto a los parámetros, en este caso se denominan argumentos, ya que para el subprograma invocado son argumentos, aunque para el invocante sean parámetros. En C, los parámetros se pasan por valor. Es decir, tal y como hemos hecho en el ejemplo. Pero hay lenguajes, como Pascal, en los que se permite el paso por referencia. Es decir, si un parámetro es modificado dentro del código de un subprograma en el que

se le ha pasado como parámetro, esta modificación permanece al salir del código del subprograma. Por lo tanto, no se utilizaría una dirección temporal local para guardar el valor del parámetro pasado a un subprograma sino su dirección absoluta (conocida por el invocante). Pero esto sale fuera del objeto de este libro.

Llegamos ahora a la línea 13. Aquí se pone la etiqueta de comienzo del programa principal. Se llena el registro índice (IX) con la dirección donde va a estar alojado su RA, que se obtiene de la tabla de símbolos (en realidad se apunta dos direcciones más adelante). Además, se pone el código de la primera llamada a un subprograma. Lo primero que se hace es inventar una etiqueta para señalar la siguiente instrucción después de la llamada (aquella que se copia en la dirección relativa #-1[.IX] en el subprograma invocado). Se utiliza una etiqueta porque no se sabe exactamente de qué dirección se trata. La etiqueta debe ser única para cada llamada. Por lo tanto, es una buena opción llevar un contador de llamadas y añadirlo al nombre del subprograma al que se llama. Una vez inventada la etiqueta, usamos su dirección para cargarla en la dirección #-1[.IX].

En la línea 15 se pone el código del salto al código del subprograma (en este caso, como no hay parámetros, no se añade nada más).

En la línea 16 se recupera el valor devuelto, que está en la dirección #-2[.IX]. Esta línea es la línea etiquetada donde deberá devolver el control el subprograma invocado.

El valor devuelto se copia en la siguiente temporal disponible. En este caso en la 9001. Luego, viene el código generado por la primera asignación.

Para la segunda llamada, primero se pone el registro índice IX con el valor adecuado (como *f1* sólo utiliza una temporal, el valor de IX para *f2* es el de *f1* más 3. Una por la temporal utilizada, otra para el valor devuelto y una más para la dirección de retorno). Como esta función tiene un parámetro, habrá que poner su valor en la dirección correspondiente relativa al registro .IX del RA de la función a la que se va a llamar. En el caso de *f2*, tiene reservadas las dos primeras direcciones para el valor devuelto y la dirección de retorno y la tercera es para el parámetro. En esta dirección es donde hay que poner el valor del parámetro. El parámetro contiene su valor en la dirección temporal global 9002 (obtenida de la evaluación de la expresión). Por lo tanto, su contenido se copia en la dirección #0[.IX] y luego se llama a la función tal y como se hizo en el caso de *f1*. La otra llamada que queda se hace igual.

Si hubiera más parámetros, por cada uno de ellos habría que hacer esta operación.

Hemos considerado que la dimensión de cada parámetro es 1, pero si es mayor, copiaríamos cada una de las direcciones a partir de la primera y en número igual a la dimensión. Lo mismo haríamos para el valor devuelto. Si su dimensión es mayor que 1, copiaríamos cada una de las direcciones en temporales consecutivas en número igual a la

dimensión del tipo devuelto.

El CI necesario es directo. Para la llamada se utiliza el nombre de la función y la etiqueta que señala la dirección de retorno. Para copiar cada parámetro, se utiliza un CI en que una dirección sea la del parámetro y la dirección del resultado es la relativa a .IX. Es decir, copiar el contenido de la dirección del parámetro en la dirección correspondiente en el RA del subprograma.

Veamos el mapa de memoria que resulta de la ejecución del **Programa 1**.

```
0   BR /26                      28                          56
1                               29  MOVE #34, #-1[.IX]      57  MOVE /9003, /9000
2   MOVE #10, #0[.IX]           30                          58
3                               31                          59
4                               32  BR /2                   60  MOVE #5, /9004
5   MOVE #0[.IX], #-2[.IX]      33                          61
6                               34  MOVE #-2[.IX], /9001    62
7   MOVE #-1[.IX], .A           35                          63  MOVE #10005, .IX
8                               36                          64
9   MOVE .A, .PC                37  MOVE /9001, /9000      55
10                              38                          66  MOVE /9004, #0[.IX]
11  MOVE #0[.IX], #1[.IX]       39                          67
12                              40  MOVE /9000, /9002       68
13  MOVE #1, #2[.IX]            41                          69  MOVE #74, #-1[.IX]
14                              42                          70
15                              43  MOVE #10005, .IX        71
16  ADD #1[.IX], #2[.IX]        44                          72  BR /11
17                              45                          73
18  MOVE .A, #3[.IX]            46  MOVE /9002, #0[.IX]    74  MOVE #-2[.IX], /9005
19                              47                          75
20  MOVE #3[.IX], #-2[.IX]      48                          76
21                              49  MOVE #54, #-1[.IX]      77  MOVE /9005, /9000
22  MOVE #-1[.IX], .A           50                          78
23                              51                          79
24  MOVE .A, .PC                52  BR /11                  80  HALT
25                              53                          81
26  MOVE #10002, .IX            54  MOVE #-2[.IX], /9003   82
27                              55                          83
```

```
x ⟶ 9000  6
t1 ⟶ 9001 10
t2 ⟶ 9002 10
t3 ⟶ 9003 11
t4 ⟶ 9004 5
t5 ⟶ 9005 6
```

```
f1  dev ⟶ 10000     10
f1  Dir ret ⟶ 10001 34
f1  t1 ⟶ 10002      10
```

f2 dev ⟶ 10003 6

f2 Dir ret ⟶ 10004 74

f2 t1 ⟶ 10005 5

f2 t2 ⟶ 10006 5

f2 t3 ⟶ 10007 1

f2 t4 ⟶ 10008 6

En este mapa de memoria hemos optado por incluir en la zona de código no las instrucciones en código máquina, sino las instrucciones en ensamblador (las direcciones en blanco en realidad son parte de la instrucción anterior, que ocupa más de una dirección de memoria).

En el código se han sustituido las etiquetas por los valores reales (este proceso lo hace el programa ensamblador).

A partir de la dirección 9000 están las variables globales y las temporales globales (la dirección 9000 es de la variable global *x*).

A partir de la dirección 10000 están los RA. Aunque el RA comienza en la dirección 10000, el registro IX apunta dos direcciones más adelante, después de las dos direcciones reservadas al valor devuelto y a la dirección de retorno. Cuando se cierra el ámbito de un subprograma se calcula el comienzo del RA del siguiente y se incluye en su entrada de la tabla de símbolos.

Vemos que en la dirección 10003 está el último valor devuelto por la función *f2*, que es 6. En la dirección 10004 está la dirección de retorno de la última llamada a la función *f2*. La dirección 10005 contiene el valor del parámetro que se le ha pasado a *f2*, es decir, el número 5. El resto son temporales utilizadas.

Aunque en estos ejemplos hemos utilizado los valores 9000 y 10000 arbitrariamente para alojar los datos globales y locales, ya se comentó en secciones anteriores que estos valores es posible obtenerlos de manera que los datos vayan a continuación exactamente del código. De esa manera, no llegaría el caso en que se solapen ambos y, por tanto, el programa no funcione correctamente.

Nota. Hay que tener mucho cuidado y acordarse de que el registro IX apunta dos direcciones más adelante que el RA.

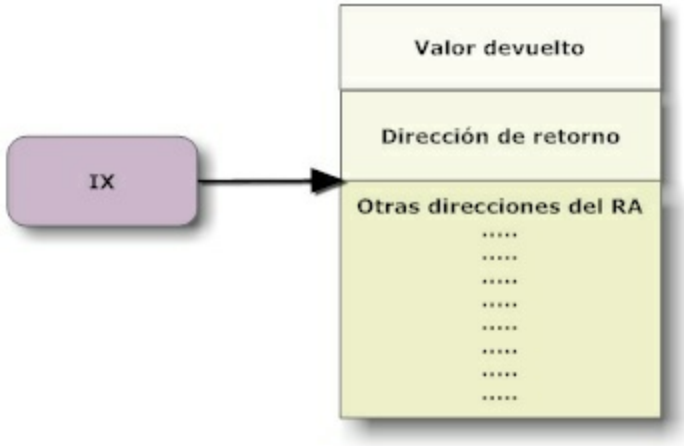


***Figura 8.12**. RA para entornos estáticos*

Es muy importante distinguir entre el código que se genera dentro de los subprogramas y el código de las llamadas. También hay que distinguir entre el código de los subprogramas y los datos manejados por ese código. Los datos se encuentran en el RA del subprograma y el código que maneja estos datos está dentro del código del subprograma. Quizás un gráfico permita ver mejor estas diferencias.

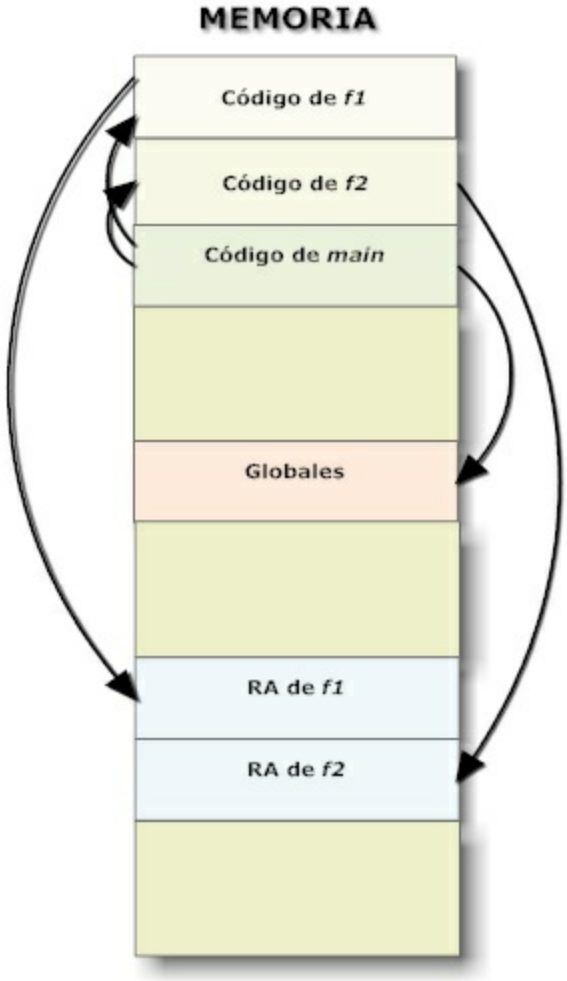


Figura 8.13**. Organización de la memoria para el **Programa 1

8.16 Secuencia de acciones en subprogramas recursivos

Si se permite la utilización de subprogramas recursivos, ya no es posible mantener un entorno de ejecución totalmente estático. Esto se debe a que en la recursividad, si un subprograma se llama a sí mismo, cada llamada debe tener su propio RA ya que los datos con los que opera el subprograma son diferentes. Estos RA suelen apilarse en memoria tal y como hemos hecho antes para subprogramas diferentes o utilizar la pila para apilarlos. Nosotros vamos a seguir la técnica de situar los sucesivos RA uno detrás de otro en la memoria.

Cada llamada a un subprograma apila un RA con los datos propios de la llamada

y cuando se sale del código y se devuelve el control al invocante, el RA se elimina. Por lo tanto, en un mismo instante de la ejecución del programa es posible que haya en memoria más de un RA de un mismo subprograma.

Para poder manejar esta información, es necesario añadir un puntero a cada RA que apunte al RA anterior (*enlace de control*). Por lo tanto, en vez de reservar 2 direcciones al valor devuelto y a la dirección de retorno, se reserva otra dirección para apuntar al RA anterior. Por lo tanto, cuando finaliza la ejecución de un subprograma, el valor del registro índice IX se debe llenar con la dirección del RA anterior. Al registro IX se le suele llamar *puntero de cuadro*.

Hay que mantener la norma de que el puntero de cuadro no apunta al comienzo del RA sino a la primera dirección después de las tres direcciones reservadas. Si el subprograma tuviera parámetros, apuntaría al primer parámetro. Si no tuviera parámetros, apuntaría a la primera variable local. Y si no tuviese tampoco variables locales, apuntaría a la primera temporal.

Tras esta remodelación del RA, quedaría así:

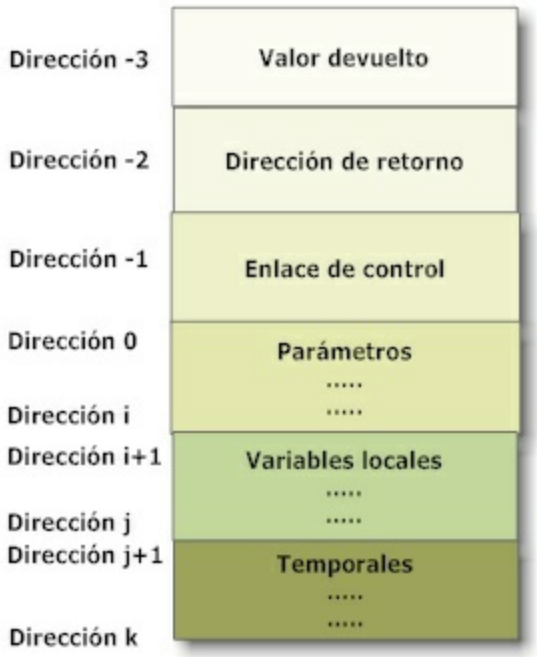


***Figura 8.14**. RA para subprogramas recursivos*

Como en este caso no se sabe la dirección concreta donde está cada RA, hay que calcularla en el propio código ensamblador en tiempo de ejecución. Para ello, se dota de alguna instrucción más a la secuencia de llamada y retorno.

Utilizaremos el siguiente programa para ver el código que debemos generar. El programa está escrito en C.

```
int x;
```

```
int factorial(int y) {

if(y<2) {

return 1;

} else {

return factorial(y-1) * y;

}

}

main() {

x = factorial(6);

}
```

Un posible código final en ENS2001 sería:

1	MOVE #10000, .IX	Pone en IX la dirección donde comenzarán a colocarse los RA
2	BR /MAIN	Salta al programa principal
3	FACTORIAL: MOVE #0[.IX], #1[.IX]	Guarda el contenido del parámetro *y* en una temporal
4	MOVE #2, #2[.IX]	Guarda el valor 2 en otra temporal
5	CMP #1[.IX], #2[.IX]	Compara ambos valores
6	BN $5	Salta 5 direcciones si *y*<2 (El número de direcciones a saltar dependerá de cuántas direcciones ocupe la instrucción siguiente)
7	MOVE #0, #3[.IX]	Si *y* no es menor que 2, pone un 0 en la siguiente temporal
8	BR $3	Salta para evitar poner un 1 en la siguiente temporal
9	MOVE #1, #3[.IX]	Si *y*<2 pone un 1 en la siguiente temporal
10	CMP #3[.IX], #0	Comprueba si y era o no menor que 2
11	BZ /ELSE_1	Si *y* no es menor que 2 salta al ***else***
12	MOVE #1, #4[.IX]	Carga la siguiente temporal con un 1
13	MOVE #4[.IX], #-3[.IX]	Coloca el valor de retorno en la dirección correspondiente
14	MOVE #-2[.IX], .A	Coloca la dirección de retorno en su dirección correspondiente
15	MOVE .A, .PC	Salta a la dirección de retorno
16	BR /FIN_1	Se salta el ***else***
17	ELSE_1: MOVE #0[.IX], #9[.IX] MOVE #1,	Reserva las direcciones relativas a IX de la 5 a la 7 para los tres valores fijos del RA y la 8 para el parámetro de la función que se llama. Utiliza la siguiente temporal para cargar el valor del parámetro *y*

18	#10[.IX]	Carga en la siguiente temporal el valor 1
19	SUB #9[.IX], #10[.IX]	Resta *y*-1
20	MOVE .A, #11[.IX]	El resultado lo pone en otra temporal
21	MOVE #11[.IX], #8[.IX]	Copia el valor en el parámetro que se le pasa a la función *factorial*
22	MOVE .IX, #7[.IX]	Pone el valor del *enlace de control* para la función que se va a llamar
23	ADD #8, .IX	Calcula el valor de IX para que apunte al siguiente RA, reservando el espacio para el RA actual, que ocupa de la dirección relativa a IX 0 hasta la 8
24	MOVE .A, .IX	Pone el valor en IX
25	MOVE /FACTORIAL_1, #-2[.IX]	Pone en el RA que se va a llamar la dirección de retorno correspondiente
26	BR /FACTORIAL	Salta a la función que se va a llamar
27	FACTORIAL_1: MOVE #-1[.IX], .IX	Tras la vuelta de la función, restaura el valor de IX
28	MOVE #0[.IX], #6[.IX]	Recicla las temporales que había antes de hacer la llamada. De modo que la última que se había utilizado era la 4 relativa a IX. La 5 relativa a IX es donde está el valor devuelto por la función. Pone el valor del parámetro en la siguiente temporal
29	MUL #5[.IX], #6[.IX]	Multiplica el valor devuelto por la llamada a la función por *y*
30	MOVE .A, #7[.IX]	El resultado lo pone en una temporal
31	MOVE #7[.IX], #-3[.IX]	Pone el valor para devolver
32	MOVE #-2[.IX], .A	Pone la dirección de retorno
33	MOVE .A, .PC	Salta a la dirección de retorno
34	FIN_1: MOVE #-2[.IX], .A	Por si no se ha llegado a retornar, al final se vuelve a poner el código para poder retornar
35	MOVE .A, .PC	Retorna si no se ha retornado antes
36	MAIN: MOVE #6, #4[.IX]	Se reserva de la dirección relativa a IX 0 a la 2 para los valores fijos de la llamada y la dirección relativa a IX 3 para el parámetro de la llamada. Se pone el valor 6 en la siguiente temporal
37	MOVE #4[.IX], #3[.IX]	Copia el valor en el parámetro que se le pasa a la función *factorial*
38	MOVE .IX, #2[.IX]	Pone el valor del enlace de control para la función que se va a llamar
39	ADD #3, .IX	Calcula el valor de IX para que apunte al siguiente RA, reservando el espacio para el RA actual, que ocupa de la dirección relativa a IX 0 hasta la 3
40	MOVE .A, .IX	Pone el valor en IX
41	MOVE /FACTORIAL_2, #-2[.IX]	Pone en el RA que se va a llamar la dirección de retorno correspondiente
42	BR /FACTORIAL	Salta a la función que se va a llamar
43	FACTORIAL_2: MOVE #-1[.IX], .IX	Tras la vuelta de la función, restaura el valor de IX
44	MOVE #0[.IX], /9000	Pone el valor devuelto en la dirección de la variable global *x*, que se supone que está en la dirección 9000
45	HALT	Finaliza el programa principal y por tanto todo el programa

Hay dos puntos importantes a la hora de afrontar la compilación de subprogramas: la compilación del código del subprograma y la compilación de la llamada al subprograma.

8.16.1 Compilación del cuerpo del subprograma

Las tareas a realizar cuando aparece el código de un subprograma al existir subprogramas recursivos son:

1. Comprobar que en la tabla de símbolos no existe un símbolo en el mismo ámbito y con el mismo nombre que el subprograma. Si no existe, se pasa al punto 2, y si existe, se lanza un error semántico para indicar que el identificador no es válido.
2. Si no existe el símbolo, se inserta en la tabla de símbolos actual una nueva entrada con el nombre del subprograma y con un indicador de que se trata de un subprograma (en el momento que se pueda saber si es una función o un procedimiento, habría que señalarlo en la tabla de símbolos). Cuando se sepa si es una función o un procedimiento habrá que indicar en la tabla de símbolos el tipo que devuelve o si no devuelve nada.
3. Las temporales comienzan a contar desde la dirección relativa a 0 y además se abre un nuevo ámbito.
4. Si existen parámetros, se insertan en la tabla de símbolos con la dirección relativa al índice 0, 1, etc., dependiendo del número y de la dimensión de los parámetros. Por ejemplo, si el primer parámetro es de tipo entero, su dirección sería la 0. Si el segundo es de tipo vector de enteros de dimensión 3, su dirección sería 1. Si el siguiente parámetro es de tipo entero, su dirección sería 4 ya que la 1, 2 y 3 corresponden al vector. Y así sucesivamente. La inserción de parámetros implica que en la entrada del subprograma en la tabla de símbolos debe insertarse información acerca de los símbolos y su orden de aparición. De esta manera, cuando se haga una llamada a un subprograma, se podrá comprobar que los parámetros que se le pasan son los adecuados en cuanto a orden y número.
5. Una vez que están metidos los parámetros y el tipo de la función o nulo si es un procedimiento, las temporales comienzan a contar a partir de la dirección asignada al último parámetro.
6. Las variables locales, si las hay, irán ocupando direcciones temporales consecutivas.
7. El código generado, que necesitará de temporales, las irá poniendo a continuación de las variables locales.
8. Al final del código del subprograma, se comprueba si existe retorno se trata de una función. Si es una función y no hay retorno, se debe generar

un error semántico.
9. Tras la última comprobación, se debe cerrar el ámbito y, por tanto, eliminar las entradas en la tabla de símbolos comenzando por los parámetros (se debe conservar en la entrada del subprograma en la tabla de símbolos la información relativa al número de parámetros y su tipo) y siguiendo por las variables locales y tipos locales. Al final, debe quedar sólo la entrada en la tabla de símbolos del subprograma.

Hay que tener en cuenta que el código de las instrucciones dentro de los subprogramas empleará direccionamiento respecto al índice IX para los parámetros, variables locales y temporales locales. En cambio, si se hace referencia a subprogramas externos o variables globales, se utilizará la tabla de símbolos para localizar las direcciones.

En cuanto al retorno, se debe almacenar en la dirección relativa #-3[.IX] el valor devuelto y realizar el salto a la dirección de retorno, previamente guardada en #-2[.IX] por el invocante.

8.16.2 Compilación de la llamada al subprograma

La mayor parte de las tareas para poder implementar subprogramas se deja para el invocante. Las acciones y comprobaciones a realizar son:

1. Se comprueba si el subprograma existe en la tabla de símbolos. Si no existe, se lanza un error semántico.
2. Si el subprograma existe, se reserva espacio para el RA de la llamada. Se reservan 3 direcciones.
3. Por cada parámetro, se reserva tanto espacio en el RA como espacio ocupe el parámetro. Previamente se debe comprobar si el tipo y el orden del parámetro son los correctos respecto a la información que hay guardada en la tabla de símbolos.
4. Por cada parámetro, se copia el valor obtenido al evaluar la expresión que lo calcula en la siguiente temporal respecto a las tres primeras reservadas y las reservadas a los parámetros anteriores (las temporales utilizadas se reciclan para calcular el siguiente parámetro).
5. Pone en IX la dirección del enlace de control y lo actualiza para que apunte al siguiente RA (debe añadirle tantas unidades como espacio ocupe el RA actual).
6. Se guarda la etiqueta de retorno en el nuevo RA, se salta a la función, y

tras recibir otra vez el control, se restaura el valor IX con el valor anterior.

Todo esto parece muy complejo pero con unos cuantos ejemplos se entenderá mejor.

Supongamos un programa en C:

```
Línea 1 → int x;

Línea 2 →

Línea 3 → int suma(int a, int b) {

Línea 4 → return a + b;

Línea 5 → }

Lìnea 6 →

Línea 7 → int resta(int a, int b) {

Línea 8 → return a - b;

Línea 9 → }

Línea 10 →

Línea 11 → main() {

Línea 12 → x = suma(1,2);

Línea 13 → x = resta(3,4);

Línea 14 → }
```

Vamos a ver el contenido de la tabla de símbolos, el análisis semántico y la generación de código en cada punto del programa.

Vamos a suponer que generamos código en un ambiente en que se permiten subprogramas recursivos. Vamos a suponer que el tipo entero tiene el código 0 en la tabla de tipos, que no vamos a incluir. Vamos a utilizar una sola tabla de símbolos para todo. Las variables globales y temporales globales las pondremos a partir de la dirección 9000. Los RA irán a partir de la dirección 10000.

```
1    MOVE #10003, .IX Pone en IX la dirección donde comenzarán a colocarse los RA. En realidad IX apunta al primer parámetro
2    BR /MAIN         Salta al código del programa principal
```

Línea 1 → int x;

Se comprueba si en la tabla de símbolos, en el ámbito actual hay ya alguna entrada con este nombre.

0	x	variable	0	-1	null	9000	0

Línea 3 → int suma(

Se comprueba si en la tabla de símbolos, en el ámbito actual hay ya alguna entrada con este nombre.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	0	[]	-1	0

```
3    SUMA:  Se pone la etiqueta de comienzo del código de la función
```

La siguiente temporal será la dirección relativa 0.

Línea 3 → int a,

Se comprueba si el parámetro ya existe en la tabla de símbolos en el ámbito actual.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	1	[0]	-1	0
2	a	parámetro	0	-1	null	0	1

La siguiente temporal será la dirección relativa 1.

Línea 3 → int b) {

Se comprueba si el parámetro ya existe en la tabla de símbolos en el ámbito actual.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	a	parámetro	0	-1	null	0	1
3	b	parámetro	0	-1	null	1	1

La siguiente temporal será la dirección relativa 2.

Línea 4 → return a + b;

0	x	variable	0	-1	null	9000	0

1	suma	función	0	2	[0,0]	-1	0
2	a	parámetro	0	-1	null	0	1
3	b	parámetro	0	-1	null	1	1

3 SUMA: MOVE #0[.IX], #2[.IX] Pone el primer parámetro en una temporal

La siguiente dirección temporal es la 3.

4 MOVE #1[.IX], #3[.IX] Pone el segundo parámetro en una temporal

La siguiente temporal es la 4.

5 ADD #2[.IX] , #3[.IX] Suma los dos valores
6 MOVE .A , #4[.IX] El resultado en la siguiente temporal

La siguiente temporal es la 5.

7 MOVE #4[.IX], #-3[.IX] Pone el valor a devolver
8 MOVE #-2[.IX], .A Pone la dirección de retorno
9 MOVE .A , .PC Retorna

Línea 5 → }

Comprueba que existe un retorno y es del tipo adecuado. Cierra este ámbito, eliminando todas las entradas en la tabla de símbolos.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0

Línea 7 → int resta(

Se comprueba si en la tabla de símbolos, en el ámbito actual hay ya alguna entrada con este nombre.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	0	[]	-1	0

10 RESTA: Se pone la etiqueta de comienzo del código de la función

La siguiente temporal será la dirección relativa 0.

Línea 7 → int a,

Se comprueba si el parámetro ya existe en la tabla de símbolos en el ámbito actual.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	1	[0]	-1	0
3	a	parámetro	0	-1	null	0	1

La siguiente temporal será la dirección relativa 1.

Línea 7 → , int b) {

Se comprueba si el parámetro ya existe en la tabla de símbolos en el ámbito actual.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	2	[0,0]	-1	0
3	a	parámetro	0	-1	null	0	1
4	b	parámetro	0	-1	null	1	1

La siguiente temporal será la dirección relativa 1.

Línea 8 → return a – b;

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	2	[0,0]	-1	0
3	a	parámetro	0	-1	null	0	1
4	b	parámetro	0	-1	null	1	1

```
10    SUMA: MOVE #0[.IX], #2[.IX] Pone el primer parámetro en una temporal
```

La siguiente dirección temporal es la 3.

```
11    MOVE #1[.IX], #3[.IX] Pone el segundo parámetro en una temporal
```

La siguiente temporal es la 4.

```
12    SUB #2[.IX] , #3[.IX] Suma los dos valores
13    MOVE .A , #4[.IX]     El resultado en la siguiente temporal
```

La siguiente temporal es la 5.

```
14    MOVE #4[.IX], #-3[.IX] Pone el valor a devolver
15    MOVE #-2[.IX], .A      Pone la dirección de retorno
16    MOVE .A , .PC          Retorna
```

Línea 9 → }

Comprueba que existe un retorno y es del tipo adecuado. Cierra este ámbito, eliminando todas las entradas en la tabla de símbolos.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0

2	resta	función	0	2	[0,0]	-1	0

Línea 11 → main() {

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	2	[0,0]	-1	0

17 MAIN: Se pone la etiqueta del programa principal

Línea 12 → x = suma(1,2);

Se comprueba que existe en este ámbito una función llamada *suma*. Se comprueba que esta función tiene dos parámetros de tipo entero. Se comprueba tanto que devuelve un valor entero como que comprueba que *x* es del mismo tipo que el devuelto por *suma*.

Las direcciones 9001 en adelante (la 9000 está ocupada por una variable global) pueden utilizarse para las temporales del programa principal. También puede utilizarse el programa principal como si estuviera en un RA. Lo haremos de esta manera. Por lo tanto, la primera temporal disponible será la dirección relativa 0.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	2	[0,0]	-1	0

18	MAIN: MOVE #1, #5[.IX]	Se reserva espacio en el RA actual para las tres direcciones reservadas y dos más para los dos parámetros. Si los parámetros fueran de dimensión mayor que 1, habría que reservar más espacio. Se pone un 1 en la siguiente temporal
19	MOVE #5[.IX], #3[.IX]	Se pone en el RA el primer parámetro
20	MOVE #2, #6[.IX]	Se pone un 2 en la siguiente temporal
21	MOVE #6[.IX], #4[.IX]	Se pone el segundo parámetro
22	MOVE .IX, #2[.IX]	Se guarda el enlace de control anterior
23	ADD #4, .IX	Calcula el valor de IX para que apunte al siguiente RA, reservando el espacio para el RA actual, que ocupa de la dirección relativa a IX 0 hasta la 4
24	MOVE .A, .IX	Guarda el valor de la suma en IX
25	MOVE /SUMA_1, #-2[.IX]	Pone la etiqueta de retorno
26	BR /SUMA	Salta al código de la función
27	SUMA_1: MOVE #-1[.IX], .IX	Deja IX como estaba y se finaliza la llamada. El valor que devuelve la función se queda en #0[.IX]
28	MOVE #0[.IX], /9000	Asigna a *x* el valor devuelto por la función

Línea 13 → x = resta(3,4);

Se comprueba que existe en este ámbito una función llamada *resta*. Se comprueba

que esta función tiene dos parámetros de tipo entero. Se comprueba tanto que devuelve un valor entero como que comprueba que *x* es del mismo tipo que el devuelto por *resta*.

Las direcciones 9001 en adelante (la 9000 está ocupada por una variable global) pueden utilizarse para las temporales del programa principal. También puede utilizarse el programa principal como si estuviera en un RA. Lo haremos de esta manera. Por lo tanto, la primera temporal disponible será la dirección relativa 0 ya que hemos reciclado las anteriores.

0	x	variable	0	-1	null	9000	0
1	suma	función	0	2	[0,0]	-1	0
2	resta	función	0	2	[0,0]	-1	0

29	MAIN: MOVE #3, #5[.IX]	Se reserva espacio en el RA actual para las tres direcciones reservadas y dos más para los dos parámetros. Si los parámetros fueran de dimensión mayor que 1, habría que reservar más espacio. Se pone un 1 en la siguiente temporal
30	MOVE #5[.IX], #3[.IX]	Se pone en el RA el primer parámetro
31	MOVE #4, #6[.IX]	Se pone un 2 en la siguiente temporal
32	MOVE #6[.IX], #4[.IX]	Se pone el segundo parámetro
33	MOVE .IX, #2[.IX]	Se guarda el enlace de control anterior
34	ADD #4, .IX	Calcula el valor de IX para que apunte al siguiente RA, reservando el espacio para el RA actual, que ocupa de la dirección relativa a IX 0 hasta la 4
35	MOVE .A, .IX	Guarda el valor de la suma en IX
36	MOVE /RESTA_1, #-2[.IX]	Pone la etiqueta de retorno
37	BR /RESTA	Salta al código de la función
38	RESTA_1: MOVE #-1[.IX], .IX	Deja IX como estaba y se finaliza la llamada. El valor que devuelve la función se queda en #0[.IX]
39	MOVE #0[.IX], /9000	Asigna a *x* el valor devuelto por la función

Línea 14 → }

Se libera la memoria.

Se puede apreciar en el código anterior que hay muchas decisiones que hay que tomar para la generación del código, como por ejemplo, dónde situar la globales, dónde poner los RA, decidir si reciclar temporales o no, etc.

Es el diseñador del compilador el que debe decidir según sus necesidades o gustos la manera de enfocar estas decisiones.

8.17 Secuencia de acciones en subprogramas locales

Hay lenguajes de programación que permiten definir subprogramas dentro de otros subprogramas. Esta funcionalidad crea una serie de problemas adicionales a la

hora de generar el compilador adecuado. Por ejemplo, como el direccionamiento dentro de un RA es relativo al mismo RA, si queremos acceder a una variable de un RA padre, su dirección relativa será respecto a su RA, por lo tanto hay un solapamiento de direcciones relativas. Con un ejemplo lo entenderemos mejor.

Supongamos que tenemos este código en Pascal:

```
program p;

procedure p1;

var x:integer;

procedure p2;

var y:integer;

begin

x := 10;

end;

begin

end;

begin

end.
```

La variable *x*, local a *p1*, estará en la dirección relativa a su RA. Esta dirección relativa será la primera temporal disponible, que es la 0. Pero en el RA de p2, la dirección en la que está la variable *y* es también la relativa a su RA en 0. Por lo tanto, cuando hagamos la asignación *x:=10*, en realidad estaremos asignando 10 a la variable *y* y no a la *x*. El problema es que para cada RA, las direcciones son relativas al comienzo del RA. Y si intentamos acceder a una variable que esté en un subprograma padre del nuestro, su entrada en la tabla de símbolos será relativa a nuestro RA ya que coinciden.

Por lo tanto, será necesario implementar un mecanismo que nos permita buscar en un RA padre, o abuelo, o bisabuelo, etc., la ubicación de una variable que no esté en el subprograma actual.

Para implementar esta necesidad, se puede optar por diferentes modelos. Uno de ellos podría ser ir saltando de RA en RA mediante el valor que hay en la dirección del *enlace de control* de cada RA. La búsqueda se pararía cuando se encontrara el RA adecuado. Pero el número de saltos no se puede conocer en tiempo de compilación, por lo que la implementación de esta técnica no se suele emplear debido a su dificultad de implementación. A esta técnica se la conoce como *encadenamiento*.

Hay otras dos técnicas más efectivas y fáciles de implementar.

8.17.1 Encadenamiento de accesos

Esta técnica consiste en añadir un nuevo campo al RA de manera que este campo apunta al RA del subprograma padre según está en el código del programa a compilar. Como sabemos el orden de anidamiento de un subprograma respecto a otro, podemos acceder al RA adecuado dando tantos saltos a través de este enlace, que se llama *enlace de acceso*, como orden de anidamiento haya entre la variable que intentamos acceder y el subprograma donde queremos utilizarla. Este valor se conoce en tiempo de compilación, por lo que su implementación en el código final es inmediata.

Este encadenamiento genera una serie de código adicional. Si el anidamiento es muy profundo, se generaría demasiado código, por lo que sería recomendable aplicar otra técnica diferente.

Este campo se utiliza en aquellos lenguajes que admiten anidamiento de subprogramas (por ejemplo, Pascal o MODULA 2). En cambio, en C no es necesario ya que todos los subprogramas tienen como padre el ámbito global.

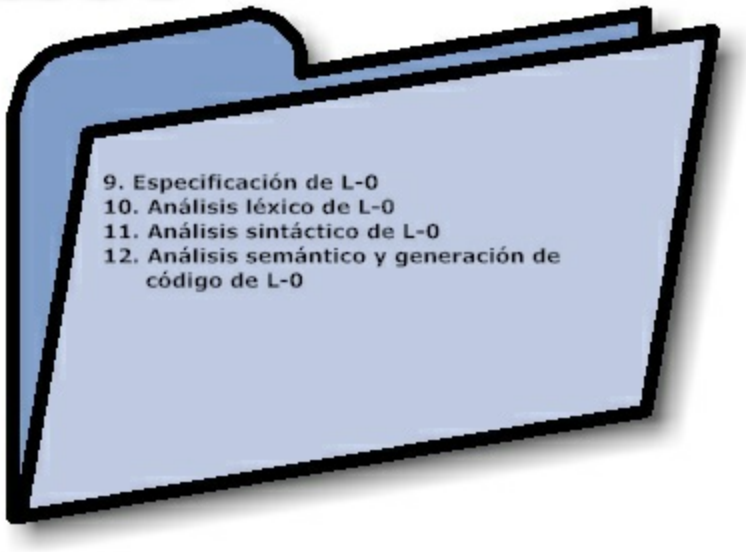
8.17.2 Display

Esta técnica se usa para evitar la creación de demasiado código adicional mediante la técnica de *encadenamiento de accesos*.

Consiste en la creación de un vector global al compilador, que en tiempo de ejecución guarde los *enlaces de acceso* necesarios. Estos datos se deben guardar en otra zona de la memoria a tal efecto. A este vector se le conoce con el nombre de *display*.

Para acceder a una variable de un subprograma, se consulta en la tabla de símbolos el nivel de anidamiento del subprograma donde está declarada la variable, y con este dato se puede acceder al elemento del vector del *display* correspondiente, que nos indicará la dirección adecuada.

Parte II. Implementación de L-0



CAPÍTULO 9

ESPECIFICACIÓN DE L-0

9.1 Introducción

Con esta práctica se pretende construir un traductor para un sencillo lenguaje para lógica de proposiciones.

Se va a utilizar como lenguaje de apoyo Java y como herramientas de ayuda para construir los analizadores: JLex y Cup. JLex se utiliza para el análisis léxico y Cup para el sintáctico, semántico y la generación de código, que en este caso es código final en Java. Es decir, no se genera código intermedio sino directamente código final.

En realidad, la salida de un programa en lenguaje L-0 es un archivo de texto con el resultado de la ejecución del programa. Es decir, no se obtiene un ejecutable sino directamente la salida del programa ejecutado.

En el apéndice A se estudiará el funcionamiento de las herramientas de apoyo que vamos a utilizar.

El programa a compilar se escribirá en un archivo de texto con cualquier nombre y extensión. El resultado de la traducción será obtenido en otro archivo de texto con el mismo nombre que el programa a compilar pero con la extensión .log.

A continuación, veremos las características que va a tener el lenguaje L-0.

9.2 Instrucciones

Un programa consiste en una seria de instrucciones seguidas de un terminador, que en este caso es un punto y coma.

Los espacios en blanco se ignoran (salvo dentro de las cadenas de texto).

