Capítulo 1

Introducción a los lenguajes de programación

En este capítulo se exponen las nociones básicas necesarias para entender los lenguajes de programación y se introducen conceptos que serán utilizados durante el resto del libro. Comienza explicando las notaciones sintácticas y semánticas para la descripción de lenguajes de programación y después profundiza en el diseño de lenguajes, realizando un recorrido por diferentes conceptos como tipos de datos, procedimientos y ambientes, expresiones, tipos abstractos de datos o modularización. Aunque estos conceptos se presentan de forma resumida, el texto contiene referencias a fuentes externas que el lector puede consultar para ampliar el contenido sobre algún aspecto concreto.

1.1 Introducción

Sir Charles Antony Richard Hoare, inventor del algoritmo Quicksort (quizá el algoritmo de computación más utilizado del mundo) dijo en una ocasión [14]:

> *El propósito principal de un lenguaje de programación es ayudar al programador en la práctica de su arte.*

Para comprender los principios de diseño de los modernos lenguajes de programación es necesario conocer algo de su historia. Esta historia comienza realmente en el siglo XIX, cuando Charles Babbage idea la máquina analítica. Esta máquina estaba diseñada para funcionar utilizando tarjetas perforadas. Estas tarjetas ya se conocían puesto que eran las utilizadas en el telar de Jacquard, una máquina inventada por Joseph Marie Jacquard que era capaz de crear telas a partir de patrones grabados en tarjetas de este tipo. Aunque Babbage no terminó nunca de construirla, su máquina analítica se considera la primera computadora. Una matemática contemporánea de Babbage, llamada Ada Lovelace, llegó incluso a crear programas para esta máquina analítica, aunque evidentemente nunca pudo comprobar si funcionaban. Por ello a Babbage se le considera

el padre de la computadora y a Ada la primera programadora de la historia (el lenguaje Ada, creado para el ministerio de defensa estadounidense se llamó así en su honor).

Mucho más recientemente, en los años 30 del siglo XX se construyeron las primeras máquinas capaces de realizar ciertas operaciones, si bien normalmente estaban orientadas a realizar algún tipo concreto de cálculo científico. Existe cierto acuerdo en que el ENIAC, desarrollado en el año 1946, puede considerarse el primer computador realmente de propósito general. El ENIAC no se programaba utilizando un lenguaje de programación, sino cableando directamente sus circuitos.

Por esta época, John von Neumann propuso una arquitectura diferente para las computadoras, donde el programa se almacena en la máquina antes de ejecutarse. Esto permitió a partir de entonces evitar tener que cablear todos los componentes para cada nuevo programa. Es entonces donde realmente empieza la historia de los lenguajes de programación.

Desde un punto de vista coloquial, un lenguaje de programación es una notación para comunicarle a una computadora lo que deseamos que haga. Desde un punto de vista formal, podemos definirlo como un sistema notacional para describir computaciones en una forma legible tanto para la máquina como para el ser humano.

Esta definición formal es la que ha estado guiando la evolución de los lenguajes de programación. En esta evolución podemos distinguir cinco generaciones de lenguajes de programación:

Primera generación. A esta generación pertenece el lenguaje máquina. El lenguaje máquina consiste exclusivamente en secuencias de ceros y unos y es el único lenguaje que entienden las computadoras modernas. Cada computadora dispone de lo que se denomina un conjunto de instrucciones que son las operaciones que esa computadora entiende (sumar, restar, cargar de memoria, guardar en memoria, etc). Estas operaciones se indican mediante secuencias de ceros y unos. El principal inconveniente del lenguaje máquina es que es difícil de entender por un humano. Los programas escritos en lenguaje máquina generalmente sólo funcionan en un modelo de máquina específico, dado que cada computadora define su propio código de secuencias de ceros y unos.

Segunda generación. Pertenecen a esta generación los lenguajes ensambladores. Estos lenguajes establecen una serie de reglas mnemotécnicas que hacen más sencilla la lectura y escritura de programas. Estas reglas mnemotécnicas consisten simplemente en asociar nombres legibles a cada una de las instrucciones soportadas por la máquina (ADD, SUB, LOAD, STORE, etc.). Los programas escritos en lenguaje ensamblador no son directamente ejecutables por la máquina, puesto que ésta sólo entiende instrucciones codificadas como ceros y unos, pero es muy sencillo traducirlos a lenguaje máquina. El lenguaje ensamblador aún se utiliza hoy en día para programar *drivers* para dispositivos o determinadas partes de los sistemas operativos.

Tercera generación. A esta generación pertenecen los lenguajes como C, FORTRAN o Java. Estos lenguajes se denominan lenguajes de alto nivel, porque están bastante alejados del lenguaje máquina y son mucho más legibles por el hombre. Para convertir los programas escritos en estos lenguajes de alto nivel en código máquina entendible por la computadora empiezan a hacer falta complejos programas que los traduzcan. Estos traductores se denominan compiladores. Los lenguajes de alto nivel son necesarios para programar grandes sistemas software, como sistemas operativos (Linux, Windows), aplicaciones para la web (Facebook, Tuenti, Twitter) o aplicaciones para móviles. Estos lenguajes facilitan el mantenimiento y la evolución del software. Los primeros lenguajes de tercera generación fueron FORTRAN, LISP, ALGOL y COBOL. Los cuatro surgieron a finales de los 50.

Cuarta generación. Los lenguajes de la cuarta generación son lenguajes de propósito específico, como SQL, NATURAL o ABAP. Estos lenguajes no están diseñados para programar aplicaciones complejas, sino que fueron diseñados para solucionar problemas muy concretos. Por ejemplo, SQL es el lenguaje empleado para describir consultas, inserciones o modificaciones de bases de datos. El lenguaje está específicamente diseñado para ello. Otro ejemplo son los lenguajes que incluyen paquetes estadísticos como SPSS que permiten manipular grandes cantidades de datos con fines estadísticos.

Quinta generación. Los lenguajes de quinta generación son los utilizados principalmente en el área de la inteligencia artificial. Se trata de lenguajes que permiten especificar restricciones que se le indican al sistema, que resuelve un determinado problema sujeto a estas restricciones. Algunos ejemplos de lenguajes de quinta generación son Prolog o Mercury.

Cada generación ha tratado de resolver los problemas detectados en la generación anterior. Así, los lenguajes de segunda generación surgieron a partir de la necesidad de evitar tener que recordar las instrucciones de una computadora concreta como una secuencia de ceros y unos. Para los humanos es más sencillo recordar palabras (como ADD o LOAD) que estas secuencias. Los lenguajes ensambladores se crearon con este propósito.
Sin embargo, los lenguajes de segunda generación eran poco convenientes para crear programas complejos. A medida que el software se hacía más complejo era más difícil mantenerlo debido a lo poco estructurado que es el lenguaje ensamblador. Los lenguajes ensambladores básicamente no permitían modularizar el código más allá de crear macros que podían reutilizarse en otras partes del programa. Esto llevó a la aparición de los lenguajes de tercera generación. Las características proporcionadas por estos lenguajes han variado muchísimo, desde la aparición de los primeros lenguajes de alto nivel como FORTRAN o ALGOL que incorporaron el concepto de subprograma, hasta los modernos lenguajes como Java, C# o Ruby que incluyen conceptos de más alto nivel como clases o paquetes.

La aparición de los lenguajes de tercera generación introdujo un problema realmente complejo: la traducción de los programas escritos en estos lenguajes a código máquina (secuencias de ceros y unos entendible por la computadora). El lenguaje ensamblador era muy sencillo de traducir, dado que la relación entre instrucciones en lenguaje ensamblador e instrucciones de código máquina era de uno a uno: básicamente las instrucciones en ensamblador no eran más que mnemotécnicos que ayudaban a recordar el nombre de las operaciones, en lugar de tener que escribir la secuencia de ceros y unos correspondiente. Sin embargo, en los lenguajes de tercera generación, una instrucción del lenguaje podía requerir varias decenas de instrucciones de código máquina. Para traducir los programas escritos en estos lenguajes se crearon los compiladores. Tradicionalmente, un compilador es un programa que traduce un programa escrito en un determinado lenguaje de programación a código máquina. De los programas traductores, y en general de los procesadores de lenguajes, se hablará en el capítulo 2.

Los lenguajes de programación tienen orígenes muy diversos. En ocasiones un lenguaje se crea para explorar los límites de las máquinas. En otras ocasiones se diseñan para incorporar características deseadas por los usuarios finales (los programadores que van a utilizarlo), como en el caso de COBOL (orientado a la construcción de aplicaciones de negocios, como bancos) y Ada (diseñado para el ministerio de defensa estadounidense). El tipo de aplicación que se va a construir puede determinar fuertemente el lenguaje a utilizar. Por ejemplo, para aplicaciones en tiempo real, donde el tiempo de respuesta es primordial, como los programas que controlan centrales nucleares o el piloto automático de un avión, se suele utilizar Ada o C. Para aplicaciones web, sin embargo, no es muy habitual utilizar C y se tiende más hacia lenguajes como Java, Python, PHP o Ruby.

Se puede ver un lenguaje de programación como una interfaz entre la máquina y el usuario (u otros programas). Desde este punto de vista surgen tres cuestiones a las que hay que dar respuesta: primero, cuál es la estructura (sintaxis) y el significado (semántica) de las construcciones permitidas por el lenguaje. Segundo, cómo traducirá el compilador dichas construcciones a lenguaje máquina. Tercero, cuán útil es el lenguaje para el programador. En otras palabras, si es suficientemente expresivo, legible o simple. Estas son las cuestiones que se estudiarán en el resto de este capítulo.

1.2 Sintaxis

Cuando aparecieron los primeros lenguajes de tercera generación, los compiladores se construían para un lenguaje concreto basándose en la descripción del lenguaje de programación correspondiente. Esta descripción se proporcionaba en lenguaje natural (inglés) en forma de manuales de referencia del lenguaje y constaba de tres partes: la descripción léxica del lenguaje, que especifica cómo se combinan símbolos para formar palabras; la descripción sintáctica, que establece las reglas de combinación de dichas palabras para formar frases o sentencias; y la descripción semántica, que trata del significado de las frases o sentencias en sí. El problema de realizar la descripción en lenguaje natural es la

ambigüedad: el lenguaje natural está lleno de ambigüedades y es difícil ser muy preciso en la descripción de estos elementos del lenguaje.

Por ello, a finales de los años 50, mientras diseñaba el lenguaje Algol 58 para IBM, John Backus ideó un lenguaje de especificación para describir la sintaxis de este lenguaje. Posteriormente, Peter Naur retomó su lenguaje de especificación y lo aplicó para definir la sintaxis de Algol 60 y lo llamó Backus Normal Form. Finalmente, este lenguaje de especificación acabó llamándose BNF (de Backus-Naur Form) y ha sido el estándar de facto para la descripción de la sintaxis de muchos lenguajes de computadora, como lenguajes de programación o protocolos de comunicación.

En lingüística, la sintaxis es la ciencia que estudia los elementos de una lengua y sus combinaciones. Es la parte que se encarga de establecer la estructura que deben tener las sentencias del lenguaje. En los lenguajes de computadora no es diferente: la sintaxis de un lenguaje establece las normas que han de cumplir los diferentes elementos del lenguaje para formar sentencias válidas de ese lenguaje. Para describir la sintaxis de los lenguajes de programación se pueden utilizar diferentes notaciones. El lenguaje BNF, o algún derivado como EBNF (*Extended BNF*), es la notación más habitualmente utilizada. Sin embargo, en ocasiones también se utiliza una notación gráfica denominada diagramas sintácticos.

Desde un punto de vista formal la sintaxis de un lenguaje de programación es un lenguaje independiente del contexto. Un lenguaje independiente del contexto es un lenguaje que se puede describir con una gramática independiente del contexto. Este tipo de gramáticas se estudian como parte de la teoría de lenguajes formales, que define las propiedades de los diferentes tipos de lenguajes formales. Estas propiedades son las que permiten realizar razonamientos matemáticos sobre los lenguajes y, en última instancia, construir herramientas que permitan reconocer una determinada cadena de un programa como perteneciente al lenguaje de programación cuya sintaxis viene definida por una determinada gramática independiente del contexto. La construcción de compiladores, y en concreto de las diferentes fases de análisis de un compilador, se asienta en la teoría de lenguajes formales.

Las notaciones BNF y EBNF, así como los diagramas sintácticos, permiten representar textualmente (las primeras) o gráficamente (la segunda) una gramática independiente del contexto.

1.2.1 Gramáticas independientes del contexto

Una **gramática independiente del contexto** es una 4-tupla $G(T,N,S,P)$, donde T es el **alfabeto** del lenguaje, N es el conjunto de **símbolos no terminales**, S es un símbolo no terminal especial denominado **símbolo inicial**, y P es el conjunto de **producciones** de la gramática. El lenguaje definido por una gramática independiente de contexto G se denota por $L(G)$ y se denomina **lenguaje independiente del contexto**.

En una gramática independiente del contexto las producciones son de la forma:

$$N \rightarrow w, \tag{1.1}$$

donde $N \in N$ y $w \in \{N \bigcup T\}*$. Al no terminal N se le denomina **antecedente**, y a la cadena de símbolos w **consecuente**.
Una producción puede interpretarse como una **regla de reescritura** que permite sustituir un elemento no terminal de una cadena por el consecuente de alguna de las producciones en las que ese no terminal actúa como antecedente. Esta sustitución se conoce como **paso de reescritura** o **paso de derivación**. El lenguaje definido por una gramática independiente del contexto son todas aquellas cadenas que se pueden derivar en uno o más pasos de derivación desde el símbolo inicial S.
Supóngase que se desea describir formalmente el lenguaje, que denominaremos $L1$, de las cadenas sobre el alfabeto $\{a, b, +\}$ en el que el símbolo terminal $+$ aparece siempre entre alguno de los símbolos a o b. La siguiente gramática describe dicho lenguaje:

$$\begin{array}{l} S \rightarrow A + A | A \\ A \rightarrow a | b | S \end{array} \tag{1.2}$$

El lenguaje $L1$ incluye, por ejemplo, la cadena $a+b+b$. Si la gramática describe de forma precisa el lenguaje, entonces esta cadena debe poder ser generada en sucesivos pasos de derivación a partir del símbolo inicial S:

$$S \Rightarrow A+A \Rightarrow a+A \Rightarrow a+S \Rightarrow a+A+A \Rightarrow a+b+A \Rightarrow a+b+b \tag{1.3}$$

Las gramáticas pueden utilizarse para decidir si una cadena pertenece al lenguaje: basta comprobar si es posible generar la cadena partiendo del símbolo inicial realizando sustituciones utilizando las producciones de la gramática.

1.2.2 Notación BNF

La notación BNF representa una forma compacta de escribir gramáticas independientes de contexto. Con este formalismo, los símbolos no terminales se encierran entre los paréntesis angulares ($<$ y $>$), como en $<$ *expresion* $>$, y las producciones se expresan mediante el siguiente formato:

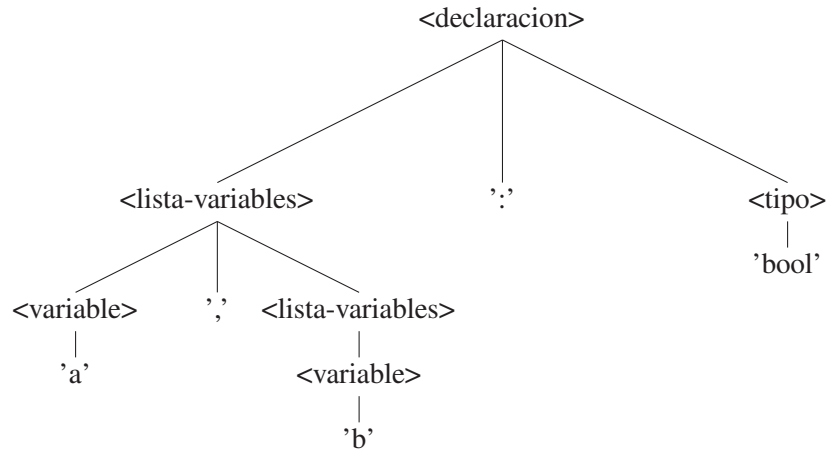
$$< antecedentes > ::= < consecuentes > \tag{1.4}$$

No existe un consenso claro sobre la forma de representar los símbolos terminales, pero en ocasiones se expresan entre comillas simples o dobles, como en $'if'$, $'for'$ o $'int'$, aunque esto no es obligatorio. Las gramáticas BNF se utilizan para definir la sintaxis de los lenguajes de programación. Por ejemplo, la siguiente gramática en notación BNF define la parte de declaraciones de variables de un lenguaje:

$$
\begin{array}{l}
< declaracion > ::= < lista - variables > ' : ' < tipo > \\
< lista - variables > ::= < variable > \mid < variable > ' , ' < lista - variables > \\
< variable > ::= ' a' \mid ' b' \mid ' c' \\
< tipo > ::= ' bool' \mid ' int' \mid ' float' \mid ' char'
\end{array}
\tag{1.5}
$$

Árboles de análisis sintáctico

Para determinar si una cadena representa una expresión o enunciado válido en un lenguaje cuya sintaxis viene definida por una gramática BNF se utilizan las producciones de la gramática. Así, se trata de obtener la cadena buscada a partir del símbolo inicial en sucesivos pasos de derivación. Este proceso da lugar a lo que se conoce como **árbol de análisis sintáctico**. Se comienza por el símbolo inicial de la gramática, que será la raiz del árbol. En un primer paso, en base a la cadena buscada, se sustituye dicho símbolo por alguno de los consecuentes de las producciones en las que el símbolo inicial aparece como antecedente. En el árbol de análsis sintáctico esto se representa añadiendo cada símbolo del consecuente como hijo del símbolo inicial. Este proceso se repite, realizando sustituciones, hasta que todas las hojas del árbol son símbolos terminales. Por ejemplo, si se tiene el lenguaje definido por la gramática 1.5, la cadena $a,b : bool$ tiene el siguiente árbol de análisis sintáctico:



Este árbol de análisis sintáctico se corresponde con la siguiente derivación:

```
1 <declaracion> =>
2 <lista-variables> ':' <tipo> =>
3 <variable> ',' <lista-variables> ':' <tipo> =>
4 'a' ',' <lista-variables> ':' <tipo> =>
5 'a' ',' <variable> ':' <tipo> =>
6 'a' ',' 'b' ':' <tipo> =>
7 'a' ',' 'b' ':' 'bool'
```

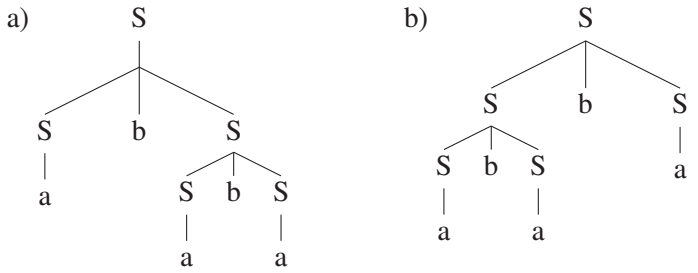


Figura 1.1: *Dos árboles sintácticos para la cadena* ababa

Nótese que en ocasiones en un determinado paso de derivación podemos elegir entre varios no terminales. En este caso se ha utilizado la regla de sustituir el no terminal más a la izquierda primero, y proceder sustituyendo de izquierda a derecha. Determinados algoritmos de análisis sintáctico proceden realizando sustituciones por la izquierda (sustituyen en cada paso el no terminal más a la izquierda), mientras que otros proceden sustituyendo el no terminal más a la derecha.
Considérese a continuación la gramática BNF para el lenguaje de las cadenas de *a*es y *b*es donde toda *b* aparece siempre entre dos *a*es:

$$< S >::=< S > b < S > | a \tag{1.6}$$

La cadena *ababa* pertenece al lenguaje definido por la gramática 1.6, o lo que es lo mismo, puede obtenerse mediante sucesivos pasos de derivación desde el símbolo inicial:

$$\begin{aligned} &< S > \Rightarrow < S > b < S > \Rightarrow < S > b < S > b < S > \Rightarrow ab < S > b < S > \Rightarrow \\ &abab < S > \Rightarrow ababa \end{aligned} \tag{1.7}$$

Esta derivación se puede representar mediante dos árboles sintácticos diferentes, como se muestra en la Figura 1.1.
Cuando una misma cadena se puede representar mediante dos o más árboles sintácticos se dice que la gramática es **ambigua**. La ambigüedad es un problema en el contexto de los lenguajes de programación porque un analizador sintáctico no puede decidir cuál de los dos árboles sintácticos construir. Análogamente, cuando para toda cadena del lenguaje existe únicamente un árbol sintáctico se dice que la gramática es no ambigua.
El problema de la ambigüedad radica en que cambia el significado dado a la cadena. Si se considera el lenguaje de las expresiones aritméticas de sustracción, podría pensarse en construir una gramática como la siguiente:

$$< expresion >::=< expresion >' -' < expresion > | < constante > \tag{1.8}$$

Según la gramática 1.8, la expresión $2-1-1$ se puede representar como los dos árboles sintácticos de la Figura 1.2. Estos dos árboles representan diferente asociatividad

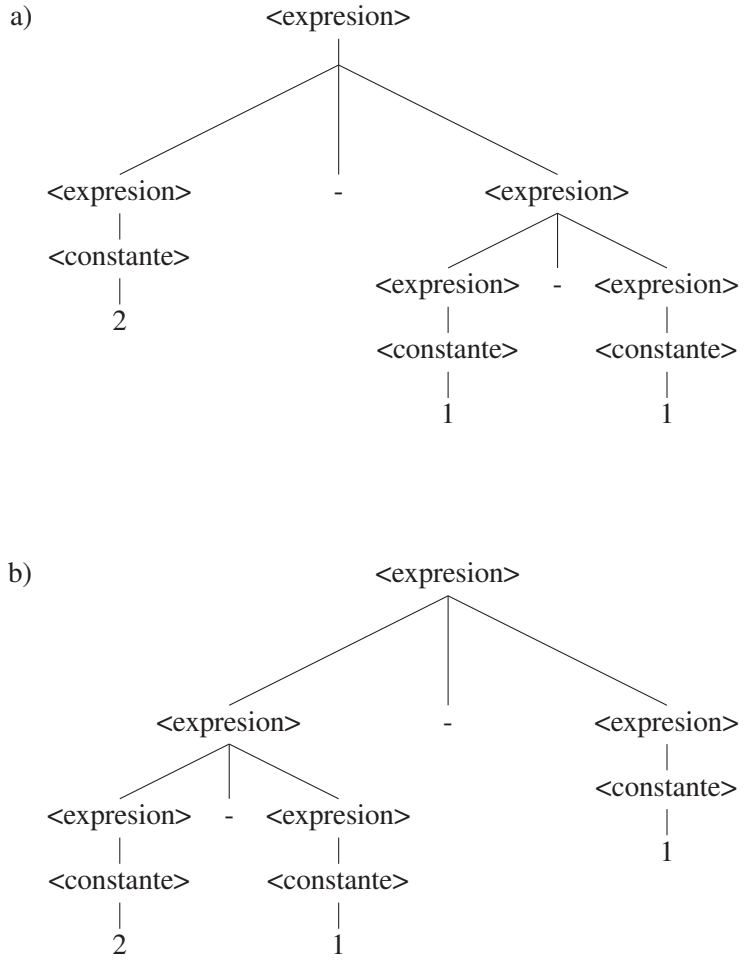


Figura 1.2: *Dos árboles sintácticos para la expresión* $2-1-1$

para la resta. El árbol a) representa la expresión $2-(1-1)=2$ (asociatividad por la derecha), mientras que el árbol b) representa la expresión $2-1-1=0$ (asociatividad por la izquierda). Cuando una gramática para un lenguaje de programación es ambigua, es necesario eliminar la ambigüedad y, además, hacerlo de tal forma que prevalezca el árbol sintáctico adecuado en cada caso.
Es posible escribir una gramática que no tenga los problemas de la gramática ambigua 1.8 eliminando las producciones que son recursivas por la derecha:

$$< expresion >::=< expresion >' -' < constante > | < constante > \qquad (1.9)$$

Esta nueva forma de describir el lenguaje de las expresiones aritméticas de sustracción no presenta problemas de ambigüedad. La cadena $2-1-1$ ahora sólo tiene un árbol

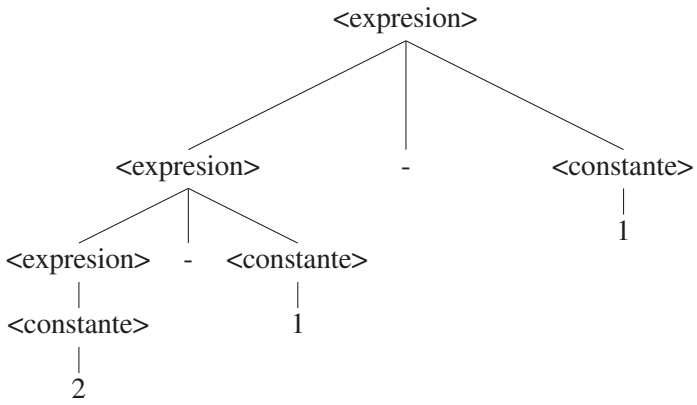


Figura 1.3: *Árbol sintáctico para la expresión* $2-1-1$ *según la gramática 1.9*

posible, el mostrado en la Figura 1.3.