Las diferentes instrucciones que se admiten son:

- Asignación: consta de dos partes separadas por el signo igual (“= “). La parte izquierda debe ser una variable lógica y la derecha puede ser cualquier expresión lógica o proposición lógica.
- Imprimir expresión (writelog(expresion)): imprime un 1 si la expresión es verdadera y un 0 si es falsa.
- Imprimir cadena (writestr(cadena)): imprime una cadena encerrada entre dobles comillas.

- Imprimir retorno (writeintro()): imprime un retorno de carro.
- Imprimir tabla de verdad (writetabla(expresion)): imprime la tabla de verdad de una expresión. En este caso, los valores de las variables lógicas que están en la expresión no se toman en cuenta.

9.3 Variables lógicas

Las variables lógicas deben comenzar por una letra o el símbolo "_" y después cualquier letra o número. No se admiten otros símbolos.

Las variables lógicas pueden valer 0 o 1. No hace falta declararlas antes de utilizarlas.

Si una variable lógica no tiene valor antes de ser utilizada, su valor es 0 (falso).

9.4 Operadores

Los operadores que podemos utilizar son:

- AND ("*")
- OR ("+")
- NOT ("-")
- ENTONCES ("->")
- DOBLEENTONCES ("<->")
- TAUTOLOGIA (tauto(expresion)): nos devuelve un 0 si la expresión no es una tautología y un 1 si sí lo es. No toma en cuenta el valor de las variables lógicas.
- CONTRADICCION (contra(expresion)): nos devuelve un 0 si la expresión no es una contradicción y un 1 si sí lo es. No toma en cuenta el valor de las variables lógicas.
- DECIDIBLE (deci(expresion)): nos devuelve un 0 si la expresión no es decidible y un 1 si sí lo es. No toma en cuenta el valor de las variables lógicas.

El orden de los operadores es: - ,* , + , -> , <->

9.5 Expresiones

Las expresiones lógicas constan de variables lógicas y constantes lógicas (1 o 0) relacionadas por medio de operadores. Para aclarar mejor todo, se pueden utilizar paréntesis.

9.6 Ejemplo de programa válido

El siguiente programa es un programa válido en L-0:

```
b=tauto( a -> a );

writestr("Es (a -> a) una tautologia? ");

writelog(b);

writeintro();

c=contra( a -> -a );

writestr("Es (a -> -a) una contradiccion? ");

writelog(c);

writeintro();

d=deci( a -> a );

writestr("Es decidible (a -> a) ? ");

writelog(d);

writeintro();

writestr("La tabla de verdad de (a + b * c) es : ");

writetabla( a + b * c );
```

El resultado de su ejecución debe ser:

```
Es (a -> a) una tautologia? 1

Es (a -> -a) una contradiccion? 0

Es decidible (a -> a) ? 1

La tabla de verdad de (a + b * c) es:

< a,b,c,: ? >

< 0 0 0 : 0 >
```

< 0 0 1 : 0 >

< 0 1 0 : 0 >

< 0 1 1 : 1 >

< 1 0 0 : 1 >

< 1 0 1 : 1 >

< 1 1 0 : 1 >

< 1 1 1 : 1 >

CAPÍTULO 10
ANÁLISIS LÉXICO DE L-0

10.1 Preparativos

Como vamos a utilizar la herramienta JLex (en el apéndice A se explicará la utilización de las diferentes herramientas), lo primero es crear un archivo de texto con la extensión .jlex. Por ejemplo, le vamos a llamar "L-0.jlex".

Le añadimos las importaciones que necesitamos, que son:

```
import java_cup.runtime.Symbol;

import java.io.IOException;
```

La primera importación permite utilizar la clase *Symbol* de la herramienta Cup. Esto es necesario porque ambas herramientas están conectadas, o más bien Cup utiliza Jlex cada vez que requiere un token.

Después se añaden las directivas adecuadas y se añaden dos métodos que sirven para devolver los tokens y para señalar la línea que se está procesando actualmente y el token actual. Esto es necesario para que en caso de error, se señale dónde está. Se utiliza una clase que se llama *InformacionCodigo* que se encarga de guardar en campos estáticos la línea actual y el token actual. Además, se devuelve el token EOF cuando se termina el fichero.

La directiva que devuelve el token EOF cuando se llega al final del fichero es:

```
%eofval{

{ return Token(sym.EOF); }

%eofval}
```

10.2 Patrones

Ahora debemos utilizar una serie de patrones para definir los posibles tokens que se admitirán:

```
EspacioOTerminador=[\ \t\f\r|\n|\r\n]

Digito = [0-9]

Letra = [A-Za-z_]

Num = [0-1]
```

```
Alfanumerico = ({Letra}|{Digito})

Identificador = ({Letra})({Alfanumerico})*

CadenaTexto = \"([\x20-\x21\x23-\xFE])*\"
```

Hay que destacar que un identificador se compone de una letra o el símbolo "_" seguido de una letra o un dígito.

También hay que señalar que una cadena de texto se compone de cualquier número de caracteres entre comillas dobles.

Los espacios o terminadores serán ignorados.

En cuanto a los números que se admiten sólo serán válidos el 0 y el 1.

10.3 Tokens válidos

En la siguiente sección se definirán los tokens válidos y se devolverá su código. Como en este caso no se necesitan estados, se utilizará sólo el estado inicial.

```
<YYINITIAL> "-" { return Token(sym.NOT); }

<YYINITIAL> "->" { return Token(sym.ENTONCES); }

<YYINITIAL> "<->" { return Token(sym.DOBLEENTONCES); }

<YYINITIAL> "=" { return Token(sym.ASIGNACION); }

<YYINITIAL> "*" { return Token(sym.AND); }

<YYINITIAL> "+" { return Token(sym.OR); }

<YYINITIAL> ";" { return Token(sym.PUNTOCOMA); }

<YYINITIAL> "(" { return Token(sym.PARENTIZQ); }

<YYINITIAL> ")" { return Token(sym.PARENTDER); }
```

Hay que tener cuidado con dejar un espacio en blanco entre cada sección dentro de la misma línea. Los nombres en mayúsculas son los identificadores de los diferentes tokens que luego se emplearán en la herramienta Cup.

El resto de tokens válidos son:

```
<YYINITIAL> tauto { return Token(sym.TAUTO); }

<YYINITIAL> contra { return Token(sym.CONTRA); }
```

```
<YYINITIAL> deci { return Token(sym.DECI); }

<YYINITIAL> writelog { return Token(sym.WRITELOG); }

<YYINITIAL> writestr { return Token(sym.WRITESTR); }

<YYINITIAL> writeintro { return Token(sym.WRITEINTRO); }

<YYINITIAL> writetabla { return Token(sym.WRITETABLA); }

<YYINITIAL> {CadenaTexto} { return Token(sym.CADENA,yytext()); }

<YYINITIAL> {Identificador} { return Token(sym.ID,yytext()); }

<YYINITIAL> {Num} { return Token(sym.NUMERO,yytext()); }

<YYINITIAL> {EspacioOTerminador}+ { }

<YYINITIAL>.|\n{System.err.println("Carácter no permitido: "+yytext()); }
```

La última línea señala que en caso de que el token que se lee no corresponda a ninguno de los anteriores es que será un error léxico y se sacará un mensaje indicándolo.

El método *yytext()*, dotado por el analizador, devuelve los caracteres de que se compone el lexema actual. Esto es necesario ya que por ejemplo un identificador no tiene un lexema previamente conocido, sino que sólo debe responder a un patrón.

Después de esto, se debe procesar este archivo con la herramienta JLex y se obtendrá el analizador léxico que será utilizado por la herramienta Cup.

CAPÍTULO 11

ANÁLISIS SINTÁCTICO DE L-0

11.1 Preparativos

El análisis sintáctico es el que guía todo el proceso de traducción. Se usa la herramienta Cup (explicada en el apéndice A). Tanto el análisis sintáctico como el semántico y la generación de código se incluyen en un único archivo de textos con extensión .cup. Lo vamos a llamar "L-0.cup".

Lo primero que aparece en el archivo son las importaciones de librerías necesarias.

A continuación, está la sección *action conde*. En esta sección es donde vamos a colocar los métodos que necesitemos en el proceso de traducción.

Después tenemos la sección *parser code*, que contendrá los métodos por defecto que provee Cup para el tratamiento de errores, el método que nos va a servir para inicializar el proceso de análisis y para lanzarlo (mediante el método *main*) y una serie de objetos de diferentes clases que se utilizarán como variables globales.

Por último, se encuentra la sección donde situamos los terminales, los no terminales, las precedencias y las reglas de la gramática (en las siguientes fases del análisis incluiremos insertadas las acciones semánticas y de generación de código).

Después de completado el archivo, se procesará y obtendremos el párser, que es el encargado del proceso de compilación.

11.2 Inicialización y arranque

Antes de nada, vamos a utilizar una clase para generar el código intermedio. En realidad, podríamos generarlo directamente pero como ese es el proceso que se seguirá en ejemplos más complejos, lo haremos así por uniformidad. Esta clase está dotada de una serie de métodos y atributos para manipular las diferentes instrucciones que se produzcan como consecuencia de la generación de código.

Debemos tener una variable global en la sección *parser code* para el código intermedio. Cuando inicializamos el proceso de análisis, creamos una instancia de esa clase. El constructor indica el nombre del archivo donde se generará el resultado del proceso de traducción. Ese archivo tendrá el mismo nombre del programa a traducir y la extensión .log.

```
parser code {:

static TablaSimbolos ts;
```

```
static CodigoIntermedio codigoIntermedio;

static String ficheroCodigoIntermedio=null;

static String codFuente;

public void error(String mensaje) {

System.out.println("ERROR lin:"+InformacionCodigo.linea+

" tok:"+InformacionCodigo.token+" => "+mensaje);

}

public static void inicializar() throws IOException {

ts = new TablaSimbolos();

codigoIntermedio = new CodigoIntermedio(

ficheroCodigoIntermedio);

codigoIntermedio.abrirFicheroEscritura();

}

public static void main(String args[]) throws Exception {

if (args.length != 1)

System.out.println(Textos.faltaFichero);

else {

try {

Yylex lexico = new Yylex(new FileReader(args[0]));

String name = (String)args[0];

codFuente= name.substring(0,name.lastIndexOf("."));

ficheroCodigoIntermedio = codFuente + ".log";

inicializar();

new parser(lexico).parse();
```

```
} catch (FileNotFoundException e1) {

System.out.println(Textos.ficheroNoAbierto);

}

}

}
// Muestra el texto de un error
public void report_error(String message, Object info) {
error(message);
}
// Muestra un error de sintaxis
public void syntax_error(Symbol actual) {
error("Error SINTACTICO");
}
// Muestra el texto de un error irrecuperable
public void report_fatal_error(String message, Object info) {
error(message);
}
// Muestra un mensaje cuando no se puede seguir //analizando
public void unrecovered_syntax_error(Symbol actual) {
}
:}
```

Vemos que se han incluido otras variables globales, una por ejemplo para guardar la tabla de símbolos y otra que indica el nombre del archivo donde está el código fuente y que se le pasa al programa mediante un argumento al método *main*.

Los mensajes para lanzar mensajes de error los suministra Cup. Aunque el método *error* se construye para sacar por pantalla los diferentes mensajes de error con información sobre la línea donde se han producido y los tokens implicados. La información al respecto se extrae de los atributos estáticos de la clase *InformacionCodigo*.

La clase *parser* es la encargada del proceso de compilación. En concreto el método *parse()*.

Toda la sección *parser code* es una sección de atributos y métodos estáticos, por lo que desde otra sección, nos referiremos a ellos poniendo el nombre de la clase delante. Es decir, por ejemplo si queremos referirnos al atributo *condigoIntermedio* desde un método de la sección *action code*, lo haremos así: *parser.codigoIntermedio*.

También se hace uso de una clase llamada *Textos*. Esta clase va a contener tantos atributos estáticos del tipo *String* como mensajes queramos sacar por pantalla.

11.3 Situación de terminales y no terminales

A continuación de la sección *parse code,* vamos a poner los nombres de los terminales y no terminales de nuestra gramática. Delante pondremos la palabra terminal o no terminal. Podemos agrupar varios separados por comas.

Los terminales los sacamos directamente del archivo .jlex y son estos:

```
terminal PUNTOCOMA;

terminal PARENTIZQ;

terminal PARENTDER;

terminal NOT;

terminal ENTONCES;

terminal DOBLEENTONCES;

terminal ASIGNACION;

terminal AND;

terminal OR;

terminal WRITESTR;

terminal WRITELOG;

terminal WRITEINTRO;

terminal WRITETABLA;

terminal TAUTO;

terminal CONTRA;

terminal DECI;

terminal String CADENA;

terminal String NUMERO;
```

```
terminal String ID;
```

Delante del nombre del terminal se pone el tipo al que pertenece. En nuestro caso son todas cadenas de caracteres; por ejemplo, el valor 1 estaría representado por la cadena "1", el identificador *ab*1 por la cadena "ab1", etc.

Otra cosa que debemos declarar es la precedencia de los operadores. Debemos situarlos en orden de menor a mayor precedencia. En este caso sería:

```
precedence left DOBLEENTONCES;

precedence left ENTONCES;

precedence left OR;

precedence left AND;

precedence right NOT;
```

Delante del operador debemos poner si la precedencia es por la izquierda o por la derecha.

Ahora es el momento de crear las reglas necesarias para el análisis sintáctico. Al mismo tiempo, crearemos los no terminales necesarios. Cada vez que creemos un no terminal, deberemos declararlo antes de poder utilizarlo en una regla. Por ejemplo, un programa en L-0 consta de una serie de sentencias. Las primeras reglas podrían ser:

```
non terminal proa;

non terminal sentencias;

non terminal sentencia;

non terminal nulo;

prog ::= sentencias;

sentencias ::= sentencias sentencia | nulo;

nulo ::= ;
```

Las declaraciones se suelen poner todas juntas y junto a las de los terminales. Las reglas se suelen poner al final de las declaraciones.

Lo primero que hemos definido es que un programa consta de sentencias. Las sentencias constan de una serie de elementos llamados *sentencia* o nada. Ahora definiremos qué es una sentencia.

11.4 Sentencias

Las sentencias en L-0 pueden ser de diferentes tipos, según la especificación que hemos hecho del lenguaje. En concreto:

```
sentencia ::= asig PUNTOCOMA |

sentwritestr PUNTOCOMA |

sentwritelog PUNTOCOMA |

sentwriteintro PUNTOCOMA |

sentwritetabla PUNTOCOMA;
```

Tenemos la sentencia de asignación, la sentencia de escribir una cadena, la de escribir el contenido de una variable lógica, la de escribir un salto de carro y la de escribir una tabla de verdad.

Todos estos no terminales habrá que incluirlos en la declaración de no terminales. Los no terminales pueden también tener tipo como los terminales, pero en principio ninguno de estos tiene tipo.

11.5 Expresiones

Las expresiones son quizás la parte más importante. Mediante las expresiones podemos operar con variables y con valores. Las expresiones se combinan también mediante los operadores para obtener otras expresiones.

En nuestro caso, vamos a utilizar una clase para guardar la información de una expresión. En concreto, vamos a guardar el contenido. En lenguajes compilados no se guarda el contenido sino más información, además de la dirección en memoria donde estará el contenido. Vamos a conservar esta clase, aunque se podría no utilizar, para habituarnos a que es necesaria para lenguajes compilados.

El no terminal *expresión* se debe declarar en la zona de no terminales con el tipo *Expresion* y las reglas con las expresiones son:

```
non terminal Expresion expresion;

//Expresiones

expresion ::= NOT expresion

|
```

```
expresion DOBLEENTONCES expresion
|
expresion ENTONCES expresion
|
expresion AND expresion
|
expresion OR expresion
|
NUMERO
|
PARENTIZQ expresion PARENTDER
|
identificador
|
senttauto
|
sentcontra
|
sentdeci
;
```

Vemos que aparecen otros terminales y no terminales. Se definirán y explicarán más adelante.

No hemos incluido las tres últimas sentencias en la sección de sentencias porque en realidad son funciones que devuelven un valor con el que se puede operar en las expresiones.

Una expresión puede ser, como se indica en la gramática, una constante lógica, un NUMERO (sólo el 0 o el 1). También puede ser una variable lógica (identificador).

identificador ::= **ID**;

Un identificador no es más que un elemento terminal. Contendrá en cada momento el lexema que represente a una variable.

En cuanto al tipo al que pertenece una expresión, se verá su utilidad en el análisis semántico y la generación de código.

11.6 Asignación

Una asignación consta de una parte izquierda, que debe ser una variable lógica y una parte derecha que es una expresión. Por lo tanto, la regla de la gramática sería:

asig ::= identificador **ASIGNACION** expresión;

11.7 Sentencias de escritura

Las sentencias de escritura son:

```
//Sentencia writelog

sentwritelog ::= WRITELOG PARENTIZQ expresion PARENTDER;

//Sentencia writeintro

sentwriteintro ::= WRITEINTRO PARENTIZQ PARENTDER;

//Sentencia writestr

sentwritestr ::= WRITESTR PARENTIZQ CADENA PARENTDER;

//Sentencia writetabla

sentwritetabla ::= WRITETABLA PARENTIZQ matriz PARENTDER;
```

Las tres primeras son evidentes, pero en la última aparece un no terminal que indica una matriz. Esta matriz es en realidad una tabla de verdad. Por lo que se puede escribir una tabla de verdad.

11.8 Tablas de verdad

Nuestro lenguaje permite imprimir tablas de verdad. No tenemos más que indicar una expresión lógica y se podrá imprimir su tabla de verdad (independientemente de los valores que las variables lógicas tengan en ese momento).

La operatividad es parecida a la de las expresiones. Sólo que en este caso sólo se permite incluir algunos operadores y variables lógicas (no constantes).

```
matriz ::= NOT matriz

|

matriz DOBLEENTONCES matriz

|

matriz ENTONCES matriz

|

matriz AND matriz

|

matriz OR matriz

|

PARENTIZQ matriz PARENTDER

|

identificador

;
```

11.9 Funciones

Aunque les hemos llamado sentencias, en realidad devuelven una constante lógica, es decir, un 0 o un 1 en función de que la tabla de verdad que se les pasa como argumento cumpla una serie de condiciones.

```
//Sentencia tautologia

senttauto ::= TAUTO PARENTIZQ matriz PARENTDER;

//Sentencia contradiccion

sentcontra ::= CONTRA PARENTIZQ matriz PARENTDER;
```

```
//Sentencia decidible


sentdeci ::= DECI PARENTIZQ matriz PARENTDER;
```

Hemos hablado poco de los tipos de los terminales y no terminales, pero es que los tipos se utilizarán sólo a partir del análisis semántico.

Una cosa que no hay que olvidar es que todos los no terminales que aparecen deben estar declarados en la zona de declaración de terminales y no terminales

CAPÍTULO 12

ANÁLISIS SEMÁNTICO Y GENERACIÓN DE CÓDIGO DE L-0

12.1 Preparativos

En este lenguaje, el análisis semántico prácticamente no existe ya que el análisis sintáctico impide poder cometer errores semánticos.

Por ejemplo, como no hay que declarar las variables lógicas antes de ser usadas, no hay que comprobar al usar una variable lógica si antes había sido declarada o no. Cuando aparece una variable lógica, se mira en la tabla de símbolos y si no estaba, se añade con el valor que tenga. Si estaba, solamente se pone con el valor que tenga.

Por lo tanto, lo único que hay que hacer es gestionar la tabla de símbolos y generar el código oportuno.

12.2 Tabla de símbolos

En este lenguaje tan sencillo, la tabla de símbolos es también muy simple. Contiene entradas para las variables. Cada entrada es un objeto de la clase *Simbolo*. Esta clase contiene dos atributos y una serie de métodos para su manipulación. Los dos atributos contienen información del nombre de la variable lógica y el contenido (un 0 o un 1).

```
public class Simbolo {

String id;

int valor;

Simbolo() {

id="";

valor=0;

}

Simbolo(String id,int valor) {
```

```
this.id=id;

this.valor=valor;

}

int getValor() {

return valor;

}

String getId() {

return id;

}

}
```

La tabla de símbolos es una estructura que contiene una lista de símbolos (objetos de la clase *Simbolo*). Los métodos que utiliza son también los mínimos, insertar símbolos, ver el valor de un símbolo, comprobar si un símbolo existe, sacar un listado de la tabla y cambiar el valor de un símbolo. No existe ningún método para sacar un símbolo ya que no es necesario.

```
//Esta clase contiene la tabla de simbolos

import java.util.*;

public class TablaSimbolos {

public HashMap tabla;

public int tamano;

TablaSimbolos() {

tabla=new HashMap();

tamano=0;

}

//Inserta un simbolo

public void insertar(String identificador,int valor) {
```

```
if(!existe(identificador)){

tamano++;

tabla.put(new Integer(tamano),new Simbolo(identificador,valor));

}

}

//¿Existe un simbolo?

public boolean existe(String identificador) {

boolean retorno=false;

for(int i=1;i<=tamano;i++) {

Simbolo s=new Simbolo();

s=(Simbolo)tabla.get(new Integer(i));

if(s.getId().equals(identificador)) {

retorno=true;
break;
}
}
return retorno;
}
//Obtener valor
public int getValor(String identificador) {
int retorno=0;
for(int i=1;i<=tamano;i++) {
Simbolo s=new Simbolo();
s=(Simbolo)tabla.get(new Integer(i));
if(s.getId().equals(identificador)) {
retorno=s.getValor();
break;
}
}
return retorno;
}
public void putSimbolo(String identificador,int valor) {
if(!existe(identificador)) insertar(identificador,valor);
else {
for(int i=1;i<=tamano;i++) {
Simbolo s=new Simbolo();
s=(Simbolo)tabla.get(new Integer(i));
```

```
if(s.getId().equals(identificador)) {
tabla.put(new Integer(i),new Simbolo(identificador,valor));
break;
}
}
}
}
public void listar() {
for(int i=1;i<=tamano;i++) {
Simbolo s=new Simbolo();
s=(Simbolo)tabla.get(new Integer(i));
System.out.println(s.getId()+" "+s.getValor());
}
}
}
```

Como ya se ha comentado en un capítulo anterior, tenemos una variable global que guarda la tabla de símbolos.

Veremos a continuación la generación de código.

12.3 Tratamiento de expresiones

La información sobre las expresiones va a estar almacenada en un objeto de la clase *Expresion*. En realidad sólo se guarda el valor de la expresión, es decir, su contenido. Por ejemplo, si una expresión representa a una variable lógica, la clase almacenará el valor de esa variable lógica, esto es, un 0 o un 1.

El no terminal *expresion* es de la clase *Expresion*.

non terminal Expresion expresion;

Las acciones semánticas serán:

```
//Expresiones
expresion ::= NOT expresion:e
{: RESULT=semNO(e); :}
| expresion:e1 DOBLEENTONCES expresion:e2
{: RESULT=semDOBLEENTONCES(e1,e2); :}
| expresion:e1 ENTONCES expresion:e2
{: RESULT=semENTONCES(e1,e2); :}
| expresion:e1 AND expresion:e2
{: RESULT=semAND(e1,e2); :}
| expresion:e1 OR expresion:e2
{: RESULT=semOR(e1,e2); :}
| NUMERO:i
{: RESULT=semNuevoNumero(i); :}
| PARENTIZQ expresion:e PARENTDER
{: RESULT=e; :}
| identificador:i
{: RESULT=semValorIdentificador(i); :}
```

```
| senttauto:s
{: RESULT=semTautologia(s); :}
| sentcontra:s
{: RESULT=semContradiccion(s); :}
| sentdeci:s
{: RESULT=semDecidible(s); :}
;
```

Se utiliza una serie de métodos que deben estar implementados en la sección *action code*. Estos métodos deben devolver un objeto de la clase *Expresion*. Este objeto debe contener un valor lógico como resultado de la manipulación que se haga con los parámetros que se le pasan al método.

Comencemos con el método *semNO(e):*

```
Expresion semNO(Expresion e) {
int valor=e.getContenido();
if(valor==0) valor=1; else valor=0;
return new Expresion(valor);
}
```

Se le pasa como parámetro un objeto del tipo *Expresion*. Y se devuelve otro objeto del mismo tipo pero con el contenido negado.

Las demás manipulaciones lógicas son:

```
Expresion semAND(Expresion e1,Expresion e2) {
int v1=e1.getContenido();
int v2=e2.getContenido();
if(v1*v2==0) v1=0; else v1=1;
return new Expresion(v1);
}
Expresion semOR(Expresion e1,Expresion e2) {
int v1=e1.getContenido();
int v2=e2.getContenido();
if( v1!=0 || v2!=0) v1=1; else v1=0;
return new Expresion(v1);
}
Expresion semENTONCES(Expresion e1,Expresion e2) {
int v1=e1.getContenido();
int v2=e2.getContenido();
return semOR(semNO(e1),e2);
}
Expresion semDOBLEENTONCES(Expresion e1,Expresion e2) {
int v1=e1.getContenido();
int v2=e2.getContenido();
if((v1==0 && v2==0) || (v1==1 && v2==1)) v1=1;else v1=0;
return new Expresion(v1);
}
```

El método *semNuevoNumero(i)* se encarga de tomar el lexema del terminal ***NUMERO*** y meterlo en una expresión que es la que se devuelve. Por ejemplo, si el

lexema del terminal es “1”, metería en una nueva expresión el valor 1 y la devolvería.

```
Expresion semNuevoNumero(String s) {

int valor=0;

valor=Integer.parseInt(s);

return new Expresion(valor);

}
```

El método *semValorIdentificadir(i)* devuelve el valor que tiene en ese momento la variable lógica i (el valor lo busca en la tabla de símbolos) metido en un objeto de la clase *Expresion*.

```
Expresion semValorIdentificador(String i) {

return new Expresion(semGetValor(i));

}

int semGetValor(String identificador){

if(parser.ts.existe(identificador))

return parser.ts.getValor(identificador);

else return 0;

}
```

Si la variable no está aún en la tabla de símbolos, le supone el valor lógico falso (0).

Hay tres funciones que también se utilizan en las expresiones. Las tres devuelven valores de verdad (0 o 1).

12.3.1 La función *tautología*

Esta función devuelve un valor de verdad (0 o 1), dependiendo de si la matriz que representa una tabla de verdad tiene todos sus valores a 1 o no.

La matriz es la manera de representar una tabla de verdad de una proposición lógica. Por ejemplo, la tabla de verdad de la proposición lógica *x OR y* sería:

```
0 0 0
0 1 1
1 0 1
```

Esta tabla se guarda en una matriz de dimensión 2 con valores de tipo cadena. En este caso, sería esta matriz:

“x” “y” null
“0” “0” “0”
“0” “1” “1”
“1” “0” “1”
“1” “1” “1”

```
String[][] matriz = new String[5][3];
```

Por lo tanto, si todos los valores de la última columna de la matriz son “1” se trata de una tautología, y por lo tanto habría que devolver un 1.

La regla de la gramática que devuelve la matriz de cadenas es:

```
//Sentencia tautologia

senttauto ::= TAUTO PARENTIZQ matriz:m PARENTDER {: RESULT=m; :};
```

Y el método que devuelve una expresión con un 0 o un 1 dependiendo de si la matriz no es o sí es una tautología es:

```
Expresion semTautologia(String[][] m) {

int resultado=1;

for(int fila=1;fila<m.length;fila++) {

if((m[fila][(m[0].length)-1]).equals(“0”)) {

resultado=0;

break;

}

}

return new Expresion(resultado);

}
```

12.3.2 La función *contradicción*

Esta función se utiliza para ver si una tabla de verdad de una proposición contiene en su última columna todo ceros, lo que implicaría que la proposición es una

contradicción. Se hace lo mismo que para la tautología.

```
//Sentencia contradiccion

sentcontra ::= CONTRA PARENTIZQ matriz:m PARENTDER {: RESULT=m; :};

Expresion semContradiccion(String[][] m) {

int resultado=1;

for(int fila=1;fila<m.length;fila++) {

if((m[fila][(m[0].length)-1]).equals("1")) {

resultado=0;

break;

}

}

return new Expresion(resultado);

}
```

12.3.3 La función *decidible*

Con esta función se puede comprobar si una proposición es decidible, es decir, si hay alguna combinación en su tabla de verdad en la que salga un 1.

La regla de la gramática es:

```
//Sentencia decidible

sentdeci ::= DECI PARENTIZQ matriz:m PARENTDER {: RESULT=m; :};
```

El método empleado es:

```
Expresion semDecidible(String[][] m) {

int resultado=0;

for(int fila=1;fila<m.length;fila++) {

if((m[fila][(m[0].length)-1]).equals("1")) {
```

```
resultado=1;

break;

}

}

return new Expresion(resultado);

}
```

Hay que distinguir entre las reglas de la gramática, lo primero que aparece y los métodos lanzados, que deben estar definidos en la sección *action code*.

12.4 Operaciones con tablas de verdad

Es posible combinar mediante operadores diferentes tablas de verdad para obtener una tabla de verdad como resultado. Esto es necesario para poder realizar las tablas de verdad a partir de las variables lógicas. Las tablas de verdad, como se ha comentado antes, están representadas en matrices bidimensionales.

Las matrices que representan las tablas de verdad no sólo pueden operarse entre sí para obtener otras más complejas, sino que se pueden imprimir.

Las reglas que permiten esto son:

```
matriz ::= NOT matriz:m {: RESULT=semMatrizNOT(m); :}

| matriz:m1 DOBLEENTONCES matriz:m2

{: RESULT=semMatrizDOBLEENTONCES(m1,m2); :}

| matriz:m1 ENTONCES matriz:m2

{: RESULT=semMatrizENTONCES(m1,m2); :}

| matriz:m1 AND matriz:m2

{: RESULT=semMatrizAND(m1,m2); :}

| matriz:m1 OR matriz:m2

{: RESULT=semMatrizOR(m1,m2); :}

| PARENTIZQ matriz:m PARENTDER

{: RESULT=m; :}

| identificador:i

{: RESULT=semMatrizIdentificador(i); :}

;

//Sentencia writetabla
```

```
sentwritetabla ::= WRITETABLA PARENTIZQ matriz:m PARENTDER
{: semEscribeTabla(m); :};
```

El código que hace estas operaciones con las matrices tiene poco que explicar.

```
String[][] semMatrizAND(String[][] m1,String[][] m2) {
Vector v=new Vector();
for(int columna=0;columna<(m1[0].length)-1;columna++) {
if(!v.contains(m1[0][columna])) v.addElement(m1[0][columna]);
}
for(int columna=0;columna<(m2[0].length)-1;columna++) {
if(!v.contains(m2[0][columna])) v.addElement(m2[0][columna]);
}
int tamano=elevado(2,v.size());
String binario;
String[][] r=new String[tamano+1][v.size()+1];
for(int columna=0;columna<(r[0].length)-1;columna++)
r[0][columna]=(String)v.elementAt(columna);
for(int fila=1;fila<r.length;fila++) {
StringBuffer longitud=new StringBuffer();
binario=Integer.toBinaryString(fila-1);
for(int i=0;i<v.size()-binario.length();i++)
longitud.append("0");
longitud.append(binario);
for(int columna=0;columna<v.size();columna++) {
r[fila][columna]=longitud.substring(columna,columna+1);
}
}
for(int fila=1;fila<m1.length;fila++) {
for(int columna=0;columna<(m1[0].length)-1;columna++) {
m1[fila][columna]=m1[0][columna]+"="+m1[fila][columna];
}
}
for(int fila=1;fila<m2.length;fila++) {
for(int columna=0;columna<(m2[0].length)-1;columna++) {
m2[fila][columna]=m2[0][columna]+"="+m2[fila][columna];
}
}
for(int fila=1;fila<r.length;fila++) {
StringBuffer c=new StringBuffer();
for(int columna=0;columna<(r[0].length)-1;columna++) {
c.append(r[0][columna]+"="+r[fila][columna]);
}
int r1=0;
boolean resultado1;
for(int fila1=1;fila1<m1.length;fila1++) {
resultado1=true;
for(int              columna=0;columna<(m1[0].length)-1;columna++)              {
if(c.toString().indexOf(m1[fila1][columna])<0) {
resultado1=false;
break;
}
```

```
}
if(resultado1==true) {
r1=Integer.parseInt(m1[fila1][(m1[0].length)-1]);
break;
}
}
int r2=0;
boolean resultado2;
for(int fila2=1;fila2<m2.length;fila2++) {
resultado2=true;
for(int columna=0;columna<(m2[0].length)- 1;columna++) {
if(c.toString().indexOf(m2[fila2][columna])<0) {
resultado2=false;
break;
}
}
if(resultado2==true) {
r2=Integer.parseInt(m2[fila2][(m2[0].length)-1]);
break;
}
}
if(r1*r2==0) r[fila][(r[0].length)-1]="0";
else r[fila][(r[0].length)-1]="1";
}
return r;
}
// …….
// …….
```

12.5 La asignación

Esta operación es muy sencilla. Consiste en asignar un valor a una variable lógica. El valor viene de una expresión. Es decir, se copia el valor de la expresión en la entrada de la tabla de verdad de la variable lógica. Si la variable lógica no está aún declarada en la tabla de símbolos, se añadirá previamente. Si ya estaba, se sustituirá su valor por el que proviene de la expresión.

La regla de la gramática es:

```
asig ::= identificador:i ASIGNACION expresion:e
{: semAsignar(i,e.getContenido()); :};
```

Los métodos utilizados y situados en la sección *action code* son:

```
void semPutSimbolo(String identificador,int valor){
parser.ts.putSimbolo(identificador,valor);
}
void semAsignar(String i,int v) {
semPutSimbolo(i,v);
}
```

12.6 Operaciones de impresión

Con estas operaciones se pretende poder sacar al archivo de resultados diferentes valores o texto.

Las reglas utilizadas son:

```
//Sentencia writelog
sentwritelog ::= WRITELOG PARENTIZQ expresion:e
PARENTDER {: semEscribir(String.valueOf(e.getContenido())); :};
//Sentencia writeintro
sentwriteintro ::= WRITEINTRO PARENTIZQ
PARENTDER {: semEscribirLn(“”); :};
//Sentencia writestr
sentwritestr ::= WRITESTR PARENTIZQ CADENA:c
PARENTDER {: semEscribir(c.toString()); :};
//Sentencia writetabla
sentwritetabla ::= WRITETABLA PARENTIZQ matriz:m
PARENTDER {: semEscribeTabla(m); :};
```

writelog imprime el valor de una expresión lógica.

writeintro provoca un salto de línea en la impresión.

writestr imprime la cadena que se le pasa como argumento.

writetabla imprime la matriz o tabla de verdad de una proposición.

Los métodos utilizados, y que se implementan en la sección *action code* son:

```
void semEscribir(String cadena) {
parser.codigoIntermedio.escribir(cadena);
}
void semEscribirLn(String cadena) {
parser.codigoIntermedio.escribirLn(cadena);
}
void semEscribeTabla(String[][] m) {
StringBuffer t=new StringBuffer();
t.append(“< “);
for(int columna=0;columna<(m[0].length)-1;columna++) {
t.append(m[0][columna]+”,”);
}
t.append(“: ? >”);
parser.codigoIntermedio.escribirLn(t.toString());
for(int fila=1;fila<m.length;fila++) {
StringBuffer s=new StringBuffer();
s.append(“< “);
```

```
for(int columna=0;columna<(m[0].length)-1;columna++) {
s.append(m[fila][columna]+" ");
}
s.append(": "+m[fila][m[0].length-1]+" ");
s.append(">");
parser.codigoIntermedio.escribirLn(s.toString());
}
}
```

Por último, el fin del programa provoca una acción que cierra el archivo donde se guarda el resultado de la traducción.

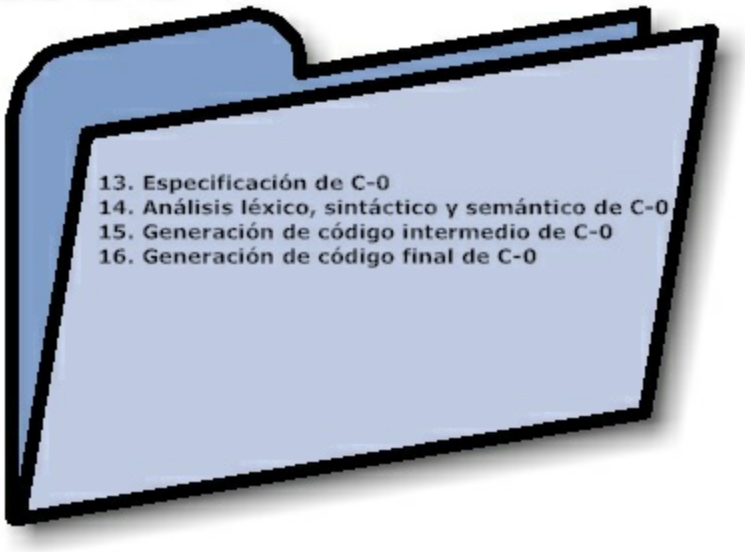
La regla es:

prog ::= sentencias {: semFinPrograma(); :};

El método utilizado es:

```
void semFinPrograma() {
parser.codigoIntermedio.cerrarFicheroEscritura();
}
```

Parte III. Implementación de C-0



CAPÍTULO 13

ESPECIFICACIÓN DE C-0

13.1 Introducción

En este capítulo vamos a describir el lenguaje del que vamos a crear el compilador. Como el objetivo de esta práctica es introducirnos al diseño e implementación de compiladores, vamos a crear un compilador de un lenguaje extremadamente sencillo. Lo llamaremos **C-0**.

Figura 13.1

13.2 Tokens

También podemos llamarlos lexemas. Un programa es una secuencia de tokens de varios tipos: operadores, delimitadores, identificadores, palabras reservadas, constantes, etc. Por ejemplo, la secuencia de caracteres abc123 representa un único token. Dos tokens se pueden separar mediante un espacio en blanco, un operador o un delimitador.

13.3 Constantes

En **C-0** utilizaremos sólo un tipo de constantes, las constantes de cadena. Una constante de cadena es cualquier ristra de caracteres tipo ASCII entre comillas dobles. El carácter "\" sirve para destacar caracteres espaciales, como puede ser \n que representa un salto de línea.

13.4 Operadores y delimitadores

Se consideran los siguientes operadores y delimitadores:

- Delimitadores : **" ( ) ; { }**
- Operadores aritméticos : **+** (suma) **–** (resta) ***** (producto) **/** (división)
- Operadores relacionales : **<** (menor) **>** (mayor) **==** (igual) **¡=** (distinto)
- Operadores lógicos : **| |** (or) **&&** (and)

- Operador de asignación : =

La precedencia de los operadores es la siguiente de mayor a menor prioridad:

( ) * / + - > < == ¡= && || =

13.5 Identificadores y palabras reservadas

Un identificador consiste en una secuencia de caracteres, dígitos o caracteres de subrayado que comienzan por una letra o por un carácter de subrayado. Los identificadores son utilizados para nombrar las entidades del programa como las variables. Dos identificadores con los mismos caracteres sólo que unos están en mayúsculas y otros están en minúsculas son considerados diferentes.