Notación EBNF

La **notación EBNF** (*Extended BNF*) es una forma derivada de BNF que fue introducida para describir la sintaxis del lenguaje Pascal, y que actualmente es un estándar ISO (ISO/IEC 14977). EBNF simplifica la notación BNF mediante el uso de paréntesis para agrupar símbolos (( )), llaves para representar repeticiones ({ }), corchetes para representar una parte optativa ([ ]) –correspondiente a una cardinalidad de 0 ó 1–, o representando los símbolos terminales entre comillas sencillas. Además, se permite el uso del cuantificadores: $\{\}*$ para representar 0 ó más veces, y $\{\}+$ para representar una cardinalidad de 1 ó más veces.
La gramática EBNF 1.10 representa una posible gramática para expresiones aritméticas. Como puede verse, no se explicita la recursividad en la definición de $< expresion >$ y de $< termino >$.

$$\begin{array}{l} < expresion > ::= < termino > \{('+' \mid '-') < termino >\}* \\ < termino > ::= < factor > \{('*' \mid '/') < factor >\}* \\ < factor > ::= ' a' \mid 'b' \mid 'c' \mid '('< expresion >')' \end{array} \tag{1.10}$$

Árboles de sintaxis abstracta

En ocasiones no es necesario representar cada detalle sintáctico en el árbol. Por ejemplo, para conocer la estructura de un determinado fragmento de código no es necesario representar en el árbol sintáctico cada elemento sintáctico (paréntesis, llaves, puntos y comas). Por ello los traductores muchas veces trabajan sobre una versión reducida del árbol de análisis sintáctico denominada árbol de sintaxis abstracta. En este tipo de árboles gran parte de los símbolos han sido suprimidos, y el árbol es mucho más conciso.

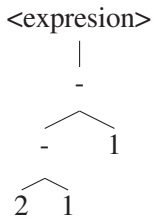


Figura 1.4: *Árbol de sintaxis abstracta para la expresión* $2 - 1 - 1$ *según la gramática 1.9*

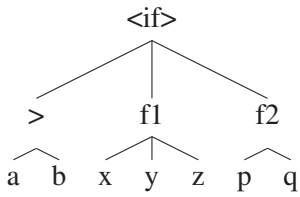


Figura 1.5: *Árbol de sintaxis abstracta para una sentencia if*

Por ejemplo, normalmente en un árbol de sintaxis abstracta las expresiones aritméticas binarias se representan con un nodo del que cuelgan los operandos, como se muestra en la Figura 1.4. Este árbol es mucho más conciso que el árbol sintáctico de la Figura 1.3, pero representa exactamente la misma expresión aritmética.
De igual forma, una expresión condicional puede representarse como un subárbol donde la raiz es una etiqueta que indica bifurcación, y tres nodos hijos: el subárbol para la condición, el subárbol para la sentencia o conjunto de sentencias de la rama que se ejecuta si la condición es verdadera y el subárbol para la sentencia o conjunto de sentencias de la rama que se ejecuta si la condición es falsa. Esta representación puede observarse en la Figura 1.5. En este árbol, las funciones invocadas se representan especificando el nombre de la función en la raíz del subárbol correspondiente y a continuación un hijo por parámetro. En los árboles de sintaxis abstracta desaparecen la mayoría de construcciones gramaticales para mostrar la estructura de una forma concisa. Así, se eliminan los símbolos de puntuación, paréntesis (que no son necesarios dado que dicha información ya está implícita en el árbol), corchetes y todos aquellos símbolos no terminales que puedan sustituirse directamente por sus consecuentes sin pérdida de información. Así, la Figura 1.4 omite los símbolos no terminales $factor$ y $termino$ cuya única función es forzar la precedencia y asociatividad de los operadores, centrando los subárboles en los operadores correspondientes. En este caso precedencia y asociatividad ya quedan fijados por la estructura que tiene el árbol, por tanto esa información es superflua.

Diagramas sintácticos

Los diagramas sintácticos representan en forma gráfica una gramática independiente del contexto. En un diagrama sintáctico, los símbolos no terminales se representan habitual-

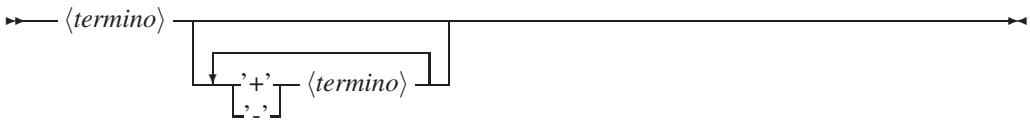


Figura 1.6: *Diagrama sintáctico correspondiente a la gramática 1.11*

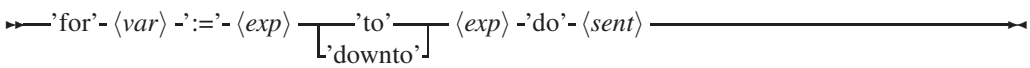


Figura 1.7: *Diagrama sintáctico correspondiente a la gramática 1.12*

mente mediante rectángulos y los símbolos terminales se representan por círculos. Una palabra reconocida se representa como un camino entre la entrada (izquierda) y la salida (derecha). Sin embargo, en ocasiones los símbolos de la gramática se representan con la notación BNF. Esta última será la convención que se utiliza en este texto.
La Figura 1.6 muestra la siguiente producción mediante un diagrama sintáctico:

$$< expresion > ::= < termino > \{('+' \mid '-') < termino >\} \tag{1.11}$$

La gramática 1.12, correspondiente a la sintaxis de `for` de Pascal, se representaría en forma de diagrama sintáctico como se muestra en la Figura 1.7.

$$< sent - for > ::= ' for' < var > ':=' < exp > ('to' \mid 'downto') < exp > ' do' < sent > \tag{1.12}$$

1.3 Semántica básica

La sintaxis de un lenguaje especifica cómo deben disponerse los componentes elementales del lenguaje para formar sentencias válidas. La semántica se encarga de comprobar el correcto sentido de las sentencias construidas. La especificación semántica del lenguaje establece cómo se comportan los diferentes elementos del lenguaje. Considérese la siguiente sentencia del castellano:

```
He comido un poco de todo.
```

Esta sentencia es válida desde el punto de vista sintáctico: está bien formada, tiene un sujeto y un predicado, un verbo, etc; también lo es desde el punto de vista semántico, la frase tiene un significado concreto. La sentencia que se muestra a continuación, pese a ser correcta sintácticamente (sólo cambia el verbo respecto al ejemplo anterior), no es correcta semánticamente:

```
He dormido un poco de todo.
```

La semántica del lenguaje se encarga de determinar qué construcciones sintácticas son válidas y cuáles no. Para ello existen varias notaciones, como la semántica denotacional, la semántica axiomática o la semántica operacional. Sin embargo, estas notaciones son bastante complejas y se apoyan fuertemente en formalismos matemáticos. Por ello, para la definición semántica del lenguaje generalmente se utiliza una variación denominada *gramáticas atribuídas*. Las gramáticas atribuídas son una extensión de las gramáticas independientes del contexto utilizadas para la definición sintáctica de los lenguajes de programación. Se verán más en detalle estos tipos de gramáticas en el capítulo 2.

1.4 Tipos de datos

Un tipo de datos representa un conjunto de valores y un conjunto de operaciones definidas sobre dichos valores. Casi todos los lenguajes de programación incluyen tipos de datos primitivos (o predefinidos) para cuyos valores disponen de construcciones sintácticas especiales. Además, la mayoría de los lenguajes de programación permiten al programador definir nuevos tipos a partir de los ya existentes.
Los tipos de datos se pueden agrupar en dos clases:

Tipos de datos simples. Son aquellos que no pueden descomponerse en otros tipos de datos más elementales. Se incluyen dentro de esta clase los tipos de datos entero, carácter, lógico (booleano), real, enumerado o subrango.

Tipos de datos estructurados. Son aquellos que se componen de tipos de datos más básicos. Ejemplos de tipos de datos estructurados son los registros, arrays, listas, cadenas de caracteres o punteros.

1.4.1 Tipos de datos simples

Los tipos de datos simples son tipos básicos que vienen definidos por el lenguaje de programación y que sirven de base para la construcción de otros tipos de datos (como los tipos estructurados o los tipos definidos por el usuario). El número de tipos de datos simples proporcionados varía de un lenguaje a otro. Además, en algunos casos, el mismo tipo de dato está presente en el lenguaje como dos tipos distintos con distinta precisión. Esto sucede en Java con los tipos de datos `float` y `double`, que representan números reales pero con precisiones distintas. Lo mismo sucede en Haskell con los tipos enteros `Int` e `Integer`: `Int` representa un valores de tipo entero dentro del rango $(-2^{29}, 2^{29} - 1)$; mientras que `Integer` representa números enteros de precisión arbitraria.

Entero

El tipo entero en los lenguajes de programación es normalmente un subconjunto ordenado del conjunto infinito matemático de los enteros. El conjunto de valores que el tipo entero puede representar en un lenguaje viene determinado por el diseñador del lenguaje en ciertos casos, o por el diseñador del compilador en otros. Por ejemplo, en Java el tamaño del conjunto de los enteros viene determinado por la especificación del lenguaje, mientras que en C la precisión de la mayoría de los tipos viene determinada por el diseñador del compilador y suele depender de la arquitectura específica para la que se construyó ese compilador.
El tipo de datos de los enteros suele incluir las operaciones típicas entre valores de este tipo: aritméticas (tanto unarias como binarias), relacionales y de asignación.
La implementación del tipo de datos suele consistir típicamente en una palabra de memoria (32 bits en arquitecturas x86) donde se almacena el valor entero en complemento a 2, aunque no cualquier combinación de bits es válida para representar valores. Es posible que ciertos bits estén reservados para propósitos específicos de la implementación. Por ejemplo, algunos lenguajes reservan dentro de estos 32 bits un bit para indicar desbordamiento (una operación cuyo resultado no cabe dentro del rango de enteros soportado), otros pueden utilizar algún bit como descriptor del tipo de datos, etc.
Algunos lenguajes soportan operaciones a nivel de bit sobre operandos de tipo entero, como desplazamiento, desplazamiento circular, etc. El siguiente código, escrito en C, equivale a multiplicar el primer operando por 2 elevado a la potencia indicada por el segundo operando:

```
1 y = x << 4;
```

Real

El tipo de datos real representa números decimales en notación de punto flotante o punto fijo. Se trata de un subconjunto del conjunto infinito matemático de los números reales. Este tipo de datos incluye las operaciones típicas: aritméticas (unarias y binarias), relacionales y de asignación. Generalmente los lenguajes proporcionan al menos dos tipos de datos reales con distinta precisión, típicamente uno de 32 y otro de 64 bits.
La implementación de números reales suele estar basada en el estándar IEEE 754, donde el número real se divide en una mantisa y un exponente. Diferentes implementaciones de este tipo de datos pueden establecer diferente tamaño de mantisa y exponente. El estándar IEEE 754 define valores especiales para representar los conceptos de *infinito*, *-infinito* y *Not a Number* (`NaN`). Por ejemplo, en Java la siguiente expresión da como resultado `NaN`:

```
1 System.out.println(0.0 / 0.0);
```

Esta otra expresión da como resultado `Infinity`:

```
System.out.println(1.0 / 0.0);
```

Booleano

El tipo de datos booleano consiste en dos únicos valores posibles: verdadero y falso. Incluye las operaciones lógicas típicas: `and`, `or`, `not`, etc. Aunque teóricamente estos valores podrían almacenarse en un único bit, dado que no es posible direccionar un único bit, la implementación suele consistir al menos en un byte.
Algunos lenguajes no incluyen este tipo de dato y en su lugar lo representan mediante otros tipos. Por ejemplo, en C los valores booleanos se representan con enteros, donde todo valor distinto de cero es equivalente al valor verdadero, y el valor cero representa el valor falso.

Carácter

El tipo de datos carácter es la principal vía de intercambio de información entre el programador (o el usuario del programa) y la computadora. Los valores de este tipo de dato se toman de una enumeración fija. En algunos casos puede ser la tabla ASCII (como en C), en otros casos puede ser una tabla más amplia (como Unicode). La implementación en memoria dependerá del tamaño de la tabla. La tabla ASCII se puede representar con un solo byte. Los caracteres Unicode, en cambio, pueden requerir uno, dos o cuatro bytes dependiendo de la implementación utilizada.
Los valores del tipo de datos carácter están ordenados (según la ordenación de la tabla en la que están basados), por lo que habitualmente pueden compararse utilizando operadores relacionales.
Normalmente, los lenguajes soportan ciertos caracteres especiales que pueden representar, por ejemplo, saltos de línea, tabulados, etc. Además, en algunos casos, el programador dispone de funciones predefinidas que le permiten discriminar si cierto carácter es un dígito, una letra, un signo de puntuación, etc.

Enumerados

Los tipos enumerados definen un número reducido de valores ordenados. El tipo se define por enumeración de todos los valores posibles. Es habitual encontrar dentro de este tipo operaciones como sucesor o predecesor, además de los operadores relacionales. El siguiente es un ejemplo de definición de un tipo enumerado en Pascal:

```
type
  DiaSemana = (Lunes, Martes, Miercoles, Jueves, Viernes, Sabado
      , Domingo);
```

Algunos lenguajes implementan internamente los valores del tipo enumerado como enteros, como es el caso de C.

Subrango

Los valores de tipo subrango se crean especificando un intervalo de otro tipo de datos simple. Soportan las mismas operaciones que el tipo de datos básico del que derivan. Pascal es uno de los lenguajes que permiten la definición de subrangos de tipos de datos básicos. Las siguientes definiciones proporcionan un nuevo tipo basado en algunos de los tipos básicos de Pascal:

```
1 type
2   DiaSemana = (Lunes, Martes, Miercoles, Jueves, Viernes, Sabado
        , Domingo);
3   Laborable = Lunes .. Viernes;
4   Mes = 1 .. 12
```

La definición de tipos subrango permite en ocasiones hacer el código más legible y reducir la posibilidad de errores. En el ejemplo anterior, la definición del tipo `Mes` como un subrango de enteros entre 1 y 12, podría ser utilizada en un procedimiento `setMes` del tipo `Fecha`. El compilador asegurará que la función `setMes` no es invocada con un valor entero fuera del rango especificado para `Mes`.

```
1 procedure setMes(m : Mes; var fecha : Fecha);
2 begin
3   fecha.mes := m;
4 end;
```

1.4.2 Tipo de datos estructurados

Los tipos de datos estructurados son tipos de datos que se construyen a partir de otros tipos, que pueden ser a su vez simples o estructurados. Esto permite crear jerarquías de composición de tipos tan complejas como sea necesario. Estos tipos de datos presentan ciertas propiedades:

Número de componentes. Puede ser fijo, como suele suceder en el caso de arrays o registros, o variable, como en el caso de listas, pilas, colas, etc.

Tipo de cada componente. El tipo de los componentes puede ser homogéneo, como en el caso de los arrays, o heterogéneo, como en el caso de los registros.

Acceso a los componentes. La forma de acceder a los componentes puede ser mediante un índice (arrays, listas) o mediante el nombre de un campo (registros).

Organización de los componentes. Los componentes de una estructura de datos pueden organizarse linealmente (como arrays o ficheros), o bien se pueden disponer en múltiples dimensiones.

Las operaciones sobre estos tipos de datos estructurados suelen ser:

Selección de componentes. Se selecciona un componente de la estructura. En este sentido hay que distinguir entre estructuras de acceso directo (como los arrays) y estructuras de acceso secuencial (como los ficheros).

Inserción o eliminación de componentes. Los tipos de datos estructurados de tamaño variable permiten la inserción y eliminación de elementos.

Creación y destrucción de tipos de datos estructurados. El lenguaje propociona construcciones sintácticas específicas para crear y destruir valores de tipos de datos estructurados.

Con estructuras completas. No es habitual que existan operaciones sobre estructuras completas, sin embargo algunos lenguajes sí permiten operaciones como, por ejemplo, sumas de vectores (arrays), asignación de registros (copia campo a campo de un registro) o unión de conjuntos.

Atendiendo al formato de almacenamiento de los valores de estos tipos de datos podemos distinguir entre dos tipos de representación interna: secuencial y vinculada. En la **representación secuencial** toda la estructura de datos se encuentra almacenada en memoria en un único bloque contiguo que incluye tanto el descriptor del tipo como los valores de sus componentes. En la **representación vinculada** el valor se divide en varios bloques de memoria no contiguos, interrelacionados por un puntero llamado vínculo. La estructura de datos tiene que contener un espacio de memoria reservado para este vínculo.
Las estructuras de datos de tamaño fijo generalmente utilizan la representación secuencial, mientras que las de tamaño variable usan la representación vinculada. Es importante notar que, en general, es más sencillo obtener varios bloques de memoria no contiguos más pequeños que uno contiguo más grande. Por tanto, el sistema puede indicar que no hay memoria suficiente para reservar espacio para un tipo de datos con representación secuencial, pero sí la hay para tipos de datos con representación vinculada. Esto se debe a la fragmentación de la memoria. En los lenguajes con gestión automática de la memoria estos problemas pueden atenuarse, ya que el sistema desfragmenta la memoria automáticamente de forma periódica. Durante la desfragmentación, los objetos de datos en memoria se mueven, agrupando los bloques libres de memoria no contiguos. Un ejemplo conocido de lenguaje con gestión automática de memoria es Java.

Arrays

Los arrays son estructuras de datos de tamaño fijo donde todos los elementos son del mismo tipo. Esto hace muy apropiada la representación secuencial para representarlos.

Conociendo la dirección de memoria del primer elemento del array y el tamaño de cada uno de sus elementos, la posición del i-ésimo elemento es $posicion[i] = posicion[0] + i \times T$, donde T es el tamaño del tipo de datos que contiene el array.
En determinados lenguajes es posible definir exactamente el rango de índices que se desean utilizar para el array, como en el caso de Pascal, donde es posible una declaración como la siguiente:

```
1 var a : array [-5..5] of integer;
```

El array `a` de la declaración anterior contiene 11 elementos. El índice del primer elemento es -5 y el índice del último elemento 5. Otros lenguajes no permiten este tipo de construcciones, en cuyo caso normalmente el índice del primer elemento es cero, y si el array tiene tamaño n, el índice del último elemento es $n-1$.
Algunos lenguajes de programación chequean en tiempo de ejecución que la posición del array accedida es correcta, es decir, que dicha posición se encuentra dentro de los límites de declarados. En estos lenguajes el acceso a una posición determinada es un poco menos eficiente, pero aseguran que no se accederá a una posición de memoria no válida (una posición de memoria que no corresponde a la memoria reservada para el array).
La mayoría de los lenguajes permiten declarar arrays bidimensionales, tridimensionales, etc. Estos arrays se declaran indicando el tamaño (o los índices) para cada dimensión. En el siguiente ejemplo en Java se declara un array bidimensional cuya primera dimensión es de tamaño 10 y cuya segunda dimensión es de tamaño 100:

```
1 int[][] matriz = new int[10][100];
```

Los arrays bidimensionales pueden verse como arrays de arrays. Cada posición del array original es a su vez un array del tamaño que indica la segunda dimensión. Igual que sucede con los arrays unidimensionales, el tipo de datos que contiene un array bidimensional es siempre el mismo.
Los arrays bidimensionales también se representan en memoria de forma secuencial. Podemos ver el ejemplo anterior como una matriz de 10 filas y 100 columnas. La representación secuencial de esta matriz consistiría en disponer en primer lugar las 100 columnas de la fila 0 (dado que en Java los arrays comienzan siempre en 0), a continuación las 100 columnas de la fila 1, y así sucesivamente hasta la fila 9.
Es importante notar que en algunos lenguajes de programación es necesario reservar memoria para cada una de las columnas de cada fila. Este enfoque también es útil si no se conoce de antemano el tamaño del array, dado que generalmente los índices sólo se pueden especificar mediante una constante. Así, el fragmento anterior podría haber sido escrito en Java de la siguiente forma (donde `numColumnas` es una variable definida en alguna parte del método o pasada como parámetro):

```
// No se especifica el tamaño de la segunda dimensión
int[][] matriz = new int[10][];
for(int i = 0; i < 10; i++) {
        matriz[i] = new int[numColumnas];
}
```

Registros

Un registro es una estructura de datos compuesta de un número fijo de elementos que pueden tener distinto tipo. Cada uno de los elementos de un registro se denomina campo y tiene asociado un nombre que permite identificarlo, como en el siguiente ejemplo:

```
struct Fraccion {
        int numerador;
        int denominador;
};
```

El acceso a los campos de un registro se realiza habitualmente mediante lo que se denomina la notación punto: anteponiendo la variable del tipo registro al nombre del campo y separándolos por un punto:

```
Fraccion f1;
f1.numerador = 1;
f1.denominador = 2;
```

Los registros se representan en memoria de forma secuencial. Es posible calcular en tiempo de compilación la dirección de memoria de todos los campos como un desplazamiento desde la dirección de memoria del comienzo del registro, dado que éstos tienen tamaño fijo y se conoce el tamaño de cada uno de los campos del mismo.

Listas

Una lista es una colección ordenada de datos. En este sentido se parecen a los arrays. Sin embargo las listas son dinámicas: pueden crecer y disminuir durante la ejecución del programa conforme se añaden y eliminan elementos de la misma. Estos elementos pueden ser del mismo tipo o de tipos distintos, generalmente dependiendo del lenguaje. La representación más común para las listas es la representación vinculada. La lista se representa como un puntero al primer elemento. Este elemento típicamente contiene el dato propiamente dicho y un puntero al siguiente elemento. A este tipo de implementación se la denomina lista enlazada. También es posible implementar una lista con dos punteros, uno apuntando al elemento siguiente y otro apuntando al elemento anterior, por motivos de eficiencia. Este tipo de listas se denominan doblemente enlazadas.

En determinados lenguajes las listas representan un tipo de datos más del propio lenguaje y por tanto se pueden declarar variables de dicho tipo. Esto es especialmente habitual en los lenguajes funcionales (como LISP o Haskell) y en los lenguajes con tipado dinámico (como PHP o Ruby). En otros lenguajes las listas son un tipo de dato definido por el programador que generalmente se proporcionan como librerías. Es el caso de lenguajes como Java o C, en los que es necesario incluir dichas librerías para poder utilizar las definiciones incluidas en las mismas.

1.5 Expresiones y enunciados

Una expresión en un lenguaje de programación es un bloque de construcción básico que puede acceder a los datos del programa y que devuelve un resultado. A partir de las expresiones es posible construir bloques sintácticos más complejos, como enunciados. Las expresiones son el único bloque de construcción de programas de que disponen algunos lenguajes como es el caso de los lenguajes funcionales (como Haskell, LIPS o Scheme).
Un enunciado, también denominado sentencia, es el bloque de construcción en el que se basan los lenguajes imperativos (o de base imperativa como los lenguajes orientados a objetos). En este tipo de lenguajes, mediante enunciados de diferentes tipos se establece la secuencia de ejecución de un programa. Estos enunciados pueden ser de asignación, de alternancia, de composición, etc.

1.5.1 Expresiones

Además de tipos de datos y variables de esos tipos, en un lenguaje hay también constantes (o literales). Una constante es un valor, normalmente de un tipo primitivo, expresada literalmente. Por ejemplo, `5` y `'Hola Mundo'` son constantes. Algunas constantes están predefinidas en el lenguaje, como `nil` en Pascal, `null` en Java o `NULL` en C. También las constantes `true` y `false`, en aquellos lenguajes que soportan los tipos booleanos, suelen estar predefinidas.
Una expresión puede ser una constante, una variable, una invocación de subprograma que devuelva un valor, una expresión condicional (como `?:` en C o Java), o un operador cuyos operandos son a su vez expresiones. Las expresiones representan valores, es decir, su evaluación da como resultado un valor. Este valor puede ser almacenado en una variable, pasado como argumento a un subprograma o utilizado a su vez en una expresión.
Los operadores utilizados en las expresiones tienen una aridad que se refiere al número de operandos que esperan. Un operador puede verse como un tipo especial de función, donde los argumentos son los operandos. Normalmente los operadores son unarios (un operando), binarios (dos operandos) o ternarios (tres operandos). Un ejemplo de operador unario es el operador de cambio de signo (`-`): `-4`. Los operadores unarios pueden ir

antes o después del operando. Por ejemplo, el operador ++ que incrementa en una unidad el operando sobre el que se aplica, puede ir antes o después, con comportamientos ligeramente diferentes en cada caso. No hay unanimidad sobre cómo se utilizan los operandos unarios. Por ejemplo, desreferenciar un puntero en Pascal se hace utilizando el operador ↑ en notación postfija (a continuación del operando): `miPuntero↑ := miValor`. En C sin embargo el operador de desreferenciación, `*`, es prefijo: `*miPuntero := miValor`. En los lenguajes imperativos, es habitual que los operadores binarios sean infijos (como los operadores matemáticos usuales): `+`, `-`, `*`, `%`, `div`, `mod`, etc. Un caso especial es la asignación. En algunos lenguajes la asignación es también una expresión, que devuelve el valor que es asignado. En estos lenguajes es posible encadenar asignaciones: `a = b = c`. También es posible utilizar asignaciones allí donde se espere una expresión: `if(a = getChar())`. El problema es que la legibilidad disminuye y es fácil cometer errores como: `while(a = NULL)`, en lugar de `while(a == NULL)` en un lenguaje donde el operador de comparación por igualdad es ==.