Las palabras reservadas son identificadores que tienen un significado especial. Las palabras reservadas de **C-0** son:

main if while else putw puts int break

13.6 Tipos de datos

Sólo vamos a considerar un tipo de dato, el tipo entero, que se representa con la palabra reservada ***int***.

13.7 Sentencias de control de flujo

Se consideran dos tipos de sentencias:

- ***if(condicion)sentencia1;else sentencia2;***
- ***while(condicion)sentencia;***

La parte del ***else*** no siempre aparece, por lo que se puede omitir.

Si aparece un ***break*** dentro de un bloque ***while***, se deberá salir inmediatamente de dicho bloque.

13.8 Instrucciones de entrada-salida

Sólo se consideran funciones de salida por pantalla: ***puts*** para salida de cadenas de caracteres y **p*utw*** para salida de expresiones enteras.

13.9 Declaración de variables

La declaración de variables se debe hacer al principio de las sentencias del programa. Las variables sólo pueden ser de tipo entero y se declaran de una en una. Se pone delante del identificador de la variable el tipo. En nuestro caso, sólo ***int***.

Las variables no se inicializan al ser declaradas, por lo que si no les damos valor a lo largo del programa, su valor será impredecible.

13.10 Programa principal

En **C-0** no se admiten subprogramas, sólo el programa principal. Este comienza con ***main() {*** y termina con ***}***.

13.11 Sentencia if-then-else

Esta sentencia consta de la palabra reservada ***if*** seguida de la condición entre paréntesis, luego una llave abierta, las sentencias y llave cerrada. Opcionalmente se le puede encadenar otro bloque con ***else***, llave abierta, sentencias y llave cerrada. Por ejemplo:

```
if(x==1) {

y=12;

} else {

y=13;

}
```

O también:

```
if(x!=12) {

y=32;

}
```

13.12 Sentencia while

Esta sentencia consta de la palabra ***while*** seguida de la condición entre paréntesis y un grupo de sentencias entre llaves. Por ejemplo:

```
while(x>2) {

x=x+32;

}
```

13.13 Ejemplo de programa válido

```
int x;

int y;

main() {

x=6;

y=1;

while(x>0) {

y=y*x;

x=x-1;

}

puts("El factorial de 6 es : ");

putw(y);

puts("\n");

puts("El valor de x+1 es : ");

putw(x+1);

}
```

El resultado de la ejecución de este programa sería:

```
El factorial de 6 es: 720

El valor de x+1 es: 1
```

CAPÍTULO 14

ANÁLISIS LÉXICO, SINTÁCTICO Y SEMÁNTICO DE C-0

14.1 Análisis léxico

El análisis léxico lo vamos a realizar con ayuda de JLex.

Llamaremos *C-0.jlex* al archivo donde tenemos nuestro análisis léxico.

yytext() devuelve la cadena de caracteres leídos. *yyline* va señalando la línea actual de la entrada.

Conforme se van leyendo caracteres del fichero a procesar, se van suministrando los tokens y se van guardando en un objeto de la clase *InformacionCodigo* el último token leído y la línea actual (esto se hace para en el caso de que haya un error, podamos informar al usuario dónde está y en qué token está).

La clase *InformacionCodigo* es:

```
// Esta clase se utiliza para guardar y recuperar

//informacion de punto del codigo

// fuente por el que se va analizando

class InformacionCodigo {

public static int linea;

public static String token;

public static void guardarInformacionCodigo(int l,String t) {

linea = l;

token = t;

}

}
```

Una vez creados los dos archivos, los pasos a seguir son:

1. Procesamos el archivo *jlex* : *java JLex.Main Compilador.jlex*
2. Renombramos el archivo *Compilador.jlex.java* a *Yylex.java*
3. Compilamos los archivos de tipo Java: *javac Yylex.java sym.java InformacionCodigo.java*

14.2 Análisis sintáctico

El análisis sintáctico es una tarea que vamos a realizar con la ayuda de Cup.

Lo primero que vamos a hacer es crear el esqueleto del archivo *C-0.cup*.

En la sección *parse code* ponemos cuatro métodos que nos suministra Cup y dos de ellos los hemos reescrito para que saquen un mensaje de error junto con la línea donde se ha producido el error y el lexema causante del mismo (esta información la obtenemos de la clase *InformacionCodigo*).

También hemos incluido el método *main* que acepta un argumento con el nombre del programa a compilar.

Observamos que ya fuera de esa sección declaramos los terminales de la gramática (que son los mismos que hemos definido en el archivo *Compilador.jlex*) y los no terminales, que nos van a servir para definir las reglas de nuestra gramática.

Además, definimos la precedencia de los operadores (de menor a mayor), para que en caso de poder aplicar más de uno de ellos, se elija primero el de más precedencia (por ejemplo, no es lo mismo aplicar la suma antes que el producto que al revés: 1+2*3=7 si la precedencia de * es mayor que la de +, en caso contrario sería 9).

NOTA. Primero se definen los de menor precedencia y luego los de mayor.

La sentencia *start with* nos indica cuál es la primera regla que se va a procesar.

Bueno, ya tenemos nuestra primera versión de la gramática. En esta primera versión sólo hemos definido una regla que dice que un programa consta primero de una sección de declaraciones (puede no tener ninguna declaración) y luego una sección de código (Cuerpo) obligatoria.

Ahora ya podemos procesar los archivos *.jlex* y *.cup* y compilar las clases obtenidas (no hay que preocuparse porque al procesar el archivo *.cup* salgan algunos *warnings* avisándonos de que hay terminales no usados, es normal, aún no hemos terminado de definir la gramática).

Es hora de ir completando nuestra gramática. Ya tenemos la primera regla, que un programa se compone de una parte declarativa y de un cuerpo. En la parte declarativa definimos las variables que luego vamos a utilizar en el cuerpo. Según nuestra gramática, sólo hay un tipo de datos, el entero. Además, cada variable se declara por separado. Pues bien, con estos datos, ya podemos crear las reglas para las declaraciones de variables.

Una declaración se compone del terminal INT seguido del terminal IDENTIFICADOR y después un punto y coma. La regla sería:

Declaracion ::= INT IDENTIFICADOR PTOCOMA;

Pero se admiten varias declaraciones, por lo que debemos crear otra regla para ello:

Declaraciones ::= Declaracion Declaraciones | Declaracion;

Esta última regla indica que una serie de declaraciones puede ser o bien una sola declaración o una declaración a continuación de las declaraciones anteriores. Debemos añadir el nuevo no terminal y las nuevas reglas a nuestro archivo *.cup*.

```
non terminal Declaracion;

Declarciones ::= Declaracion Declaraciones | Declaracion;

Declaracion ::= INT IDENTIFICADOR PTOCOMA;
```

Ahora toca definir las reglas para el cuerpo del programa.

Tal y como dijimos en la definición de nuestro lenguaje, el cuerpo debe comenzar con:

```
main() {

…..

}
```

Es decir, debemos crear otra regla:

Cuerpo ::= MAIN LPAREN RPAREN LLLAVE BloqueSentencias RLLAVE;

El bloque de sentencias es otro no terminal. Nos quedaría así:

```
non terminal BloqueSentencias;

Cuerpo ::= MAIN LPAREN RPAREN LLLAVE BloqueSentencias RLLAVE;
```

Ahora ya podemos ver si un pequeño programa de ejemplo es procesado. Para poder compilar un programa, debemos utilizar las clases que hemos creado, además de las clases que suministra Cup. Es decir, que en el CLASSPATH debemos incluir tanto la ruta a nuestras clases como la ruta al directorio donde está Cup. Luego, ya podemos compilar nuestro programa (por ejemplo, le llamaremos *programa.c*).

java parser programa.c

Probemos con el siguiente programa:

```
int x;
```

```
int y;

main() {

}
```

Que no debe sacar ningún mensaje de error.

Si no ha habido ningún error, todo irá bien. Ahora nos toca definir las reglas para las sentencias. Vamos a permitir que no haya ninguna sentencia (el cuerpo esté vacío) o que haya una o más sentencias.

```
non terminal Sentencias, Sentencia;

BloqueSentencias ::= Sentencias | ;

Sentencias ::= Sentencias Sentencia | Sentencia;
```

Vemos que hemos definido dos nuevos no terminales y dos nuevas reglas, una para indicar que el bloque de sentencias puede tener sentencias o estar vacío y otra para indicar que las sentencias se componen de una o más sentencias.

Ya podemos definir las reglas para las sentencias. Una sentencia puede ser un bloque *if-then-else*, un bloque *while*, una asignación, imprimir un entero o imprimir una cadena. Para todo ello, vamos a definir lo que son las expresiones.

Las expresiones son operaciones con números e identificadores (por ejemplo, la suma de un número y el contenido de una variable).

```
non terminal Expresion;
Expresion ::= Expresion SUMA Expresion |
Expresion RESTA Expresion |
Expresion PRODUCTO Expresion |
Expresion DIVISION Expresion |
ENTERO |
ID |
LPAREN Expresion RPAREN ;
```

Vemos que una expresión puede estar formada por operaciones entre expresiones, una expresión entre paréntesis, un identificador o un número entero.

Ahora vamos a ver cómo se definen las condiciones en los bloques *if-then-else* y *while*.

```
non terminal Condicion;
Condicion ::= Expresion OR Expresion |
Expresion AND Expresion |
Expresion IGUAL Expresion |
Expresion DISTINTO Expresion |
Expresion MAYOR Expresion |
Expresion MENOR Expresion |
LPAREN Condicion RPAREN;
```

Ya vemos que una condición puede componerse de varias combinaciones de expresiones o incluso de una condición entre paréntesis. Como en nuestra gramática no trabajamos con tipos booleanos, una condición no puede ser un identificador sólo (ya que el identificador no puede ser booleano).

Ya estamos en condiciones de ver cómo serían las reglas para los bloques.

```
non terminal SentIf, SentElse;

SentIf ::= IF LPAREN Condicion RPAREN

LLLAVE BloqueSentencias RLLAVE SentElse;

SentElse ::= ELSE LLLAVE BloqueSentencias RLLAVE |;
```

Vemos que se han declarado dos nuevos no terminales y también que la sentencia *if* debe tener la condición entre paréntesis y el bloque de sentencias entre llaves, y opcionalmente puede tener un bloque *else*. También podemos apreciar que puede haber *if* anidados.

Ahora, definimos la sentencia *while*.

```
non terminal SentWhile;

SentWhile ::= WHILE LPAREN Condicion RPAREN

LLLAVE BloqueSentencias RLLAVE;
```

Esta sentencia se compone de una condición entre paréntesis y un bloque de sentencias entre llaves. También se permite el anidamiento.

Vamos ahora con la sentencia de asignación. Se compone de una parte izquierda que debe ser un identificador y una parte derecha que debe ser una expresión. La sentencia debe terminar con un punto y coma.

```
non terminal SentAsignacion;

SentAsignacion ::= ID ASIGNAR Expresion PTOCOMA;
```

Otra de las posibles sentencias es la de imprimir tanto enteros como cadenas.

```
non terminal SentPutw, SentPuts;

SentPutw ::= PUTW LPAREN Expresion RPAREN PTOCOMA;

SentPuts ::= PUTS LPAREN CADENATEXTO RPAREN PTOCOMA;
```

Se puede imprimir tanto una cadena de texto como una expresión (mediante *puts*

y *putw*).

Por último, nos queda la sentencia para salir de un bloque *while*. En este caso, esta sentencia debería admitirse sólo dentro de un bloque *while*, por lo que habría que hacer algunas transformaciones en la gramática. Pero para no liar mucho la gramática, vamos a comprobar que la sentencia de salida del bloque está correctamente situada no en el análisis sintáctico sino más adelante, en el análisis semántico.

```
non terminal SentBreak;
SentBreak ::= BREAK PTOCOMA;
```

Ya podemos poner la lista de sentencias disponibles.

```
Sentencia ::= SentIf |

SentWhile |

SentAsignacion |

SentPutw |

SentPuts |

SentBreak;
```

Y con esto, podemos dar por finalizado el análisis sintáctico.

Vamos a probar si se puede analizar un programa más complejo. Por ejemplo:

```
int x;

int y;

main() {

x=1-12;

y=x*32;

x=(32/12)*y;

if(x>7) {

while(y>0) {

y=y-1;
```

```
if(y>1) {

break;

}

}

}

puts("Cadena\n");

puts("de texto");

putw(x+y);

}
```

Tras compilarlo, no debe aparecer ningún mensaje de error.

Una aspecto a tener en cuenta es que en esta fase, sólo se realiza el análisis sintáctico, por lo que no podemos detectar errores semánticos (por ejemplo, si hay una división por cero, si una variable está inicializada antes de ser usada, etcétera).

14.3 Análisis semántico

El análisis semántico consiste en verificar que los tipos en las expresiones son compatibles, que las funciones devuelven tipos adecuados, etcétera. Aunque algunas de las acciones no tendremos que realizarlas debido a que nuestro lenguaje es muy simple y no lo requiere.

Lo primero que debemos hacer es gestionar los símbolos de nuestro lenguaje, los ámbitos y los tipos. Para ello, se utilizan unas estructuras de datos que son: la tabla de símbolos, la tabla de tipos y la pila de ámbitos.

Hay muchas maneras de gestionar estas estructuras, pero lo normal es que haya una tabla de símbolos y otra de tipos para cada ámbito (un ámbito contiene las variables locales, los tipos locales, etc.). Cada función o procedimiento tiene su propio ámbito y cuando necesitamos un símbolo o tipo dentro del ámbito, lo primero es buscar si está declarado en el mismo, y si no lo está, buscar a ver si lo está en un ámbito que lo englobe. Lo mismo que ocurre para funciones y procedimientos, ocurre cuando entramos en un bloque de una sentencia *if-then-else* o *while*, por ejemplo. Pero como en nuestro lenguaje no hay ni funciones, ni procedimientos y además no se permite declarar nada dentro de los bloques de sentencias condicionales ni iterativas, en realidad sólo tenemos un ámbito. Por lo tanto, podemos prescindir de la gestión de ámbitos. Por lo que sólo habrá una tabla de símbolos y una tabla de tipos. En realidad, podríamos prescindir también de la tabla de tipos ya que sólo hay un tipo. Pero la vamos a incluir por si se nos ocurre

ampliar los tipos permitidos.

En resumen, vamos a usar una clase para gestionar la tabla de tipos y la de símbolos. En principio, la de tipos sólo contiene el tipo *int*.

Cada símbolo de la tabla de símbolos (TS) debe almacenar varios campos: código (el número de orden en la tabla) y tipo al que pertenece (código del tipo en la tabla de tipos). Más adelante nos daremos cuenta de que hay que incluir más información, pero por ahora tenemos suficiente.

Creamos la clase, los atributos y métodos necesarios para gestionarla. Esta tabla va a utilizar dos tipos de clases: *Simbolo* y *Tipo*.

Al añadir un símbolo o un tipo, se le asigna un código consecutivo que corresponde a su posición en la lista de símbolos y tipos respectivamente (contando desde 0). También hay que notar que al crear la tabla, se añaden los tipos básicos (en nuestro caso, sólo el tipo *int*).

Una vez que ya sabemos gestionar las tablas, añadiremos los símbolos correspondientes que se definan en la sección de declaraciones. Cada vez que vayamos a añadir un nuevo símbolo, comprobaremos que no existe en la tabla, y si existe debemos enviar un mensaje de error.

Como trataremos en esta sección con algunos mensajes de error, vamos a crear una clase que los centralice todos. Esta nueva clase la vamos a llamar *Textos*.

Le iremos añadiendo los diferentes mensajes de error.

Además, crearemos un método en el archivo *.cup* para sacar por pantalla el aviso con la línea donde se ha producido, el lexema y el mensaje de error.

```
parser code {:

……..

//Muestra un mensaje de error con linea, token y mensaje

public void error(String mensaje) {

report_error("ERROR -> Linea: "+InformacionCodigo.linea+

" Lexema: "+InformacionCodigo.token+" "+mensaje);

}

……..

:}
```

Ahora ya tenemos las bases para añadir un nuevo símbolo a la tabla de símbolos o sacar un mensaje de error para advertir que el símbolo se ha redeclarado.

Las acciones semánticas se ponen a continuación de donde se pone el terminal o no terminal objeto de la acción. Además, se puede utilizar una variable pasada por el terminal o no terminal con su lexema a la acción semántica.

Antes de nada, debemos crear un objeto con las tablas. Este objeto se crea al iniciar el proceso de análisis. Lo incluiremos todo (tanto el objeto de la clase *Tabla* como los métodos del análisis semántico) en la sección *action code*.

```
action code {:

Tabla tabla;

void inicializar() {

tabla = new Tabla();

}

:}

Programa ::= {:

inicializar();

:}

Declaraciones Cuerpo | Cuerpo;
```

También hemos llamado al método para inicializar las tablas.

La primera comprobación será cuando declaremos una variable, la añadiremos a la tabla de símbolos comprobando antes que no esté ya incluida. Los métodos a utilizar son:

```
action code {:

…….

boolean existeSimbolo(String id) {

return tabla.existeSimbolo(id);

}

void addSimbolo(String id) {

tabla.addSimbolo(id);
```

```
}

…….

:}
```

Ahora, debemos señalar que el terminal ID es del tipo *String* para que podamos utilizarlo como una variable para añadirla en la tabla.

```
terminal String ID;

Declaracion ::= INT ID:id PTOCOMA

{:

if(existeSimbolo(id)) {

parser.error(Textos.simboloRedeclarado);

} else {

addSimbolo(id);

}

:}

;
```

Vemos que el método *error* hay que llamarlo como método del objeto *parser,* ya que está en la sección *parse code*.

También podemos ver cómo se utilizan las variables de los lexemas de la gramática para procesarlos en el código Java. Detrás del terminal o no terminal de la gramática, y separado por dos puntos se le pone el nombre de la variable y luego esa variable (que contiene el valor del terminal o no terminal) puede ser utilizada en el código Java.

Ya hemos comprobado si al declarar una variable, ya ha sido declarada antes o no.

Ahora debemos comprobar si tras utilizar una variable, esta ha sido declarada previamente o no. Esto se hace en una de las reglas de las expresiones. Además, en las asignaciones.

```
Expresion ::= Expresion SUMA Expresion |

Expresion RESTA Expresion |
```

```
Expresion PRODUCTO Expresion |

Expresion DIVISION Expresion |

ENTERO |

ID:id

{:

if(existeSimbolo(id)) {

} else {

parser.error(Textos.simboloNoDeclarado);

}

:}

| LPAREN Expresion RPAREN ;

SentAsignacion ::= ID:id

{:

if(existeSimbolo(id)) {

} else {

parser.error(Textos.simboloNoDeclarado);

}

:}

ASIGNAR Expresion PTOCOMA;
```

Vemos que aparece otro tipo de mensaje de error. Por lo que debemos incluirlo en la clase *Textos*.

```
class Textos {

……..

static final String simboloNoDeclarado =
```

```
"El simbolo no ha sido declarado previamente";

…….

}
```

Otro tema que habría que solucionar sería la comprobación de si en una expresión se produce un desbordamiento o una división por 0. Pero estos dos aspectos no se pueden hacer en tiempo de compilación, por lo que habrá que hacerlo en tiempo de ejecución (en la generación de código final).

Como nuestro lenguaje es muy simple, nos evitamos hacer otro tipo de comprobaciones semánticas (las referentes a los tipos, las funciones, los procedimientos, etc.).

Nos queda una última comprobación, que es si la sentencia *break* está dentro de un bloque *while*. Esto es más complicado de hacer. Para ello, vamos a utilizar un contador de sentencias *while* que se vaya incrementando cada vez que se lea un lexema *while* y se irá decrementando cuando salgamos de su bloque de código. Podemos llamar a este contador, por ejemplo, *cuentaWhiles*. Lo inicializamos a 0 en el método *inicializar()*.

```
action code {:

……..

int cuentaWhiles;

void inicializar() {

tabla = new Tabla();

cuentaWhiles = 0;

}

…….

:}

SentWhile ::= WHILE LPAREN Condicion RPAREN LLLAVE

{:

cuentaWhiles++;

:}
```

```
BloqueSentencias

{:

cuentaWhiles--;

:}

RLLAVE;

SentBreak ::= BREAK

{:

if(cuentaWhiles>0) {

} else {

parser.error(Textos.breakSinWhile);

}

:}

PTOCOMA;
```

Y creamos un nuevo atributo en la clase *Textos*.

```
class Textos {

…….

static final String breakSinWhile =

“Se ha utilizado un break fuera de un bloque while”;

…….

}
```

Y con esto queda finalizado el análisis semántico. Hay que tener en cuenta que si nuestro lenguaje fuera algo más complejo, habría que hacer bastantes más comprobaciones semánticas.

CAPÍTULO 15

GENERACIÓN DE CÓDIGO INTERMEDIO DE C-0

15.1 Introducción

Una vez que ya hemos realizado las comprobaciones semánticas (aunque en nuestro reducido lenguaje han sido muy pocas), hay que enfocar la generación del código en ensamblador. Pero esto sería una locura hacerlo directamente. Lo que se suele hacer es crear un paso intermedio para reducir un poco la complejidad.

En esta fase, intentaremos crear una serie de instrucciones en un pseudolenguaje (inventado por nosotros) que nos permita, a partir de él, generar el código final (en ensamblador) de una manera más sencilla.

Este pseudolenguaje (que sería algo parecido a los bytecodes de Java) debe ser independiente de la plataforma. Es decir, independiente del ensamblador de la máquina. Y se hace así por dos cosas: por ser una manera de preparar nuestro compilador para cualquier plataforma y por simplificar un poco el trabajo de crear directamente nuestro programa en ensamblador.

A este pseudolenguaje se le llama código intermedio (CI). En nuestro caso, vamos a inventarnos uno sobre la marcha.

Vamos a utilizar una técnica que se llama código de tres direcciones.

15.2 Código de tres direcciones

El código de tres direcciones tiene su origen en la necesidad de gestionar las expresiones del lenguaje (sumas, restas, etc.).

Cuando tenemos una expresión, generalmente consta del nombre de la expresión, los dos operandos y el resultado.

Debemos saber, además, que vamos a escribir y leer sobre la memoria del computador (en sus direcciones de memoria). Por lo tanto, vamos a utilizar las direcciones de memoria para guardar valores de los resultados de las operaciones.

Por ejemplo, si queremos hacer la suma de dos números, primero debemos guardar cada número en una dirección de memoria, y luego aplicar la operación suma entre esas direcciones de memoria, y el resultado, lo guardamos en otra dirección de memoria. Es decir, utilizamos tres direcciones y el nombre de la operación.

Supongamos que tenemos esta expresión:

```
x=a+6;
```

Vamos a suponer que el valor de la variable *x* lo tenemos en la dirección de memoria 9000 y *a* lo tenemos en la dirección 9001. Para obtener la suma de *a* y *6*, podemos tomar como primer operando la dirección donde está *a*, de segundo operando el valor *6* y el resultado lo podemos guardar por ejemplo en la dirección 9002. Después, no tenemos más que cargar el contenido de la dirección 9002 en la dirección donde está el valor de *x*, es decir, en la dirección 9000.

Podríamos pensar que hay un desperdicio de direcciones ya que podríamos haber cargado la dirección 9000 con el contenido del resultado de la operación de suma, pero esto es posible en algunos casos, y para estandarizar el sistema a otros casos, se sigue el mismo mecanismo en todos ellos, aunque no sea el más eficiente.

Vamos a poner otro ejemplo:

```
x=12;

y=(x+1)*3;

z=x+y;
```

Supongamos que la dirección donde tenemos la variable *x* es la 9000, la de la *y* es 9001 y la de la *z* es 9002.

Los pasos a seguir serían:

1. Cargamos 12 en la dirección 9003.
2. Cargamos el contenido de la dirección 9003 en la dirección 9000.
3. Cargamos 1 en la dirección 9004.
4. Sumamos los contenidos de las direcciones 9000 y 9004 y el resultado lo ponemos en la dirección 9005.
5. Cargamos 3 en la dirección 9006.
6. Multiplicamos los contenidos de las direcciones 9005 y 9006 y el resultado lo ponemos en la dirección 9001.
7. Sumamos los contenidos de las direcciones 9000 y 9901 y lo guardamos en la dirección 9007.
8. Cargamos la dirección 9002 con el contenido de la dirección 9007.

De aquí podemos sacar varios tipos de operaciones, cada una de las cuales puede tener su propio CI.

Por ejemplo, cargar una dirección con el contenido de otra dirección:

```
CARGAR_DIRECCION op1 null resultado
```

Cargar una dirección con un valor:

```
CARGAR_VALOR op1 null resultado
```

Sumar el contenido de dos direcciones:

```
SUMAR op1 op2 resultado
```

Multiplicar el contenido de dos direcciones:

```
MULTIPLICAR op1 op2 resultado
```

Ahora, vamos a codificar nuestro ejemplo en CI:

1. CARGAR_VALOR 12 null 9003
2. CARGAR_DIRECCION 9003 null 9000
3. CARGAR_VALOR 1 null 9004
4. SUMAR 9000 9004 9005
5. CARGAR_VALOR 3 null 9006
6. MULTIPLICAR 9005 9006 9001
7. SUMAR 9000 9001 9007
8. CARGAR_DIRECCION 9007 null 9002

Podemos ver que no todas las direcciones se utilizan en todas las instrucciones de nuestro CI.

Hemos creado un pseudolenguaje, independiente de la máquina y del sistema operativo. Sólo en el momento de pasar estas instrucciones a un lenguaje ensamblador concreto es cuando este pseudolenguaje dejará de ser multiplataforma.

Habría maneras de reducir el número de direcciones utilizadas, pero para nuestro propósito es adecuado hacerlo así porque no pretendemos hacer un compilador comercial sino enseñar de una manera clara la manera de construirlo.

Podemos apreciar que nuestras instrucciones constan del tipo de operación, dos operandos y un resultado. Esto se puede manejar mediante una clase en Java con cuatro atributos. En principio, vamos a utilizar todos los atributos del tipo *String* ya que más adelante veremos que hay instrucciones que requieren cadenas y no números como en este caso.

La clase la llamaremos *Cuadrupla*.

15.3 Espacio de direcciones

Ya hemos visto que las variables y los resultados parciales de las expresiones

deben guardar su contenido en direcciones de memoria. Para ello, cada vez que declaremos una variable le debemos asignar una dirección de memoria única. También debemos ir asignando direcciones de memoria consecutivas a los resultados de las expresiones. Por lo tanto, debemos tener un contador de direcciones que iremos incrementando conforme vayamos declarando variables y conforme vayamos obteniendo resultados parciales de las expresiones.

Le podemos llamar a este contador de direcciones, por ejemplo *cuentaDirecciones*. Lo declararemos en la sección *action code*. Habrá que inicializarlo en el método inicializar.

Ahora, debemos ver cómo vamos a organizar la memoria para ver qué valor le damos inicialmente al contador de direcciones.

Vamos a ir poniendo el código del programa a partir de la dirección de memoria 0, y los datos los vamos a situar más adelante, en un espacio aparte, por ejemplo, a partir de la dirección 10000 (como se trata de un compilador sencillo y de aprendizaje, no vamos a suponer que el programa que queramos a compilar va a ser demasiado extenso y por eso, a su código le reservamos sólo 10000 direcciones de memoria).

Ahora ya podemos ver cómo quedaría el trozo del archivo *.cup*:

```
      action code {:

……..

int cuentaDirecciones;

void inicializar() {

      tabla = new Table();

      cuentaWhiles = 0;

      cuentaDirecciones = 9999;

}

……..

      :}
```

¿Por qué hemos puesto 9999 en vez de 10000? Porque antes de asignar la dirección, la incrementamos una unidad. Así, *cuentaDirecciones* siempre apunta a la última dirección asignada.

15.4 Asignación de direcciones a variables

La asignación de direcciones a las variables es muy sencilla, cuando nos encontremos en la gramática la declaración de una variable, la metemos en la tabla de símbolos, incrementamos el contador de direcciones y asignamos la dirección al símbolo declarado.

Para ello, debemos dotar a la clase *Simbolo* de un nuevo atributo para guardar la dirección. La clase quedaría así:

```
      class Simbolo {
……
private int direccion;
……..
void setDireccion(int d) {
      direccion = d;
}
int getDireccion() {
      return direccion;
}
……
      }
```

Ahora, necesitamos crear los métodos adecuados en la clase *Tabla*:

```
      class Tabla {
……
void setSimbolo(Simbolo s) {
      int cod = s.getCod();
      tablaSimbolos.setElementAt(s,cod);
}
void setDireccionSimbolo(String id,int d) {
      Simbolo simbolo = getSimbolo(id);
      simbolo.setDireccion(d);
      setSimbolo(simbolo);
}
……..
      }
```

Creamos un método en la sección *action code* para que haga esta tarea:

```
      action code {:

      …….

void setDireccionSimbolo(String id,int dir) {

      tabla.setDireccionSimbolo(id,dir);
```

```
}
```

```
……..
```

```
:}
```

Y en la gramática:

```
Declaracion ::= INT ID:id PTOCOMA
```

```
{:
```

```
if(existeSimbolo(id)) {
```

```
parser.error(Textos.simboloRedeclarado);
```

```
} else {
```

```
addSimbolo(id);
```

```
cuentaDirecciones++;
```

```
setDireccionSimbolo(id,cuentaDirecciones);
```

```
}
```

```
:}
```

```
;
```

15.5 Asignación de direcciones a expresiones y condiciones

Como ya se ha comentado anteriormente, los resultados de la evaluación de las expresiones deben guardarse también en una dirección de memoria. En caso de que nuestra gramática tuviera más de un tipo de datos, habría que comprobar que los tipos son compatibles al evaluar expresiones. Pero en nuestro caso, no tenemos que hacerlo al tener un solo tipo de datos. Por lo tanto, la clase que vamos a crear para contener las direcciones de las expresiones va a ser muy simple. Le llamaremos *Expresion*:

```
class Expresion {
```

```
int direccion;
```

```
Expresion(int d) {
```

```
        direccion = d;

    }

    int getDireccion() {

        return direccion;

    }

        }
```

Ahora, modificaremos la gramática para alojar en las direcciones correspondientes los resultados intermedios de la evaluación de expresiones. El no terminal *Expresion* debe ser del tipo *Expresion*. El no terminal *Condicion* también debe ser evaluado y, por lo tanto, debe ser del tipo *Expresion*.

```
        non terminal Expresion Expresion;

        non terminal Expresion Condicion;
```

Además, el terminal *ENTERO* debe ser del tipo *String* y luego lo convertiremos a *int*.

```
        terminal String ENTERO;
```

Las reglas de evaluación de expresiones quedarían así:

```
        Expresion ::= Expresion:e1 SUMA Expresion:e2

        {:

        RESULT=suma(e1,e2);

        :}

        | Expresion:e1 RESTA Expresion:e2

        {:

        RESULT=resta(e1,e2);

        :}

        | Expresion:e1 PRODUCTO Expresion:e2

        {:
```

```
RESULT=producto(e1,e2);

:}

| Expresion:e1 DIVISION Expresion:e2

{:

RESULT=division(e1,e2);

:}

| ENTERO:e

{:

RESULT=entero(e);

:}

| ID:id

{:

if(existeSimbolo(id)) {

RESULT=identificador(id);

} else {

parser.error(Textos.simboloNoDeclarado);

RESULT=identificador(null);

}

:}

| LPAREN Expresion:e RPAREN

{:

RESULT=e;

:}

;
```

Ahora, tenemos que implementar los métodos que se utilizan en las reglas:

```
action code {:
……….
Expresion suma(Expresion e1,Expresion e2) {
cuentaDirecciones++;
return new Expresion(cuentaDirecciones);
}
Expresion resta(Expresion e1,Expresion e2) {
cuentaDirecciones++;
return new Expresion(cuentaDirecciones);
}
Expresion producto(Expresion e1,Expresion e2) {
cuentaDirecciones++;
return new Expresion(cuentaDirecciones);
}
Expresion division(Expresion e1,Expresion e2) {
cuentaDirecciones++;
return new Expresion(cuentaDirecciones);
}
Expresion entero(String e) {
cuentaDirecciones++;
return new Expresion(cuentaDirecciones);
}
Expresion identificador(String id) {
cuentaDirecciones++;
return new Expresion(cuentaDirecciones);
}
…….
:}
```

Ahora, la pregunta es para qué sirven estos métodos que parece que sólo incrementan el contador de direcciones. Los vamos a utilizar más adelante para la generación de CI, pero por ahora, sólo van a incrementar el contador de direcciones y van a devolver una expresión con la dirección donde se va a guardar el resultado de la evaluación de la expresión.

Para las condiciones, vamos a hacer lo mismo, aunque el significado que le daremos en la generación de CI será diferente. Pero eso será más adelante. Por ahora:

```
Condicion ::= Condicion:c1 OR Condicion:c2

{:

RESULT=or(c1,c2);

:}
```

```
|
Condicion:c1 AND Condicion:c2
{:
RESULT=and(c1,c2);
:}
|
Expresion:e1 IGUAL Expresion:e2
{:
RESULT=igual(e1,e2);
:}
|
Expresion:e1 DISTINTO Expresion:e2
{:
RESULT=distinto(e1,e2);
:}
|
Expresion:e1 MAYOR Expresion:e2
{:
RESULT=mayor(e1,e2);
:}
|
Expresion:e1 MENOR Expresion:e2
{:
RESULT=menor(e1,e2);
```

```
:}

|

LPAREN Condicion:c RPAREN

{:

RESULT=c;

:}

;
```

Y el código de los métodos utilizados es:

```
action code {:

……..

Expresion or(Expresion c1,Expresion c2) {

cuentaDirecciones++;

return new Expresion(cuentaDirecciones);

}

Expresion and(Expresion c1,Expresion c2) {

cuentaDirecciones++;

return new Expresion(cuentaDirecciones);

}

Expresion mayor(Expresion e1,Expresion e2) {

cuentaDirecciones++;

return new Expresion(cuentaDirecciones);

}

Expresion menor(Expresion e1,Expresion e2) {

cuentaDirecciones++;
```

```
return new Expresion(cuentaDirecciones);

}

Expresion igual(Expresion e1,Expresion e2) {

cuentaDirecciones++;

return new Expresion(cuentaDirecciones);

}

Expresion distinto(Expresion e1,Expresion e2) {

cuentaDirecciones++;

return new Expresion(cuentaDirecciones);

}

……..

:}
```

Nos queda la sentencia de asignación, que debe cargar el contenido de la dirección de memoria dada por la expresión en la parte derecha de la asignación en la dirección donde se guarda el valor de la variable en la parte izquierda de la asignación.

Sería algo así:

```
SentAsignacion ::= ID:id

{:

if(existeSimbolo(id)) {

} else {

parser.error(Textos.simboloNoDeclarado);

}

:}

ASIGNAR Expresion:e PTOCOMA

{:
```

```
asignacion(id,e);

:}

;
```

Ahora, definimos el método nuevo:

```
action code {:

……..

void asignacion(String id,Expresion e) {

}

……

:}
```

Por ahora, el método no hace nada, pero más adelante nos servirá para generar el CI correspondiente.

15.6 CI de expresiones

Vamos a crear las sentencias de código intermedio conforme las vayamos necesitando. Pero antes, debemos saber que a medida que se analiza el programa a compilar, se deben ir generando las sentencias de CI; por lo tanto, antes debemos tener abierto un archivo de código intermedio donde iremos guardando las cuádruplas que se vayan generando. Al final, utilizaremos ese archivo para generar el código final.

Por lo tanto, vamos a crear una clase que se puede llamar *CondigoIntermedio* que guardará una lista de cuádruplas. Hemos decidido que se guarde esa lista en disco para poder ver las cuádruplas generadas y poder resolver los posibles errores que salgan.

El fichero de CI lo llamaremos igual que el programa a compilar pero con la extensión *.ci*. Para ello, utilizaremos una variable en la sección *parser code*, estática que guarde el nombre del archivo de entrada.

En la sección *action code* creamos un nuevo atributo de la clase *CodigoIntermedio* para guardar las cuádruplas.

Y en el método que inicializa el proceso de análisis, añadimos la inicialización del CI:

```
action code {:

……..
```

```
void inicializar() {

tabla = new Tabla();

cuentaWhiles = 0;

cuentaDirecciones = 9999;

String nombre = parser.nombreFichero.substring(0,

parser.nombreFichero.lastIndexOf(".”));

codigoIntermedio = new CodigoIntermedio(nombre+”.ci”);

try {

codigoIntermedio.abrirFicheroEscritura();

} catch (IOException e) {

System.out.println(Textos.ficheroCiNoExiste);

codigoIntermedio.cerrarFicheroEscritura();

}

}

…….

:}
```

Hemos incluido un nuevo texto de aviso de error. Lo añadiremos a la clase *Textos*.

```
class Textos {

……

static final String ficheroCiNoExiste =

“El fichero del CI no existe”;

}
```

También debemos crear un método para cerrar el archivo de CI al final de todo el proceso.

```
action code {:

……

void cerrarCI() {

codigoIntermedio.cerrarFicheroEscritura();

}

:}
```

Y en la regla de la gramática:

```
Cuerpo ::= MAIN LPAREN RPAREN LLLAVE BloqueSentencias

{:

cerrarCI();

:}

RLLAVE;
```

Ya tenemos todo preparado para generar el CI de las expresiones. Comencemos con la suma:

```
action code {:

…….

Expresion suma(Expresion e1,Expresion e2) {

cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“SUMAR”,

String.valueOf(e1.getDireccion()),

String.valueOf(e2.getDireccion()),

String.valueOf(cuentaDirecciones)));

return new Expresion(cuentaDirecciones);

}
```

```
……

:}
```

Y ahora continuamos con el resto.

```
Expresion resta(Expresion e1,Expresion e2) {
cuentaDirecciones++;
codigoIntermedio.guardarCuadrupla(new Cuadrupla(“RESTAR”,
String.valueOf(e1.getDireccion()),
String.valueOf(e2.getDireccion()),
String.valueOf(cuentaDirecciones)));
return new Expresion(cuentaDirecciones);
}
Expresion producto(Expresion e1,Expresion e2) {
cuentaDirecciones++;
codigoIntermedio.guardarCuadrupla(new Cuadrupla(“MULTIPLICAR”,
String.valueOf(e1.getDireccion()),
String.valueOf(e2.getDireccion()),
String.valueOf(cuentaDirecciones)));
return new Expresion(cuentaDirecciones);
}
Expresion division(Expresion e1,Expresion e2) {
cuentaDirecciones++;
codigoIntermedio.guardarCuadrupla(new Cuadrupla(“DIVIDIR”,
String.valueOf(e1.getDireccion()),
String.valueOf(e2.getDireccion()),
String.valueOf(cuentaDirecciones)));
return new Expresion(cuentaDirecciones);
}
```

En el caso de que la expresión sea un número entero, debemos cargar la dirección de memoria resultante con el valor del entero.

```
Expresion entero(String e) {
cuentaDirecciones++;
codigoIntermedio.guardarCuadrupla(new Cuadrupla(
“CARGAR_VALOR”,e,null,
String.valueOf(cuentaDirecciones)));
return new Expresion(cuentaDirecciones);
}
```

En el caso de que sea el nombre de una variable, debemos cargar el contenido de la dirección donde está la variable en la dirección de resultado.

```
Expresion identificador(String id) {
```

```
cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“CARGAR_DIRECCION”,

String.valueOf((tabla.getSimbolo(id)).getDireccion()),

null,String.valueOf(cuentaDirecciones)));

return new Expresion(cuentaDirecciones);

}
```

15.7 CI de condiciones

Para las condiciones haremos lo mismo que para las expresiones. Más adelante veremos que el resultado de un *or* o un *and* va a ser un 0 o un 1, dependiendo de si es falsa o verdadera la condición. También veremos que el resultado que vamos a guardar de una condición va a ser un 0 o un 1.

```
Expresion or(Expresion c1,Expresion c2) {

cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“OR”,

String.valueOf(c1.getDireccion()),

String.valueOf(c2.getDireccion()),

String.valueOf(cuentaDirecciones)));

return new Expresion(cuentaDirecciones);

}

Expresion and(Expresion c1,Expresion c2) {

cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“AND”,

String.valueOf(c1.getDireccion()),
```

```
String.valueOf(c2.getDireccion()),

String.valueOf(cuentaDirecciones)));

return new Expresion(cuentaDirecciones);

}

Expresion mayor(Expresion e1,Expresion e2) {

cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“MAYOR”,

String.valueOf(e1.getDireccion()),

String.valueOf(e2.getDireccion()),

String.valueOf(cuentaDirecciones)));

return new Expresion(cuentaDirecciones);

}

Expresion menor(Expresion e1,Expresion e2) {

cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“MENOR”,

String.valueOf(e1.getDireccion()),

String.valueOf(e2.getDireccion()),

String.valueOf(cuentaDirecciones)));

return new Expresion(cuentaDirecciones);

}

Expresion igual(Expresion e1,Expresion e2) {

cuentaDirecciones++;

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“IGUAL”,

String.valueOf(e1.getDireccion()),
```

```
String.valueOf(e2.getDireccion()),
String.valueOf(cuentaDirecciones)));
return new Expresion(cuentaDirecciones);
}
Expresion distinto(Expresion e1,Expresion e2) {
cuentaDirecciones++;
codigoIntermedio.guardarCuadrupla(new Cuadrupla("DISTINTO",
String.valueOf(e1.getDireccion()),
String.valueOf(e2.getDireccion()),
String.valueOf(cuentaDirecciones)));
return new Expresion(cuentaDirecciones);
}
```

15.8 CI de asignación

En el caso de una asignación, el CI cargará la dirección de la variable a la izquierda de la asignación con la dirección de la expresión a la derecha de la asignación.

```
void asignacion(String id,Expresion e) {
codigoIntermedio.guardarCuadrupla(new Cuadrupla(
"CARGAR_DIRECCION",
String.valueOf(e.getDireccion()),null,
String.valueOf((tabla.getSimbolo(id)).getDireccion())));
}
```

Vamos a comprobar que hasta ahora va todo correcto. Procesaremos el siguiente programa:

```
int x;
```

```
int y;

main() {

x=1-12;

y=x*32;

x=(32/12)*y;

if(x>7) {

while(y>0) {

y=y-1;

if(y>1) {

break;

}

}

}

puts(“Cadena\n”);

puts(“de texto”);

putw(x+y);

}
```

Tras procesarlo, debemos obtener el siguiente fichero de CI con el contenido:

```
CARGAR_VALOR 1 null 10002
CARGAR_VALOR 12 null 10003
RESTAR 10002 10003 10004
CARGAR_DIRECCION 10004 null 10000
CARGAR_DIRECCION 10000 null 10005
CARGAR_VALOR 32 null 10006
MULTIPLICAR 10005 10006 10007
CARGAR_DIRECCION 10007 null 10001
CARGAR_VALOR 32 null 10008
CARGAR_VALOR 12 null 10009
DIVIDIR 10008 10009 10010
```

```
CARGAR_DIRECCION 10001 null 10011
MULTIPLICAR 10010 10011 10012
CARGAR_DIRECCION 10012 null 10000
CARGAR_DIRECCION 10000 null 10013
CARGAR_VALOR 7 null 10014
MAYOR 10013 10014 10015
CARGAR_DIRECCION 10001 null 10016
CARGAR_VALOR 0 null 10017
MAYOR 10016 10017 10018
CARGAR_DIRECCION 10001 null 10019
CARGAR_VALOR 1 null 10020
RESTAR 10019 10020 10021
CARGAR_DIRECCION 10021 null 10001
CARGAR_DIRECCION 10001 null 10022
CARGAR_VALOR 1 null 10023
MAYOR 10022 10023 10024
CARGAR_DIRECCION 10000 null 10025
CARGAR_DIRECCION 10001 null 10026
SUMAR 10025 10026 10027
```

Si es así, va todo bien.

15.9 CI de bloques if-then-else

Para la generación de CI para estos bloques de código, debemos tener en cuenta que se pueden dar anidados. Es decir, bloques *if-then-else* dentro de otros. Ahora es el momento de ver cómo se va a organizar el código final para un bloque de este tipo ya que se requerirán algunas variables adicionales.

Cuando se lee una sentencia del tipo *if-then-else*, detrás del *if*, tenemos una condición, que si es *false* se producirá un salto a lo que hay detrás del *else*. Por lo tanto, debemos generar un código que produzca un salto en ciertas circunstancias. Pero para producir un salto, debemos saber a dónde vamos a saltar. En este momento de la generación del CI no podemos saberlo. Por lo tanto, echamos mano del ensamblador y vemos que hay lo que se llaman etiquetas. Son descriptores de direcciones de memoria. De manera que una dirección de memoria equivale a un descriptor o etiqueta. De esta forma, no tenemos por qué saber la dirección del salto, solamente debemos saltar a la dirección señalada por una etiqueta. Más tarde, ponemos la etiqueta en el lugar correspondiente (al comienzo del bloque *else*).

Pero para que esto funcione bien, las etiquetas no se deben repetir. Por ello, tendremos una pila de números que indicarán el orden de anidamiento de los bloques *if-then-else*.

Aparte, tendremos un contador de bloques que se irá incrementando siempre para que no se repitan las etiquetas ya que las vamos a crear a partir del número que nos dé el contador de bloques.

La necesidad de emplear una pila y un contador la veremos con este ejemplo. Sea

el trozo de programa:

```
1: if(x>1) {

2 : if(z<2) {

3 : x=x+z;

4 : } else {

5 : x=x-z;

6 : }

7 : x=x+9;

8 : if(z>1) {

9 : x=x+1;

10 : }

11 : }
```

Llamaremos *cuentaIf* al contador de bloques. Llamaremos a la pila *Pila* y supondremos que tenemos dos métodos básicos para *apilar* y *desapilar* en la pila. Además, tenemos un método para ver la cima de la pila *verCima*.

Inicialmente, la pila está vacía y el contador está a 0.

Línea 1: cuentaIf++=1, apilar(cuentaIf)=apilar(1) . Si la condición es falsa, saltar a la etiqueta ELSE_verCima=ELSE_1

Línea 2: cuentaIf++=2, apilar(cuentaIf)=apilar(2) . Si la condición es falsa, saltar a la etiqueta ELSE_verCima=ELSE_2

Línea 4: Saltar a la etiqueta FINIF_verCima=FINIF_2. Ponemos la etiqueta ELSE_verCima=ELSE_2

Línea 6: Ponemos la etiqueta FINIF_verCima=FINIF_2. Desapilamos una unidad desapilar()

Línea 8: cuentaIf++=3, apilar(cuentaIf)=apilar(3) . Si la condición es falsa, saltar a la etiqueta ELSE_verCima=ELSE_3

Línea 10: Saltar a la etiqueta ELSE_verCima=ELSE_3. Ponemos la etiqueta ELSE_verCima=ELSE_3. Saltar a la etiqueta FINIF_verCima=FINIF_3. Poner la etiqueta FINIF_verCima=FINIF_3. Desapilamos una unidad desapilar()

Línea 11: Saltar a la etiqueta ELSE_verCima=ELSE_1. Ponemos la etiqueta ELSE_verCima=ELSE_1. Saltar a la etiqueta FINIF_verCima=FINIF_1. Poner la etiqueta FINIF_verCima=FINIF_1. Desapilamos una unidad desapilar()

Esto parece un poco enredoso, pero vamos a aclararlo mejor. Lo primero es crear una nueva instrucción de CI que llamaremos así "ETIQUETA". Los dos operandos serán *null* y el resultado es el nombre de la etiqueta a poner. Por ejemplo, la cuádrupla *"ETIQUETA",null,null,"ELSE_1"* pondrá una etiqueta que se llame *ELSE_1*.

Para la pila, crearemos una clase. La podemos llamar *Pila*.

En el archivo *.cup* debemos declarar e inicializar estos cambios:

```
action code {:

……

int cuentaIf;

Pila pilaIf;

void inicializar() {

……..

cuentaIf = 0;

pilaIf = new Pila();

……

}

……

:}
```

En este momento, ya podemos incluir las acciones en los bloques de *if-then-else*.

```
SentIf ::= IF LPAREN

{:

cuentaIf++;

pilaIf.apilar(cuentaIf);
```

```
:}

Condicion:c

{:

condicion(c,pilaIf.verCima());

:}

RPAREN LLLAVE BloqueSentencias RLLAVE

{:

saltarEtiqueta(“FINIF”,pilaIf.verCima());

ponerEtiqueta(“ELSE”,pilaIf.verCima());

:}

SentElse

{:

ponerEtiqueta(“FINIF”,pilaIf.verCima());

pilaIf.desapilar();

:}

;
```

Y los métodos:

```
action code {:

……..

void condicion(Expresion e,int n) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“SALTAR_CONDICION”,

String.valueOf(e.getDireccion()),

null,
```

```
"ELSE_"+String.valueOf(n)));

}
void saltarEtiqueta(String eti,int n) {
codigoIntermedio.guardarCuadrupla(new Cuadrupla(
"SALTAR_ETIQUETA",
null,
null,
eti+"_"+String.valueOf(n)));
}
void ponerEtiqueta(String eti,int n) {
codigoIntermedio.guardarCuadrupla(new Cuadrupla(
"ETIQUETA",
null,
null,
eti+"_"+String.valueOf(n)));
}
……..
:}
```

Vemos que hemos creado tres nuevas instrucciones de CI. La primera es para provocar un salto a una etiqueta si la condición dada por el primer operando es falsa (en concreto, si el contenido de la dirección que representa el primer operando es 0).

La segunda provoca un salto incondicional a la etiqueta dada. Y la tercera pone una etiqueta.

15.10 CI de bloques while

La generación de CI para este tipo de bloque es muy parecida a la anterior. Pero en este caso hay que poner una etiqueta al inicio de la sentencia y al final del bloque se pone un salto a la etiqueta de inicio de la sentencia y después una etiqueta de final del bloque. De manera que en la condición, si el resultado es falso, se salta a la etiqueta que está al final del bloque.

Necesitamos también un contador y una pila. Les llamaremos *cuentaBucle* (ya teníamos uno que se llamaba *cuentaWhiles*) y *pilaBucle,* respectivamente.

Se deben declarar en la sección *action code* e inicializar en el método de inicialización (como es similar al anterior, no lo pondremos aquí).

Veremos cómo queda la gramática:

```
SentWhile ::= WHILE LPAREN

{:

cuentaBucle++;
```

```
pilaBucle.apilar(cuentaBucle);

ponerEtiqueta("BUCLE",pilaBucle.verCima());

:}

Condicion:c

{:

condicion2(c,pilaBucle.verCima());

:}

RPAREN LLLAVE

{:

cuentaWhiles++;

:}

BloqueSentencias

{:

cuentaWhiles--;

:}

RLLAVE

{:

saltarEtiqueta("BUCLE",pilaBucle.verCima());

ponerEtiqueta("FINBUCLE",pilaBucle.verCima());

pilaBucle.desapilar();

:}

;
```

Y el nuevo método:

```
void condicion2(Expresion e,int n) {
```

```
codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“SALTAR_CONDICION”,

String.valueOf(e.getDireccion()),

null,

“FINBUCLE_”+String.valueOf(n)));

}
```

15.11 CI de putw

Esta sentencia imprime un entero en la pantalla. El entero debe estar contenido en una dirección, por lo que su CI es sencillo:

```
“IMPRIMIR_ENTERO” , op1 , null , null
```

De manera que en op1 está la dirección.

La gramática sería:

```
SentPutw ::= PUTW LPAREN Expresion:e

{:

imprimirW(e);

:}

RPAREN PTOCOMA;
```

Y el método:

```
void imprimirW(Expresion e) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“IMPRIMIR_ENTERO”,

String.valueOf(e.getDireccion()),

null,

null));
```

```
}
```

15.12 CI de puts

Esta sentencia debe imprimir una cadena en la pantalla. Para crear el CI de esta sentencia, es necesario ver cómo suele implementarse en ensamblador la impresión de cadenas. Lo normal es que se haga mediante una instrucción que permita imprimir los datos que hay a partir de una cierta dirección. Luego, guardará cada uno de los caracteres en código ASCII de la cadena (el último suele ser un carácter 0).

En el ensamblador de ENS2001 se imprimen cadenas con una instrucción de impresión y un operando que es una etiqueta. Esta etiqueta señala el comienzo de la cadena a imprimir. Por lo tanto, debemos generar tantas etiquetas como cadenas a imprimir (cada etiqueta debe ser única).

Para hacer esto, utilizaremos otro contador, en este caso, de cadenas. Lo podemos llamar *cuentaCadenas* y lo declaramos e inicializamos en los mismos lugares que los demás contadores.

```
action code {:

…….

int cuentaCadenas;

……

void inicializar() {

…..
cuentaCadenas = 0;
…….
}
……..
:}
```

Además, vamos a poner todas las cadenas al final del código (para no mezclar los datos con el código), por lo que cuando se genere todo el código del programa, se generarán todas las cadenas. Por lo tanto, necesitamos una lista para guardar provisionalmente las cadenas que vayamos encontrando y luego recuperarlas al final (como el contador de cadenas comienza con 1, el número de orden en la lista+1 indicará la cadena correspondiente).

La clase para guardar las cadenas es *Lista*.

Habrá que declararla e inicializarla en su lugar correspondiente:

```
action code {:
……
Lista listaCadenas;
```

```
……..
void inicializar() {
……
listaCadenas = new Lista();
…….
}
…….
:}
```

Ahora veremos cómo queda la gramática:

```
terminal String CADENATEXTO;
SentPuts ::= PUTS LPAREN CADENATEXTO:c
{:
cuentaCadenas++;
imprimirS(c,cuentaCadenas);
:}
RPAREN PTOCOMA;
```

Ya sólo queda implementar el método *imprimirS* y crear el método para poner las cadenas al final del código (y situarlo para que se ejecute al final de todo el proceso de generación de código).

```
action code {:

……

void imprimirS(String c,int cuenta){

listaCadenas.addCadena(c);

codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“IMPRIMIR_CADENA”,

“CADENA_”+String.valueOf(cuenta),

null,

null));

}

……..

:}

Cuerpo ::= MAIN LPAREN RPAREN LLLAVE BloqueSentencias

{:
```

```
generarCadenas();

cerrarCI();

:}

RLLAVE;

action code {:

……..

void generarCadenas() {

for(int i=0;i<listaCadenas.size();i++) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“PONER_CADENA”,

“CADENA_”+String.valueOf(i+1),

null,

listaCadenas.getCadena(i)));

}

}

……..

:}
```

Nos queda sólo una cosa, finalizar el programa (para que el ensamblador sepa que no debe procesar más instrucciones). Utilizaremos la cuádrupla “FIN” null null null.

En la gramática:

```
Cuerpo ::= MAIN LPAREN RPAREN LLLAVE BloqueSentencias

{:

finPrograma();

generarCadenas();

cerrarCI();
```

```
generarCF();

:}
```

RLLAVE;

Y el nuevo método es:

```
void finPrograma() {

codigoIntermedio.guardarCuadrupla(new Cuadrupla(

“FIN”,null,null,null));

}
```

Y con esto, hemos finalizado la generación de código intermedio.

Por ejemplo, el siguiente programa:

```
int x;

int y;

main() {

x=10;

y=0;

puts(“Se debe escribir 10 : “);

putw(x);

puts(“\n”);

while(x>0) {

y=y+x;

x=x-1;

}

puts(“Se debe escribir 55 : “);

putw(y);
```

```
puts("\n");

if(x==0) {

puts("Esto se debe leer\n");

} else {

puts("Esto no se debe leer\n");

}

puts("Se debe escribir 11 : ");

putw(x+(y/5));

puts("\n");

}
```

Debería generar el siguiente CI:

```
CARGAR_VALOR 10 null 10002

CARGAR_DIRECCION 10002 null 10000

CARGAR_VALOR 0 null 10003

CARGAR_DIRECCION 10003 null 10001

IMPRIMIR_CADENA CADENA_1 null null

CARGAR_DIRECCION 10000 null 10004

IMPRIMIR_ENTERO 10004 null null

IMPRIMIR_CADENA CADENA_2 null null

ETIQUETA null null BUCLE_1

CARGAR_DIRECCION 10000 null 10005

CARGAR_VALOR 0 null 10006

MAYOR 10005 10006 10007

SALTAR_CONDICION 10007 null FINBUCLE_1
```

```
CARGAR_DIRECCION 10001 null 10008

CARGAR_DIRECCION 10000 null 10009

SUMAR 10008 10009 10010

CARGAR_DIRECCION 10010 null 10001

CARGAR_DIRECCION 10000 null 10011

CARGAR_VALOR 1 null 10012

RESTAR 10011 10012 10013

CARGAR_DIRECCION 10013 null 10000

SALTAR_ETIQUETA null null BUCLE_1

ETIQUETA null null FINBUCLE_1

IMPRIMIR_CADENA CADENA_3 null null

CARGAR_DIRECCION 10001 null 10014

IMPRIMIR_ENTERO 10014 null null

IMPRIMIR_CADENA CADENA_4 null null

CARGAR_DIRECCION 10000 null 10015

CARGAR_VALOR 0 null 10016

IGUAL 10015 10016 10017

SALTAR_CONDICION 10017 null ELSE_1

IMPRIMIR_CADENA CADENA_5 null null

SALTAR_ETIQUETA null null FINIF_1

ETIQUETA null null ELSE_1

IMPRIMIR_CADENA CADENA_6 null null

ETIQUETA null null FINIF_1

IMPRIMIR_CADENA CADENA_7 null null
```

```
CARGAR_DIRECCION 10000 null 10018

CARGAR_DIRECCION 10001 null 10019

CARGAR_VALOR 5 null 10020

DIVIDIR 10019 10020 10021

SUMAR 10018 10021 10022

IMPRIMIR_ENTERO 10022 null null

IMPRIMIR_CADENA CADENA_8 null null

FIN null null null

PONER_CADENA CADENA_1 null “Se debe escribir 10 : “

PONER_CADENA CADENA_2 null “\n”

PONER_CADENA CADENA_3 null “Se debe escribir 55 : “

PONER_CADENA CADENA_4 null “\n”

PONER_CADENA CADENA_5 null “Esto se debe leer\n”

PONER_CADENA CADENA_6 null “Esto no se debe leer\n”

PONER_CADENA CADENA_7 null “Se debe escribir 11 : “

PONER_CADENA CADENA_8 null “\n”
```

CAPÍTULO 16

GENERACIÓN DE CÓDIGO FINAL DE C-0

16.1 Introducción

Ahora es el momento de preparar todo para generar un archivo con el código final partiendo del código intermedio. Al archivo lo llamaremos igual que el nombre del programa pero con la extensión *.ens* (que es la que utiliza nuestro emulador).

El trabajo a realizar es leer todas las instrucciones guardadas en el objeto *codigoIntermedio*, procesarlas una a una e ir escribiendo el resultado en el archivo del código final.

16.2 Preparativos

Para la gestión del código final, vamos a utilizar una nueva clase que llamaremos *CodigoFinal*.

En cuanto a la acción a llevar en la gramática sería:

```
Cuerpo ::= MAIN LPAREN RPAREN LLLAVE BloqueSentencias
{:
generarCadenas();
cerrarCI();
generarCF();
:}
RLLAVE;
```

Y el nuevo método:

```
action code {:

……..

void generarCF() {

CodigoFinal codigoFinal = new CodigoFinal(

codigoIntermedio,parser.nombreFichero);

try {

codigoFinal.traducirCodigo();

} catch (Exception e) {}

}
```

```
:}
```

16.3 Introducción a Ens2001

Ens2001 es un emulador de un ensamblador estándar.

Sólo vamos a ver las instrucciones que necesitamos.

La memoria en Ens2001 comienza en la dirección 0 y termina en la 65535, es decir, tiene 64 Kb. Cada unidad de memoria es de 16 bits, por lo que en principio, sólo puede contener números de 16 bits (se supone que no se van a utilizar números mayores).

Hay varios registros, el registro PC (contador de programa), A (Acumulador), IX (Registro índice) y los registros de propósito general del R0 al R9 (hay algunos más pero por ahora no nos interesan).

Los registros son también de 16 bits, por lo que sólo pueden contener números de 16 bits (desde el 0 al 65535).

Los biestables de estado que vamos a utilizar son: Z (cero).

El registro PC se encarga de ir indicando en cada momento la posición de memoria que se va a ejecutar seguidamente.

El acumulador va guardando los resultados de las operaciones de suma, resta, multiplicación, etc.

El registro IX es un registro índice.

Los demás registros de propósito general los vamos a utilizar de manera auxiliar para guardar resultados intermedios.

Los modos de direccionamiento que vamos a utilizar son:

- Direccionamiento inmediato: para cargar los registros o direcciones de memoria con valores concretos.
- Direccionamiento directo a registro: para utilizar el valor contenido en un registro.
- Direccionamiento directo a memoria: para utilizar el contenido de una dirección de memoria.
- Direccionamiento indirecto: para utilizar la dirección señalada por el contenido de un registro.
- Direccionamiento relativo a registro índice: para utilizar la dirección apuntada por el contenido del registro índice más o menos cierta cantidad de unidades.

- Direccionamiento relativo a contador de programa: lo mismo que el anterior pero respecto al PC.

Vamos a ir cogiendo cada una de las etiquetas de CI y viendo qué código necesitamos generar.

En Ens2001, si no se indica lo contrario, un programa se carga a partir de la dirección 0. Nosotros lo haremos así. En cuanto a los datos, los pondremos a partir de la dirección 10000, excepto las cadenas, que irán guardadas al final del código.

En cuanto a los desbordamientos y divisiones por 0, dejaremos que Ens2001 envíe los mensajes de error oportunos.

Para referirnos a los registros, se utiliza un punto seguido del nombre del registro (por ejemplo, .A, .R1, etc.).

Para referirnos a una dirección de memoria, se utiliza una barra y la dirección o una barra y el nombre de una etiqueta que representa a la dirección (por ejemplo, /9453 o /DIR).

Para referirnos al contenido de un registro, lo encerramos entre corchetes (por ejemplo, [.R4]).

Para referirnos a la dirección respecto a un índice se pone # seguido del incremento o decremento, corchetes, nombre del registro y corchetes (por ejemplo, #3[.IX] o #-2[.PC]).

Para referirnos a la dirección respecto al contador de programa ponemos $ seguido del incremento o decremento (por ejemplo, $3 o $-2).

A partir de este momento, cuando vayamos procesando las diferentes instrucciones del CI iremos viendo cómo generar las instrucciones en ensamblador.

16.4 CARGAR_DIRECCION op1 null res

Carga el contenido de la dirección de memoria señalada por el primer operando en la dirección señalada por el resultado.

MOVE /op1 , /res

Por ejemplo, copiar el contenido de la dirección 10001 en la dirección 10002:

MOVE /10001 , /10002

En la clase *CodigoFinal*, haríamos esta modificación:

```
.....

class CodigoFInal {
```

```
…….

// Procesa la cuadrupla

private void procesarCuadrupla(

Cuadrupla cuadrupla)throws IOException {

String op1,op2,inst,res;

String linea =

“ “;

op1 = cuadrupla.op1;

op2 = cuadrupla.op2;

inst = cuadrupla.nombre;

res = cuadrupla.res;

if(inst.equals(“CARGAR_DIRECCION”)) {

escribirLinea(linea+”MOVE /”+op1+” , /”+res);

} else

if(inst.equals(……..) {

} else

if(inst.equals(……..) {

}

}

}
```

En adelante, sólo pondremos el condicional para cada una de las instrucciones de CI.

16.5 CARGAR_VALOR op1 null res

Carga un valor en la dirección señalada por el resultado.

MOVE #op1 , /res

Por ejemplo, carga el valor 15 en la dirección 10003:

MOVE #15 , /10003

```
if(inst.equals("CARGAR_VALOR")) {

escribirLinea(linea+"MOVE #"+op1+" , /"+res);

}
```

16.6 SUMAR op1 op2 res

Suma el contenido de las direcciones de los operadores y el resultado lo carga en la dirección del resultado.

ADD /op1 , /op2

MOVE .A , /res

Como el resultado de las operaciones aritméticas se guarda en el acumulador, hay que cargar en la dirección del resultado el contenido del acumulador.

Por ejemplo: sumar el contenido de las direcciones 10000 y 10001 y poner el resultado en la dirección 10002.

ADD /10000 , /100001

MOVE .A , /100002

```
if(inst.equals("SUMAR")) {

escribirLinea(linea+"ADD /"+op1+" , /"+op2);

escribirLinea(linea+"MOVE .A , /"+res);

}
```

16.7 RESTAR op1 op2 res

Lo mismo que el anterior pero restando.

```
if(inst.equals("RESTAR")) {

escribirLinea(linea+"SUB /"+op1+" , /"+op2);

escribirLinea(linea+"MOVE .A , /"+res);

}
```

16.8 MULTIPLICAR op1 op2 res

```
if(inst.equals("MULTIPLICAR")) {

escribirLinea(linea+"MUL /"+op1+" , /"+op2);

escribirLinea(linea+"MOVE .A , /"+res);

}
```

16.9 DIVIDIR op1 op2 res

```
if(inst.equals("DIVIDIR")) {

escribirLinea(linea+"DIV /"+op1+" , /"+op2);

escribirLinea(linea+"MOVE .A , /"+res);

}
```

16.10 OR op1 op2 res

En este caso, se trata de una operación lógica. Como ya dijimos los operadores deben ser o un 1 o un 0 para representar verdadero o falso. El resultado, por lo tanto, es un 1 o un 0.

```
if(inst.equals("OR")) {

escribirLinea(linea+"OR /"+op1+" , /"+op2);

escribirLinea(linea+"MOVE .A , /"+res);

}
```

16.11 AND op1 op2 res

```
if(inst.equals("AND")) {

escribirLinea(linea+"AND /"+op1+" , /"+op2);

escribirLinea(linea+"MOVE .A , /"+res);

}
```

16.12 MAYOR op1 op2 res

Esta operación es algo más complicada que las anteriores. Debemos comparar dos números y si el primero es mayor que el segundo, poner un 1 en el resultado y en caso contrario poner un 0.

Utilizaremos la instrucción en ensamblador CMP que compara dos operandos (resta el primero menos el segundo) . Si el biestable P se activa, es que el resultado es positivo o cero y por lo tanto el primer operando es mayor o igual que el segundo. Pero necesitamos saber sólo si es mayor, por lo tanto, cambiaremos el orden de los operandos y restaremos el segundo menos el primero, de manera que si N (biestable para resultados negativos) se activa es que el primero es mayor que el segundo (resultado 1) y al contrario (resultado 0).

```
if(inst.equals("MAYOR")) {

escribirLinea(linea+"CMP /"+op2+" , /"+op1);

escribirLinea(linea+"BN $5");

escribirLinea(linea+"MOVE #0 , /"+res);

escribirLinea(linea+"BR $3");

escribirLinea(linea+"MOVE #1 , /"+res);

}
```

16.13 MENOR op1 op2 res

En este caso, lo hacemos al revés que el caso anterior.

```
if(inst.equals("MENOR")) {

escribirLinea(linea+"CMP /"+op1+" , /"+op2);

escribirLinea(linea+"BN $5");

escribirLinea(linea+"MOVE #0 , /"+res);

escribirLinea(linea+"BR $3");

escribirLinea(linea+"MOVE #1 , /"+res);

}
```

16.14 IGUAL op1 op2 res

Para ver si los operandos son iguales, utilizamos el biestable Z que nos indica si el resultado es cero (es decir, son iguales).

```
if(inst.equals("IGUAL")) {

escribirLinea(linea+"CMP /"+op1+" , /"+op2);

escribirLinea(linea+"BZ $5");

escribirLinea(linea+"MOVE #0 , /"+res);

escribirLinea(linea+"BR $3");

escribirLinea(linea+"MOVE #1 , /"+res);

}
```

16.15 DISTINTO op1 op2 res

Este es el caso contrario del anterior.

```
if(inst.equals("DISTINTO")) {

escribirLinea(linea+"CMP /"+op1+" , /"+op2);

escribirLinea(linea+"BZ $5");

escribirLinea(linea+"MOVE #1 , /"+res);

escribirLinea(linea+"BR $3");

escribirLinea(linea+"MOVE #0 , /"+res);

}
```

16.16 ETIQUETA null null res

Esta instrucción simplemente pone una etiqueta con un nombre concreto. Las etiquetas se ponen en Ens2001 con el nombre seguido de dos puntos.

```
if(inst.equals("ETIQUETA")) {

String lin = res+":"+linea;
```

```
escribirLinea(lin.substring(0,linea.length())+"NOP");

}
```

16.17 SALTAR_CONDICION op1 null res

Esta instrucción consiste en saltar a la etiqueta representada por el resultado si la condición es falsa, es decir, si el valor del contenido de la dirección señalada por el primer operando es 0.

Por lo tanto, debemos comparar con 0 el contenido de la dirección señalada por *op1*. Si se activa el biestable Z es que la comparación es 0 y, por tanto, habrá que saltar a la etiqueta. Si no se activa, se sigue sin que pase nada.

```
if(inst.equals("SALTAR_CONDICION")) {

escribirLinea(linea+"CMP #0 , /"+op1);

escribirLinea(linea+"BZ /"+res);

}
```

16.18 SALTAR_ETIQUETA null null res

Esta instrucción consiste en un salto incondicional a la etiqueta señalada por el resultado.

```
if(inst.equals("SALTAR_ETIQUETA")) {

escribirLinea(linea+"BR /"+res);

}
```

16.19 IMPRIMIR_ENTERO op1 null null

La sentencia para imprimir en Ens2001 es WRINT.

```
if(inst.equals("IMPRIMIR_ENTERO")) {

escribirLinea(linea+"WRINT /"+op1);

}
```

16.20 IMPRIMIR_CADENA op1 null null

En este caso, se imprime una cadena. En realidad, se manda la orden de imprimir la cadena que está señalada por la etiqueta del primer operando. Se utiliza la instrucción WRSTR.

```
if(inst.equals("IMPRIMIR_CADENA")) {

escribirLinea(linea+"WRSTR /"+op1);

}
```

16.21 PONER_CADENA op1 null res

Esta instrucción pone una etiqueta con una cadena. La manera de hacer esto es mediante el formato en Ens2001:

Etiqueta: DATA cadena

```
if(inst.equals("PONER_CADENA")) {

String lin = op1+": DATA"+linea;

escribirLinea(lin.substring(0,linea.length())+res);

}
```

16.22 Punto y final

Finalmente, para ver si todo esto funciona, probaremos a compilar un programa:

```
int x;

int y;

main() {

x=10;

y=0;

puts("Se debe escribir 10 : ");

putw(x);

puts("\n");

while(x>0) {

y=y+x;

x=x-1;

}
```

```
puts("Se debe escribir 55 : ");
putw(y);
puts("\n");
if(x==0 && y>54) {
puts("Esto se debe leer\n");
} else {
puts("Esto no se debe leer\n");
}
puts("Se debe escribir 11 : ");
putw(x+(y/5));
puts("\n");
x=x+2;
while(x*2<y) {
while(y>x*4) {
x=x+1;
}
x=x+1;
}
puts("Se debe escribir 28 : ");
putw(x);
puts("\n");
}
```

El CI es:

```
CARGAR_VALOR 10 null 10002
```

```
CARGAR_DIRECCION 10002 null 10000

CARGAR_VALOR 0 null 10003

CARGAR_DIRECCION 10003 null 10001

IMPRIMIR_CADENA CADENA_1 null null

CARGAR_DIRECCION 10000 null 10004

IMPRIMIR_ENTERO 10004 null null

IMPRIMIR_CADENA CADENA_2 null null

ETIQUETA null null BUCLE_1

CARGAR_DIRECCION 10000 null 10005

CARGAR_VALOR 0 null 10006

MAYOR 10005 10006 10007

SALTAR_CONDICION 10007 null FINBUCLE_1

CARGAR_DIRECCION 10001 null 10008

CARGAR_DIRECCION 10000 null 10009

SUMAR 10008 10009 10010

CARGAR_DIRECCION 10010 null 10001

CARGAR_DIRECCION 10000 null 10011

CARGAR_VALOR 1 null 10012

RESTAR 10011 10012 10013

CARGAR_DIRECCION 10013 null 10000

SALTAR_ETIQUETA null null BUCLE_1

ETIQUETA null null FINBUCLE_1

IMPRIMIR_CADENA CADENA_3 null null

CARGAR_DIRECCION 10001 null 10014
```

```
IMPRIMIR_ENTERO 10014 null null

IMPRIMIR_CADENA CADENA_4 null null

CARGAR_DIRECCION 10000 null 10015

CARGAR_VALOR 0 null 10016

IGUAL 10015 10016 10017

CARGAR_DIRECCION 10001 null 10018

CARGAR_VALOR 54 null 10019

MAYOR 10018 10019 10020

AND 10017 10020 10021

SALTAR_CONDICION 10021 null ELSE_1

IMPRIMIR_CADENA CADENA_5 null null

SALTAR_ETIQUETA null null FINIF_1

ETIQUETA null null ELSE_1

IMPRIMIR_CADENA CADENA_6 null null

ETIQUETA null null FINIF_1

IMPRIMIR_CADENA CADENA_7 null null

CARGAR_DIRECCION 10000 null 10022

CARGAR_DIRECCION 10001 null 10023

CARGAR_VALOR 5 null 10024

DIVIDIR 10023 10024 10025

SUMAR 10022 10025 10026

IMPRIMIR_ENTERO 10026 null null

IMPRIMIR_CADENA CADENA_8 null null
CARGAR_DIRECCION 10000 null 10027
```

```
CARGAR_VALOR 2 null 10028
SUMAR 10027 10028 10029
CARGAR_DIRECCION 10029 null 10000
ETIQUETA null null BUCLE_2
CARGAR_DIRECCION 10000 null 10030
CARGAR_VALOR 2 null 10031
MULTIPLICAR 10030 10031 10032
CARGAR_DIRECCION 10001 null 10033
MENOR 10032 10033 10034
SALTAR_CONDICION 10034 null FINBUCLE_2
ETIQUETA null null BUCLE_3
CARGAR_DIRECCION 10001 null 10035
CARGAR_DIRECCION 10000 null 10036
CARGAR_VALOR 4 null 10037
MULTIPLICAR 10036 10037 10038
MAYOR 10035 10038 10039
SALTAR_CONDICION 10039 null FINBUCLE_3
CARGAR_DIRECCION 10000 null 10040
CARGAR_VALOR 1 null 10041
SUMAR 10040 10041 10042
CARGAR_DIRECCION 10042 null 10000
SALTAR_ETIQUETA null null BUCLE_3
ETIQUETA null null FINBUCLE_3
CARGAR_DIRECCION 10000 null 10043
CARGAR_VALOR 1 null 10044
SUMAR 10043 10044 10045
CARGAR_DIRECCION 10045 null 10000
SALTAR_ETIQUETA null null BUCLE_2
ETIQUETA null null FINBUCLE_2
IMPRIMIR_CADENA CADENA_9 null null
CARGAR_DIRECCION 10000 null 10046
IMPRIMIR_ENTERO 10046 null null
IMPRIMIR_CADENA CADENA_10 null null
FIN null null null
PONER_CADENA CADENA_1 null “Se debe escribir 10 : “
PONER_CADENA CADENA_2 null “\n”
PONER_CADENA CADENA_3 null “Se debe escribir 55 : “
PONER_CADENA CADENA_4 null “\n”
PONER_CADENA CADENA_5 null “Esto se debe leer\n”
PONER_CADENA CADENA_6 null “Esto no se debe leer\n”
PONER_CADENA CADENA_7 null “Se debe escribir 11 : “
PONER_CADENA CADENA_8 null “\n”
PONER_CADENA CADENA_9 null “Se debe escribir 28 : “
PONER_CADENA CADENA_10 null “\n”
```

Y el código final es:

```
MOVE #10 , /10002

MOVE /10002 , /10000

MOVE #0 , /10003

MOVE /10003 , /10001

WRSTR /CADENA_1

MOVE /10000 , /10004

WRINT /10004

WRSTR /CADENA_2

BUCLE_1: NOP

MOVE /10000 , /10005

MOVE #0 , /10006

CMP /10006 , /10005

BN $5

MOVE #0 , /10007

BR $3

MOVE #1 , /10007

CMP #0 , /10007

BZ /FINBUCLE_1

MOVE /10001 , /10008

MOVE /10000 , /10009

ADD /10008 , /10009

MOVE .A , /10010
```

```
MOVE /10010 , /10001

MOVE /10000 , /10011

MOVE #1 , /10012

SUB /10011 , /10012

MOVE .A , /10013

MOVE /10013 , /10000

BR /BUCLE_1

FINBUCLE_1: NOP

WRSTR /CADENA_3

MOVE /10001 , /10014

WRINT /10014

WRSTR /CADENA_4

MOVE /10000 , /10015

MOVE #0 , /10016

CMP /10015 , /10016

BZ $5

MOVE #0 , /10017

BR $3

MOVE #1 , /10017

MOVE /10001 , /10018

MOVE #54 , /10019

CMP /10019 , /10018

BN $5

MOVE #0 , /10020
```

```
BR $3

MOVE #1 , /10020

AND /10017 , /10020

MOVE .A , /10021

CMP #0 , /10021

BZ /ELSE_1

WRSTR /CADENA_5

BR /FINIF_1

ELSE_1: NOP

WRSTR /CADENA_6

FINIF_1: NOP

WRSTR /CADENA_7

MOVE /10000 , /10022

MOVE /10001 , /10023

MOVE #5 , /10024

DIV /10023 , /10024

MOVE .A , /10025

ADD /10022 , /10025

MOVE .A , /10026

WRINT /10026

WRSTR /CADENA_8

MOVE /10000 , /10027

MOVE #2 , /10028

ADD /10027 , /10028
```

```
MOVE .A , /10029

MOVE /10029 , /10000

BUCLE_2: NOP

MOVE /10000 , /10030

MOVE #2 , /10031

MUL /10030 , /10031

MOVE .A , /10032

MOVE /10001 , /10033

CMP /10032 , /10033

BN $5

MOVE #0 , /10034

BR $3

MOVE #1 , /10034

CMP #0 , /10034

BZ /FINBUCLE_2

BUCLE_3: NOP

MOVE /10001 , /10035

MOVE /10000 , /10036

MOVE #4 , /10037

MUL /10036 , /10037

MOVE .A , /10038

CMP /10038 , /10035

BN $5

MOVE #0 , /10039
```

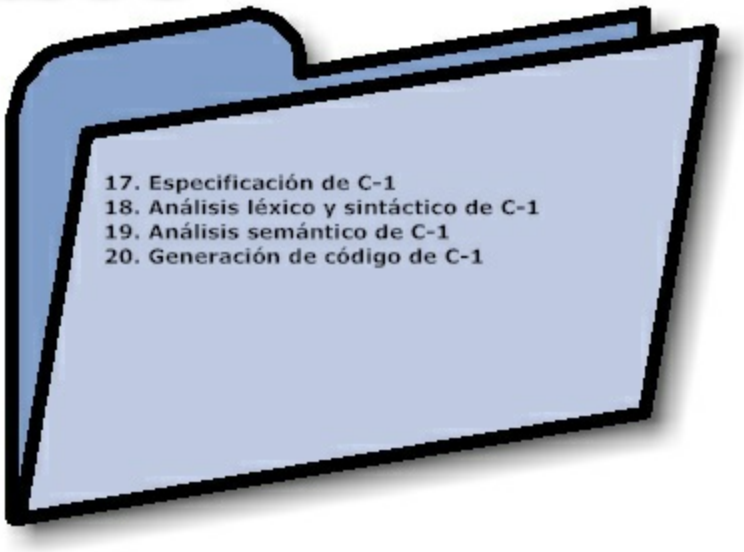
```
BR $3

MOVE #1 , /10039

CMP #0 , /10039

BZ /FINBUCLE_3

MOVE /10000 , /10040

MOVE #1 , /10041

ADD /10040 , /10041

MOVE .A , /10042

MOVE /10042 , /10000

BR /BUCLE_3

FINBUCLE_3: NOP

MOVE /10000 , /10043

MOVE #1 , /10044

ADD /10043 , /10044

MOVE .A , /10045

MOVE /10045 , /10000

BR /BUCLE_2

FINBUCLE_2: NOP

WRSTR /CADENA_9

MOVE /10000 , /10046

WRINT /10046

WRSTR /CADENA_10

HALT

CADENA_1: DATA “Se debe escribir 10 : “
```

```
CADENA_2: DATA "\n"

CADENA_3: DATA "Se debe escribir 55 : "

CADENA_4: DATA "\n"

CADENA_5: DATA "Esto se debe leer\n"

CADENA_6: DATA "Esto no se debe leer\n"

CADENA_7: DATA "Se debe escribir 11 : "

CADENA_8: DATA "\n"

CADENA_9: DATA "Se debe escribir 28 : "

CADENA_10: DATA "\n"
```

16.23 Posibles ampliaciones

El lenguaje **C-0** es muy sencillo, aunque muestra la gran potencia de las herramientas presentadas. A partir de aquí, es posible ampliar las funcionalidades del lenguaje con unos pocos cambios (en algunos casos, bastante complejos). Por ejemplo, admitir estructuras de datos (registros, vectores, etc.), admitir subprogramas (funciones y procedimientos), admitir importaciones de librerías, admitir más operadores y tipos de sentencias condicionales o de bucle, e incluso implementar la orientación a objetos (esto complicaría mucho más el lenguaje).