Por otro lado, los operadores suelen estar sobrecargados: `+` puede aplicarse a enteros y reales. Incluso puede aplicarse en algunos lenguajes a valores de tipos diferentes: en C puede sumarse un entero y un real. Algunos lenguajes permiten sobrecargar los operadores, es decir, redefinirlos para un nuevo tipo de datos. C++ por ejemplo permite redefinir el operador + para actuar sobre un tipo de datos matriz.

1.5.2 Efectos colaterales

Las expresiones suelen estar exentas de efectos colaterales, es decir, frente a las sentencias que son instrucciones que cambian el estado de la máquina, las expresiones no deberían cambiar el estado de los elementos que participan en ellas. Sin embargo, esta regla no se cumple siempre, y existen operadores que cambian el estado de los operandos sobre los que actuan. Un caso concreto son los operadores de incremento y decremento en C: `++i` incrementa `i` en una unidad. La modificación de `i` aquí es un efecto colateral. Compárese esto con la decisión tomada en Pascal de que el incremento y decremento fueran sentencias: `inc(i)` y `dec(i)`, respectivamente. Como una sentencia no puede utilizarse en una expresión, se evitan así efectos colaterales.

En determinados lenguajes la frontera entre expresiones y sentencias se ha difuminado mucho, pudiendo en ocasiones utilizar un operador como si fuera una sentencia (donde el cambio de estado es el efecto colateral causado por el operador). En C, puede escribirse `i++`; como si fuera una sentencia. Lo contrario también ocurre. Otro ejemplo en C es la asignación que es una sentencia y una expresión. En lenguajes como C y derivados (Java, C++), la asignación, además de cambiar el valor, devuelve ese mismo valor, de forma que se pueden encadenar asignaciones o utilizarse asignaciones donde sea que se espera una expresión: `a = b = c`. El operador de asignación tiene precedencia de derecha a izquierda, de forma que primero se asigna el valor de `c` a `b`, ese valor se devuelve y se asigna a `a`.

1.5.3 Evaluación de expresiones

Los lenguajes pueden realizar la evaluación de expresiones de diferentes formas. Aunque desde un punto de vista matemático está claro cuál es la ordenación de las expresiones en una expresión mayor, los lenguajes pueden elegir no evaluarla completamente por motivos de eficiencia (como ocurre en las expresiones condicionales en algunos lenguajes) o demorar el cálculo de una expresión hasta que el resultado de la misma es realmente necesario. Ambos tipos de políticas de evaluación se describen a continuación.

Evaluación en cortocircuito

En ocasiones se puede conocer el valor de determinadas expresiones sin evaluarlas completamente. Este es el caso de las expresiones con conectivas lógicas como *and* y *or*. Si el primer operando toma un determinado valor entonces el valor de la expresión completa a veces se puede conocer sin necesidad de evaluar el segundo operando. Algunos lenguajes aprovechan esta característica para evitar evaluar el segundo operando, si no es estrictamente necesario para conocer el valor de la expresión. Esto se conoce como **evaluación en cortocircuito**. Los lenguajes con evaluación en cortocircuito permiten hacer cosas como la siguiente:

```
1 while (i < a.length and a[i] == 0) {
2         ...
3 }
```

En este tipo de lenguajes se garantiza que el segundo operador no se va a evaluar si el primero es falso, porque la expresión `false and x` evalúa siempre a falso independientemente del valor de `x`. Sabemos que nunca se va a producir un desbordamiento del índice del array.

Evaluación diferida y evaluación estricta

Dependiendo del momento de evaluación de las expresiones podemos hablar de evaluación diferida o evaluación estricta. La **evaluación diferida** consiste en retardar el cálculo de las expresiones hasta que realmente son necesarias. La evaluación diferida es utilizada habitualmente en los lenguajes funcionales. Los lenguajes imperativos (como C, Java o Pascal) utilizan **evaluación estricta**, lo que quiere decir, por ejemplo, que los argumentos de una función son completamente evaluados antes de la invocación de la función. En evaluación diferida, esto no tiene porqué ser necesariamente así. Considérese el siguiente ejemplo de código C:

```
1 int maxMin(int a, int b, int exprA, int exprB) {
2         if(a > b)
3                 return exprA
```

```
4         else
5                 return exprB
6 }
```

La siguiente invocación del método fuerza la completa evaluación de los argumentos del mismo (dado que C implementa evaluación estricta):

```
1 result = maxMin(a, b, factorial(a), factorial(b));
```

En lenguaje Haskell el mismo ejemplo puede escribirse de la siguiente forma:

```
1 maxMin(Integer,Integer,Integer,Integer)->Integer
2 maxMin(a,b,exprA,exprB) = if(a > b) then exprA else exprB
```

Sin embargo, la misma invocación del método `maxMin` difiere la evaluación de los argumentos hasta saber cuál de las dos expresiones será devuelta (evaluación diferida):

```
1 maxMin(a, b, factorial(a), factorial(b))
```

Si `a` resultó mayor que `b`, entonces sólo el factorial de `a` fue evaluado. La evaluación diferida se conoce también como **evaluación perezosa** y la evaluación estricta como **evaluación ansiosa**.

1.5.4 Enunciados

En los lenguajes imperativos los enunciados se disponen en secuencias formando subprogramas e incluso programas enteros. Algunos enunciados permiten controlar el orden de ejecución de otros enunciados, posibilitando así repetir conjuntos de enunciados un número determinado de veces, o escoger entre una de varias secuencias de enunciados en base a los datos manejados por el programa.
El enunciado más básico es el enunciado de asignación que cambia el estado del programa asignando el valor obtenido por alguna expresión a alguna variable del programa. Como se ha destacado anteriormente este enunciado a veces es considerado también una expresión.

Enunciado compuesto

Un enunciado compuesto es una secuencia de enunciados que se ejecuta secuencialmente comenzando por el primer enunciado y acabando en el último. Los enunciados compuestos pueden incluirse allí donde se espere un enunciado para construir enunciados más grandes.
En Pascal, un enunciado compuesto se forma agrupando una serie de enunciados entre las palabras reservadas `begin` y `end`:

```
begin
        enunciado_1;
        enunciado_2;
        ...
end
```

Enunciados condicionales

Los enunciados condicionales permiten controlar cuál, de un grupo de enunciados (ya sean simples o compuestos) se ejecuta en base a una determinada condición que es necesario evaluar.
El enunciado condicional más habitual es el `if`. Este enunciado tiene dos formas. En la forma más básica, `if` evalúa una condición y de cumplirse ejecuta la secuencia de enunciados que contiene. El siguiente es un ejemplo de este tipo de construcción en Java:

```
if (denominador == 0) {
        System.out.println("Division por cero");
}
```

En el ejemplo anterior la secuencia de enunciados (entre llaves) a continuación de la condición del `if` sólo se ejecuta en caso de que la variable `denominador` sea cero en el momento de evaluar la condición.
La segunda forma del `if` contiene también una secuencia de enunciados a ejecutar en caso de que la condición no se cumpla (generalmente separada por la palabra reservada `else`):

```
if (denominador == 0) {
        System.out.println("Division por cero");
} else {
        System.out.println(numerador + "/" + denominador);
}
```

Otro enunciado condicional habitual en los lenguajes de programación es el enunciado `case`. Este enunciado contiene una condición que es evaluada y a continuación se comprueba la concordancia de dicho valor con una serie de alternativas. Estas alternativas exponen una constante (o rango de valores constantes) que se consideran válidos para ejecutar la secuencia de enunciados de dicha alternativa. El siguiente ejemplo muestra la sintaxis de Pascal para este tipo de enunciados:

```
case (a / 2) of
        0: begin
```

```
3                   enunciado_1;
4                   enunciado_2;
5                   ...
6           end;
7           1: begin
8                   enunciado_3;
9                   enunciado_4;
10                  ...
11          end;
12          else begin
13                  enunciado_5;
14                  enunciado_6;
15                  ...
16          end;
17  end;
```

Enunciados de iteración

Los enunciados de iteración permiten repetir un enunciado un determinado número de veces o hasta que se cumpla una condición. Los enunciados de repetición con contador pertenecen al primer caso. Estos enunciados utilizan una variable como un contador. El programador establece el valor inicial de dicha variable, cuál es el valor máximo que puede tomar y el enunciado que se debe ejecutar. Se comprueba si la variable ha superado el valor máximo y en caso contrario se ejecuta el enunciado especificado y se incremente el contador. Cuando éste llega al valor máximo se abandona el enunciado de iteración y se sigue ejecutando por el siguiente enunciado. A continuación se muestra un ejemplo de enunciado de repetición con contador en Pascal:

```
1 for i := 1 to 10 do
2           writeln(i);
```

En algunos lenguajes es posible controlar la forma de actualizar el valor del contador, como es el caso de C o Java:

```
1 for(int i = 0; i < 10; i+=2) {
2           System.out.println(i);
3 }
```

Los enunciados iterativos con condición ejecutan el enunciado especificado por el programador mientras se cumpla la condición:

```
1 int i = 0;
2 do {
3           System.out.println(i);
```

```
} while (i < 10);
```

Nótese que en el ejemplo anterior los enunciados que se encuentran dentro del enunciado iterativo se ejecutan al menos una vez, dado que la condición en este caso se evalúa al final de cada iteración. Normalmente los lenguajes proporcionan otra versión de este enunciado iterativo donde la condición se evalúa al principio. En este caso, si la condición no se cumple no se ejecutan las sentencias del enunciado iterativo. El siguiente ejemplo en Java es equivalente al anterior, pero la condición se comprueba antes de comenzar cada iteración:

```
int i = 0;
while (i < 10) {
        System.out.println(i);
}
```

Manejo de excepciones

Cualquier programa debe ser capaz de enfrentarse en un momento u otro de su ejecución a determinadas condiciones de error. En algunos casos los errores son irrecuperables (como falta de espacio en disco), en otros es posible realizar alguna acción que permita al programa recuperarse. Pero en cualquier caso no es deseable que el programa finalice sin más, al menos debería informar al usuario de lo que ha sucedido.
Algunos lenguajes de programación incorporan manejo de excepciones que posibilitan al programador capturar determinadas situaciones de error con el objeto de darles el tratamiento apropiado. En Java, por ejemplo, cuando se produce una situación de error es habitual que el código donde dicho error es detectado eleve una excepción. Una excepción es un tipo de datos que contiene información sobre el error y que puede ser capturado estableciendo enunciados específicos para ello en determinadas partes clave del programa, como por ejemplo cuando se trata de abrir un fichero. Dado que es posible que el fichero no exista, pero no se quiere que el programa termine sin más, se puede utilizar un enunciado especial para capturar el error:

```
try {
        FileReader fr = new FileReader(file);
} catch(FileNotFoundException e) {
        // Aquí típicamente utilizaríamos un fichero de log
        // Por motivos de legibilidad simplemente escribimos el
            mensaje por pantalla
        System.out.println("Fichero no encontrado: " + e.
            getMessage());
}
```

En el ejemplo anterior, el constructor de la clase `FileReader` puede lanzar distintos tipos de excepciones, entre ella `FileNotFoundException`, que indica que el fichero no pudo ser abierto porque no existía. Cuando un subprograma (en este caso el constructor de la clase) declara que es posible que se lancen excepciones en caso de error, el programador puede protegerse capturándolas y dándoles el tratamiento adecuado. En este caso simplemente se notifica por pantalla el error. Un enfoque más realista habría sido notificar la usuario el error y preguntarle de nuevo el nombre del fichero.