Parte IV. Implementación de C-1



CAPÍTULO 17

ESPECIFICACIÓN DE C-1

17.1 Introducción

Con esta práctica pretendemos ampliar las funcionalidades del lenguaje **C-0**. Vamos a reutilizar todo el código que nos sirva del compilador para **C-0** añadiéndole el código nuevo necesario para las nuevas características de **C-1**.

Las características nuevas, no implementadas en **C-0,** son:

1. La definición de tipos estructurados, en concreto registros y vectores.
2. La utilización de subprogramas: funciones y procedimientos.
3. La posibilidad de recursividad en los subprogramas.
4. La posibilidad de declaración de variables locales en los subprogramas.
5. La posibilidad de utilizar nuevos operadores.
6. La posibilidad de declarar variables conjuntamente.
7. La inclusión de comentarios.
8. Permitir sentencias condicionales sin tener que poner un bloque entre llaves cuando hay sólo una instrucción dentro del bloque.

La mayor dificultad a la hora de afrontar la creación de este nuevo compilador es la inclusión de subprogramas. Lo que implica la gestión no sólo de estos subprogramas sino la gestión de ámbitos. Lo veremos todo en los siguientes capítulos.

17.2 Tipos estructurados

Los dos nuevos tipos estructurados que se va a permitir utilizar en C-1 son: los registros y los vectores.

17.2.1 Registros

Un registro es una estructura de datos que consta de una serie de campos que pueden ser de diferentes tipos.

No se va a permitir el anidamiento en la definición de registros. Además, como sólo tenemos un tipo básico, el entero (aunque tenemos el tipo *booleano* encubierto. Un 0 es falso y cualquier otro número es verdadero), los campos de un registro sólo pueden ser de tipo entero.

Una restricción adicional sirve para definir cada campo aparte, es decir, no se puede definir conjuntamente más de un campo.

Para acceder a un campo de una variable de tipo registro, se pone el nombre de la

variable seguida de un punto y del nombre del campo. Como no hay anidamiento, sólo es posible esa forma de acceso.

La estructura de la definición de un campo es:

```
struct nombreTipo{

int campo1;

int campo2;

……..

int campon;

};
```

Vemos que como sólo tenemos el tipo primitivo entero, todos los posibles campos son del mismo tipo. También vemos que cada campo se debe definir aparte. Y además, se debe terminar con punto y coma.

Sólo se va a permitir la definición de nuevos tipos en la zona del programa principal y no de los subprogramas.

La declaración de una variable del tipo registro será así:

```
nombreTipo nombreVariable;
```

Estas restricciones podrían relajarse a costa de complicar el diseño. Se dejará para una posible ampliación (la creación de un posible compilador **C-2**).

17.2.2 Vectores

Al igual que con los registros, sólo se va a permitir definir tipos de vectores en la zona del programa principal. El tipo de los vectores sólo puede ser el entero. Además, aquí se va a realizar una modificación respecto a la estructura del lenguaje C. Además, sólo se permiten vectores unidimensionales. La manera de definir el tipo de vector será:

```
int[n] nombreTtipo;
```

Es decir, primero se pone el tipo (como sólo admitimos el tipo entero, no hay otra opción que poner ***int***). Seguido (o separado por espacios) se pone entre corchetes el tamaño del vector (de 0 a n-1). Y después el nombre del tipo y punto y coma.

La utilización de elementos de un vector fuera del rango no será comprobada por el compilador, por lo que será el usuario (el que utilice el compilador para programar) el responsable de la buena utilización de los vectores.

La declaración de una variable de tipo vector será:

```
nombreTipo nombreVariable;
```

Estas restricciones se podrán relajar para un futuro compilador más avanzado.

17.3 Declaración conjunta de variables y variables locales

Se permite la declaración conjunta de variables. Es decir, se permite declarar a la vez más de una variable. No hay más que poner el tipo delante y detrás los nombres de las variables separados por una coma. Al final se pone un punto y coma.

Es decir, se permite este tipo de declaración:

```
int x,y,z;
```

Aparte de esta nueva característica, se permite declarar variables dentro del ámbito de los subprogramas. Se llaman variables locales y deben declararse al principio, antes del código del subprograma.

Aunque es posible entremezclar declaraciones de variables y definiciones de tipos, antes de poder declarar una variable de un tipo, el tipo debe estar definido. Si no es así, el compilador lanzará un mensaje de error.

17.4 Nuevos operadores y delimitadores

Aparte de los operadores que se permiten en **C-0**, se incorporan algunos nuevos:

- Menor o igual: **>=**
- Mayor o igual: **>=**
- Módulo: **%**
- Negación lógica**: *!***
- Delimitadores: ***[ ]***

Los delimitadores se utilizan para los vectores.

17.5 Subprogramas

Esta es la novedad más importante que se va a implementar. Se trata de incluir la posibilidad de utilizar funciones y procedimientos.

Los subprogramas permiten la inclusión de argumentos. En nuestro caso, sólo se va a permitir el paso de parámetros por valor y no por referencia.

Los argumentos deben ir de uno en uno y separados por una coma. Cada argumento consta del tipo y del nombre.

Las funciones tienen esta estructura:

```
tipoDevuelto nombreFuncion(tipo1 arg1, tipo2 arg2,…., tipon argn) {

//declaración de variables e instrucciones.Una de ellas debe de ser //return valor;

}
```

Los procedimientos tienen esta:

```
void nombreProcedimiento(tipo1 arg1, tipo2 arg2,…., tipon argn) {

//declaración de variables e instrucciones

}
```

Los tipos en los argumentos pueden ser cualquiera de los definidos. En las funciones, el tipo devuelto puede ser cualquiera de los definidos.

17.6 Asignación

La asignación sigue igual, pero con la salvedad de que ahora, en la parte derecha se permite la llamada a una función.

Otra cosa más, cuando se asigna a una variable de un tipo estructurado, se asigna toda la variable. Por ejemplo, supongamos que tenemos este programa:

```
int[2] vector;

vector v1,v2;

main() {

v1[0] = 1;v1[1] = 2;

v2 = v1;v2[0] = v1[0];

}
```

En la asignación (*v2 = v1*), estamos copiando todo el vector *v1* en *v2* (los dos elementos del vector). En cambio, en la asignación (*v2[0] = v1[0]*) asignamos sólo el componente 0.

17.7 Comentarios

Se permite la inclusión de comentarios, aunque esto no afecta a las fases de análisis sintácticos ni siguientes ya que serán eliminados en la primera fase.

Los comentarios tienen esta estructura:

```
/* comentarios */
```

Se permite que los comentarios ocupen más de una línea y que se puedan anidar.

CAPÍTULO 18

ANÁLISIS LÉXICO Y SINTÁCTICO DE C-1

18.1 Introducción

Vamos a reutilizar todo el código que podamos de **C-0**. En cuanto a los archivos de JLex y Cup, en vez de llamarlos C-0 les llamaremos C-1. Es decir, el archivo que contiene el analizador léxico se llamará "C-1.jlex" y el otro "C-1.cup".

18.2 Análisis léxico

Lo primero que vamos a afrontar es la utilización de comentarios. Los comentarios deben estar encerrados entre "/*" y "*/". Como se permiten comentarios anidados, es preciso utilizar estados en el analizador léxico.

Para incluir estados, se deben declarar en la primera sección del archivo .jlex.

```
import java_cup.runtime.Symbol;

import java.lang.System;

import java.io.*;

%%

%full

%unicode

%line

%cup

%char

%state COMENTARIO
```

Debemos incluir una variable global que nos indique el orden de anidamiento de un comentario. Además, cuando se termine de analizar el archivo del programa, se deberá comprobar que los comentarios han sido cerrados todos. Se comprueba viendo si el valor de la variable que indica el orden de anidamiento es 0.

```
%{

int comentarioAnidado = 0;
```

```
String tok = "";

private Symbol Token(int token, Object lexema) throws IOException {

int linea = yyline + 1;

tok = (String)lexema;

if (token != sym.EOF)

InformacionCodigo.guardarInformacionCodigo(linea,tok);

return new Symbol(token, lexema);

}

private Symbol Token(int token) throws IOException {

return Token(token, yytext());

}

%}

%eofval{

{

if(comentarioAnidado > 0){

System.out.println("ERROR LEXICO : No se ha cerrado un comentario");

comentarioAnidado = 0;

}

else return Token(sym.EOF);

}

%eofval}
```

La siguiente sección cambia un poco al permitir estados. Su estructura es la siguiente:

```
<YYINITIAL,COMENTARIO> \r\n|\r|\n { }
```

```
        <YYINITIAL,COMENTARIO> "/*" {

        yybegin(COMENTARIO);

        comentarioAnidado++;

        }

        <COMENTARIO> "*/" {

        comentarioAnidado--;

        if(comentarioAnidado == 0) yybegin(YYINITIAL);

        }

        <YYINITIAL>  "*/"  {System.out.println("ERROR  LEXICO:  Se  intenta  cerrar  un
comentario no abierto"); }

        <COMENTARIO> [^/*] {}

        <COMENTARIO> [/*] {}

        <YYINITIAL> "(" { return Token(sym.LPAREN); }

        <YYINITIAL> ")" { return Token(sym.RPAREN); }

        <YYINITIAL> "[" { return Token(sym.LCOR); }

        <YYINITIAL> "]" { return Token(sym.RCOR); }

        <YYINITIAL> ";" { return Token(sym.PTOCOMA); }

        <YYINITIAL> "." { return Token(sym.PUNTO); }

        <YYINITIAL> "," { return Token(sym.COMA); }

        <YYINITIAL> "+" { return Token(sym.SUMA); }

        <YYINITIAL> "-" { return Token(sym.RESTA); }

        <YYINITIAL> "*" { return Token(sym.PRODUCTO); }

        <YYINITIAL> "/" { return Token(sym.DIVISION); }

        <YYINITIAL> "%" { return Token(sym.MODULO); }
```

```
<YYINITIAL> "<" { return Token(sym.MENOR); }

<YYINITIAL>"<=" { return Token(sym.MENORIGUAL); }

<YYINITIAL> ">=" { return Token(sym.MAYORIGUAL); }

<YYINITIAL> ">" { return Token(sym.MAYOR); }

<YYINITIAL> "==" { return Token(sym.IGUAL); }

<YYINITIAL> "!=" { return Token(sym.DISTINTO); }

<YYINITIAL> "||" { return Token(sym.OR); }

<YYINITIAL> "&&" { return Token(sym.AND); }

<YYINITIAL> "!" { return Token(sym.NOT); }

<YYINITIAL> "=" { return Token(sym.ASIGNAR); }

<YYINITIAL> "{" { return Token(sym.LLLAVE); }

<YYINITIAL> "}" { return Token(sym.RLLAVE); }

<YYINITIAL> "int" { return Token(sym.INT); }

<YYINITIAL> "main" { return Token(sym.MAIN); }

<YYINITIAL> "void" { return Token(sym.VOID); }

<YYINITIAL> "struct" { return Token(sym.STRUCT); }

<YYINITIAL> "if" { return Token(sym.IF); }

<YYINITIAL> "else" { return Token(sym.ELSE); }

<YYINITIAL> "while" { return Token(sym.WHILE); }

<YYINITIAL> "puts" { return Token(sym.PUTS); }

<YYINITIAL> "putw" { return Token(sym.PUTW); }

<YYINITIAL> "break" { return Token(sym.BREAK); }

<YYINITIAL> "return" { return Token(sym.RETURN); }

<YYINITIAL> {CadenaTexto}{ return Token(sym.CADENATEXTO,yytext());}
```

```
<YYINITIAL> {Letra}({Letra}|{Digito})* {return Token(sym.ID,yytext()); }

<YYINITIAL> {Digito}+ {return Token(sym.ENTERO,yytext()); }

<YYINITIAL> (" "|\n|\t|\r)+ { }

<YYINITIAL> . {System.err.println("Caracter no permitido: "+yytext()); }
```

Cuando el analizador encuentra los lexemas para el comienzo de un comentario, incrementa el valor de la variable de anidamiento de comentarios y lanza el estado donde se analizan los comentarios. Cuando el analizador encuentra los lexemas de fin de comentario, si el estado actual es el inicial, lanza un aviso de error de que se cierra un comentario no abierto. Si el estado actual es el de estar en un comentario, disminuye una unidad la variable de anidamiento, y si es 0, sale del estado de comentario y va al estado inicial.

Se han añadido nuevos lexemas. Los corchetes para los vectores, los nuevos operadores y las nuevas palabras reservadas (***void*** y ***struct***). El resto permanece igual que estaba salvo que se ha añadido el estado inicial.

18.3 Análisis sintáctico

Para este análisis utilizamos el archivo de Cup. Le añadiremos reglas nuevas, modificaremos algunas de las que tenía. Añadiremos también variables globales y métodos.

Lo primero es añadir los nuevos terminales:

```
terminal LCOR, RCOR, PUNTO, STRUCT, COMA, MODULO, MAYORIGUAL, MENORIGUAL, VOID,
NOT, RETURN;
```

Después, indicar las precedencias:

```
//Precedencias de los operadores

precedence left ASIGNAR;

precedence left DISTINTO,IGUAL,DISTINTO,MAYOR, MENOR, MAYORIGUAL, MENORIGUAL;

precedence left OR;

precedence left SUMA, RESTA;

precedence left AND;
```

```
precedence right NOT;

precedence left PRODUCTO, DIVISION, MODULO;

precedence left LCOR, RCOR;

precedence left LPAREN, RPAREN;

precedente left PUNTO;

precedente left ELSE;
```

Ahora hay que diseñar las reglas de la gramática modificada.

Un programa consta de una serie de declaraciones de variables globales y/o tipos y un cuerpo. El cuerpo consta de 0 o más declaraciones de subprogramas y de un programa principal. Por lo tanto, las primeras reglas son:

```
Programa ::= Declaraciones Cuerpo | Cuerpo;
Declaraciones ::= DeclVar | DeclTipo | DeclSub;
```

Habrá que añadir a la lista de no terminales los nuevos creados.

non terminal DeclVar, DeclTipo, DeclSub;

A partir de este momento, cuando aparezca algún no terminal nuevo, supondremos que ya se ha añadido a la lista de no terminales.

Como ahora ya se permite la declaración conjunta de variables, debemos diseñar las reglas que lo permitan. Una posible forma de hacerlo es mediante estas reglas:

```
DeclVar ::= Tipo ListaVar PTOCOMA;

ListaVar ::= ListaVar COMA UnaVar | UnaVar;

UnaVar ::= ID;

Tipo ::= ID | INT;
```

Ahora vamos con la definición de tipos estructurados:

```
DeclTipo ::= TipoStruct | TipoVect;

TipoStruct ::= STRUCT ID LLLAVE BloqueStruct RLLAVE PTOCOMA;

BloqueStruct ::= BloqueStruct DefCampo | DefCampo;
```

```
DefCampo ::= INT ID PTOCOMA;

TipoVect ::= INT LCOR ENTERO RCOR ID PTOCOMA;
```

Introduciremos los nuevos operadores:

```
Expresion ::= Expresión SUMA Expresion

| Expresion RESTA Expresion

| Expresion PRODUCTO Expresion

| Expresion DIVISION Expresion

| ENTERO

| ID

| Expresion MODULO Expresion

| SentFuncion

| LPAREN Expresion RPAREN

| CampoRegistro

| ElementoVector

;

CampoRegistro ::= ID PUNTO ID;

ElementoVector ::= ID LCOR Expresion RCOR;
```

Aparece un nuevo no terminal que es la llamada a una función. Más adelante veremos cómo es su gramática.

```
Condicion ::= Condicion OR Condicion
| Condicion AND Condicion
| Expresion IGUAL Expresion
| Expresion DISTINTO Expresion
| Expresion MAYOR Expresion
| Expresion MENOR Expresion
```

```
| Expresion MAYORIGUAL Expresion
| Expresion MENORIGUAL Expresion
| NOT Condicion
| LPAREN Condicion RPAREN
;
```

Se permite que tras una instrucción condicional se pueda poner un bloque de sentencias entre llaves o una sola sentencia y no ponerla entre llaves. Es decir, se permite:

```
if(x>0) x=x+1;
```

O

```
if(x>0) {
x=x+1;
}
```

Las reglas necesarias para esto son:

```
SentIf ::= IF LPAREN Condicion RPAREN BloqueOSentencia
SentElse ;
BloqueOSentencia ::= LLLAVE BloqueSentencias RLLAVE | Sentencia;
SentElse ::= ELSE BloqueOSentencia | ;
SentWhile ::= WHILE LPAREN Condicion RPAREN BloqueOSentencia ;
```

En cuanto a la asignación, podemos asignar a un campo de una variable de tipo estructura un valor y también a un elemento de un vector.

```
SentAsignacion ::= ParteIzq ASIGNAR Expresion PTOCOMA;
ParteIzq ::= ID | ID LCOR Expresion RCOR | ID PUNTO ID;
```

Ya sólo nos quedan las reglas para las sentencias de llamadas a funciones y procedimientos y la declaración de las mismas.

```
DeclSub ::= DeclFunc | DeclProc;

DeclFunc ::= Tipo ID LPAREN ListaArgumentos RPAREN LLLAVE Bloque RLLAVE;

DeclProc ::= VOID ID LPAREN ListaArgumentos RPAREN LLLAVE Bloque RLLAVE;

Bloque ::= DeclaracionesLocales BloqueSentencias | BloqueSentencias;
```

Ahora la lista de argumentos de la declaración:

```
ListaParametros ::= ListaArgumentos COMA Argumento | Argumento | ;

Argumento ::= Tipo ID;
```

Como sólo se permite declarar localmente variables y no definir nuevos tipos, las reglas necesarias son:

```
DeclaracionesLocales ::= DeclaracionesLocales DeclaracionLocal | DeclaracionLocal;

DeclaracionLocal ::= Tipo ListaVarLocal PTOCOMA;

ListaVarLocal ::= ListaVarLocal COMA VarLocal | VarLocal;

VarLocal ::= ID;
```

Nos queda la sentencia de retorno para funciones y la llamada de procedimientos.

```
Sentencia ::= SentIf |

SentWhile |

SentAsignacion |

SentPutw |

SentPuts |

SentBreak |

SentReturn |

SentProcedimiento;

SentReturn ::= RETURN Expresion PTOCOMA;

SentProcedimiento ::= ID LPAREN ListaParametros RPAREN PTOCOMA;

ListaParametros ::= ListaParametros COMA Parametro | Parametros | ;

Parametro ::= Expresion ;
```

Por último, la sentencia de llamada a funciones que será parte de una expresión como ya indicamos anteriormente.

*SentFuncion ::= **ID LPAREN** ListaParametros **RPAREN**;*

Con esto queda finalizado el proceso de análisis sintáctico.

A continuación, se detalla la gramática completa:

```
        //Terminales
        terminal LPAREN, RPAREN, PTOCOMA, SUMA, RESTA, PRODUCTO, DIVISION, MENOR, MAYOR,
IGUAL, DISTINTO;
        terminal OR, AND, ASIGNAR, LLLAVE, RLLAVE, INT, MAIN, IF, ELSE, WHILE, PUTS,
PUTW, BREAK;
        terminal ID, ENTERO, CADENATEXTO;
        terminal LCOR, RCOR, PUNTO, STRUCT, COMA, MODULO, MAYORIGUAL, MENORIGUAL, VOID,
NOT, RETURN;
        // No terminales
        non terminal Programa, Declaraciones, Cuerpo, Declaracion, BloqueSentencias,
Sentencias, Sentencia;
        non terminal Expresion, Condicion;
        non terminal SentIf, SentElse, SentWhile, SentAsignacion, SentPutw, SentPuts,
SentBreak, CampoRegistro, ElementoVector;
        non terminal DeclVar, DeclTipo, DeclSub, ListaVar, UnaVar, Tipo, TipoStruct,
TipoVect;
        non terminal BloqueStruct, DefCampo, SentFuncion, ParteIzq, DeclFunc, DeclProc,
Argumento;
        non terminal ListaArgumentos, DeclaracionesLocales, DeclaracionLocal,
ListaVarLocal, VarLocal;
        non terminal SentProcedimiento, SentReturn, ListaParametros, Parametros,
Parámetro, BloqueOSentencia, Bloque;
        //Precedencias de los operadores
        precedence left ASIGNAR;
        precedence left DISTINTO,IGUAL, DISTINTO, MAYOR, MENOR, MAYORIGUAL, MENORIGUAL;
        precedence left OR;
        precedence left SUMA, RESTA;
        precedence left AND;
        precedence right NOT;
        precedence left PRODUCTO, DIVISION, MODULO;
        precedence left LCOR, RCOR;
        precedence left LPAREN, RPAREN;
        precedence left PUNTO;
        precedence left ELSE;
        //Comienza con un no terminal
        start with Programa;
        //Aqui va la gramatica
        Programa ::= Declaraciones Cuerpo | Cuerpo;
        Declaraciones ::= Declaraciones Declaracion | Declaracion;
        Declaracion ::= DeclVar | DeclTipo | DeclSub;
        DeclVar ::= Tipo ListaVar PTOCOMA;
        ListaVar ::= ListaVar COMA UnaVar | UnaVar;
        UnaVar ::= ID ;
        Tipo ::= ID | INT;
        DeclTipo ::= TipoStruct | TipoVect;
        TipoStruct ::= STRUCT ID LLLAVE BloqueStruct RLLAVE PTOCOMA;
```

BloqueStruct ::= BloqueStruct DefCampo | DefCampo;

DefCampo ::= **INT ID PTOCOMA;**

TipoVect ::= **INT LCOR ENTERO RCOR ID PTOCOMA;**

DeclSub ::= DeclFunc | DeclProc;

DeclFunc ::= Tipo **ID LPAREN** ListaArgumentos **RPAREN LLLAVE** Bloque **RLLAVE;**

DeclProc ::= **VOID ID LPAREN** ListaArgumentos **RPAREN LLLAVE** Bloque **RLLAVE;**

Bloque ::= DeclaracionesLocales BloqueSentencias | BloqueSentencias;

ListaArgumentos ::= ListaArgumentos **COMA** Argumento | Argumento | ;

Argumento ::= Tipo **ID;**

DeclaracionesLocales ::= DeclaracionesLocales DeclaracionLocal | DeclaracionLocal;

DeclaracionLocal ::= Tipo ListaVarLocal **PTOCOMA;**

ListaVarLocal ::= ListaVarLocal **COMA** VarLocal | VarLocal;

VarLocal ::= **ID;**

Cuerpo ::= **MAIN LPAREN RPAREN LLLAVE** BloqueSentencias **RLLAVE;**

BloqueSentencias ::= Sentencias | ;

Sentencias ::= Sentencias Sentencia | Sentencia;

Expresion ::= Expresion **SUMA** Expresion

| Expresion **RESTA** Expresion

| Expresion **PRODUCTO** Expresion

| Expresion **DIVISION** Expresion

| **ENTERO**

| **ID**

| Expresion **MODULO** Expresion

| SentFuncion

| **LPAREN** Expresion **RPAREN**

| CampoRegistro

| ElementoVector

;

CampoRegistro ::= **ID PUNTO ID;**

ElementoVector ::= **ID LCOR** Expresion **RCOR;**

Condicion ::= Condicion **OR** Condicion

| Condicion **AND** Condicion

| Expresión **IGUAL** Expresion

| Expresion **DISTINTO** Expresion

| Expresion **MAYOR** Expresion

| Expresion **MENOR** Expresion

| Expresion **MAYORIGUAL** Expresion

| Expresion **MENORIGUAL** Expresion

| **NOT** Condicion

| **LPAREN** Condicion **RPAREN ;**

SentFuncion ::= **ID LPAREN** ListaParametros **RPAREN;**

SentIf ::= **IF LPAREN** Condicion **RPAREN** BloqueOSentencia
SentElse ;

BloqueOSentencia ::= **LLLAVE** BloqueSentencias **RLLAVE** | Sentencia;

SentElse ::= **ELSE** BloqueOSentencia | ;

SentWhile ::= **WHILE LPAREN** Condicion **RPAREN** BloqueOSentencia ;

SentAsignacion ::= ParteIzq **ASIGNAR** Expresion **PTOCOMA;**

ParteIzq ::= **ID** | **ID LCOR** Expresion **RCOR** | **ID PUNTO ID;**

SentPutw ::= **PUTW LPAREN** Expresión **RPAREN PTOCOMA;**

SentPuts ::= **PUTS LPAREN CADENATEXTO RPAREN PTOCOMA;**

SentBreak ::= **BREAK PTOCOMA;**

SentReturn ::= **RETURN** Expresion **PTOCOMA;**

SentProcedimiento ::= **ID LPAREN** ListaParametros **RPAREN PTOCOMA;**

ListaParametros ::= ListaParametros **COMA** Parametro | Parametros | ;
Parametro ::= Expresion ;
Sentencia ::= SentIf |
SentWhile |
SentAsignacion |
SentPutw |
SentPuts |
SentBreak |
SentReturn |
SentProcedimiento;

CAPÍTULO 19

ANÁLISIS SEMÁNTICO DE C-1

19.1 Introducción

En este capítulo, más extenso que los anteriores, vamos a realizar el análisis semántico. Esta tarea ha sido ya implementada en parte cuando se hizo la práctica del compilador para **C-0**. Ahora sólo vanos añadir el código nuevo y modificar el que sea necesario.

Un aspecto importante es la tabla de símbolos. Para C-0 la tabla de símbolos era bastante simple porque no utilizábamos ámbitos. Pero ahora ya tenemos subprogramas y, además, se permite la declaración de variables locales en los subprogramas. Estas nuevas funcionalidades requieren una gestión de ámbitos. Por lo tanto, la nueva tabla de símbolos (o tablas de símbolos, según vayamos a utilizar una técnica u otra) debe tener en cuenta los ámbitos.

Otra nueva funcionalidad es la creación de tipos globales. Esto requiere una tabla de tipos global. Anteriormente, como sólo se utilizaba un tipo, no era necesario mantener una tabla de tipos complicada. Como no se permite la definición de tipos locales, con una sola tabla de tipos global tenemos suficiente.

19.2 La tabla de tipos

Veamos la tabla de tipos global. Los campos necesarios son:

- *Código*: indica el número de orden en la tabla. El primero es el 0 y así sucesivamente.
- *Identificador*: es el nombre del tipo.
- *Tipo base*: en el caso de tipos estructurados, el tipo base es el tipo al que pertenece cada uno de los elementos. Por ejemplo, si tenemos un vector de enteros, el tipo base será el tipo entero. El tipo base de un campo de una estructura será el tipo del campo.
- *Dimensión*: indica la dimensión en unidades del tipo. Por ejemplo, el tipo entero tiene dimensión 1. Un vector de 10 enteros tendrá dimensión 10. En general, la dimensión de un tipo estructurado será la suma de las dimensiones de todos sus componentes.
- *Máximo*: indica, en vectores, el índice mayor. Por ejemplo, si definimos un vector de 6 elementos (del 0 al 5), el índice máximo sería el 5. No confundirlo con la dimensión.

- *Padre*: indica, para el caso de los campos de los registros, el código del tipo del registro al que pertenece.

Como sólo se permite definir tipos globales, todos son del ámbito 0, por lo que es innecesario incluir un campo en la tabla de tipos para señalar el ámbito.

La clase base para cada entrada en la tabla de tipos es:

```
class Tipo {

private int cod;

private String id;

private int tipoBase;

private int dimension;

private int maximo;

privateint padre;

Tipo() {

cod = -1;

id = “”;

tipoBase = -1;

dimension = 0;

maximo = -1;

padre = -1;

}

Tipo(int c, String i) {

cod = c;

id = i;

tipoBase = -1;
```

```
dimension = 0;

maximo = -1;

padre = -1;

}

void setCod(int c) {

cod = c;

}

void setId(String i) {

id = I;

}

int getCod() {

return cod;

}

String getId() {

return id;

}

void setTipoBase(int tb) {

tipoBase = tb;

}

int getTipoBase() {

return tipoBase;

}

void setDimension(int d) {

dimension = d;
```

```
}

int getDimension() {

return dimension;

}

void setMaximo(int m) {

maximo = m;

}

int getMaximo() {

return maximo;

}

void setPadre(int p) {

padre = p;

}

int getPadre() {

return padre;

}

}
```

Si más adelante necesitamos algún constructor más o algún método más, lo añadiremos a la clase.

Para la gestión de la tabla de tipos, hemos utilizado una clase conjuntamente con la tabla de símbolos. La clase se llama *Tabla*. Dentro, hay un atributo que se llama *tablaTipos* (de tipo *Vector*) que contiene una lista de elementos de la clase *Tipo*.

Ya tenemos algunos métodos que habíamos implementado para **C-0**. Conforme sea necesario, añadiremos nuevos métodos. Queda decir sólo que el constructor de la tabla añade por defecto el tipo primitivo “int”.

```
import java.util.Vector;
class Tabla {
```

```
private Vector tablaSimbolos;
private Vector tablaTipos;
Tabla() {
tablaSimbolos = new Vector();
tablaTipos = new Vector();
Tipo t = new Tipo(0,”int”);
t.setDimension(1);
addTipo(t);
}
void addTipo(Tipo t) {
tablaTipos.add(t);
}
void addTipo(String id) {
tablaTipos.add(new Tipo(countTipos(),id));
}
……..
……..
}
```

19.3 La tabla de símbolos

La gestión de la tabla de símbolos se complica respecto a la empleada para **C-0**. En este caso, debemos tener en cuenta los ámbitos. Los símbolos globales pertenecen al ámbito inicial, por ejemplo el 0. Conforme entramos en el código de un subprograma debemos incrementar el ámbito en una unidad. Cuando salimos, eliminamos todo el ámbito y lo decrementamos en una unidad. El eliminar todo el ámbito consiste en eliminar de la tabla de símbolos todos los símbolos de ese ámbito (ya que hemos optado por mantener una tabla única para todos los ámbitos).

Los campos necesarios para cada entrada en la tabla de símbolos son:

- *Código*: indica el número de orden dentro de la tabla. Comienza con el 0.
- *Identificador*: es el nombre del símbolo.
- *Dirección*: indica la dirección en memoria donde estará el valor. Puede ser absoluta o relativa.
- *Tipo*: es el tipo al que pertenece el símbolo.
- *Categoría*: indica si el símbolo es el de una variable, una función, un procedimiento o un argumento de un subprograma.
- *Número de parámetros*: indica el número de parámetros que admite un subprograma.
- *Parámetros*: es una lista con los tipos de los parámetros que admite un subprograma.
- Ámbito: indica el ámbito del símbolo.

La clase que vamos a utilizar para cada entrada en la tabla de símbolos es la clase *Simbolo*. Inicialmente, podemos tener esta estructura de clase, aunque si más adelante necesitamos implementar algunos métodos más, lo haremos.

```
import java.util.Vector;

class Simbolo {

private int cod;

private String id;

private int direccion;

private int tipo;

private String categoria;

private int numeroParametros;

private Vector parametros;

private int ambito;

Simbolo() {

cod = -1;

id = "";

direccion = -1;

tipo = -1;

categoria = "";

numeroParametros = 0;

parametros = new Vector();

ambito = 0;

}

Simbolo(int c, String i) {
```

```
cod = c;

id = i;

direccion = -1;

tipo = -1;

categoria = “”;

numeroParametros = 0;

parametros = new Vector();

ambito = 0;

}

void setCod(int c) {

cod = c;

}

void setId(String i) {

id = i;

}

int getCod() {

return cod;

}

String getId() {

return id;

}

void setDireccion(int d) {

direccion = d;

}
```

```
int getDireccion() {

return direccion;

}

void setTipo(int t) {

tipo = t;

}

int getTipo() {

return tipo;

}

void setCategoria(String c) {

categoria = c;

}

String getCategoria() {

return categoria;

}

void setNumeroParametros(int np) {

numeroParametros = np;

}

int getNumeroParametros() {

return numeroParametros;

}

void addParametro(int p) {

parametros.addElement(new Integer(p));

}
```

```
int getParametro(int pos) {

return ((Integer)parametros.elementAt(pos)).intValue();

}

void setAmbito(int a) {

ambito = a;

}

int getAmbito() {

return ambito;

}

}
```

Al igual que la tabla de tipos, la tabla de símbolos se engloba dentro de una clase que se llama *Tabla*. Por ahora, los métodos que tenemos para manipular la tabla de símbolos son:

```
import java.util.Vector;

class Tabla {

private Vector tablaSimbolos;

……

Tabla() {

tablaSimbolos = new Vector();

tablaTipos = new Vector();

Tipo t = new Tipo(0,”int”);

t.setDimension(1);

addTipo(t);

}

void addSimbolo(String id) {
```

```
tablaSimbolos.add(new Simbolo(countSimbolos(),id));
}
int countSimbolos() {
return tablaSimbolos.size();
}
Simbolo getSimbolo(int pos) {
return (Simbolo)tablaSimbolos.elementAt(pos);
}
Simbolo getSimbolo(String id) {
Simbolo simbolo = null;
//La busqueda comienza desde el final hacia el principio
//Esto se hace asi para cuando tengamos variables locales
//declaradas con el mismo nombre que las globales. Se
//buscara primero la local, es decir la ultima en la //tabla
for(int i=countSimbolos()-1;i>=0;i--) {
simbolo = getSimbolo(i);
if(simbolo.getId().equals(id)) {
break;
} else {
simbolo = null;
}
}
return simbolo;
}
```

```
boolean existeSimbolo(String id) {

if(getSimbolo(id)!=null) {

return true;

} else {

return false;

}

}

void setSimbolo(Simbolo s) {

int cod = s.getCod();

tablaSimbolos.setElementAt(s,cod);

}

void setDireccionSimbolo(String id,int d) {

Simbolo simbolo = getSimbolo(id);

simbolo.setDireccion(d);

setSimbolo(simbolo);

}

……..

}
```

Iremos añadiendo o modificando los métodos conforme nos vaya haciendo falta.

19.4 Análisis semántico

Antes de nada, indicar que vamos a utilizar el código que tenemos para **C-0**. Por lo tanto, las variables globales van a ser las mismas y los métodos que ya teníamos también. Solamente vamos a modificar las acciones que teníamos en la gramática para adaptarlas a los cambios.

Todos los métodos a que no refiramos aquí se implementarán en la sección *action code* del archivo "C-1.cup". Además, también incluiremos en esa sección las variables

globales que necesitemos.

Para recordar lo que ya teníamos, aquí se detallan las variables globales que tenemos actualmente:

```
action code {:

Tabla tabla;

int cuentaWhiles;

int cuentaDirecciones;

CodigoIntermedio codigoIntermedio;

int cuentaIf;

Pila pilaIf;

int cuentaBucle;

Pila pilaBucle;

int cuentaCadenas;

Lista listaCadenas;

……

:}
```

Además, el método que inicializa todo el proceso:

```
action code {:

…….

void inicializar() {

tabla = new Tabla();

cuentaWhiles = 0;

cuentaDirecciones = 9999;

cuentaIf = 0;
```

```
pilaIf = new Pila();

cuentaBucle = 0;

pilaBucle = new Pila();

cuentaCadenas = 0;

listaCadenas = new Lista();

String nombre = parser.nombreFichero.substring(0

,parser.nombreFichero.lastIndexOf(".”));

codigoIntermedio = new CodigoIntermedio(nombre+”.ci”);

try {

codigoIntermedio.abrirFicheroEscritura();

} catch (IOException e) {

System.out.println(Textos.ficheroCiNoExiste);

codigoIntermedio.cerrarFicheroEscritura();

}

}

…….

:}
```

En esta sección sólo añadiremos las acciones semánticas enfocadas al análisis semántico y a la manipulación de la tabla de tipos y símbolos. Más adelante, incluiremos las de la generación de código.

Sólo indicaremos aquellas reglas afectadas por el análisis semántico, el resto no incluirán por ahora acciones de ningún tipo.

19.4.1 Definición del tipo struct

Comprobaremos primero que cuando se defina un nuevo tipo registro (struct) no esté ya definido en la tabla de tipos. Si no está definido, lo añadirá. Utilizaremos la clase *Textos* para informar de los posibles mensajes de error. Además, añadiremos los métodos que vayamos necesitando.

```
TipoStruct ::= STRUCT ID:id

{:

if(existeTipo(id)) {

parser.error(Textos.existeTipo);

} else {

addTipo(id);

}

:}

LLLAVE BloqueStruct RLLAVE PTOCOMA;

class Textos{

……..

static final String existeTipo = "El tipo definido ya existe";

…….

}
```

En adelante, no mostraremos la estructura de la clase *Textos* ya que es similar.

En cuanto a los dos métodos que hemos utilizado, su definición está en la sección *action code*. A partir de ahora, entenderemos que todos los métodos que utilicemos estarán en esa sección.

```
action code {:

…….

boolean existeTipo(String id) {

return tabla.existeTipo(id);

}

void addTipo(String id) {

tabla.addTipo(id);
```

```
}

……..

:}
```

Hemos comprobado si el tipo estaba ya en la tabla de tipos. Si ya estaba, lanzará un mensaje de error y si no estaba, lo añadirá. Inicialmente la dimensión es 0. Pero conforme le añadamos campos, debemos aumentar la dimensión. Esto es algo más complicado. Veremos cómo hacerlo.

```
DefCampo ::= INT ID:id

{:

if(existeCampoRegistro(id)) {

parser.error(Textos.existeCampo);

} else {

addTipoCampo(id);

}

:}

PTOCOMA;
```

Vemos que cada vez que se añade un campo se debe verificar que no estaba ya definido en el mismo registro, y si ya estaba, lanza un error. Si no estaba antes definido, lo añade. Al añadirlo, debe modificarse la dimensión del registro. Los métodos implicados son:

```
boolean existeCampoRegistro(String id) {

  return tabla.existeCampoRegistro(id);

  }

  void addTipoCampo(String id) {

  tabla.addTipoCampo(id);

  }
```

Estos dos métodos utilizan sendos métodos en la clase *Tabla*:

```
boolean existeCampoRegistro(String id) {

//Como el tipo base de un registro es -1,

//eso nos va a indicar cuando hemos llegado al tipo base

boolean retorno = false;

int i = countTipos()-1;

while(i>=0) {

Tipo t = getTipo(i);

if(t.getTipoBase()<0) {

break;

} else {

if(t.getId().equals(id)) {

retorno = true;

break;

}

}

i--;

}

return retorno;

}

void addTipoCampo(String id) {

addTipo(id);

Tipo t = getTipo(id);

t.setDimension(1);

t.setTipoBase(0);
```

```
//Como el tipo base de un registro es -1,

//eso nos va a indicar cuando hemos llegado al tipo base

int i = countTipos()-1;

while(i>=0) {

Tipo j = getTipo(i);

if(j.getTipoBase()<0) {

j.setDimension(j.getDimension()+1);

t.setPadre(i);

break;

}

i--;

}

}
```

Ahora debemos hacer una elección en cuanto a la especificación. En ningún momento se ha dicho en la especificación si está permitido dar el mismo nombre de un campo de un registro que de una variable. Es decir, debemos decidir si se debe aceptar estos:

```
struct x {

int a;

int b;

}

int a;
```

Si se acepta, debemos modificar el método que comprueba si un tipo está ya en la tabla de tipos. Vamos a aceptar esa posibilidad. El método deberá tener esta implementación:

```
boolean existeTipo(String id) {
```

```
boolean retorno = false;

for(int i=countTipos()-1;i>=0;i--) {

Tipo t = getTipo(i);

if(t.getId().equals(id) && t.getPadre()<0) {

retorno = true;

break;

}

}

return retorno;

}
```

Una consideración más. En principio vamos a permitir que se pueda declarar una variable con un nombre igual a la de un tipo. Esto parece chocante, pero sintácticamente no crea ningún problema.

19.4.2 Definición del tipo vector

Debido a las restricciones que hemos puesto a la hora de especificar la gramática, hay pocas comprobaciones semánticas que haya que hacer para vectores. Solamente, hay que comprobar que el nombre del tipo no esté ya declarado en la tabla de tipos.

```
TipoVect ::= INT LCOR ENTERO:e RCOR ID:id

{:

if(existeTipo(id)) {

parser.error(Textos.existeTipo);

} else {

addTipo(id,(new Integer(e)).intValue());

}

:}

PTOCOMA;
```

El nuevo método empleado es:

```
  void addTipo(String id, int e) {

  tabla.addTipo(id,e);

  }
```

En la clase *Tabla*:

```
void addTipo(String id, int e) {

  Tipo t = new Tipo(countTipos(),id);

  t.setTipoBase(0);

  t.setDimension(e+1);

  t.setMaximo(e);

  tablaTipos.addElement(t);

  }
```

19.4.3 Declaración de variables globales

En esta sección vamos a comprobar que una variable global no ha sido declarada antes. Además, al mismo tiempo le asignaremos una dirección en su entrada de la tabla de símbolos. La asignación de dirección a una variable global va a estar referida a un índice. Es decir, la primera dirección que se va a utilizar es la 0 respecto a la zona donde estarán las variables globales. Por ejemplo, ese índice puede valer 10000 y, por tanto, la primera variable global ocupará la dirección de memoria 10000.

Podríamos haber partido la memoria en dos zonas, una para las direcciones de las variables globales y temporales globales y otra para los registros de activación (argumentos de subprogramas, variables locales y temporales locales). Pero el enfoque que vamos a tomar es suponer que el primer registro de activación va a ser el del programa principal. Los demás se irán poniendo a continuación de este. Por lo tanto, la primera dirección útil para el programa principal va a ser la 0 (relativa a un registro índice que será creado al principio del programa).

Ya tenemos un contador global de direcciones que se llama *cuentaDirecciones*.Vamos a inicializarlo con el valor 10000. Cada vez que debamos asignar una dirección, asignamos la dirección y luego lo incrementamos tantas unidades como dimensión tenga el tipo de la variable.

```
action code {:

……..

int cuentaDirecciones;

……….

void inicializar() {

……..

cuentaDirecciones = 10000;

……….

}

……….

:}
```

Un aspecto que también hay que tener en cuenta es que una variable puede ocupar más de una dirección. Por ejemplo, si la variable es de un tipo de vector con 10 elementos, la variable ocupará 10 direcciones consecutivas. En su entrada de la tabla de símbolo sólo pondremos la primera que ocupa, pero la siguiente entrada ocupará 10 direcciones después.

En la regla donde se declaran variables, debemos poner el tipo de una variable al tiempo que la introducimos en la tabla de símbolos. Para ello, podemos crear una lista de todas las variables que se declaran a la vez y luego asignar el tipo recorriendo la lista e introduciendo los datos en la tabla de símbolos. Otra opción sería crear una variable global que guarde el tipo cuando aparezca el tipo y cuando seguidamente se vayan declarando los nombres de las variables, se le asigna el tipo a la entrada en la tabla de símbolos. Nosotros vamos a utilizar esta técnica por su simplicidad. Por lo tanto, vamos a crear una nueva variable global que la podemos llamar *tipoActual*.

```
action code {:

……….

Tipo tipoActual = null;

……….

:}
```

Antes de poner el tipo actual, debemos comprobar que ese tipo está definido en la tabla de tipos.

Pero en este momento, el analizador va a encontrar un conflicto en la gramática. Se va a encontrar que no tiene potencia de cálculo para decidir cuál de estas dos reglas aplicar, *DeclVar* y *DeclFunc*:

```
Declaracion ::= DeclVar | DeclTipo | DeclSub;

DeclVar ::= Tipo ListaVar PTOCOMA;

DeclFunc ::= Tipo ID LPAREN ListaArgumentos RPAREN LLLAVE Bloque RLLAVE;

Tipo ::= ID | INT;

ListaVar ::= ListaVar COMA UnaVar | UnaVar;
```

Esto se debe a que el analizador sólo puede leer un símbolo como preanálisis y como encuentra dos reglas que comienzan con el mismo símbolo (*ID* que viene del no terminal *Tipo*) no sabe cuál de las dos aplicar.

Por ejemplo, ante la siguiente secuencia de tokens *int f*, no sabe cuál de las reglas aplicar. El siguiente código es un ejemplo de esta situación:

```
int f,g;

int f() {

return 1;

}
```

Debemos modificar la gramática para que el analizador no tenga estos problemas de decisión. Esta situación es probable que la encontremos a lo largo del diseño del compilador. Por lo que deberemos estar dispuestos a realizar cambios en la gramática a lo largo de todo el proceso de creación del compilador.

Una manera de atajar el problema es “sacando factor común”. Por ejemplo:

```
CabeceraDecl ::= Tipo UnaVar;

UnaVar ::= ID;

DeclVar ::= CabeceraDecl ListaVar PTOCOMA;
```

```
ListaVar ::= ListaVar COMA UnaVar | ;

Tipo ::= ID | INT ;

DeclFunc ::= CabeceraDecl LPAREN ListaArgumentos RPAREN LLLAVE Bloque RLLAVE;
```

Ahora ya podemos generar las acciones semánticas adecuadas. Pero tenemos ahora un escollo. Se trata de que cuando nos encontremos una cabecera de declaración no sabemos la categoría a que pertenece el símbolo. No sabemos si se trata de una variable o una función. Por lo tanto, ese dato habrá que añadirlo después de haber insertado el símbolo en la tabla de símbolos. Por defecto, vamos a poner siempre que es una variable y cuando sea una función, modificaremos la entrada correspondiente en la tabla de símbolos.

Las reglas quedan así:

```
CabeceraDecl ::= Tipo:tp
{:
if(existeTipo(tp)) {
tipoActual = getTipo(tp);
} else {
parser.error(Textos.noExisteTipo);
}
:}
UnaVar;

DeclVar ::= CabeceraDecl ListaVar PTOCOMA;

ListaVar ::= ListaVar COMA UnaVar | ;

UnaVar ::= ID:id
{:
if(existeSimbolo(id)) {
parser.error(Textos.simboloRedeclarado);
} else {
addSimbolo(id);
setTipoSimbolo(id,tipoActual.getCod());
setCategoriaSimbolo(id,”variable”);
setDireccionSimbolo(id,cuentaDirecciones);
cuentaDirecciones = cuentaDirecciones + tipoActual.getDimension();
}
:}
;

Tipo ::= ID:i
{:
RESULT = i;
:}

| INT
{:
RESULT = new String(“int”);
:}
```

```
;
DeclFunc ::= CabeceraDecl
{:
setCategoriaUltimoSimbolo("funcion");
cuentaDirecciones = cuentaDirecciones - tipoActual.getDimension();
:}
LPAREN ListaArgumentos RPAREN LLLAVE Bloque RLLAVE;
```

Los métodos son:

```
void addSimbolo(String id) {

tabla.addSimbolo(id);

}

void setDireccionSimbolo(String id,int dir) {

tabla.setDireccionSimbolo(id,dir);

}

void setTipoSimbolo(String id,int tp) {

tabla.setTipoSimbolo(id,tp);

}

void setCategoriaSimbolo(String id, String c) {

tabla.setCategoriaSimbolo(id,c);

}

void setCategoriaUltimoSimbolo(String c) {

tabla.setCategoriaUltimoSimbolo(c);

}
```

Y en la clase *Tabla*:

```
void setSimbolo(Simbolo s) {

int cod = s.getCod();

tablaSimbolos.setElementAt(s,cod);
```

```
}

void setDireccionSimbolo(String id,int d) {

Simbolo simbolo = getSimbolo(id);

simbolo.setDireccion(d);

setSimbolo(simbolo);

}

void setTipoSimbolo(String id,int t) {

Simbolo simbolo = getSimbolo(id);

simbolo.setTipo(t);

setSimbolo(simbolo);

}

void setCategoriaSimbolo(String id,String c) {

Simbolo simbolo = getSimbolo(id);

simbolo.setCategoria(c);

setSimbolo(simbolo);

}

void setCategoriaUltimoSimbolo(String c) {

Simbolo simbolo = getSimbolo(countSimbolos()-1);

simbolo.setCategoria(c);

setSimbolo(simbolo);

}
```

Un aspecto a destacar es que como hemos tratado la cabecera de una función como si se tratara de la declaración de una variable, el contador de direcciones lo hemos incrementado tantas unidades como dimensión del tipo devuelto de la función. Esto no es necesario ya que más adelante, cuando tratemos las expresiones, se asignarán las

direcciones oportunas a la expresión que toma el valor devuelto por la función. Por lo tanto, no hay que reservar direcciones para el tipo devuelto por la función. Restauraremos el contador de direcciones a su valor antes de toparse con la cabecera de una función. Eso lo hace este trozo de código:

```
DeclFunc ::= CabeceraDecl

{:

setCategoriaUltimoSimbolo("funcion");

cuentaDirecciones = cuentaDirecciones - tipoActual.getDimension();

:}

LPAREN ListaArgumentos RPAREN LLLAVE Bloque RLLAVE;
```

En resumen, sólo se reservan direcciones para las variables, pero no se reservan ni para las funciones, ni para los procedimientos, ni para los tipos.

Dentro de los subprogramas, se reserva memoria para los argumentos y las variables locales.

Aparte de la reserva de estas direcciones, se reservarán las que hagan falta como temporales para guardar datos temporales. Asimismo, se reservarán direcciones para los RA.

19.4.4 Declaración de variables locales

La declaración de variables locales es parecida a la de las variables globales. Según nuestra gramática sólo se permite declarar variables locales a continuación de la cabecera del subprograma y antes del código.

Tanto los argumentos de un subprograma como las variables locales del mismo pertenecen a un ámbito superior, en nuestro caso al ámbito 1 (el ámbito global es el 0). Como no se permite definir subprogramas anidados, sólo existen dos ámbitos, el global y uno local para cada subprograma.

Cuando el analizador se encuentra con la declaración de un subprograma, este lo inserta en la tabla de símbolos con el ámbito 0, pero a continuación, el ámbito que utiliza para los argumentos y las variables locales es el 1. Por lo tanto, al declarar los argumentos y las variables locales, se insertarán en la tabla de símbolos con el número de ámbito 1. Ahora es el momento de indicar que es posible que se declare un símbolo con el mismo nombre pero en ámbitos diferentes. Es decir, si tenemos una variable global con el nombre "x", podemos insertar en la tabla de símbolos otra variable local con el mismo nombre (ya que se diferencian en el ámbito). Cuando intentemos acceder dentro del ámbito local a un subprograma a una variable, miramos en la tabla de símbolos desde el

final hasta el principio hasta que encontremos un símbolo con el mismo nombre. Es decir, primero accedemos a las variables locales y luego a las globales en caso de que no exista una variable con el nombre que buscamos.

Para declarar una variable local, debemos asegurarnos de que no hay otra declarada en el mismo ámbito. Por lo tanto, la acción semántica a la hora de declarar una variable local es buscar que no esté ya en la tabla de símbolos pero en el mismo ámbito.

En cuanto al direccionamiento, direccionamos localmente. Es decir, comenzamos por la dirección local 0 y vamos incrementándola conforme vayamos necesitando. Para ello, debemos utilizar otro contador global de direcciones que pondremos a 0 al comienzo de la declaración de cada subprograma. A esta variable global la vamos a llamar *direccionLocal*. La debemos incluir en la zona correspondiente de la sección *action code*.

```
action code {:

…….

int direccionLocal;

……..

:}
```

Un detalle más, cuando salgamos del código de un subprograma debemos eliminar todo su ámbito local. Es decir, debemos eliminar desde el final de la tabla de símbolos hacia el principio todos los símbolos con el ámbito 1.

Con un ejemplo lo veremos mejor:

```
Línea 1 à int x;

Línea 2 à int f(int a) {

Línea 3 à int x;

Línea 4 à x = a;

Línea 5 à return x;

Línea 6 à }

Línea 7 à main() {
```

```
Línea 8 à x = f(2);

Línea 9 à }
```

Veremos el contenido de la tabla de símbolos tras procesar cada línea del programa.

0	x	0	0	variable	0	[]	0

0	x	0	0	variable	0	[]	0
1	f	1	0	función	1	[0]	0
2	a	0	0	parámetro	0	[]	1

0	x	0	0	variable	0	[]	0
1	f	1	0	función	1	[0]	0
2	a	0	0	parámetro	0	[]	1
3	x	1	0	variable	0	[]	1

0	x	0	0	variable	0	[]	0
1	f	1	0	función	1	[0]	0

Las reglas son:

```
DeclaracionesLocales ::= DeclaracionesLocales DeclaracionLocal | DeclaracionLocal;
DeclaracionLocal ::= Tipo:tp
{:
if(existeTipo(tp)) {
tipoActual = getTipo(tp);
} else {
parser.error(Textos.noExisteTipo);
}
:}
ListaVarLocal PTOCOMA;
ListaVarLocal ::= ListaVarLocal COMA VarLocal | VarLocal;
VarLocal ::= ID:id
{:
if(existeSimboloAmbito(id,1)) {
parser.error(Textos.simboloRedeclarado);
} else {
addSimbolo(id);
setTipoSimbolo(id,tipoActual.getCod());
setCategoriaSimbolo(id,"variable");
setDireccionSimbolo(id,direccionLocal);
setAmbitoSimbolo(id,1);
direccionLocal = direccionLocal + tipoActual.getDimension();
}
:}
```

```
  ;
```

Los métodos son:

```
void setAmbitoSimbolo(String id, int a) {
  tabla.setAmbitoSimbolo(id,a);
  }
  boolean existeSimboloAmbito(String id, int a) {
  return tabla.existeSimboloAmbito(id,a);
  }
```

Y en la clase *Tabla*:

```
void setAmbitoSimbolo(String id, int a) {

  for(int i=countSimbolos()-1;i>=0;i--) {

  Simbolo s = getSimbolo(i);

  if(s.getId().equals(id)) {

  s.setAmbito(a);

  setSimbolo(s);

  break;

  }

  }

  }

  boolean existeSimboloAmbito(String id, int a) {

  boolean retorno = false;

  for(int i=countSimbolos()-1;i>=0;i--) {

  Simbolo s = getSimbolo(i);

  if(s.getId().equals(id) && s.getAmbito()==a) {

  retorno = true;

  break;

  }
```

```
}

return retorno;

}
```

19.4.5 Declaración de subprogramas

Las comprobaciones semánticas que hay que hacer son únicamente para las funciones (a excepción de la comprobación de que el nombre del subprograma no esté ya en la tabla de símbolos). Hay que comprobar que contienen al menos una sentencia de retorno (utilizaremos una variable global que se llame *hayRetorno* y que es de tipo booleano).

Para las funciones, ya hemos realizado las comprobaciones semánticas a la hora de estudiar su cabecera. Nos queda introducir en la tabla los procedimientos. En este caso, tras la comprobación de que el nombre no está ya utilizado en la tabla, se introduce con el código de tipo devuelto -1 ya que no se devuelve nada. Además, no se consume ninguna variable global (ya que no hay que depositar el valor devuelto en ninguna parte).

Tanto en funciones como en procedimientos, después de insertar el nombre en la tabla de símbolos, hay que poner a 0 el contador de direcciones locales.

Al final de la declaración de un subprograma hay que eliminar su ámbito local.

Las reglas son:

```
DeclFunc ::= CabeceraDecl

{:

setCategoriaUltimoSimbolo("funcion");

direccionLocal = 0;

hayRetorno = false;

:}

LPAREN ListaArgumentos RPAREN LLLAVE Bloque

{:

eliminarAmbito(1);

if(hayRetorno == false) {
```

```
parser.error(Textos.noHayRetorno);

}

:}

RLLAVE ;

DeclProc ::= VOID ID:id

{:

if(existeSimbolo(id)) {

parser.error(Textos.simboloRedeclarado);

} else {

addSimbolo(id);

setCategoriaSimbolo(id,”procedimiento”);

direccionLocal = 0;

}

:}

LPAREN ListaArgumentos RPAREN LLLAVE Bloque

{:

eliminarAmbito(1);

:}

RLLAVE;

SentReturn ::= RETURN Expresion PTOCOMA

{:

hayRetorno = true;

:}
```

```
;
```

Los métodos son:

```
void eliminarAmbito(int a) {

tabla.eliminarAmbito(a);

}
```

Y en la clase *Tabla*:

```
void eliminarAmbito(int a) {

int i = countSimbolos()-1;

while(i>0) {

Simbolo s = getSimbolo(i);

if(s.getAmbito()>=a) {

tablaSimbolos.removeElementAt(i);

i--;

} else {

break;

}

}

}
```

19.4.6 Argumentos de subprogramas

Un subprograma puede no tener argumentos. Si los tiene, deben ir separados por una coma. No se permite el agrupamiento de argumentos, es decir, cada uno hay que declararlo independientemente con el tipo seguido del nombre del argumento.

Las comprobaciones semánticas que hay que hacer es comprobar que el tipo existe en la tabla de tipos y que el nombre del argumento no está ya declarado antes. La única manera de que esté ya declarado es que sea un argumento dentro de la declaración del mismo subprograma. Esto es así porque los argumentos corresponden al ámbito local del subprograma y como son lo primero que se declara, es imposible que exista ninguna

variable con ese nombre ya que aún no se han declarado.

En cuanto al direccionamiento, corresponde al direccionamiento local. Es decir, el primer argumento tiene la dirección relativa 0 y así sucesivamente. El ámbito es el 1. Cada vez que añadamos un argumento, hay que buscar la entrada del subprograma correspondiente y añadirle al campo *numeroParametros* una unidad y además añadir a la lista de parámetros el tipo del nuevo argumento.

Las reglas son:

```
Argumento ::= Tipo:tp ID:id

{:

if(existeTipo(tp)) {

if(existeSimboloAmbito(id,1)) {

parser.error(Textos.simboloRedeclarado);

} else {

addSimbolo(id);

setTipoSimbolo(id,getTipo(tp).getCod());

setCategoriaSimbolo(id,”parametro”);

setDireccionSimbolo(id,direccionLocal);

etAmbitoSimbolo(id,1);

direccionLocal = direccionLocal + getTipo(tp).getDimension();

setParametroUltimoSubprograma(getTipo(tp));

}

} else {

parser.error(Textos.noExisteTipo);

}

:}
```

```
;
```

Los métodos:

```
void setParametroUltimoSubprograma(Tipo tp) {

  tabla.setParametroUltimoSubprograma(tp);

  }
```

Y en la clase *Tabla*:

```
  Simbolo getUltimoSubprograma() {

  Simbolo s = null;

  for(int i=countSimbolos()-1;i>=0;i--) {

  s = getSimbolo(i);

  if(s.getCategoria().equals("funcion")                                    ||
s.getCategoria().equals("procedimiento")) {

  break;

  } else {

  s = null;

  }

  }

  return s;

  }

  void setParametroUltimoSubprograma(Tipo tp) {

  Simbolo s = getUltimoSubprograma();

  s.setNumeroParametros(s.getNumeroParametros()+1);

  s.addParametro(tp.getCod());

  }
```

19.4.7 Expresiones

La comprobación semántica que hay que hacer en las expresiones es que los operandos son compatibles en cuanto al tipo. Cuando una expresión es una variable, además hay que comprobar que dicha variable existe en la tabla de símbolos.

Vamos a utilizar una nueva clase para guardar la información referente a las expresiones. Esta nueva clase, ya utilizada para **C-0**, se llamará *Expresion*.

Los atributos que va a tener son:

1. *Dirección*: es la dirección donde se va a guardar el valor de la expresión.
2. *Tipo*: es el tipo al que pertenece la expresión y que se hereda de una expresión a otra. En realidad, guardamos sólo su código de la tabla de tipos.

Si necesitamos más atributos, los añadiremos más adelante.

```
Class Expresion {

int direccion;

int tipo;

Expresión(int d) {

direccion = d;

tipo = -1;

}

Expresion(int d, int t) {

direccion = d;

tipo = t;

}

int getDireccion() {

return direccion;

}
```

```
void setDireccion(int d) {

direccion = d;

}

int getTipo() {

return tipo;

}

void setTipo(int t) {

tipo = t;

}

}
```

Debemos declarar los no terminales de las expresiones como del tipo *Expresion*.

non terminal Expresion Expresion;

La primera expresión que vamos a analizar es:

Expresion ::= ***ENTERO****;*

En este caso, habrá que devolver un objeto del tipo *Expresion*. El tipo debe ser "int". A la hora de asignar direcciones, hemos dicho anteriormente que todos los RA van a estar referidos respecto a un índice, por lo que como ese índice es la dirección local, todas las temporales generadas serán locales, incluso para el ámbito global.

Debemos implementar un mecanismo que nos informe si estamos en un ámbito global o local. Por ejemplo, podemos utilizar una variable que nos diga el número de ámbito en que estamos. Por ejemplo, la podemos llamar *ambitoActual*. Su valor será 0 para el ámbito global y 1 para el local.

```
action code {:

…….

int ambitoActual = 0;

…….

:}
```

Esta variable tomará el valor 1 cuando entremos en el ámbito de los

subprogramas y volverá a valer 0 cuando salgamos:

```
DeclFunc ::= CabeceraDecl
{:
setCategoriaUltimoSimbolo("funcion");
direccionLocal = 0;
cuentaDirecciones = cuentaDirecciones - tipoActual.getDimension();
hayRetorno = false;
ambitoActual = 1;
:}
LPAREN ListaArgumentos RPAREN LLLAVE Bloque
{:
eliminarAmbito(1);
ambitoActual = 0;
if(hayRetorno == false) {
parser.error(Textos.noHayRetorno);
}
:}
RLLAVE;

DeclProc ::= VOID ID:id
{:
if(existeSimbolo(id)) {
parser.error(Textos.simboloRedeclarado);
} else {
```

```
addSimbolo(id);

setCategoriaSimbolo(id,"procedimiento");

direccionLocal = 0;

ambitoActual = 1;
}
:}
LPAREN ListaArgumentos RPAREN LLLAVE Bloque
{:
eliminarAmbito(1);
ambitoActual = 0;
:}
RLLAVE;
```

Ya estamos preparados para asignar dirección y tipo a la expresión correspondiente a un entero.

```
Expresion ::= ENTERO:e
{:
RESULT = entero(e);
:}
;
Expresion entero(String e) {
direccionLocal++;
return new Expresion(direccionLocal,getTipo("int").getCod());
}
```

Continuamos con el siguiente tipo de expresión:

*Expresion ::= **ID**;*

Este caso es algo diferente. Debemos comprobar que el identificador existe en la tabla de símbolos. Debemos también asignar el tipo del identificador al tipo de la expresión. Otra cosa es asignar la dirección. Lo mismo que en el caso anterior, dependiendo del ámbito, utilizamos un contador de direcciones u otro. Después, habrá que incrementar el contador de direcciones utilizado en tantas unidades como tenga la dimensión del tipo del identificador.

```
Expresion ::= ID:id
{:
RESULT = identificador(id);
:}
;
```

Los métodos empleados son:

```
Expresion identificador(String id) {
Expresion e = null;
```

```
if(existeSimbolo(id)) {
Simbolo s = getSimbolo(id);
Tipo t = getTipo(s.getTipo());
direccionLocal = direccionLocal+(t.getDimension());
e = new Expresion(direccionLocal,t.getCod());
} else {
parser.error(Textos.simboloNoDeclarado);
}
return e;
}
Simbolo getSimbolo(String id) {
return tabla.getSimbolo(id);
}
Tipo getTipo(int t) {
return tabla.getTipo(t);
}
```

La siguiente regla es muy simple:

```
Expresion ::= LPAREN Expresion:e
{:
RESULT = e;
:}
RPAREN;
```

Veamos la suma de expresiones:

*Expresion ::= Expresion **SUMA** Expresion;*

Ahora debemos comprobar que los tipos son compatibles (en nuestro caso, no permitimos la conversión de tipos, por lo que deben ser exactamente los mismos). Luego, debemos asignar una dirección nueva, como en los casos anteriores.

```
Expresion ::= Expresion:e1 SUMA Expresion:e2
{:
RESULT = suma(e1,e2);
:}
;
```

El método empleado es:

```
Expresion suma(Expresion e1,Expresion e2) {

  direccionLocal++;

  Expresion e = null;

  if(e1.getTipo()==e2.getTipo()) {

  e = new Expresion(direccionLocal,e1.getTipo());

  } else {
```

```
parser.error(Textos.tiposIncompatibles);

}

return e;

}
```

Para el resto de operadores, se hace lo mismo pero cambiando los nombres de los métodos (*resta*, *producto*, *division*, *modulo*). No los pondremos aquí por no saturar con código muy parecido.

Abordaremos en la siguiente sección el tratamiento de los registros y los elementos de los vectores.

Nos queda solamente una expresión:

Expresion ::= SentFuncion;

En este caso, el no terminal *SentFunion* debe devolver un objeto de la clase *Expresion*. Por lo tanto, sólo tenemos que modificar la regla de esta manera:

```
non terminal Expresion SentFuncion;

Expresion ::= SentFuncion:e

{:

RESULT = e;

:}

;
```

Ahora sólo nos queda comprobar que cuando se llama a una función, el nombre de esta ya está en la tabla de símbolos. Además, debemos asignar una nueva dirección para recoger el valor devuelto de la función. Y por último, comprobar que los parámetros son los adecuados tanto en número como en tipo.

Para poder procesar la lista de parámetros, crearemos una lista en memoria de parámetros. Cada vez que se declare un parámetro de la llamada, se añadirá a la lista el tipo y la dirección (ya que cada parámetro está representado por un objeto de la clase *Expresion*). En realidad, el no terminal *ListaParametros* debe ser una lista de objetos de la clase *Expresion*.

```
non terminal Expresion Parámetro;

non terminal Vector ListaParametros;
```

```
Parametro ::= Expresion:e
{:
RESULT = e;
:}
;
ListaParametros ::= ListaParametros:lp COMA Parametro:e
{:
lp.addElement(e);
RESULT = lp;
:}
| Parametro:e
{:
Vector v = new Vector();
v.addElement(e);
RESULT = v;
:}
|
{:
RESULT = new Vector();
:}
;
```

Con estas reglas, hemos preparado el terreno para poder hacer las comprobaciones oportunas para las llamadas a funciones.

```
SentFuncion ::= ID:id LPAREN ListaParametros:lp RPAREN
```

```
{:

RESULT = funcion(id,lp);

:}

;
```

Esta última regla toma el identificador de la función y las listas de parámetros (vector de expresiones) y utiliza un método para comprobar la semántica de la operación y para asignar las direcciones oportunas. Devuelve un objeto del tipo *Expresion* con el tipo que devuelve la función y con la dirección donde se va a guardar el valor devuelto.

```
SentFuncion ::= ID:id LPAREN ListaParametros:lp RPAREN

{:

RESULT = funcion(id,lp);

:}

;

Expresion funcion(String id, Vector lp) {

Expresion e = null;

if(existeSimbolo(id)) {

Simbolo s = getSimbolo(id);

Tipo t = getTipo(s.getTipo());

if(s.getNumeroParametros()!=lp.size()) {

parser.error(Textos.numeroParametrosDiferente);

} else {

if(s.comprobarTiposParametros(lp)) {

direccionLocal = direccionLocal+(t.getDimension());

e = new Expresion(direccionLocal,t.getCod());

} else { parser.error(Textos.tiposParametrosIncorrectos);
```

```
  }

  }

  } else {

  parser.error(Textos.noExisteFuncion);

  }

  return e;

  }
```

Hemos tenido que añadir un nuevo método a la clase *Simbolo*:

```
boolean comprobarTiposParametros(Vector v) {

  boolean retorno = true;

  Expresion e;

  int t;

  for(int i=0;i<v.size()-1;i++) {

  t = getParametro(i);

  e = (Expresion)v.elementAt(i);

  if(e.getTipo()!=t) {

  retorno = false;

  break;

  }

  }

  return retorno;

  }
```

19.4.8 Condiciones

En esta sección vamos a analizar la semántica de las condiciones. En este

momento debemos tomar una serie de decisiones en cuanto al diseño. En la especificación del lenguaje no se nos dijo nada respecto a si se comparan dos expresiones de tipos estructurados. Tenemos dos posibilidades, o admitir sólo la comparación de expresiones de tipos enteros, o permitir la comparación de los tipos estructurados. Si optamos por esta segunda opción, deberíamos comprobar elemento a elemento. Como esto complicaría en exceso la generación de código, optaremos por permitir la comparación de elementos de tipo entero.

El no terminal *Condicion* también es del tipo *Expresion* ya que el resultado de la evaluación de una condición se guarda como un 0 si es falsa o como un 1 si es verdadera. Por lo tanto, el tipo al que pertenece la expresión que devuelve una condición es del tipo entero.

non terminal Expresion Condicion;

Pondremos sólo un ejemplo de operador y el resto sería igual. Por ejemplo:

Condicion ::= Expresion ***OR*** *Expresion;*

Como ya hemos comentado, se debe devolver un objeto de la clase *Expresion*, previo análisis semántico. El resultado sería:

```
Condicion ::= Expresion:c1 OR Expresion:c2

{:

RESULT = or(c1,c2);

:}

;

Expresion or(Expresion c1,Expresion c2) {

Expresion e = null;

int t = getTipo("int").getCod();

if(c1.getTipo()==t && c2.getTipo()==t) {

direccionLocal++;

e = new Expresion(direccionLocal,t);

} else {

parser.error(Textos.tiposInvalidos);
```

```
}

return e;

}
```

Hay dos casos que vamos a tratar aparte. Uno es muy sencillo:

```
Condicion ::= LPAREN Condicion:c RPAREN

{:

RESULT = c;

:}

;
```

El otro caso es el del operador ***NOT,*** que tiene un solo operando. En este caso, la única comprobación semántica es que la condición sea del tipo entero.

```
Condicion ::= NOT Expresion:c

{:

RESULT = not(c);

:}

;

Expresion not(Expresion c) {

Expresion e = null;

int t = getTipo("int").getCod();

if(c.getTipo()==t) {

direccionLocal++;

e = new Expresion(direccionLocal,t);

} else {

parser.error(Textos.tipoInvalido);
```

```
}

return e;

}
```

19.4.9 Sentencia de asignación

Esta sentencia consta de dos partes, la parte izquierda y la derecha, que es una expresión. Debemos comprobar que ambas partes son del mismo tipo. En el caso de la parte izquierda, puede ser un identificador de una variable, un elemento de un vector o un campo de un registro. En el caso de que la parte izquierda sea un identificador, debemos comprobar que sea o bien una variable, o bien un parámetro, pero nunca debe ser una función o procedimiento.