1.6 Procedimientos y ambientes

Los procedimientos, denominados funciones si devuelven un valor, son subprogramas que pueden ser llamados desde otras partes del código. Típicamente tienen un nombre, una secuencia de parámetros, declaraciones locales de variables y un cuerpo de sentencias. La declaración del subprograma incluye la declaración de los parámetros, denominados **parámetros formales**. Éstos tienen un tipo y habitualmente también un nombre con el que pueden referenciarse posteriormente desde el cuerpo del subprograma. Así, el tipo de un subprograma viene definido por el tipo y número de los parámetros formales. Este tipo es comprobado cuando se realiza una llamada al subprograma y permite detectar errores semánticos en estas llamadas. El siguiente es un ejemplo de procedimiento en C donde `a` y `b` son parámetros formales:

```
1 void max(double a, double b) {
2         if(a > b)
3                 return a;
4         else
5                 return b;
6 }
```

La llamada a un procedimiento debe incluir el mismo número de parámetros, del mismo tipo y en el mismo orden que los indicados en su declaración. A los parámetros de una llamada a un procedimiento se les denomina **parámetros reales** o **parámetros actuales**. En la siguiente llamada al procedimiento `max`, `5.0` y `10.0` son los parámetros reales:

```
1 double maxValue = max(5.0, 10.0);
```

La implementación, a nivel de código objeto, de cómo se realiza una llamada a un fragmento de código que se encuentra en otra posición de memoria y que, además, puede recibir parámetros, no es trivial. Normalmente, la invocación de un subprograma implica guardar todo el estado del subprograma que llama (como los valores que se encuentran en el registro del procesador), evaluar todos los argumentos (en el caso de lenguajes con evaluación estricta), apilar los valores de los mismos en la pila de ejecución para que estén accesibles al subprograma llamado, y finalmente cambiar el valor del conta-

dor de programa (que le indica al procesador la siguiente instrucción a procesar) para que apunte a la primera instrucción del subprograma llamado. Cuando éste termina su ejecución, es necesario sacar sus parámetros de la pila, guardar el valor devuelto por la función en la misma, restaurar los valores de los registros que había antes de la llamada y finalmente cambiar el contador de programa para que apunte a la instrucción siguiente a la llamada, devolviendo el control al subprograma que llama. El tamaño de la pila de ejecución es finito, y puede agotarse.

Algunos lenguajes obligan a utilizar un tipo específico para indicar que un procedimiento no devuelve ningún valor, como `void`. Otros, utilizan una notación distinta para procedimientos y funciones, como es el caso de Pascal donde dos palabras reservadas distintas permiten diferenciar entre ambos tipos: `procedure` y `function`. Hablaremos genéricamente de subprogramas cuando no sea necesario distinguir entre los dos.

Es posible que un lenguaje permita lo que se denominan **funciones sobrecargadas**. Una función se dice que está sobrecargada si existen varias definiciones de la misma función (con el mismo nombre). Para poder distinguirlas unas de otras, las funciones sobrecargadas tienen que poder diferenciarse por el número o tipo de sus parámetros formales. Las funciones sobrecargadas se pueden utilizar cuando existen argumentos con valores por defecto o cuando tiene sentido definir la función para distintos tipos de datos. El siguiente ejemplo en C muestra una función `max` sobrecargada para dos tipos diferentes de argumentos:

```
1 int max(int a, int b) {
2   ...
3 }
4
5 double max(double a, double b) {
6   ...
7 }
```

Hay dos aspectos fundamentales relacionados con los subprogramas: el paso de parámetros y el ámbito de las variables.

1.6.1 Paso de parámetros

Existen diferentes formas de pasar los parámetros a los subprogramas. En el **paso por valor** el valor del parámetro real es copiado en el parámetro formal. Esto implica que realizar cambios en el valor del parámetro formal no afecta al parámetro actual, cuyo valor permanece invariable. En el siguiente ejemplo, la variable `a` definida en la línea 5 permanece inalterada, pese a que el parámetro formal correspondiente sí es modificado:

```
1 void inc(int v) {
2         v++;
3 }
```

```
4 ...
5 int a = 10;
6 inc(a);
```

En el **paso por referencia** la posición de memoria del parámetro real es pasada al subprograma. Existen entonces dos referencias a la misma posición de memoria: el parámetro real y el parámetro formal. Esto implica que cambios en el parámetro formal se verán reflejados en el parámetro real. El siguiente es el mismo ejemplo utilizando paso por referencia, con la sintaxis de C++:

```
1 void inc(int &v) {
2         v++;
3 }
4 ...
5 int a = 10;
6 inc(a);
```

C++ permite tanto paso por valor como paso por referencia. Por defecto, se realiza paso por valor. Si queremos que un determinado parámetro sea pasado por referencia es necesario anteponer el símbolo `&` al nombre del parámetro formal.

En el **paso por copia y restauración** (o valor y resultado) los parámetros reales son evaluados y pasados al subprograma llamado en los parámetros formales. Cuando la ejecución del subprograma termina, los valores de los parámetros formales son copiados de vuelta en las direcciones de memoria de los parámetros reales. Evidentemente, esto sólo se puede hacer con aquellos parámetros reales que representen posiciones de memoria (como variables o indexación de arrays).

Aunque Pascal no tiene paso por copia y restauración, en el siguiente ejemplo se ha intentado ejemplificar lo que pasaría con este tipo de paso de parámetros:

```
1  program copiarest;
2
3  var
4    a : integer;
5
6  procedure inc(num : integer);
7  begin
8    {En este punto se ha copiado a en el parámetro num}
9    num := num +1;
10   {En este punto sólo se ha modificado num, la variable a
11   permanece inalterada}
12 end;
13
14 begin
15   a := 10;
16   inc(a);
```

```
17   {En este punto el valor de num ha sido copiado de vuelta en a}
18 end.
```

En la **llamada por nombre** el subprograma llamado se sustituye por la llamada y los parámetros formales se sustituyen por los parámetros reales. En este tipo de llamada puede ser necesario cambiar los nombres de las variables locales del subprograma para evitar conflictos con las variables del subprograma que hace la llamada.
En el siguiente ejemplo se muestra cómo funcionaría la llamada por nombre en un lenguaje como C (aunque este lenguaje no tiene llamada por nombre). En una llamada por nombre se sustituye la llamada por el propio código de la función, cambiando los nombres de los parámetros por los nombres de las variables:

```
1 int a = 10;
2 a = a + 1; // Se ha sustituido la llamada a inc(a) por la
      implementación de inc
```

1.6.2 Ámbito de variables

En los programas se utilizan constantemente nombres para referenciar diferentes conceptos. Una variable es un identificador que se utiliza como un nombre con el que se hace referencia a una posición de memoria. Un subprograma es un identificador que hace referencia a un fragmento de código con unos parámetros de entrada y que posiblemente devuelve un valor. Una constante es un nombre para un valor que nunca cambia. Los tipos de datos definidos por el usuario (los tipos estructurados o los tipos abstractos de datos) tienen un nombre que se utiliza en declaraciones de variables o de parámetros de subprogramas.
Todos estos nombres pueden estar presentes en un programa, sin embargo, no todos son accesibles al mismo tiempo y desde cualquier parte del programa. En general, existen ciertas reglas de visibilidad que determinan qué variables son accesibles en un determinado momento. Si una variable no es accesible desde un determinado punto del programa, dicha variable no se podrá utilizar. El programa puede contener, por ejemplo, algunas variables globales que son visibles o accesibles desde cualquier parte del programa. Sin embargo, típicamente los subprogramas pueden declarar sus propias variables locales. Cuando un subprograma que declara variables locales es llamado, se enlazan los nombres de sus variables locales a sus posiciones de memoria correspondientes, de forma que puedan utilizarse en el cuerpo del subprograma. Cuando éste termina, estos enlaces se destruyen y por tanto dichas variables ya no están accesibles desde el subprograma que llamó.
La asociación de nombres a datos o subprogramas tiene lugar en lo que se denomina **ámbito** o **alcance** y es lo que define qué variables son visibles desde las diferentes partes del programa. Un ámbito normalmente es un bloque de código que puede estar asociado a una definición de subprograma, un bucle con un conjunto de sentencias asociadas,

una instrucción de bifurcación (`if`) con un bloque `then` y un bloque `else`, etc. Estos bloques pueden tener sus propias declaraciones locales y un conjunto de enunciados y se denominan ámbitos. En el ejemplo siguiente se muestra el cuerpo de una sentencia iterativa `for` con variables locales:

```
1 void swap(int[] values) {
2         int i;
3         for(i = 0; i < values.length-1; i++) {
4                 int temp = values[i];
5                 values[i] = values[i+1]
6         }
7 }
```

En este ejemplo, la función swap define un ámbito en el cual están definidos los nombres `values` e `i`. El enunciado for define su propio ámbito donde además de los anteriores está definido el nombre `temp`.
Como se puede observar los ámbitos se pueden anidar. Un ámbito más interno tiene acceso a los nombres definidos en un ámbito más externo, a menos que el ámbito más interno contenga una definición con el mismo nombre que la del ámbito externo, en cuyo caso la definición más interna oculta a la definición más externa. En el siguiente ejemplo, el parámetro formal `i` queda oculto por la declaración de `i` en el enunciado del `for`, es decir, dentro del cuerpo del `for` una referencia a `i` se resuelve accediendo a la variable definida como contador del bucle y no al parámetro formal de la función:

```
1 void swap(int i, int j) {
2         for(int i = 0; i < this.values.length-1; i++) {
3                 ...
4         }
5
6 }
```

Al salir del enunciado del `for`, vuelve a ser accesible el parámetro formal `i`, dado que se sale del ámbito del `for` y se destruyen los enlaces creados en dicho ámbito, en concreto el enlace local al `for` para `i`.
Atendiendo a cómo se realice el enlazado de nombres a datos en memoria, subprogramas, etc., podemos distinguir entre ámbito estático y ámbito dinámico. En el **ámbito estático** (también denominado **ámbito léxico**), el enlace se puede determinar simplemente en base al código fuente. Basta echar un vistazo al anidamiento de los ámbitos para saber qué nombres son accesibles desde qué ámbitos. Considérese el siguiente ejemplo en Pascal:

```
1 program ambitos;
2 type
3         TArray : array [1..3] of integer;
```

```
4  var
5          a : TArray;
6  procedure uno(i : integer);
7
8          procedure dos;
9          var j : integer;
10                 a : TArray;
11         begin
12                 a[1] := 0;
13                 a[2] := 0;
14                 a[3] := 0;
15                 intercambia(1, 2);
16         end;
17
18         procedure intercambia(i, j : integer);
19         var aux : integer;
20         begin
21                 aux := a[i];
22                 a[i] := a[j];
23                 a[j] := aux;
24                 writeln(a[1], ' ', a[2], ' ', a[3]);
25         end;
26
27 begin
28         a[1] := 1;
29         a[2] := 2;
30         a[3] := 3;
31         dos;
32 end; {uno}
33
34 begin {ambitos}
35         ...
36         uno(1);
37 end. {ambitos}
```

El programa `ambitos` define un ámbito en el cual están definidos los nombres `TArray` y `a`. En el procedimiento `uno` se tiene acceso a estos dos nombres, y además define un ámbito donde están definidos los nombres `i`, `dos` e `intercambia`. El procedimiento `dos` define un ámbito en el cual el nombre de la variable `a` del programa principal queda oculto por la variable local `a` de `dos` (línea 10). Dentro del ámbito de `dos` también está definido el nombre `j`. En el procedimiento `intercambia` se tiene acceso al nombre `a` y además define los nombres `i` y `j` (parámetros formales) y el nombre `aux` (variable local). En el ámbito estático se puede extraer la asociación de nombres del propio código del programa. A la vista del código anterior, en el ámbito estático la referencia al nombre `a` en el procedimiento `intercambia` hace referencia a la variable global `a` declarada en la línea 5. Por tanto el enunciado de la línea 24 daría lugar a la siguiente salida por pantalla:

```
1 2 1 3
```

En el **ámbito dinámico** la asociación entre nombres y objetos (datos, subprogramas, etc.) tiene lugar en tiempo de ejecución y depende del flujo de ejecución del programa. En el ejemplo anterior, utilizando ámbito dinámico, primero tendría lugar la llamada al procedimiento `uno`, y después éste realizaría la llamada al procedimiento `dos` en la línea 31. Al ejecutarse el procedimiento dos, se enlaza el nombre `a` con la definición local de la línea 10. Cuando se invoca el procedimiento `intercambia` en la línea 15, el enlace del nombre `a` dentro de `intercambia` tiene lugar con la última declaración de `a`, que fue la del procedimiento `dos`. El resultado de este programa con ámbito dinámico sería:

```
1 0 0 0
```

La diferencia con el ámbito estático es por tanto que pese a que `intercambia` no está dentro del ámbito de `dos`, al ser llamado por `dos`, hereda sus enlaces, y por tanto la variable `a` se enlaza con la declaración de `a` en `dos` y no a la declaración de `a` en el programa principal.
La mayoría de lenguajes de programación utilizan ámbito estático, que fue el utilizado en el lenguaje ALGOL. Algunas excepciones notables son LISP (aunque las versiones más recientes han pasado a utilizar ámbito estático) y las versiones iniciales de Perl. Algunos lenguajes permiten utilizar ambos enfoques (ámbito estático y ámbito dinámico) como en el caso de Common LISP.