También en este caso debemos tomar algunas decisiones en cuanto al diseño, si permitimos asignar variables de tipos estructurados conjuntamente. En este caso vamos a optar por lo más sencillo de implementar. Sólo permitiremos asignaciones del tipo entero. Un ejemplo:

```
int[2] vector;

vector v,w;

main() {

v[0] = 1;

v[1] = 2;

/* No esta permitido */

w = v;

/* Si esta permitido */

w[0] = v[0];

w[1] = v[1];

}
```

La parte izquierda de una asignación va a ser también del tipo *Expresion*. Va a guardar la dirección sobre la que se va a realizar la asignación. Pero en este caso no se van a consumir nuevas direcciones temporales para guardar los resultados ya que se va a señalar la dirección real donde hay que hacer la asignación de memoria.

non terminal ParteIzq Expresion;

Para el caso de que la parte izquierda sea una variable simplemente:

```
ParteIzq ::= ID:id

{:

Expresion e = null;

if(existeSimbolo(id)) {

Simbolo s = getSimbolo(id);

if(s.getCategoria().equals("variable") || s.getCategoria().equals("parametro")){

e = new Expresion(s.getDireccion(),s.getTipo());

} else {

parser.error(Textos.identificadorInvalido);

}

} else {

parser.error(Textos.simboloNoDeclarado);

}

RESULT = e;

:}

;
```

En el caso de asignación a un elemento de un vector, habrá que comprobar que la expresión que señala el índice es de tipo entero y que el tipo del identificador es en efecto un vector. Esto último es más difícil ya que para ello debemos comprobar que no es de tipo entero ni de tipo registro. Para hacer esta comprobación miraremos el atributo *maximo* del tipo al que pertenece el vector y si es mayor que -1 es que se trata de un vector.

```
ParteIzq ::= ID:id LCOR Expresion:e RCOR

{:
```

```
Expresion expresion = null;
if(existeSimbolo(id)) {
if(simboloEsVector(id)) {
if(e.getTipo()!=getTipo("int").getCod()) {
Simbolo s = getSimbolo(id);
expresion = new Expresion(s.getDireccion(),e.getTipo());
} else {
parser.error(Textos.tipoInvalido);
}
} else {
parser.error(Textos.noEsVector);
}
} else {
parser.error(Textos.simboloNoDeclarado);
}
RESULT = expresion;
:}
;
boolean simboloEsVector(String id) {
Simbolo s = getSimbolo(id);
Tipo t = getTipo(s.getTipo());
if(t.getMaximo()<0) {
return false;
} else {
```

```
return true;

}

}
```

Para comprobar que un identificador pertenece a una variable de tipo registro, debemos comprobar que no es de tipo entero y que no es de tipo vector. Otro aspecto que hay que comprobar para los registros es que el nombre del campo existe.

```
ParteIzq ::= ID:id1 PUNTO ID:id2

{:

Expresion e = null;

if(existeSimbolo(id1)) {

Simbolo s = getSimbolo(id1);

int t = s.getTipo();

if(simboloEsVector(id1) || t==getTipo("int").getCod()) {

parser.error(Textos.tipoInvalido);

} else {

int pos = posicionCampoRegistro(t,id2);

if(pos>-1){

e = new Expresion(s.getDireccion()+ pos,getTipo("int").getCod());

} else {

parser.error(Textos.noExisteCampo);

}

}

} else {

parser.error(Textos.simboloNoDeclarado);

}
```

```
:}

;
int posicionCampoRegistro(int t, String id) {
return tabla.posicionCampoRegistro(t,id);
}
```

Y en la clase *Tabla*:

```
int posicionCampoRegistro(int t, String id) {
int pos = -1;
for(int i = t+1;i<countTipos();i++) {
if(getTipo(i).getPadre()>-1) {
if(getTipo(i).getId().equals(id)) {
pos = i-t-1;
break;
}
} else {
break;
}
}
return pos;
}
```

Ya tenemos implementadas las comprobaciones de la parte izquierda, ahora debemos comprobar que la asignación es correcta. Sólo comprobaremos que los tipos a ambos lados de la asignación son tipo entero, que es el único en que se puede asignar valores.

```
SentAsignacion ::= ParteIzq:p ASIGNAR Expresion:e PTOCOMA
{:
if(p.getTipo()!=getTipo("int").getCod() ||
e.getTipo()!=getTipo("int").getCod()) {
parser.error(Textos.tiposInvalidos);
}
:}
;
```

Nos habíamos dejado el tratamiento de los registros y los vectores de las expresiones sin analizar. Lo hemos hecho así porque el código es muy parecido a la parte izquierda que acabamos de analizar. La única diferencia es que en las expresiones debemos asignar una nueva dirección para guardar el valor y en la parte izquierda utilizamos la dirección ya establecida previamente.

```
non terminal Expresión CampoRegistro, ElementoVector;

Expresion ::= CampoRegistro:e

{:
```

```
RESULT = e;

:}

| ElementoVector:e

{:

RESULT = e;

:}

;

CampoRegistro ::= ID:id1 PUNTO ID:id2

{:

Expresion e = null;

if(existeSimbolo(id1)) {

Simbolo s = getSimbolo(id1);

int t = s.getTipo();

if(simboloEsVector(id1) || t==getTipo("int").getCod()) {

parser.error(Textos.tipoInvalido);

} else {

int pos = posicionCampoRegistro(t,id2);

if(pos>-1){

direccionLocal++;

e = new Expresion(direccionLocal,getTipo("int").getCod());

} else {

parser.error(Textos.noExisteCampo);

}
```

```
}

} else {

parser.error(Textos.simboloNoDeclarado);

}

:}

;

ElementoVector ::= ID:id LCOR Expresion:e RCOR

{:

Expresion expresion = null;

if(existeSimbolo(id)) {

if(simboloEsVector(id)) {

if(e.getTipo()!=getTipo("int").getCod()) {

direccionLocal++;

expresion = new Expresion(direccionLocal,e.getTipo());

} else {

parser.error(Textos.tipoInvalido);

}

} else {

parser.error(Textos.noEsVector);

}

} else {

parser.error(Textos.simboloNoDeclarado);

}
```

```
RESULT = expresion;

:}

;
```

19.4.10 Sentencia de retorno de una función

En esta sentencia hay que comprobar que el tipo de la expresión de retorno es el mismo que el tipo que devuelve la función.

```
SentReturn ::= RETURN Expresion:e PTOCOMA

{:

hayRetorno = true;

int t = getTipoFuncion();

if(t!=e.getTipo()) {

parser.error(Textos.tipoDevueltoDiferente);

}

:}

;

int getTipoFuncion() {

return tabla.getTipoFuncion();

}
```

Y en la clase *Tabla*:

```
int getTipoFuncion() {

int retorno = -1;

Simbolo s;

for(int i=countSimbolos()-1;i>=0;i--) {

s = getSimbolo(i);

if(s.getAmbito()==0) {
```

```
retorno = s.getTipo();

break;

}

}

return retorno;

}
```

19.4.11 Sentencia de llamada a un procedimiento

Las comprobaciones en este caso son que el nombre del procedimiento esté en la tabla de símbolos y que los parámetros de la llamada coincidan en número y tipo con los del procedimiento llamado. Estas comprobaciones ya se han hecho para las llamadas a funciones (salvo la comprobación del tipo devuelto, que en este caso no hay). Otro aspecto en que se diferencia de las funciones es que en este caso no se reserva espacio en memoria para el tipo devuelto.

```
SentProcedimiento ::= ID:id LPAREN ListaParametros:lp RPAREN PTOCOMA

{:

if(existeSimbolo(id)) {

Simbolo s = getSimbolo(id);

Tipo t = getTipo(s.getTipo());

if(s.getNumeroParametros()!=lp.size()) {

parser.error(Textos.numeroParametrosDiferente);

} else { if(s.comprobarTiposParametros(lp)==false) {

parser.error(Textos.tiposParametrosIncorrectos);

}

}

} else {

parser.error(Textos.noExisteProcedimiento);
```

```
}

:}

;
```

19.4.12 Resto de sentencias

Para el resto de sentencias (*SentIf, SentWhile, SentBreak, SentPuts, SentPutw y SentElse),* ya tenemos el código completo para **C-0**. Lo utilizaremos ya que no ha habido ningún cambio. No lo vamos a volver a poner aquí.

Queda un aspecto importante que trataremos en el capítulo siguiente. Se trata del acceso a variables globales dentro del ámbito de variables locales. Como el direccionamiento es relativo, hay un solapamiento entre las direcciones de las variables en ambos ámbitos. Por lo tanto, si dentro del ámbito de un subprograma queremos acceder a una variable que hay en el ámbito global, debemos buscar su dirección relativa a la posición en memoria del RA global. En el RA local hay una dirección que guarda la dirección del RA de quien llamó al subprograma local. Por lo tanto, la dirección de la variable global es relativa a esa dirección (*encadenamiento de accesos*). Pero tenemos un problema, como se permite la recursión de las llamadas, no sabemos cuántos saltos hay hasta llegar al RA global.

Debido a esto, debemos utilizar otra técnica para acceder a las variables globales desde el ámbito local. Utilizaremos un indicador en la aplicación que nos diga a partir de qué dirección están las variables globales. Este indicador se guardará en un registro en el código final. Por lo tanto, tendremos dos registros índice, uno para el RA actual y otro para el RA global. Pero todo esto lo implementaremos en el capítulo siguiente, cuando estudiemos la generación de código.

CAPÍTULO 20

GENERACIÓN DE CÓDIGO DE C-1

20.1 Introducción

En **C-0** ya hicimos gran parte de la generación de código. Ahora nos queda sólo una parte, aunque importante y compleja. En este capítulo sólo explicaremos los cambios respecto a **C-0** y las nuevas funcionalidades.

El primer cambio es referente al espacio de direcciones. Vamos a utilizar para las direcciones globales un direccionamiento absoluto y para las locales uno relativo.

Podríamos utilizar direcciones relativas para ambos casos pero complicaría mucho la generación de código para el acceso a variables en ámbitos diferentes. Por lo tanto, vamos a reservar una zona de memoria que comience en la dirección 0 para el código del programa y, a continuación, van los datos de las cadenas.

Como Ens2001 sólo admite un espacio de direcciones de 64k direcciones, vamos a suponer que nuestro programa va a ocupar a lo sumo 10000 direcciones (de la 0 a la 9999). Esto no es lo más recomendable, pero como este no es un compilador comercial, lo haremos así por simplicidad.

Reservaremos desde la dirección 10000 hasta la 10999 para las variables globales y de la 11000 en adelante para los RA. Vamos a considerar que incluso el programa principal está en un RA, por lo que sus temporales serán respecto al índice que marque la posición en memoria de los RA.

Por lo tanto, inicialmente el contador de direcciones globales debe valer 10000. Y además cargaremos el registro índice para el acceso a los RA con el valor 11000.

Volvemos a repetir que en un compilador comercial se debe organizar mejor la memoria para que no se pueda dar el caso de que algunas zonas se solapen y, por tanto, se reescriban datos. En nuestro caso no lo hacemos así por simplicidad (pero si el programa es demasiado largo o tiene demasiadas variables globales o locales, puede darse el caso del solape y, por tanto, de un funcionamiento incorrecto).

Por lo tanto, cargaremos el contador global con su valor:

```
action code {:

…….

nt direccionRA;

……
```

```
void inicializar() {

……..

cuentaDirecciones = 10000;

direccionRA = 11000;

…….

}

…….

:}
```

La primera instrucción de código que debemos insertar es la que cargue un registro índice con el valor a partir del cual se sitúen los RA. La siguiente es un salto al programa principal (para que no vaya a ejecutarse inicialmente el código de ningún subprograma).

```
Programa ::= {:

inicializar();

saltarMain(direccionRA);

:}

Declaraciones Cuerpo | Cuerpo;

void saltarMain(int d) {

codigoIntermedio.guardarCuadrupla(new
Cuadrupla("CARGAR_IX",String.valueOf(d+3),null,null));

codigoIntermedio.guardarCuadrupla(new
Cuadrupla("SALTAR_ETIQUETA",null,null,"MAIN"));

}
```

Hay que notar que IX apunta en realidad a la primera temporal disponible después de las reservadas al valor devuelto, a la dirección de retorno y a la dirección del RA llamante.

El siguiente programa:

```
int x;
```

```
main() {

x = 0;

}
```

Debe dar este código intermedio (CI):

CARGAR_IX 11003 null null

SALTAR_ETIQUETA null null MAIN

Las declaraciones no generan ningún código, por lo que pasamos directamente a las expresiones.

20.2 CI de expresiones

En **C-0** ya vimos el CI de las expresiones que teníamos. Ahora seguirá todo igual. Pero hay una diferencia en el código final. Se trata de que ahora las temporales están referidas a un índice y no lo están respecto a direcciones absolutas.

20.2.1 Suma, resta, producto, multiplicación, división y módulo

Ya utilizamos un método para la comprobación semántica de la suma. Lo mismo hacemos para las comprobaciones semánticas de las demás operaciones con expresiones. Vamos a incrustar dentro de ese método las instrucciones necesarias para la generación del CI correspondiente.

```
Expresion suma(Expresión e1,Expresión e2) {

direccionLocal++;

Expresión e = null;

if(e1.getTipo()==e2.getTipo()) {

e = new Expresión(direccionLocal,e1.getTipo());

codigoIntermedio.guardarCuadrupla(new                    Cuadrupla("SUMAR",
String.valueOf(e1.getDireccion()), String.valueOf(e2.getDireccion()),

String.valueOf(direccionLocal)));

} else {

parser.error(Textos.tiposIncompatibles);

}
```

```
return e;

}
```

Para el resto de operaciones con expresiones, es exactamente igual, salvo el nombre del método y el identificador de la cuádrupla del CI. Para las condiciones es lo mismo. Pero en ese caso en el código final se pondrá un 1 para verdadero y un 0 para falso en la dirección temporal donde se guarde el resultado. Para ver mejor el funcionamiento, mirar en el apéndice B.

20.2.2 CI para enteros

El CI que se genera cuando se encuentra un entero es:

```
Expresion entero(String e) {
direccionLocal++;
codigoIntermedio.guardarCuadrupla(new Cuadrupla("CARGAR_VALOR",e,null,
String.valueOf(direccionLocal)));
return new Expresion(d,getTipo("int").getCod());
}
```

20.2.3 CI para identificadores

Cuando aparece el nombre de una variable como expresión, se generaría este código:

```
Expresion identificador(String id) {
Expresion e = null;
if(existeSimbolo(id)) {
Simbolo s = getSimbolo(id);
Tipo t = getTipo(s.getTipo());
direccionLocal = direccionLocal+(t.getDimension());
e = new Expresion(direccionLocal,t.getCod());
codigoIntermedio.guardarCuadrupla(new Cuadrupla("CARGAR_DIRECCION",
String.valueOf((getSimbolo(id)).getDireccion()),
null,String.valueOf(direccionLocal)));
} else {
parser.error(Textos.simboloNoDeclarado);
}
return e;
}
```

20.2.4 CI para funciones

Entramos en la parte más complicada de la generación de código intermedio.

Cada vez que se hace una llamada a una función hay que realizar ciertas tareas. Lo

primero es calcular dónde se va a ubicar el RA de la función que vamos a llamar. Para ello, calculamos cuál es la última dirección que estamos utilizando y colocamos el RA después. Después, se reserva memoria en el nuevo RA para el valor devuelto, la dirección de retorno y la dirección del RA actual. Luego, se pone en el nuevo RA la dirección donde debe volver el control cuando se vuelva de la llamada. Esa dirección será relativa al punto desde donde estemos llamando. Para ello, ponemos una etiqueta a continuación de la instrucción de llamada y esa etiqueta señalará la dirección del retorno.

Después ponemos la dirección del RA actual en el RA invocado.

Más tarde, procesamos los parámetros y los valores de los mismos los iremos poniendo a continuación de las direcciones que hemos reservado en el RA invocado.

Por último, hacemos un salto al código de la función que vamos a llamar mediante su etiqueta. Para ello, es necesario que cuando aparezca el código de la declaración de una función se señale con una etiqueta con el nombre de la función.

Vamos a ver un ejemplo para entender mejor el proceso:

```
int x;
int inc(int a) {
return a+1;
{
main() {
x = inc(0);
}
```

Vamos a ver el código final necesario y luego veremos el CI que lo generaría:

```
MOVE #11003 , .IX      Carga el registro índice con el valor de la dirección del RA
BR /MAIN               Salta al código del programa principal
INC: MOVE #1 , #0[.IX] Pone un 1 en la siguiente dirección temporal disponible en el RA actual.
MOVE #0[.IX] #1[.IX]   , Pone la dirección del parámetro en la siguiente temporal disponible
ADD #1[.IX] #1[.IX]    , Suma los valores de las dos temporales
MOVE .A #2[.IX]        , Pone el resultado en la siguiente temporal disponible
MOVE #2[.IX] #-3[.IX]  , Pone el valor devuelto en su lugar en el RA actual
MOVE #-2[.IX] , .A     Toma la dirección de retorno y
MOVE .A .PC            , La pone para que sea la siguiente instrucción a procesar (es como un salto)
MAIN: MOVE #0 #4[.IX]  El RA del invocado se va a situar a continuación de la última temporal utilizada. Como la única que hemos utilizado ha
                       , sido una global, el RA de la función que se va a invocar estará a partir de la 0. Se reservan la 0, 1 y 2 fijas y la 3 para el
                       parámetro. La siguiente disponible estará en la 4
MOVE #4[.IX] #3[.IX]   , Se pone el parámetro en su lugar del nuevo RA
MOVE .IX #2[.IX]       , Ponemos la dirección del RA actual en el RA invocado
```

```
MOVE .IX , .A  Preparamos para calcular el índice del nuevo RA
ADD #0 , .A
MOVE .A , .IX  Hacemos que IX apunte al nuevo RA
MOVE
#INC_1         , Situamos en el nuevo RA la dirección de retorno
#-2[.IX]
BR /INC          Saltamos a la función
INC_1:
MOVE
#-1[.IX] , .IX   Restauramos IX para que apunte al RA actual después de volver de la llamada. El valor del RA actual lo toma de donde lo guardó en el RA invocado
MOVE
#0[.IX]        , Ponemos el valor devuelto por la función en la dirección de la variable global, que era la 10000
/10000
HALT             Fin del programa
```

Una cosa a tener en cuenta, cada llamada a una misma función debe tener una etiqueta de retorno única. Por lo tanto, debemos tener un contador de llamadas para cada función o un contador para todas las llamadas. Es más cómodo esto último. Añadiremos el valor del contador a la etiqueta de retorno de la llamada y lo incrementaremos. El mismo contador nos sirve para las llamadas a procedimientos.

```
action code {:

……..

int numeroLlamada = 0;

……..

:}

SentFuncion ::= ID:id LPAREN ListaParametros:lp RPAREN

{:

numeroLlamada++;

RESULT = funcion(id,lp);

:}

;

SentProcedimiento ::= ID:id LPAREN ListaParametros:lp RPAREN PTOCOMA

{:

if(existeSimbolo(id)) {
```

```
Simbolo s = getSimbolo(id);

if(s.getNumeroParametros()!=lp.size()) {

parser.error(Textos.numeroParametrosDiferente);

} else {

if(s.comprobarTiposParametros(lp)==false) {

parser.error(Textos.tiposParametrosIncorrectos);

}

numeroLlamada++;

}

} else {

parser.error(Textos.noExisteProcedimiento);

}

:}

;
```

En el caso especial de que un parámetro de una llamada sea otra llamada, hay un conflicto en las direcciones locales. Para resolver este problema, crearemos una pila en la que guardaremos el contador de dirección local actual y el número de parámetro que estamos procesando en la llamada (el primer parámetro será el de índice 1, el segundo el 2, etc.).

Cada vez que encontremos un identificador de una llamada a función o procedimiento apilaremos estos dos valores y los restauraremos cuando finalicemos con el código generado en la llamada.

Para ello, crearemos una clase que podemos llamar *Temporal* y una pila para apilar objetos de esta clase:

```
class Temporal {

int direccionBase;

int indiceParametro;
```

```
Temporal(int db, int ip) {

direccionBase = db;

indiceParametro = ip;

}

int getBase() {

return direccionBase;

}

int getIndice() {

return indiceParametro;

}

}

import java.util.Vector;

class Temporales {

Vector pila;

Temporales() {

pila = new Vector();

}

void apilar(int db, int ip) {

pila.addElement(new Temporal(db,ip));

}

void desapilar() {

pila.removeElement(pila.size()-1);

}

int getBase() {
```

```
return ((Temporal)pila.elementAt(pila.size()-1)).getBase();

}

int getIndice() {

return ((Temporal)pila.elementAt(pila.size()-1)).getIndice();

}

}
```

Debemos tener una variable de este tipo y, además, necesitamos una variable del tipo *Temporal* para guardar las direcciones del RA siguiente y del parámetro que estamos procesando. Utilizaremos una variable que se llame por ejemplo, *temporales*.

```
action code {:

…….

Temporales pilaTemporales = new Temporales();

Temporal temporales;

……….

:}
```

Cuando llamemos a una función o procedimiento, apilamos los valores actuales de las temporales y creamos los nuevos para el RA del invocado. Al final de la invocación, desapilamos y ya tenemos los valores que había antes de hacer la llamada.

Veremos todo el proceso para funciones y luego para procedimientos.

```
SentFuncion ::= ID:id

{:

numeroLlamada++;

apilarTemporales(id);

:}

LPAREN ListaParametros:lp RPAREN

{:
```

```
procesaLlamada(id);

desapilarTemporales();

RESULT = funcion(id,lp);

:}

;

void apilarTemporales(String id) {

pilaTemporales.apilar(direccionLocal,-1);

temporales = new Temporal(direccionLocal+3,1);

direccionLocal = direccionLocal + getDimensionParametros(id);

}

void desapilarTemporales() {

direccionLocal = pilaTemporales.getBase();

pilaTemporales.desapilar();

}

int getDimensionParametros(String id) {

return tabla.getDimensionParametros(id);

}

void procesaLlamada(String id) {

Simbolo simbolo = getSimbolo(id);

codigoIntermedio.guardarCuadrupla(new Cuadrupla("FINLLAMADA1",

String.valueOf(temporales.getBase()),

String.valueOf(temporales.getBase()+1),null));

codigoIntermedio.guardarCuadrupla(new Cuadrupla("FINLLAMADA2",

id,String.valueOf(numeroLlamada),null));
```

```
}
```

En la clase *Tabla*:

```
int getDimensionParametros(String id) {

Simbolo s = getSimbolo(id);

int suma = 0;

for(int i=0;i<s.getNumeroParametros()-1;i++) {

suma = suma + getTipo(s.getParametro(i)).getDimension();

}

return suma;

}
```

El CI que debemos generar para procesar la llamada necesita dos cuádruplas porque hay que pasarle más información de la que se puede pasar con una sola.

Nos queda procesar cada uno de los parámetros.

Debemos procesar cada parámetro al ser localizado. Por lo tanto, la regla para leer cada parámetro quedaría así:

```
Parametro ::= Expresion:e

{:

RESULT = procesaParametro(e);

:}

;
```

El método *procesaParametro* debe generar el CI para cada parámetro y devolver un objeto de la clase *Expresion*.

```
Expresion procesaParametro(Expresion e) {

Tipo tipo = getTipo(e.getTipo());

if(tipo.getCod()!=getTipo("int").getCod()) {

parser.error(Textos.tipoInvalido);
```

```
        } else {

        codigoIntermedio.guardarCuadrupla(new
Cuadrupla("PARAMETRO",String.valueOf(e.getDireccion()),

        null,String.valueOf(temporales.getBase()+temporales.getIndice())));

        temporales.setIndice(temporales.getIndice()+1);

        }

        return e;

        }
```

Una nueva restricción es que sólo se permite pasar como parámetro un elemento del tipo entero.

Debemos incluir un nuevo método en la clase *Temporal*:

```
void setIndice(int i) {

  indiceParametro = i;

  }
```

Nos queda sólo el procesamiento del retorno. Adaptamos la regla:

```
  SentReturn ::= RETURN Expresion:e PTOCOMA

  {:

  hayRetorno = true;

  int t = getTipoFuncion();

  if(t!=e.getTipo()) {

  parser.error(Textos.tipoDevueltoDiferente);

  } else {

  procesaRetorno(e);

  }

  :}
```

```
;

void procesaRetorno(Expresion e) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla("RETORNO",

String.valueOf(e.getDireccion()),null,null));

}
```

20.2.5 CI para procedimientos

El CI para procedimientos es muy parecido al que se genera para funciones. La regla quedaría así:

```
SentProcedimiento ::= ID:id

{:

numeroLlamada++;

apilarTemporales(id);

:}

LPAREN ListaParametros:lp RPAREN PTOCOMA

{:

if(existeSimbolo(id)) {

Simbolo s = getSimbolo(id);

if(s.getNumeroParametros()!=lp.size()) {

parser.error(Textos.numeroParametrosDiferente);

} else {

if(s.comprobarTiposParametros(lp)==false) {

parser.error(Textos.tiposParametrosIncorrectos);

}

procesaLlamada(id);
```

```
}

} else {

parser.error(Textos.noExisteProcedimiento);

}

desapilarTemporales();

:}

;
```

20.2.6 CI para campos de registros

Cuando una expresión es el campo de un registro, hay que generar un CI que copie el contenido que hay en la dirección asignada en la tabla de símbolos al campo en una temporal. Modificamos la regla:

```
CampoRegistro ::= ID:id1 PUNTO ID:id2

{:

Expresion e = null;

if(existeSimbolo(id1)) {

Simbolo s = getSimbolo(id1);

int t = s.getTipo();

if(simboloEsVector(id1) || t==getTipo("int").getCod()) {

parser.error(Textos.tipoInvalido);

} else {

int pos = posicionCampoRegistro(t,id2);

if(pos>-1){

direccionLocal++;

e = new Expresion(direccionLocal,getTipo("int").getCod());

ciCampoRegistro(s.getDireccion()+pos,direccionLocal);
```

```
} else {

parser.error(Textos.noExisteCampo);

}

}

} else {

parser.error(Textos.simboloNoDeclarado);

}

RESULT = e;

:}

;

void ciCampoRegistro(int d1, int d2) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla("CARGAR_DIRECCION",

String.valueOf(d1),null,String.valueOf(d2)));

}
```

20.2.7 CI para elementos de un vector

Este caso es más complicado que el anterior. En tiempo de compilación no podemos saber la dirección exacta de un elemento concreto de un vector ya que el índice que indica el elemento nos viene dado como resultado de la evaluación de una expresión. Por tanto, se deja para el programador la tarea de verificar que se accede a un elemento correcto. Nosotros sólo generaremos el código necesario para acceder a un elemento, dependiendo del índice que se pase, pero sin verificar si ese índice cae dentro o fuera del rango.

```
ElementoVector ::= ID:id LCOR Expresion:e RCOR

{:

Expresion expresion = null;

if(existeSimbolo(id)) {

if(simboloEsVector(id)) {

if(e.getTipo()==getTipo("int").getCod()) {
```

```
        expresion = new Expresion(-1,e.getTipo());

        ciExpresionVector(e.getDireccion(),getSimboloid().getDireccion());

        } else {

        parser.error(Textos.tipoInvalido);

        }

        } else {

        parser.error(Textos.noEsVector);

        }

        } else {

        parser.error(Textos.simboloNoDeclarado);

        }

        RESULT = expresion;

        :}

        ;

        void ciExpresionVector(int d1, int d2) {

        codigoIntermedio.guardarCuadrupla(new
Cuadrupla("VECTOR",String.valueOf(d1),String.valueOf(d2),null));

        }
```

¿Por qué hemos devuelto una expresión con dirección -1? Es una manera de indicar que vamos a utilizar una dirección para un vector. Como no podemos saber la dirección en tiempo de compilación vamos a utilizar un registro adicional para poner el valor que obtengamos en tiempo de ejecución para esa dirección. Luego no tenemos más que acceder al contenido que señale ese registro.

20.3 CI para asignaciones

Para las asignaciones tenemos tres posibilidades, una es que la parte izquierda sea un identificador de una variable sencilla, un identificador que señale a un elemento de un vector o un identificador que señale a un campo de un registro.

```
SentAsignacion ::= ParteIzq:p ASIGNAR Expresion:e PTOCOMA

{:

if(p.getTipo()!=getTipo("int").getCod() ||

e.getTipo()!=getTipo("int").getCod()) {

parser.error(Textos.tiposInvalidos);

} else {

asignacion(p,e);

}

:}

;

void asignacion(Expresion e1,Expresion e2) {

codigoIntermedio.guardarCuadrupla(new
Cuadrupla("CARGAR_DIRECCION",String.valueOf(e2.getDireccion()),null,String.valueOf(e1.getI

}
```

20.3.1 Asignación a una variable sencilla

En este caso, lo único que habría que hacer es copiar el contenido de la dirección temporal donde está el resultado de la parte derecha en la dirección que señala la tabla de símbolos de la variable de la parte izquierda.

20.3.2 Asignación a un campo de un registro

Este caso es parecido al anterior. En tiempo de compilación se conoce la dirección de un campo de un registro. Por lo tanto, se copia el contenido de una dirección en otra.

20.3.3 Asignación a un elemento de un vector

Este caso es diferente a los dos anteriores ya que no se conoce en tiempo de compilación nada más que dónde está situado el primer elemento del vector pero no se sabe cuál es el índice. Por lo tanto, la dirección que se pasa indicando dónde está el elemento de la parte izquierda de la asignación no señala al elemento real sino al primer elemento del vector. Esto causaría un error, por lo que debemos realizar otro tipo de asignación para partes izquierdas con elementos de vectores. Vamos a utilizar una nueva temporal para señalar la dirección donde se va a indicar el elemento señalado por el índice.

```
ParteIzq ::= ID:id LCOR Expresion:e RCOR
{:
Expresion expresion = null;
if(existeSimbolo(id)) {
if(simboloEsVector(id)) {
if(e.getTipo()==getTipo("int").getCod()) {
Simbolo s = getSimbolo(id);
ciExpresionVector(e.getDireccion(),s.getDireccion());
expresion = new Expresion(-1,e.getTipo());
} else {
parser.error(Textos.tipoInvalido);
}
} else {
parser.error(Textos.noEsVector);
}
} else {
parser.error(Textos.simboloNoDeclarado); }
RESULT = expresion;
:}
;
```

20.4 Sentencias condicionales y bucles

El código para estas sentencias es el mismo que hicimos para **C-0**, por lo que no lo vamos a repetir aquí. Sólo nos queda en esta sección implementar el CI para la sentencia de ruptura de un bucle, es decir, para la sentencia *break*. Esta sentencia debe provocar un salto a la etiqueta que señala el final de un bucle *while*.

```
SentBreak ::= BREAK
{:
if(cuentaWhiles>0) {
romper(pilaBucle.verCima());
} else {
parser.error(Textos.breakSinWhile);
}
:}
PTOCOMA;

void romper(int n) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla("SALTAR_ETIQUETA",null,null,

"FINBUCLE_"+String.valueOf(n)));

}
```

El resto de instrucciones y la generación de CI son las mismas que para C-0.

20.5 Sentencias para imprimir

Estas sentencias son también iguales que para C-0, por lo que el código es el mismo. No lo vamos a repetir otra vez aquí.

20.6 Declaración de funciones y procedimientos

Lo primero que debemos poner es el CI para situar una etiqueta con el nombre del subprograma al comienzo del código que se va a crear para el procesamiento del subprograma. Lo último es poner el código necesario para volver de la llamada.

```
DeclFunc ::= CabeceraDecl

{:

ciPonerEtiqueta(getIdFuncion());

setCategoriaUltimoSimbolo("funcion");

direccionLocal = 0;

cuentaDirecciones = cuentaDirecciones - tipoActual.getDimension();

hayRetorno = false;

ambitoActual = 1;

:}

LPAREN ListaArgumentos RPAREN LLLAVE Bloque

{:

eliminarAmbito(1);

ambitoActual = 0;

procesaRetorno();

if(hayRetorno == false) {

parser.error(Textos.noHayRetorno);

}

:}

RLLAVE;
```

```
DeclProc ::= VOID ID:id

{:

if(existeSimbolo(id)) {

parser.error(Textos.simboloRedeclarado);

} else {

ciPonerEtiqueta(id);

addSimbolo(id);

setCategoriaSimbolo(id,”procedimiento”);

direccionLocal = 0;

ambitoActual = 1;

}

:}

LPAREN ListaArgumentos RPAREN LLLAVE Bloque

{:

eliminarAmbito(1);

ambitoActual = 0;

procesaRetorno();

:}

RLLAVE;

void ciPonerEtiqueta(String id) {

codigoIntermedio.guardarCuadrupla(new Cuadrupla(“ETIQUETA”,null,null,id));

}

String getIdFuncion() {
```

```
return tabla.getIdFuncion();

}

void procesaRetorno() {

codigoIntermedio.guardarCuadrupla(new Cuadrupla("RETORNO","-1",null,null));

}
```

Y en la clase *Tabla*:

```
String getIdFuncion() {

String retorno = "";

Simbolo s;

for(int i=countSimbolos()-1;i>=0;i--) {

s = getSimbolo(i);

if(s.getAmbito()==0) {

retorno = s.getId();

break;

}

}

return retorno;

}
```

Para el método principal, también hay que poner la etiqueta. Le llamamos "MAIN":

```
Cuerpo ::= MAIN LPAREN RPAREN LLLAVE

{:

ciPonerEtiqueta("MAIN");

direccionLocal = -1;
```

```
:}

BloqueSentencias

{:

finPrograma();

generarCadenas();

cerrarCI();

generarCF();

:}

RLLAVE;
```

Ponemos la dirección local a -1 porque el programa principal tiene un RA como los demás subprogramas.

20.7 Finalización

Los procedimientos para finalizar el proceso de generación del CI y la traducción del CI al código final son los mismos que se han implementado para **C-0**, por lo que no vamos a repetirlos aquí.

20.8 Generación de código final

Vamos a utilizar toda la infraestructura que ya teníamos para este menester en **C-0**. Sólo vamos a modificar algunos detalles.

Como en este lenguaje vamos a utilizar dos tipos de direcciones, las relativas y las absolutas, debemos saber cuándo nos referimos a unas o a otras. La manera de diferenciarlas va a ser comparando la dirección con la dirección donde van a estar situadas las variables globales. Si la dirección que indica la cuádrupla es mayor o igual que la dirección donde están las globales, se trata de una dirección global. En cambio, si es una dirección menor, se trata de una dirección local, es decir, relativa. Para ello, vamos a utilizar un atributo en la clase *CodigoFinal* que es la dirección a partir de la cual están las variables globales. Esta diferenciación sólo la utilizaremos para las cuádruplas que traten con direcciones de variables que puedan ser globales. En el constructor indicaremos esta dirección.

```
private int direccion = -1;

// Constructor
```

```
public CodigoFinal(CodigoIntermedio CI,String nombrePrograma, int d) {

codigoIntermedio = CI;

direccion = d;

String nombre = nombrePrograma.substring(0,nombrePrograma.lastIndexOf(“.”));

ficheroCF = nombre.concat(“.ens”);

}
```

Utilizaremos igualmente un atributo en nuestro programa principal para guardar ese valor ya que como el código final se genera al final de todo el proceso de análisis, el valor de la dirección que señala a las variables globales ya se habrá incrementado. Llamaremos por ejemplo, *direccionesGlobales* a ese valor. Antes de utilizar el atributo *cuentaDirecciones*, cargaremos su valor en el atributo *direccionesGlobales*.

```
action code {:

……

int direccionesGlobales;

……

void inicializar() {

…….

cuentaDirecciones = 10000;

direccionesGlobales = cuentaDirecciones;

……

}

…….

:}
```

El método que lanza la traducción del CI al código final es:

```
void generarCF() {
```

```
CodigoFinal codigoFinal = new CodigoFinal(codigoIntermedio,

parser.nombreFichero,direccionesGlobales);

try {

codigoFinal.traducirCodigo();

} Catch(Exception e) {}

}
```

En el método de la clase *CodigoFinal* que se encarga de traducir el código se utilizará el valor de la dirección indicada para generar un código u otro. Por ejemplo, en la generación del código final para la suma de expresiones:

```
// Procesa la cuadrupla

private void procesarCuadrupla(Cuadrupla cuadrupla)throws IOException{

String op1,op2,inst,res;

int o1,o2,r;

String linea = " ";

op1 = cuadrupla.op1;

op2 = cuadrupla.op2;

inst = cuadrupla.nombre;

res = cuadrupla.res;

if(inst.equals("CARGAR_DIRECCION")) {

o1 = Integer.parseInt(op1);

if(o1<0)op1="[.R1]";else if(o1<direccion)op1="#"+op1+"[.IX]";else op1="/"+op1;

r = Integer.parseInt(res);

if(r<0)res="[.R1]";else if(r<direccion)res="#"+res+"[.IX]";

else res="/"+res;
```

```
escribirLinea(linea+"MOVE "+op1+" , "+res);

} else if(inst.equals("SUMAR")) {

o1 = Integer.parseInt(op1);

if(o1<0)op1="[.R1]";else op1="#"+op1+"[.IX]";

o2 = Integer.parseInt(op2);

if(o2<0)op2="[.R1]";else op2="#"+op2+"[.IX]";

r = Integer.parseInt(res);

if(r<0)res="[.R1]";else res="#"+res+"[.IX]";

escribirLinea(linea+"ADD "+op1+" , "+op2);

escribirLinea(linea+"MOVE .A , "+res);

} else

……

}
```

Este mismo sistema se debe hacer en todas las cuádruplas en las que pueda haber direcciones relativas o absolutas. Hay que destacar que una dirección que valga -1 indica que el valor está en el registro R1 (lo utilizamos para los vectores).

En el apéndice B se detallarán los CI y los códigos finales correspondientes para **C-1**.

20.9 Ampliación para C-2

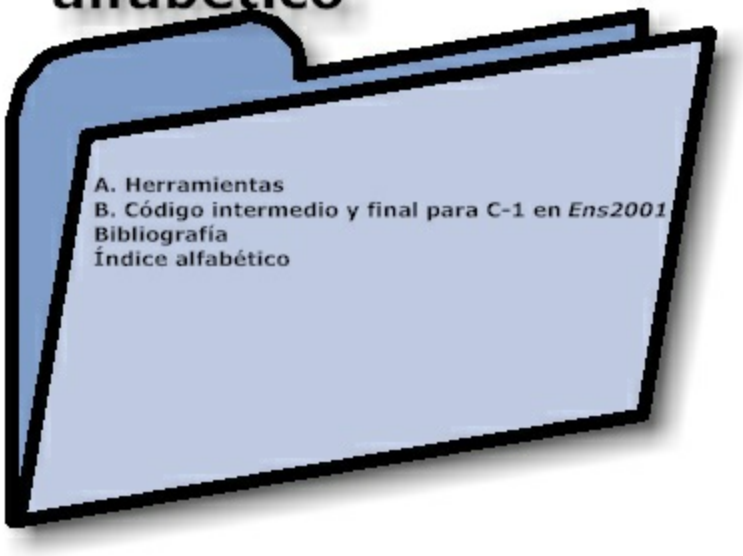
Hemos realizado algunas restricciones iniciales en la especificación de **C-1** y algunas más a lo largo de su desarrollo. Para un posible compilador para C, que podríamos llamar **C-2,** se deberían implementar estas restricciones y algunas funcionalidades más. Pero esto ya se deja a cargo del lector para un trabajo personal.

Las posibles nuevas funcionalidades de **C-2** podrían ser:

- Permitir la declaración e inicialización de variables al mismo tiempo.
- Introducir nuevos tipos primitivos como, por ejemplo, los números reales y las cadenas de caracteres.

- Nuevas sentencias de control, como, por ejemplo, los bloques *swich* y *for.*
- Permitir la definición de tipos locales.
- Permitir matrices de más de una dimensión.
- Permitir la asignación conjunta de variables de tipos estructurados.
- Permitir el paso de parámetros de tipos estructurados en funciones y procedimientos.
- Etc.

Parte V. Apéndices, bibliografía e índice alfabético



APÉNDICE A
HERRAMIENTAS

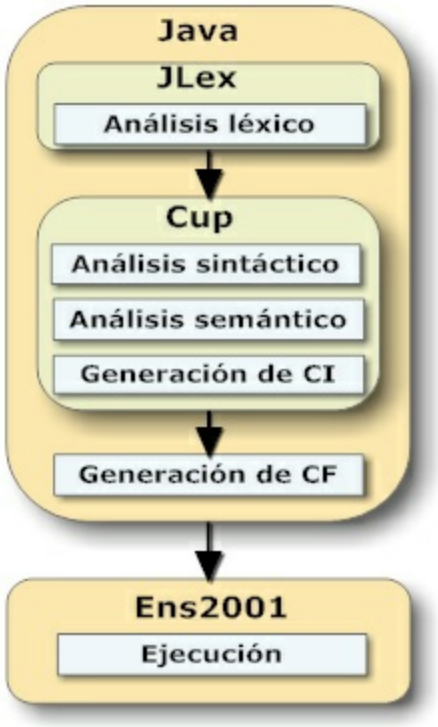
A.1 Herramientas

Hoy en día, la creación de un compilador no es como era hace unas décadas. Antes se hacía todo a partir de cero, por lo que la labor era ingente. Actualmente, hay a disposición de los desarrolladores multitud de herramientas de ayuda que les liberan de las tareas más tediosas, y a la vez más sistematizadas.

Por lo tanto, vamos a utilizar estas herramientas como apoyo para la creación de nuestro sencillo compilador.

El proceso de desarrollo de un compilador sigue una serie de pasos que son:

1. Análisis léxico (AL)
2. Análisis sintáctico (ASI)
3. Análisis semántico (ASE)
4. Generación de código intermedio (CI)
5. Generación de código final (CF)



Todo compilador tiene un lenguaje de base que es con el que se va a programar todo el proceso de creación del compilador. Antes se utilizaba directamente ensamblador, pero hoy en día se utilizan lenguajes más amigables, como pueden ser Pascal, C o Java. Nosotros utilizaremos Java, principalmente porque es multiplataforma y porque las herramientas más potentes disponibles utilizan y están realizadas en Java.

En la fase de AL se procesa el programa de entrada para ver si las palabras o los lexemas son los adecuados.

Para esta primera fase, vamos a utilizar un analizador léxico llamado JLex.

Para la segunda fase, ASI, vamos a utilizar otra herramienta llamada Cup. En esta fase se analiza que la ordenación de los diferentes lexemas del programa responda a la sintaxis del lenguaje del que queremos hacer el compilador.

Para la fase de análisis semántico, también vamos a utilizar Cup. En esta fase se va a comprobar que el programa que queremos procesar tiene significado (por ejemplo, que detecte las divisiones por 0, etc.).

Para la generación de CI, que es una especie de sucesión de códigos que luego serán traducidos a lenguaje ensamblador, vamos a utilizar Cup y Java.

Para la generación de CF, vamos a utilizar Java y el código final será un lenguaje ensamblador estándar. ¿Por qué no utilizamos el de cualquier procesador, por ejemplo alguno de la familia x86? Porque disponemos de otra herramienta que es un emulador de ensamblador y que corre bajo Windows y nos permite ver el código, el contenido de las direcciones de memoria, el contenido de los registros, etc.

El emulador de ensamblador que vamos a utilizar se llama ENS2001.

De todas formas, se podría generar CF para cualquier tipo de ensamblador.

Las técnicas que vamos a aprender son muy utilizadas en muchos entornos, por ejemplo, la creación de sencillos lenguajes para programar robots, diseño de lenguajes para programar autómatas, traductores de unos lenguajes a otros (en este caso, convertimos un programa en un lenguaje por otro programa equivalente en otro lenguaje), analizadores de textos en diferentes formatos (HTML, XML, etc.) y un largo etcétera.

Resumiendo, vamos a utilizar:

- Java (http://java.sun.com)
- JLex (http://www.cs.princeton.edu/~appel/modern/java/JLex/)
- Cup (http://www.cs.princeton.edu/~appel/modern/java/CUP/)
- ENS2001 (http://usuarios.lycos.es/ens2001/)

A.2 Instalación de las herramientas

A.2.1 Java

Debemos instalar el JDK. Para ello, podemos seguir las indicaciones que hay en la

página de Sun.

Aunque básicamente consiste en añadir el camino al directorio donde estén los ejecutables de Java (en el directorio */bin* del SDK) en la variable de entorno PATH, y crear una nueva variable de entorno que se llame CLASSPATH y que contenga el camino al directorio */lib* del SDK.

A.2.2 JLex

Creamos un directorio nuevo, por ejemplo *jdir* y lo incluimos en la variable de entorno CLASSPATH. Creamos un directorio dentro de *jdir* y que debemos llamar *jdir/JLex*. Dentro metemos la clase Java *Main.java* que hemos descargado de la página de JLex. Después, la debemos compilar con *javac* (*javac Main.java*) y se habrán creado las clases necesarias de JLex.

A partir de este momento, ya podemos analizar cualquier archivo de análisis léxico mediante el comando *java JLex.Main fichero.jlex*.

A.2.3 CUP

El proceso de instalación de Cup es parecido. Descargamos de la página de Cup un archivo comprimido. Lo descomprimimos de manera que se nos quede dentro de *jdir* el siguiente directorio *jdir/java_cup* y dentro los archivos de clases ya descomprimidos.

Luego, debemos compilar las clases descargadas mediante el comando *javac java_cup/*.java java_cup/runtime/*.java*. Habremos obtenido las clases necesarias. Utilizaremos la herramienta así : *java java_cup.Main fichero.cup*.

A.2.4 ENS2001

Esta herramienta es un emulador de un lenguaje estándar ensamblador. Hay una versión para Windows que es la que vamos a utilizar. No requiere instalación, sólo bajarse el programa en un archivo comprimido y descomprimirlo en cualquier directorio.

A.3 Uso de las herramientas

A.3.1 Uso de JLex

La pieza básica de todo el proceso es Cup, y JLex le irá suministrando a Cup los tokens (o lexemas) según los requiera.

Lo primero es crear el fichero para el análisis léxico con JLex. Este fichero especificará los lexemas aceptables por nuestro compilador. Por ejemplo, si vamos a utilizar la sentencia *while* para los bucles pero no la *for*, el analizador léxico dará un aviso de error si se encuentra una sentencia *for*.

El formato de especificación es el siguiente:

```
Código de Usuario
```

```
%%

Directivas JLex

%%

Expresiones Regulares
```

Los caracteres %% se usan para separar cada una de las secciones.

En la primera sección pondremos, por ejemplo, las importaciones de clases que podamos necesitar. Por ejemplo:

```
import java_cup.runtime.Symbol;

import java.io.*;
```

En la sección de directivas podemos poner algunas como:

%cup

En la que le informamos a JLex que vamos a utilizar Cup.

En esta sección también podemos incluir macros que resuman ciertas expresiones regulares útiles. Por ejemplo:

NUMERO = [1-9][0-9]*

Que indica que NUMERO es cualquier número entero compuesto por un número del 1 al 9 seguido por 0 o más números del 0 al 9.

Para más información sobre expresiones en JLex, hay que acudir a su página web.

En la sección de expresiones regulares definiremos las expresiones regulares que indican los lexemas de nuestro lenguaje, y qué debemos hacer una vez detectados dichos lexemas. Por ejemplo:

```
“[a-b][a-b0-9]*” { return new Symbol(sym.ID); }

{NUMERO} { return new Symbol(sym.NUMERO,new Integer(yytext()); }
```

Que nos reconocería identificadores que comiencen con la letra a o b y después tengan 0 o más letras a, b, o números del 0 al 9. Y además, reconocería números.

Por ejemplo, reconocería estos lexemas:

```
a12

b

b0223
```

```
1

123

102

20

aa23
```

Pero daría error para:

```
c12

01

ad

12a
```

Vemos que a la derecha de la expresión regular, se ponen las acciones a tomar cuando se detecta dicha expresión regular. En el primer caso se devuelve un objeto de la clase *Symbol* que será tomado por Cup.

Las directivas %{ y %} permiten encapsular código Java que se va a incluir en la clase Java producida por JLex. La forma correcta de utilizar estas directivas es:

```
%{

<código_Java>

%}
```

Es decir, las líneas que contengan las directivas %{ y %} no deberán contener ningún símbolo adicional, y el código Java deberá encontrarse entre estas líneas. No se deben utilizar identificadores que comiencen por yy, para evitar colisiones con los identificadores del código Java producido por JLex.

%line: si se incluye esta directiva, la variable *yyline*, de tipo *int*, indica la línea en la que empieza el token que está siendo reconocido. La primera línea del programa que se le pasa al analizador léxico es la línea 0. Las directivas %eofval{ y %eofval} permiten especificar el token que devuelve el analizador léxico cuando se llega al final del fichero. Estas directivas encapsulan el código Java que se va ejecutar cuando se invoque el método que proporciona el analizador léxico para producir el siguiente token (es decir, si se utiliza la directiva %cup, cuando se invoca el método next_token) y se ha llegado al final del fichero de entrada al analizador léxico. Por lo tanto, el código Java encapsulado por estas directivas debe terminar con una sentencia Java *return* que haga que se devuelva el

objeto Java correspondiente al token utilizado cuando se llega al final del fichero (si se utiliza la directiva %cup, este objeto Java habrá de ser de la clase *Symbol*). La forma correcta de utilizar estas directivas es:

```
%eofval{

<código_Java>

%eofval}
```

Es decir, las líneas que contengan las directivas %eofval{ y %eofval} no deberán contener ningún símbolo adicional, y el código Java deberá encontrarse entre estas líneas. Nótese que esta directiva es obligatoria si se utiliza JLex para producir un analizador léxico que va a ser utilizado por un analizador sintáctico generado por CUP, ya que CUP requiere que se le notifique por medio de un token cuando se llega al final del fichero de entrada, y la acción por defecto de JLex es devolver null. Una vez que se ha llegado al carácter fin de línea del fichero de entrada, el analizador léxico continuará ejecutando el código indicado por estas directivas si se pide sucesivas veces un nuevo token.

La última construcción que vamos a ver de esta sección es la posibilidad de definir macros. Una macro permite dar un nombre a una cierta expresión regular para poder utilizar ese nombre en lugar de la expresión regular en la definición de las expresiones regulares asociadas a cada token. Una macro tiene la forma:

<nombre>=<definicion>

<nombre> es el nombre que le vamos a dar a la expresión regular. Ha de estar formado por letras, números o el símbolo "_", empezando por una letra o el símbolo "_". <definicion> es la expresión regular que se define utilizando las reglas que se explicarán más adelante. La tercera parte del fichero de entrada para JLex consiste en una serie de reglas que permiten obtener los tokens. Cada regla consta del siguiente formato:

<expression> { <action> }

<expression> especifica la expresión regular que define las cadenas de símbolos de entrada que pueden activar una regla. Cada vez que se solicita un nuevo token al analizador léxico generado por JLex, este consume un cierto número de caracteres del fichero de texto que contiene el programa que se le pasa al analizador léxico. Dichos caracteres deben formar una cadena que pertenezca al lenguaje definido por la expresión regular de alguna de las reglas que permiten obtener los tokens.

El analizador generado por JLex funciona de forma que se intenta que cada token producido por el analizador léxico consuma la cadena de caracteres lo más larga que sea posible. Puede ocurrir que dicha cadena de longitud máxima pertenezca a varios de los lenguajes definidos por las expresiones regulares de reglas distintas. En ese caso, se

activa la regla que está primero en la especificación JLex. Por esta razón se suele colocar la regla para reconocer identificadores después de las reglas para reconocer las palabras clave del lenguaje de programación, para dar prioridad a estas.

Las expresiones regulares no deben contener espacios en blanco, ya que un espacio en blanco se interpreta como el fin de la expresión regular. Si se desea utilizar el espacio en blanco en una expresión regular, se deberá escribir entre comillas:" ".

Los siguientes caracteres son metacaracteres, ya que tienen significados especiales en la definición de una expresión regular:

? * + | () ^ $. [ ] { } " \

Los demás caracteres se interpretan en una expresión regular como ellos mismos como símbolos del alfabeto del lenguaje que define la expresión regular.

Dadas dos expresiones regulares *a* y *b* codificadas en el formato de JLex, la expresión regular *ab* representa la concatenación de *a* y *b*.

Dadas dos expresiones regulares *a* y *b* codificadas en el formato de JLex, la expresión regular *a*|*b* representa la unión de *a* y *b*.

Las siguientes secuencias de escape, entre otras, son reconocidas por JLex:

- \n: salto de línea.
- \t: tabulador.
- \r: retorno de carro.

El punto (.) es una expresión regular que define el lenguaje de todas las cadenas que no contienen el carácter salto de línea.

Los metacaracteres rodeados de comillas (") pierden su significado especial y se interpretan como ellos mismos como símbolos del alfabeto del lenguaje que define la expresión regular. La única excepción es \" que representa al carácter". Por ejemplo, la expresión regular "*\"*" representa en JLex a la cadena de caracteres *"*.

Para utilizar una macro para representar una expresión regular, se escribe el nombre de la macro rodeado de los caracteres { y }.

* representa el cierre reflexivo-transitivo. + representa el cierre transitivo.

(...) Los paréntesis se utilizan para agrupar expresiones regulares.

[...] Los corchetes se utilizan para representar un conjunto de caracteres. Existen varias formas de definir conjuntos de caracteres dentro de unos corchetes. Aquí sólo explicamos una de ellas. *a*-*b* indica que el rango de caracteres del *a* al *b*, ambos inclusive, están incluidos en el conjunto de caracteres. Pueden especificarse varios rangos de

caracteres dentro de una expresión [...]. Por ejemplo, la expresión regular [a-zA-Z] es el conjunto de las letras mayúsculas y minúsculas.

Un fichero JLex debe aceptar cualquier cadena de símbolos de entrada. Esto se consigue en el ejemplo poniendo al final estas dos líneas:

```
(" "|\n|\t|\r)+ { }

. { System.out.println("Caracter Ilegal en linea" + yyline);}
```

Dentro del código Java de una acción se pueden utilizar la variable *yyline* de tipo *int* (explicado más arriba) y el método *yytext()*, que devuelve un objeto, de tipo *String*, que contiene la cadena de caracteres que se ha consumido para activar la regla. Esto nos permite, por ejemplo, recuperar de *yytext* la cadena de caracteres que forma el nombre de un token identificador.

A.3.2 Uso de Cup

Cup es la herramienta principal de todo el proceso. Se encarga de ir demandando a JLex los lexemas válidos a analizar y genera el código necesario para el análisis sintáctico y semántico. Además, genera el CI.

Cup analiza las reglas de la gramática del lenguaje a compilar y genera una serie de acciones a partir del análisis de dichas reglas.

Cup distingue entre terminales (devueltos por JLex) y no terminales, que son las reglas de la gramática.

Dentro de cada regla, Cup puede ejecutar código Java para realizar su trabajo de análisis.

Un fichero de entrada para CUP consta de las siguientes partes:

1. Definición de paquete y sentencias import.
2. Sección de código de usuario.
3. Declaración de símbolos terminales y no terminales.
4. Declaraciones de precedencia.
5. Definición del símbolo inicial de la gramática y las reglas de producción.

En la primera sección se pone el nombre del paquete (si lo tiene) y las sentencias de importación. Al menos, debe contener la siguiente importación:

import java_cup.runtime.;*

El código de usuario es código Java que se va a incluir en el analizador. Se puede introducir en algunas partes. Una de ellas es:

```
parser code {:

...

:};
```

En la sección de símbolos terminales y no terminales se pueden incluir estos y, además, pueden tener como tipo uno de los válidos por Java.

Para declarar símbolos terminales y no terminales, se utiliza la siguiente sintaxis:

```
terminal [<nombre_clase>] nombre1, nombre2, ... ;

non terminal [<nombre_clase>] nombreA, nombreB, ... ;
```

En CUP, es posible definir niveles de precedencia y la asociatividad de símbolos terminales. Las declaraciones de precedencia de un fichero CUP consisten en una secuencia de construcciones que comienzan por la palabra clave *precedence*. A continuación, viene la declaración de asociatividad, que puede tomar los valores *left*, *right* y *nonassoc*. Finalmente, la construcción termina con una lista de símbolos terminales separados por comas, seguido del símbolo punto y coma.

Para definir el símbolo inicial de la gramática, se utiliza la construcción *start with...*:

```
start with Prog;
```

Para definir todas las reglas de producción que tengan a un mismo símbolo no terminal como antecedente, se escribe el símbolo no terminal en cuestión, seguido de ::= y, a continuación, las reglas de producción que le tengan como antecedente, separadas por el símbolo | . Después de la última regla de producción se termina con punto y coma.

Recordemos que cuando se presentó la declaración de símbolos terminales y no terminales, se dijo que estos podían tener asociado un objeto Java. Los identificadores que vienen después de un símbolo terminal o no terminal representan variables Java en las que se guarda el objeto asociado a ese símbolo terminal o no terminal. Estas variables Java pueden ser utilizadas en la parte {: ... :} que viene a continuación.

Entre {: ... :} se incluye el código Java que se ejecutará cuando se reduzca la regla de producción en la que se encuentre dicha cláusula {: ... :}.

Si el no terminal antecedente tiene asociada una cierta clase Java, obligatoriamente dentro de la cláusula {: ... :} habrá una sentencia RESULT=...;. El objeto Java guardado en la variable RESULT será el objeto Java que se asocie al no terminal antecedente cuando se reduzca la regla de producción en la que se encuentre esa cláusula {: ... :}.

Cuando se produce un error en el análisis sintáctico, CUP invoca a los siguientes

métodos:

```
public void syntax_error(Symbol s);

public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception;
```

Una vez que se produce el error, se invoca el método *syntax_error*. Después, se intenta recuperar el error (el mecanismo que se utiliza para recuperar errores no se explica en este documento; si no se hace nada especial, el mecanismo para recuperar errores falla al primer error que se produzca). Si el intento de recuperar el error falla, entonces se invoca el método *unrecovered_syntax_error*. El objeto de la clase Symbol representa el último token consumido por el analizador. Estos métodos se pueden redefinir dentro de la declaración *parser code* {: ... :}.

Para más información sobre Cup, hay que remitirse a su página web.

Un ejemplo básico es una calculadora:

```
/* calculadora.jlex*/

import java_cup.runtime.Sybol;

%%

%eofval{

{ System.exit(0); }

%eofval}

%cup

NUMBER = [1-9][0-9]*

%%

{NUMBER} { return new Symbol(sym.NUMERO, new Integer(yytext())); }

“+” { return new Symbol(sym.MAS); }

“-” { return new Symbol(sym.MENOS); }

“*” { return new Symbol(sym.POR); }

“/” { return new Symbol(sym.DIV); }

“(“ { return new Symbol(sym.PARI); }
```

```
"(" { return new Symbol(sym.PARI); }

")" { return new Symbol(sym.PARD); }

";" { return new Symbol(sym.FIN); }

.|\n { }

/*calculadora.cup*/

/* Importamos las clases necesarias del paquete cup.jar */

import java_cup.runtime.*;

/**

* Aquí ponemos el código que se usará para comenzar a parsear la *entrada.

*/

parser code {:

public static void main(String args[]) throws Exception {

// La clase Ylex es creada por el analizador léxico

// Jlex (ver sección siguiente).

new parser(new Ylex(System.in)).parse();

}

:}

/* Aquí especificamos los terminales del lenguaje. */

terminal MAS, MENOS, POR, DIV, PARI, PARD, FIN;

/**

* Este terminal tiene un valor entero. Recuerda que le dábamos el *valor en el código del analizador léxico, al darle como parámetro un *valor entero al objeto Symbol.

*/

terminal Integer NUMERO;

/* Lista de no terminales. */
```

```
non terminal expr_list, expr_part;

/**
* Aquí están los no terminales con valor entero, que son con los que
* podemos hacer cálculos, y podemos escribirlos de la forma expr_e:l
* (por ejemplo, no se podría hacer expr_list:l, ya que a ese no
* terminal no le damos valor.
*/
non terminal Integer expr_e;
non terminal Integer expr_t;
non terminal Integer expr_f;
non terminal Integer expr_g;
non terminal Integer expr_h;
/* Aquí especificamos la precedencia de los operadores. */
precedence left MAS;
precedence left MENOS;
precedence left POR;
precedence left DIV;
/**
* Ahora comenzamos con las reglas de producción.
*/
/**
* Estas dos reglas son nuevas. Nos sirven para encadenar varias
* expresiones separadas por un ';'
*/
```

```
        expr_list ::= expr_list expr_part

        | expr_part;

        expr_part ::= expr_e:e {: System.out.println("= "+e); :} FIN;

        /* E <- E + T | T */

        expr_e ::= expr_e:e1 MAS expr_t:r {: RESULT=new Integer(e1.intValue() +
r.intValue()); :}

        | expr_t:e {: RESULT=e; :};

        /* T <- T - F | F */

        expr_t ::= expr_t:e1 MENOS expr_f:r {: RESULT=new Integer(e1.intValue() -
r.intValue()); :}

        | expr_f:e {: RESULT=e; :};

        /* F <- F * G | G */

        expr_f ::= expr_f:e1 POR expr_g:r {: RESULT=new Integer(e1.intValue() *
r.intValue()); :}

        | expr_g:e {: RESULT=e; :};

        /* G <- G / H | H */

        expr_g ::= expr_g:e1 DIV expr_h:r {: RESULT=new Integer(e1.intValue() /
r.intValue()); :}

        | expr_h:e {: RESULT=e; :};

        /* H <- (E) | n */

        expr_h ::= PARI expr_e:e PARD {: RESULT=e; :}

        | NUMERO:n {: RESULT=n; :};
```

Para ejecutar este compilador de prueba (en concreto, no se trata de un compilador completo, ya que sólo se realizan las fases de análisis pero no las de generación de CI y CF), debemos seguir estos pasos:

- *java JLex.Main calculadora.jlex*
- Obtenemos un archivo Java que se llama *calculadora.jlex.java* que debemos renombrar a *Yylex.java*

- *java java_cup.Main calculadora.cup*
- Compilamos todos los archivos obtenidos *javac parser.java sym.java Yylex.java*
- Creamos un archivo de texto con el programa a compilar, por ejemplo, le llamamos *programa.txt*
- Lo compilamos con *java parser programa.txt*
- Saldrá por consola el programa ejecutado.

Esto que hemos hecho no es en realidad un compilador sino un intérprete de un sencillo lenguaje.

Por ejemplo, si el contenido del archivo programa.txt fuera este:

4+3*(2-7)+20;

El resultado obtenido sería:

= 9

APÉNDICE B
CÓDIGO INTERMEDIO Y FINAL PARA C-1 EN ENS2001

B.1 Introducción

En este apéndice vamos a mostrar una tabla con el código intermedio y final utilizado para **C-1**. El que se ha utilizado para **C-0** está explicado en la parte de implementación del compilador **C-0**.

Antes de nada, hay que hacer unas consideraciones: se utilizan dos operadores de tipo cadena y un resultado de tipo cadena. Hay casos en que hay que convertirlos en valores enteros para comprobar si se trata de direccionamiento relativo o directo. Otro aspecto a tener en cuenta es que para los vectores, la dirección está ubicada en un registro, en concreto el R1. Para saber si se va a utilizar un registro o no, si la dirección es -1 hay que utilizar el valor del registro R1. En caso contrario será una dirección o bien relativa (si es menor que un cierto valor) o bien absoluta (si es mayor o igual que ese cierto valor).

B.2 Tabla de código intermedio y final para Ens2001

CARGAR_DIRECCION,op1,null,res	MOVE op1 , res
CARGAR_VALOR,op1,null,res	MOVE #op1 , res
SUMAR,op1,op2,res	ADD op1 , op2 MOVE .A , res
RESTAR,op1,op2,res	SUB op1 , op2 MOVE .A , res
MULTIPLICAR,op1,op2,res	MUL op1 , op2 MOVE .A , res
DIVIDIR,op1,op2,res	DIV op1 , op2 MOVE .A , res
MODULO,op1,op2,res	MOD op1 , op2 MOVE .A , res
OR,op1,op2,res	OR op1 , op2 MOVE .A , res
AND,op1,op2,res	AND op1 , op2 MOVE .A , res
NOT,op1,null,res	CMP #1 , op1 BZ $5 MOVE #1 , res BR $3 MOVE #0 , res
MAYOR,op1,op2,res	CMP op2 , op1 BN $5 MOVE #0 , res BR $3 MOVE #1 , res
MENOR,op1,op2,res	CMP op1 , op2 BN $5 MOVE #0 , res

	BR $3 MOVE #1 , res
MAYORIGUAL,op1,op2,res	CMP op1 , op2 BN $5 MOVE #1 , res BR $3 MOVE #0 , res
MENORIGUAL,op1,op2,res	CMP op2 , op1 BN $5 MOVE #1 , res BR $3 MOVE #0 , res
IGUAL,op1,op2,res	CMP op1 , op2 BZ $5 MOVE #0 , res BR $3 MOVE #1 , res
DISTINTO,op1,op2,res	CMP op1 , op2 BZ $5 MOVE #1 , res BR $3 MOVE #0 , res
ETIQUETA,null,null,res	res: NOP
SALTAR_CONDICION,op1,null,res	CMP #0 , op1 BZ , /res
SALTAR_ETIQUETA,null,null,res	BR /res
IMPRIMIR_ENTERO,op1,null,null	WRINT op1
IMPRIMIR_CADENA,op1,null,null	WRSTR op1
PONER_CADENA,op1,null,res	op1: DATA res
FIN,null,null,null	HALT
CARGAR_IX,op1,null,null	MOVE #op1 , .IX
VECTOR,op1,op2,null	MOVE op1 , .A ADD .A , #op2 MOVE .A , .R1
FINLLAMADA1,op1,op2,null	MOVE .IX , #op1[.IX] MOVE .IX , .A ADD #op2 , .A MOVE .A , .IX
FINLLAMADA2,op1,op2,null	MOVE #LLAMADAop2 , #-2[.IX] BR /op1 LLAMADAop2: NOP MOVE #-1[.IX] , .IX
PARAMETRO,op1,null,res	MOVE op1 , res
RETORNO,op1,null,null	MOVE op1 , #-3[.IX] MOVE #-2[.IX] , .A MOVE .A , .PC

B.3 Ejemplo de programa en C-1

En esta sección presentamos un ejemplo de programa para **C-1** con su código intermedio y código final en Ens2001.

```
struct estructura{

int s1;

int s2;
```

```
};

int[5] vector;

int x,y;

vector v;

estructura s;

/* Funcion recursiva */

int factorial(int p) {

if(p<=1)return p;

else return factorial(p-1)*p;

}

/* Procedimento */

void sumaVector(int s) {

int i,suma;

i=0;

suma=0;

while(i<s) {

suma =suma+v[i]; /* Prueba un vector */

i=i+1;

}

puts("Debe escribirse 15 : ");

putw(suma);puts("\n");

}

main() {

x = 1;
```

```
y=0;

x = x+factorial(6);/* Prueba la funcion recursiva */

puts("Debe escribirse 721 : ");putw(x);

puts("\n");

while(y<6) { /* Prueba un bucle */

v[y]=y; /* Prueba la asignacion a un vector */

y=y+1;

}

s.s2=y; /* Prueba la asignacion a un registro */

puts("Debe escribirse 6 : ");

putw(s.s2);puts("\n");

sumaVector(s.s2); /* Prueba un registro */

}
```

El código intermedio generado es:

```
CARGAR_IX 11003 null null

SALTAR_ETIQUETA null null MAIN

ETIQUETA null null factorial

CARGAR_DIRECCION 0 null 2

CARGAR_VALOR 1 null 3

MENORIGUAL 2 3 4

SALTAR_CONDICION 4 null ELSE_1

CARGAR_DIRECCION 0 null 5

RETORNO 5 null null

SALTAR_ETIQUETA null null FINIF_1
```

```
ETIQUETA null null ELSE_1

CARGAR_DIRECCION 0 null 6

CARGAR_VALOR 1 null 7

RESTAR 6 7 8

PARAMETRO 8 null 9

FINLLAMADA1 8 9 null

FINLLAMADA2 factorial 2 null

CARGAR_DIRECCION 0 null 7

MULTIPLICAR 6 7 8

RETORNO 8 null null

ETIQUETA null null FINIF_1

RETORNO -1 null null

ETIQUETA null null sumaVector

CARGAR_VALOR 0 null 4

CARGAR_DIRECCION 4 null 1

CARGAR_VALOR 0 null 5

CARGAR_DIRECCION 5 null 2

ETIQUETA null null BUCLE_1

CARGAR_DIRECCION 1 null 6

CARGAR_DIRECCION 0 null 7

MENOR 6 7 8

SALTAR_CONDICION 8 null FINBUCLE_1

CARGAR_DIRECCION 2 null 9

CARGAR_DIRECCION 1 null 10
```

```
VECTOR 10 10002 null

SUMAR 9 -1 11

CARGAR_DIRECCION 11 null 2

CARGAR_DIRECCION 1 null 12

CARGAR_VALOR 1 null 13

SUMAR 12 13 14

CARGAR_DIRECCION 14 null 1

SALTAR_ETIQUETA null null BUCLE_1

ETIQUETA null null FINBUCLE_1

IMPRIMIR_CADENA CADENA_1 null null

CARGAR_DIRECCION 2 null 15

IMPRIMIR_ENTERO 15 null null

IMPRIMIR_CADENA CADENA_2 null null

RETORNO -1 null null

ETIQUETA null null MAIN

CARGAR_VALOR 1 null 0

CARGAR_DIRECCION 0 null 10000

CARGAR_VALOR 0 null 1

CARGAR_DIRECCION 1 null 10001

CARGAR_VALOR 6 null 2

PARAMETRO 2 null 5

FINLLAMADA1 4 5 null

FINLLAMADA2 factorial 3 null

CARGAR_DIRECCION 2 null 10000
```

```
IMPRIMIR_CADENA CADENA_3 null null

CARGAR_DIRECCION 10000 null 3

IMPRIMIR_ENTERO 3 null null

IMPRIMIR_CADENA CADENA_4 null null

ETIQUETA null null BUCLE_2

CARGAR_DIRECCION 10001 null 4

CARGAR_VALOR 6 null 5

MENOR 4 5 6

SALTAR_CONDICION 6 null FINBUCLE_2

CARGAR_DIRECCION 10001 null 7

VECTOR 7 10002 null

CARGAR_DIRECCION 10001 null 8

CARGAR_DIRECCION 8 null -1

CARGAR_DIRECCION 10001 null 9

CARGAR_VALOR 1 null 10

SUMAR 9 10 11

CARGAR_DIRECCION 11 null 10001

SALTAR_ETIQUETA null null BUCLE_2

ETIQUETA null null FINBUCLE_2

CARGAR_DIRECCION 10001 null 12

CARGAR_DIRECCION 12 null 10009

IMPRIMIR_CADENA CADENA_5 null null

CARGAR_DIRECCION 10009 null 13

IMPRIMIR_ENTERO 13 null null
```

```
IMPRIMIR_CADENA CADENA_6 null null

CARGAR_DIRECCION 10009 null 14

PARAMETRO 14 null 17

FINLLAMADA1 16 17 null

FINLLAMADA2 sumaVector 4 null

FIN null null null

PONER_CADENA CADENA_1 null “Debe escribirse 15 : “

PONER_CADENA CADENA_2 null “\n”

PONER_CADENA CADENA_3 null “Debe escribirse 720 : “

PONER_CADENA CADENA_4 null “\n”

PONER_CADENA CADENA_5 null “Debe escribirse 6 : “

PONER_CADENA CADENA_6 null “\n”
```

Y el código final en Ens2001 es:

```
MOVE #11003 , .IX

BR /MAIN

factorial: NOP

MOVE #0[.IX] , #2[.IX]

MOVE #1 , #3[.IX]

CMP #3[.IX] , #2[.IX]

BN $5

MOVE #1 , #4[.IX]

BR $3

MOVE #0 , #4[.IX]

CMP #0 , #4[.IX]
```

```
BZ /ELSE_1

MOVE #0[.IX] , #5[.IX]

MOVE #5[.IX] , #-3[.IX]

MOVE #-2[.IX] , .A

MOVE .A , .PC

BR /FINIF_1

ELSE_1: NOP

MOVE #0[.IX] , #6[.IX]

MOVE #1 , #7[.IX]

SUB #6[.IX] , #7[.IX]

MOVE .A , #8[.IX]

MOVE #8[.IX] , #9[.IX]

MOVE .IX , #8[.IX]

MOVE .IX , .A

ADD #9 , .A

MOVE .A , .IX

MOVE #LLAMADA2 , #-2[.IX]

BR /factorial

LLAMADA2: NOP

MOVE #-1[.IX] , .IX

MOVE #0[.IX] , #7[.IX]

MUL #6[.IX] , #7[.IX]

MOVE .A , #8[.IX]

MOVE #8[.IX] , #-3[.IX]
```

```
MOVE #-2[.IX] , .A

MOVE .A , .PC

FINIF_1: NOP

MOVE #-2[.IX] , .A

MOVE .A , .PC

sumaVector: NOP

MOVE #0 , #4[.IX]

MOVE #4[.IX] , #1[.IX]

MOVE #0 , #5[.IX]

MOVE #5[.IX] , #2[.IX]

BUCLE_1: NOP

MOVE #1[.IX] , #6[.IX]

MOVE #0[.IX] , #7[.IX]

CMP #6[.IX] , #7[.IX]

BN $5

MOVE #0 , #8[.IX]

BR $3

MOVE #1 , #8[.IX]

CMP #0 , #8[.IX]

BZ /FINBUCLE_1

MOVE #2[.IX] , #9[.IX]

MOVE #1[.IX] , #10[.IX]

MOVE #10[.IX] , .A

ADD .A , #10002
```

```
MOVE .A , .R1

ADD #9[.IX] , [.R1]

MOVE .A , #11[.IX]

MOVE #11[.IX] , #2[.IX]

MOVE #1[.IX] , #12[.IX]

MOVE #1 , #13[.IX]

ADD #12[.IX] , #13[.IX]

MOVE .A , #14[.IX]

MOVE #14[.IX] , #1[.IX]

BR /BUCLE_1

FINBUCLE_1: NOP

WRSTR /CADENA_1

MOVE #2[.IX] , #15[.IX]

WRINT #15[.IX]

WRSTR /CADENA_2

MOVE #-2[.IX] , .A

MOVE .A , .PC

MAIN: NOP

MOVE #1 , #0[.IX]

MOVE #0[.IX] , /10000

MOVE #0 , #1[.IX]

MOVE #1[.IX] , /10001

MOVE #6 , #2[.IX]

MOVE #2[.IX] , #5[.IX]
```

```
MOVE .IX , #4[.IX]

MOVE .IX , .A

ADD #5 , .A

MOVE .A , .IX

MOVE #LLAMADA3 , #-2[.IX]

BR /factorial

LLAMADA3: NOP

MOVE #-1[.IX] , .IX

MOVE #2[.IX] , /10000

WRSTR /CADENA_3

MOVE /10000 , #3[.IX]

WRINT #3[.IX]

WRSTR /CADENA_4

BUCLE_2: NOP

MOVE /10001 , #4[.IX]

MOVE #6 , #5[.IX]

CMP #4[.IX] , #5[.IX]

BN $5

MOVE #0 , #6[.IX]

BR $3

MOVE #1 , #6[.IX]

CMP #0 , #6[.IX]

BZ /FINBUCLE_2

MOVE /10001 , #7[.IX]
```

```
MOVE #7[.IX] , .A

ADD .A , #10002

MOVE .A , .R1

MOVE /10001 , #8[.IX]

MOVE #8[.IX] , [.R1]

MOVE /10001 , #9[.IX]

MOVE #1 , #10[.IX]

ADD #9[.IX] , #10[.IX]

MOVE .A , #11[.IX]

MOVE #11[.IX] , /10001

BR /BUCLE_2

FINBUCLE_2: NOP

MOVE /10001 , #12[.IX]

MOVE #12[.IX] , /10009

WRSTR /CADENA_5

MOVE /10009 , #13[.IX]

WRINT #13[.IX]

WRSTR /CADENA_6

MOVE /10009 , #14[.IX]

MOVE #14[.IX] , #17[.IX]

MOVE .IX , #16[.IX]

MOVE .IX , .A

ADD #17 , .A

MOVE .A , .IX
```

```
MOVE #LLAMADA4 , #-2[.IX]

BR /sumaVector

LLAMADA4: NOP

MOVE #-1[.IX] , .IX

HALT

CADENA_1: DATA “Debe escribirse 15 : “

CADENA_2: DATA “\n”

CADENA_3: DATA “Debe escribirse 720 : “

CADENA_4: DATA “\n”

CADENA_5: DATA “Debe escribirse 6 : “

CADENA_6: DATA “\n”
```

BIBLIOGRAFÍA

Libros y manuales

- *Breve introducción a Cup*. Universidad Carlos III. Dto. de Ingeniería Telemática.
- *Breve introducción a JLex*. Universidad Carlos III. Dto. de Ingeniería Telemática.
- **Aho, A. V. y otros**. *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana. 1990
- **Alfonseca, M. y otros**. *Teoría de lenguajes, gramáticas y autómatas*. Ediciones Universidad y Cultura. 1987
- Álvarez, F.J. *Ens2001- Manual de usuario*. 2002
- **Cueva Lovelle, J. M**. *Análisis léxico en procesadores de lenguaje*. Universidad de Oviedo. Dto. de Informática. 2000
- **Cueva Lovelle, J. M**. *Lenguajes, gramáticas y autómatas*. Universidad de Oviedo. Dto. de Informática. 2001
- **Eckel, B**. *Piensa en Java*. Prentice Hall. 2002
- **Fernández, G. y otros**. *Fundamentos de informática*. Alianza. 1987
- **Gálvez Rojas, S. y otros**. *Java a tope: Traductores y compiladores con Lex/Yacc, JFlex/Cup y JavaCC*. Edición electrónica. 2005
- **Garrido Alenda, A. y otros**. *Diseño de compiladores*. Publicaciones Universidad de Alicante. 2002
- **Hopcroft, J. E. y otros**. *Introducción a la teoría de autómatas, lenguajes y computación*. Pearson Educación. 2002
- **Juan Fuente, A. A. y otros.** *Tablas de símbolos*. Apuntes Universidad de Oviedo. Dto. de Informática. 2004
- **Miguel, P**. *Fundamentos de los computadores*. Ed. Paraninfo. 1996
- **Teufel, B. y otros**. *Compiladores: conceptos fundamentales*. Addison-Wesley Iberoamericana. 1995

Software

- Java (http://java.sun.com)
- JLex (http://www.cs.princeton.edu/~appel/modern/java/JLex/)
- Cup (http://www.cs.princeton.edu/~appel/modern/java/CUP/)

- ENS2001 (http://usuarios.lycos.es/ens2001/)