1.7 Tipos abstractos de datos y módulos

Hasta los años 70 las definiciones de tipos de datos se limitaban a especificar la estructura de un nuevo tipo de datos en base a otros tipos ya existentes [21]. Por ejemplo, en Pascal el programador puede definir un nuevo tipo de datos `Alumno`, como de tipo registro con una serie de campos (nombre, apellidos, número de expediente, nota media, etc.) cada uno de los cuales es de un tipo simple o compuesto.
Sin embargo, tal como se estableció en relación a los tipos de datos básicos (o primitivos) de los lenguajes, los tipos de datos estructurados definidos por el usuario típicamente también tienen un conjunto de operaciones asociadas. Si consideramos, por ejemplo, el caso de una lista rápidamente pensamos en operaciones como insertar un elemento, obtener el elemento *i*-ésimo, eliminar un elemento, etc. A partir de la década de los 70 comenzó a pensarse en los tipos de datos como los propios datos junto con las operaciones que se pueden realizar sobre ellos. De esta forma surgieron los **tipos abstractos de datos**. En un tipo abstracto de datos tenemos:

- Un conjunto de **datos** que se acojen a una definición de tipo (que puede ser más o menos compleja).

- Un conjunto de **operaciones** abstractas sobre datos de ese tipo.
- **Encapsulamiento** de los datos de forma que la manipulación de los mismos sólo pueda producirse a través de las operaciones definidas para el tipo abstracto de datos.

En algunos lenguajes sólo es posible simular tipos abstractos de datos, dado que la encapsulación no forma parte del propio lenguaje, como en el caso de Pascal. Sin embargo, hoy día la mayoría de los lenguajes de programación modernos proporcionan mecanismos de encapsulación que permiten implementar tipos abstractos de datos. En el caso del lenguaje Ada se introdujo el concepto de paquete que permitía encapsular definiciones de tipos de datos y las operaciones permitidas sobre estos tipos. En el caso de C++ o Java se proporciona el concepto de clase que encapsula datos y operaciones dentro de la misma unidad sintáctica. De la programación orientada a objetos se hablará en el capítulo 3.
La encapsulación de la información es una parte muy importante del tipo abstracto de datos. Mediante la encapsulación se evita que los usuarios del tipo abstracto de datos tengan que conocer la implementación concreta del tipo. Esto ayuda a pensar en términos de abstracciones. Por ejemplo, considérese el siguiente programa en el cual la función `estadoCivil` devuelve el estado civil de una persona dado su dni:

```
1  program estadoCivil;
2
3  var
4          dni, estado : String;
5
6  function estadoCivil(dni : String) : String;
7  begin
8          ...
9  end;
10
11 begin
12         dni := readln('DNI=');
13         estado := estadoCivil(dni);
14         if estado = 'SOLTERO' then
15                 ...
16         else
17                 ...
18 end.
```

El problema con el fragmento de código anterior es que no podemos estar seguros de que la comparación en la línea 11 sea correcta. Podría ser que la función `estadoCivil` devolviera simplemente los valores `'s'` y `'c'` para denotar los estados soltero y casado respectivamente, o que utilizara las cadenas completas pero en minúsculas. El problema se deriva de que se está utilizando un tipo de datos demasiado genérico (`string`) para

representar un conjunto muy restringido de valores (soltero/casado). Además, el compilador no nos puede proporcionar ninguna ayuda en la línea 11. Para el compilador la comparación de dos cadenas de caracteres es correcta siempre. Evidentemente, este código podría ser mejorado incluyendo una documentación adecuada para la función `estadoCivil`, e incluso definiendo los diferentes valores posibles como constantes. Sin embargo, con este enfoque el compilador sigue siendo de poca ayuda: si el programador que diseña la función `estadoCivil` se olvida de usar las constantes a la hora de devolver el valor, puede que la cadena devuelta no se corresponda con el valor de alguna de estas constantes.

Estos problemas se evitan fácilmente utilizando el tipo de datos enumerado de Pascal. Un tipo enumerado define un conjunto finito de valores y asigna a cada uno de ellos un nombre, de forma que pueda ser unívocamente identificado y distinguido del resto. Una variable de un tipo enumerado sólo puede contener uno de estos valores y se puede comparar por igualdad con los nombres de cada uno de los valores del enumerado:

```
1  program estadoCivil2;
2
3  type
4          TEstadoCivil = (SOLTERO, CASADO);
5
6  var
7          estado : TEstadoCivil;
8          dni : string;
9
10 function estadoCivil(dni : string) : TEstadoCivil;
11
12 begin
13         dni := readln('DNI=');
14         estado := estadoCivil(dni);
15         if estado = S then
16                 ...
17         else
18                 ...
19 end.
```

En esta nueva versión del programa anterior el compilador puede ayudar al programador a identificar errores. Dado que la comparación en la línea 9 es incorrecta (no existe un valor `S` en el enumerado `TEstadoCivil`) el compilador señalará el error al programador. No existe ahora posibilidad de utilizar valores en la comparación que no se correspondan con los valores que devuelve la función, dado que el conjunto de valores se define como un enumerado. Ahora se utiliza la abstracción adecuada para el valor devuelto por la función `estadoCivil`.

La encapsulación de información del tipo abstracto de datos también ayuda a olvidarnos de las partes constituyentes de un tipo para pensar en el todo. El siguiente ejemplo podría

ser una implementación para dar de alta un alumno en una base de datos de alumnos:

```
1 procedure insertar(nombre, apellidos, dni : string);
2 begin
3         ...
4 end;
```

Este procedimiento obliga al programador a conocer los detalles de todos los campos de datos de un tipo `Alumno`. Es más, sin información del contexto se podría pensar que este procedimiento se puede invocar, en el contexto de una aplicación para gestionar la información de una universidad, tanto para alumnos como para profesores, porque no hay ninguna información que limite la aplicabilidad del procedimiento. Si invocáramos el procedimiento con los datos de un profesor, este profesor acabaría dado de alta en la base de datos de alumnos. Considérese ahora el mismo ejemplo utilizando un tipo abstracto de datos `Alumno`:

```
1 procedure insertar(alumno : TAlumno);
2 begin
3         ...
4 end;
```

Ahora no hay ninguna duda de que el procedimiento `insertar` sólo se puede utilizar con alumnos. Si se considera que existe el tipo abstracto de datos `TProfesor` (análogamente al tipo `TAlumno`), entonces un intento de invocar el procedimiento `insertar` con una variable de tipo `TProfesor` resultaría en un error de compilación. Pero además, desde el punto de vista del programador resulta mucho más cómodo fijar su atención en la abstracción alumno en lugar de tener que manejar directamente los datos de los que se compone dicha abstracción (nombre, apellidos, dni).
Algunos lenguajes, como Java, mediante el uso de la herencia y el polimorfismo (que se estudiarán en el capítulo 3) permiten llevar más allá aún este concepto de abstracción. Java proporciona, como parte del conjunto de librerías incluido en la plataforma Java *Standard Edition*, implementaciones de diferentes estructuras de datos como listas, pilas, colas, tablas hash, etc. Por ejemplo, en el caso concreto del tipo de datos lista, este lenguaje proporciona dos implementaciones distintas de listas, una basada en arrays y otra basada en listas enlazadas:

1. `ArrayList` es una implementación basada en arrays. Para conseguir el comportamiento dinámico de una lista, cuando el array se acerca a su tamaño máximo, se reserva espacio para un array mayor y se copian todos los elementos en este nuevo array. Esta operación es costosa, por eso se recomienda utilizar esta clase en situaciones donde insertar y eliminar elementos es poco habitual y lo habitual es realizar consultas, porque una consulta en una lista de tipo `ArrayList` equivale a acceder a una posición del array.

2. `LinkedList` es una implementación basada en listas enlazadas. En esta implementación cada elemento de la lista incluye una referencia hacia el siguiente elemento. Esta implementación está recomendada cuando se realizan numerosas actualizaciones de la lista (inserciones y borrados), pero es mas ineficiente cuando se quiere acceder a una posición concreta de la misma, dado que hay que recorrer la lista desde el principio hasta la posición pedida.

Los detalles concretos de implementación de una lista en Java pueden quedar ocultos al programador si éste utiliza la interfaz `List` común a ambas clases. Esta interfaz define un conjunto de operaciones que se pueden realizar sobre cualquier lista sin especificar cómo se realizan. La implementación de las operaciones queda delegada a las clases `ArrayList` y `LinkedList`. Por tanto, si al programador se le proprociona una variable de tipo `List`, no solamente se le está proporcionando una abstracción de una lista que le permite pensar en términos de operaciones sobre listas, sino que se le oculta la implementación concreta de esa lista. En el siguiente ejemplo, no se puede saber a priori (ni en la mayoría de los casos hace falta saberlo) qué tipo de lista se está utilizando:

```
1 public class Algoritmos {
2
3         public static void quicksort(List lista) {
4                 ...
5         }
6
7 }
```

La mayoría de lenguajes de programación permiten agrupar definiciones de tipos abstractos de datos en unidades que proporcionan un nivel de abstracción aún mayor: los **módulos**, también denominados unidades o paquetes en otros lenguajes. Un módulo generalmente es un conjunto cohesivo de definiciones de tipos abstractos de datos (o clases en los lenguajes orientados a objetos). En Ada los módulos se denominan paquetes. En ocasiones los paquetes se dividen en dos ficheros: un fichero que contiene las declaraciones de los tipos de datos y sus operaciones y otro fichero que contiene las implementaciones de dichas operaciones. En Java, por ejemplo, los paquetes siguen una estructura jerárquica que se corresponde con la estructura de directorios donde se encuentran las clases de cada paquete. Cada paquete puede contener uno o varios ficheros con definiciones de clases o interfaces, y un paquete puede contener a su vez otros paquetes formando una estructura jerárquica.
El hecho de utilizar paquetes posibilita además disponer de un espacio de nombres que ayuda a evitar colisiones. En Java, por ejemplo, el paquete `java.util` contiene la definición de una clase `List`. Pero una clase con el mismo nombre existe también en el paquete `java.awt` que define el comportamiento y la apariencia de una lista de elementos seleccionables en una interfaz gráfica de usuario. Las dos clases se pueden distinguir porque están definidas en diferentes paquetes. Cada paquete define lo que se denomina

un espacio de nombres. Para hacer referencia a la clase `List` del paquete `java.util`, el nombre de la clase se cualifica con el nombre del paquete, es decir, el nombre completo de la clase sería la concatenación del nombre del paquete y el nombre de la clase (habitualmente separados por puntos): `java.util.List`. Si se quisiera hacer referencia a la clase `List` del paquete `java.awt` se utilizaría el nombre `java.awt.List`. La mayoría de lenguajes de programación proporcionan alguna forma de importar el espacio de nombres de forma que no haga falta cualificar el nombre de cada tipo de datos con el nombre del paquete donde está definido. Continuando con el ejemplo anterior, en Java es posible importar la clase `List` del paquete `java.util` mediante el enunciado:

```
1 import java.util.List;
```

A partir de ese momento todas las referencias a `List` se resuelven como referencias a `java.util.List`.

1.8 Ejercicios resueltos

1. Escribe una gramática BNF para la definición de registros en Pascal. Se considera que el símbolo <id> representa un identificador válido del lenguaje. El siguiente es un ejemplo de definición de registro en Pascal:

```
1         TAlumno = record
2                 nombre : string;
3                 apellidos : string;
4                 edad : integer;
5                 asignaturas : TListaAsignaturas;
6         end;
```

```
1   <registro> ::= <id> '=' 'record' <lista-campos> 'end' ';'
2   <lista-campos> ::= <id> ':' <id> ';' <lista-campos>
3   <lista-campos> ::= <id> ':' <id> ';'
```

2. Escribe la gramática del ejericio anterior utilizando la notación EBNF. ¿Es más compacta esta representación? ¿Cuál es la principal diferencia?

```
1   <registro> ::= <id> '=' 'record' <lista-campos> 'end' ';'
2   <lista-campos> ::= { <id> ':' <id> ';' }+
```

   La gramática es más compacta, dado que sólo necesita una producción para el no terminal `<lista-campos>`.

La principal diferencia estriba en la posibilidad de especificar repeticiones sin tener que introducir reglas recursivas.

3. ¿Cuál de las siguientes gramáticas independientes del contexto genera el lenguaje $a^n b^n$?

   a) $I ::= aIb|ab$

   b) $I ::= abI|\varepsilon$

   c) $I ::= aB|\varepsilon; B ::= Ib$

   Tanto la gramática a) como la gramática c) generan el lenguaje $a^n b^n$.

4. Considérese la siguiente gramática BNF que define la forma de construir expresiones con conectivas lógicas en algún lenguaje de programación:

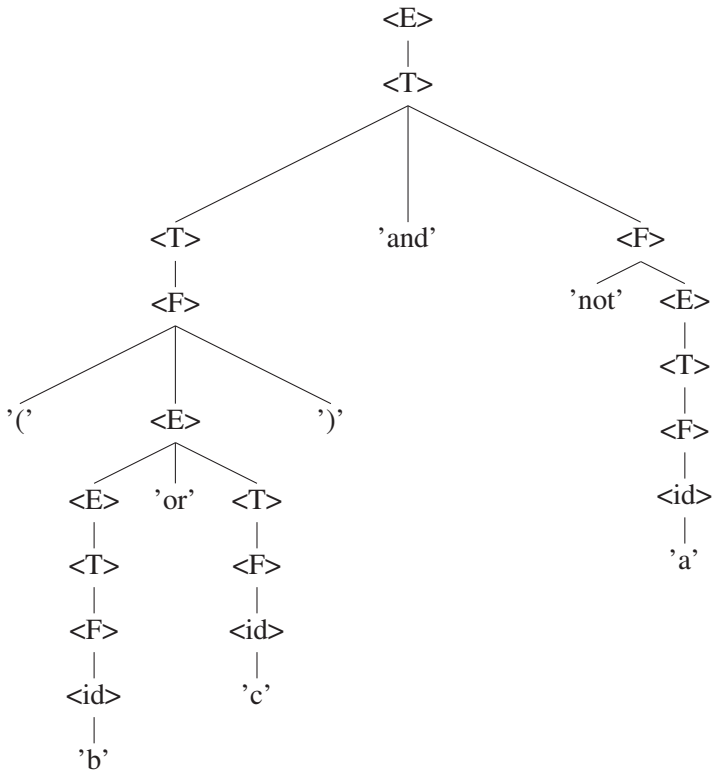
```
1  <E> ::= <E> 'or' <T> | <T>
2  <T> ::= <T> 'and' <F> | <F>
3  <F> ::= 'not' <E> | '(' <E> ')' | <id>
4  <id> ::= 'a' | 'b' | 'c'
```

   Construye los árboles de análisis sintáctico de las siguientes cadenas:

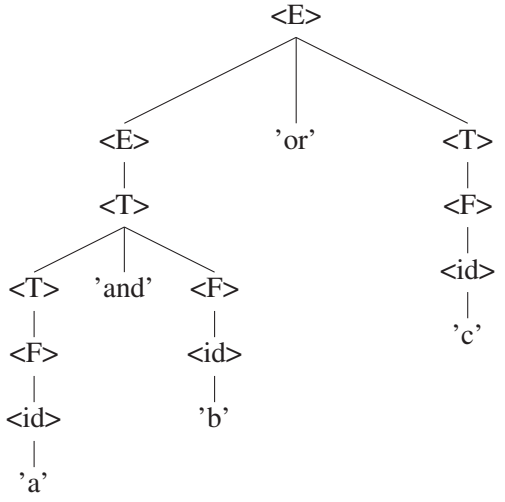
   (a) $('b'or'c')'and'\ 'not'\ 'a'$

   (b) $'a'and'b'or'c'$

(a)



(b)



5. Dada la siguiente gramática BNF, indica si esta gramática es ambigua. Justifícalo.

```
1 <S> ::= <B> | <C> <C> |  ε
2 <B> ::= 'a' <A>
3 <A> ::= <S> 'a'
4 <C> ::= 'a' <S> 'a'
```

La gramática es ambigua. La cadena *aaaa* se puede derivar de varias formas diferentes. Las siguientes son dos derivaciones posibles para esta cadena:

```
<S> => <B> => 'a' A => 'a' <S> 'a' => 'a' <B> 'a'
  => 'a' 'a' <A> 'a' => 'a' 'a' <S> 'a' 'a'
  => 'a''a''a''a'
```

```
<S> => <C> <C> => 'a' <S> 'a' <C>
  => 'a' <S> 'a' 'a' <S> 'a' => 'a' 'a' 'a' <S> 'a'
  => 'a' 'a' 'a' 'a'
```

6. Dado el siguiente código en C, ¿cuál sería la salida por pantalla?

```
int errorCode;
char *message;

void log() {
        printf("Found error %d: %s\n", errorCode, message);
}

void checkPositive(int *array, int length) {
        int errorCode = 0;
        char *message;

        int i;
        for(i = 0; i < length; i++) {
                if(array[i] < 0) {
                        errorCode = 20;
                }
        }
        if(errorCode != 0) {
                message = "Check positive failed";
                log();
        }
}

int main(void) {

        errorCode = 10;
        message = "File not found";
        log();

        int test[5] = {1,2,3,4,-1};
```

```
31          checkPositive(test, 5);
32
33          return EXIT_SUCCESS;
34 }
```

La línea 1 es generada en la llamada a `log` en la línea 28 del programa. En la llamada a log se utilizan las variables `errorCode` y `message` definidas en las líneas 1 y 2, respectivamente, y cuyos valores han sido asignados en las líneas 26 y 27.

```
1 Found error 10: File not found
2 Found error 10: File not found
```

La línea 2 de la salida por pantalla es generada por la llamada a `log` dentro de la función `checkPositive` en la línea 20. La función `log` accede a las variables definidas en las líneas 1 y 2, cuyos valores no han variado, por tanto se imprimen exactamente los mismos valores. Las variables `errorCode` y `message` definidas en la función `checkPositive` en las líneas 9 y 10 no son accesibles desde `log`, dado que en C el ámbito es estático, y por tanto, las variables definidas dentro de la función `checkPositive` sólo son accesibles desde dentro de esta función, y su ámbito se pierde al salir de ésta o pasar el control a otra función (como en el caso de `log`).

7. ¿Cuál sería la salida por pantalla del código del ejercicio anterior si C tuviera ámbito dinámico?

```
1 Found error 10: File not found
2 Found error 20: Check positive failed
```

En el caso de la línea 1, los nombres de las variables `errorCode` y `message` utilizados en la llamada a `log` de la línea 28 se corresponden con las definiciones de las líneas 1 y 2.

Sin embargo, en el caso de la línea 2, si consideramos ámbito dinámico, en las la llamada a `log` de la línea 20 se habrían enlazado las nombres de las variables `errorCode` y `message` con las definiciones locales de las líneas 9 y 10. En el momento de producirse la llamada a `log` estas variables tienen los valores 20 y "`Check positive failed`", respectivamente. De ahí el resultado.

1.9 Ejercicios propuestos

1. JSON (*JavaScript Object Notation* - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos,

mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del lenguaje de programación JavaScript. Se puede encontrar información sobre este formato en la página web http://www.json.org. En este ejercicio se considerará un subconjunto de JSON denominado SimpleJSON que sólo permite letras, números y los símbolos { } : . , [ ].

El lenguaje SimpleJSON tiene las siguientes características:

- Un objeto es una lista de pares nombre-valor. Un objeto comienza con { (llave de apertura) y termina con } (llave de cierre). Cada nombre se repersenta con un string y está seguido por : (dos puntos) y un valor. Los pares nombre-valor están separados por , (coma). Es posible que un objeto no tenga ningún par nombre-valor.
- Un fichero está formado por un único objeto.
- Un array es una colección de valores. Un array comienza por [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por , (coma). Es posible que un array no tenga ningún valor.
- Un valor puede ser una cadena de caracteres, un número, `true`, `false`, `null`, o un objeto o un array.

A continuación se muestran algunos ejemplos en formato SimpleJSON:

```
{ "nombre" : "antonio",
  "apellido" : "gonzalez",
  "edad" : 34,
  "conduce" : true }
```

```
{ "valor1" : 0.254,
  "valor2" : 34,
  "arrayValores" : [ 3, -0.3, 3e+13, 1.1e33 ],
  "arrayObjetos" : [
        { "nombre" : "antonio", "apellido" : "gonzalez" },
        { "nombre" : "felipe", "apellido" : "perez" },
        { "nombre" : "fernando", "apellido" : "rubio" } ] }
```

```
{"menu": {
        "header": "svg viewer",
        "items": [
                {"id": "open"},
                {"id": "openNew", "label": "open new"},
                null,
                {"id": "zoomin", "label": "zoom in"},
                {"id": "zoomout", "label": "zoom out"},
```

```
                    {"id": "originalview", "label": "original
                        view"},
                    null,
                    {"id": "quality"},
                    {"id": "pause"},
                    {"id": "mute"}
            ]
}}
```

Se pide diseñar una gramática independiente del contexto en notación BNF que defina el lenguaje SimpleJSON. Cuando sea necesario indicar en la gramática que puede aparecer una cadena de caracteres se puede hacer uso del no terminal $< cadena >$. No es necesario definir el no terminal cadena. En el caso de los números se puede utilizar el no terminal $< numero >$.

2. Se pide escribir la gramática independiente del contexto del ejercicio anterior en notación EBNF.

3. Dibuja el árbol de derivación para los ficheros que se muestran a continuación:

```
{ "nombre" : "antonio",
  "apellido" : "gonzalez",
  "edad" : 34,
  "conduce" : true }
```

```
{ "nombre" : "antonio",
  "edad" : 34,
  "conduce" : true,
  "pareja" : null,
  "hijos" : [
        { "nombre": "juan", "edad" : 3 },
        { "nombre": "lucia", "edad" : 2 }]}
```

En el caso de que haya que derivar el no terminal $< cadena >$ o $< numero >$, se pondrá una línea desde el no terminal hasta la palabra o el número que represente, respectivamente. Por ejemplo:

<cadena>
|
'antonio'

4. Dada la siguiente gramática para un lenguaje de programación con manejo de excepciones, donde $< sent >$ representa cualquier sentencia del lenguaje, incluyendo la sentencia try... catch e $< id >$ representa un identificador válido:

```
<sent-try> ::= 'try' '{' <sent-list> '}' <catch-list>
<sent-list> ::= <sent> <sent-list> | <sent>
<sent> ::= <sent-try> | ...
<catch-list> ::= <catch> <catch-list> | <catch>
<catch> ::= 'catch' '('<id> <id>')' '{'<sent-list>'}'
```

Se pide dibujar los árboles de análisis sintáctico de los siguientes fragmentos de código:

```
try {
        int v1 = matriz[i][j];
        int v2 = matriz[j][i];
} catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Índice fuera de límites: " + e.
           getMessage());
}
```

```
try {
        FileReader fr = new FileReader("message.eml");
        readMessage(fr);
} catch (FileNotFoundException e) {
        // Un sistema de logging muy básico
        System.out.println("No se encuentra el fichero: " +
            e.getMessage());
} catch(IOException e) {
        System.out.println("Excepción no esperada: " + e.
           getMessage());
}
```

```
try {
        fr = new FileReader("config.ini");
} catch (FileNotFoundException e) {
        try {
                fr = new FileReader("/home/user/config.ini"
                   );
        } catch(FileNotFoundException e) {
                System.out.println("No se encuentra el
                   fichero de configuración");
        }
}
```

5. Reescribe la gramática BNF del ejercicio 4 en formato EBNF.

1.10 Notas bibliográficas

Para la elaboración de este capítulo se han consultado diferentes fuentes. Para la primera parte (sintaxis y semántica) se ha utilizado sobre todo el material del libro [1].
Para la parte de teoría de lenguajes, se utilizaron las fuentes [17, 21]. Estas fuentes profundizan mucho más en los tipos de datos y su implementación de lo que se ha profundizado aquí.