Compiladores

Curso: 2001/2002
Alumna: Laura M. Castro Souto
Profesores: Bernardino Arcay Varela
José Carlos Dafonte Vázquez

Índice general

Capítulo 1

Introducción

La asignatura de **Compiladores** se divide en dos grandes bloques: ***análisis*** y ***síntesis***[1], cada uno de los cuales se estructura en diferentes partes, que veremos:

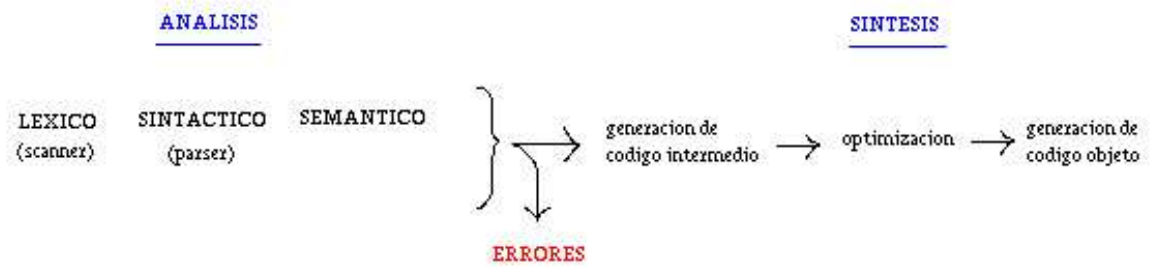


Figura 1.1: Esquema de las tareas de un compilador.

[1]Entre ambas tendrá lugar un parcial.

Capítulo 2

Lenguajes, Gramáticas, Autómatas y Traductores

2.1. Lenguajes y Gramáticas

2.1.1. Definiciones

Sea $\mathcal{A} = \{a, b, c, d, e\}$ un **alfabeto** (conjunto de caracteres). Llamamos:

√ **Tira de caracteres** o **palabra** a cualquier concatenación de caracteres del alfabeto: $\alpha = abc$.

√ **Distancia** o **longitud de una tira** al número de elementos (caracteres) de la tira: $\|\alpha\| = 3$. La **tira nula** es la tira de distancia 0 y suele representarse por ε o λ.

√ **Potencia de una tira** a la tira que resulta de yuxtaponer una consigo misma n veces: para $\alpha = ab$ se tiene $\alpha^0 = \lambda$, $\alpha^1 = ab$, $\lambda^2 = abab$, $\alpha^n = \underbrace{ab \ldots ab}_{n\text{veces}}$. Es decir:

$$\mathbf{A}^n = \mathbf{A}^{n-1} \cdot \mathbf{A}$$

con $\mathbf{A}^0 = \lambda$.

La potencia deriva del producto cartesiano: $A \cdot B = \{xy / x \in A \wedge y \in B\}$.

√ **Cierre transitivo** a: $\mathbf{A}^+ = \bigcup_{i>0} \mathbf{A}^i$

√ **Cierre reflexivo** a: $\mathbf{A}^* = \bigcup_{i\geqslant 0} \mathbf{A}^i$

De lo anterior se deduce que $\mathbf{A}^* = \mathbf{A}^+ \cup \{\lambda\}$.

DEMOSTRACIÓN:

$$\mathrm{A}^* = \bigcup_{i\geqslant 0} \mathrm{A}^i = \mathrm{A}^0 \cup \left(\bigcup_{i>0} \mathrm{A}^i \right) = \{\lambda\} \cup \left(\bigcup_{i>0} \mathrm{A}^i \right) = \mathrm{A}^+ \cup \{\lambda\}$$

2.1.2. Gramáticas. Tipos

Una **gramática** es una upla $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$, donde

N conjunto de símbolos no terminales

T conjunto de símbolos terminales

P conjunto de producciones

S **axioma** de la gramática

Según la forma de sus reglas de producción, Chomsky clasificó las gramáticas en cuatro tipos :

Gramáticas Tipo 0: con estructura de frase, se caracterizan porque los elementos de $\mathcal{P}$ son de la forma $\alpha \to \beta$ con $\alpha \in (\mathcal{N} \cup \mathcal{T})^+$ y $\beta \in (\mathcal{N} \cup \mathcal{T})^*$. Son las gramáticas *más generales* (admiten producciones de cualquier tipo) y se asocian con *autómatas bidireccionales*.

Gramáticas Tipo 1 o sensibles al contexto: sus producciones son de la forma $\alpha \mathrm{A} \beta \to \alpha\gamma\beta$ con $\alpha, \beta \in (\mathcal{N} \cup \mathcal{T})^*$; $\mathrm{A} \in \mathcal{N}$; $\gamma \in (\mathcal{N} \cup \mathcal{T})^+$. En estas gramáticas, la sustitución de un no terminal por una cadena depende de los símbolos que tenga alrededor (contexto).

Gramáticas Tipo 2 o de contexto libre: los elementos de $\mathcal{P}$ son de la forma $\mathrm{A} \to \alpha$ con $\mathrm{A} \in \mathcal{N}$ y $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$. Son las que se usan para lenguajes de programación.

Gramáticas Tipo 3 o regulares: sus producciones son siempre $\mathrm{A} \to \alpha\mathrm{B}$ o $\mathrm{A} \to \alpha$, con $\mathrm{A}, \mathrm{B} \in \mathcal{N}$ y $\alpha \in \mathcal{T}$. Se asocian con *autómatas finitos*.

Las más importantes para nosotros serán las gramáticas de tipo 3 o gramáticas regulares, ya que derivan conjuntos regulares que representaremos por medio de autómatas finitos y nos ayudarán a construir analizadores léxicos (*scanners*). Las gramáticas de tipo 2 o gramáticas independientes del contexto harán lo propio cuando nos ocupemos del análisis sintáctico (*parsers*).

Definición

Llamamos **derivación** $\alpha \Rightarrow \beta$ al conjunto de producciones que nos llevan de α a β:

$$\begin{array}{l} \alpha = \delta \mathrm{A} \mu \\ \beta = \delta \gamma \mu \end{array} \quad \text{y} \quad \exists \mathrm{A} \to \gamma \in \mathcal{P}$$

También se puede usar la notación $\alpha \Rightarrow^* \beta$ o bien $\alpha \Rightarrow^+ \beta$ para indicar que hay al menos una derivación.

2.1.3. Lenguajes

Dada una gramática $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$, definimos el **lenguaje generado por** $\mathcal{G}$, $\mathbf{L}(\mathcal{G})$, como:

$$\mathbf{L}(\mathcal{G}) = \{x / S \Rightarrow^* x \wedge x \in \mathcal{T}^*\}$$

Llamamos **formas sentenciales**, $\mathbf{D}(\mathcal{G})$, a:

$$\mathbf{D}(\mathcal{G}) = \{x / S \Rightarrow^* x \wedge x \in (\mathcal{N} \cup \mathcal{T})^*\}$$

Claramente, se verifica que $\mathbf{L}(\mathcal{G}) \subset \mathbf{D}(\mathcal{G})$.

DEMOSTRACIÓN:

Sea $x \in \mathrm{L}(\mathcal{G})$, entonces se cumple que $x \in \mathcal{T}^*$ y $S \Rightarrow^* x$. Equivalentemente,

$$\begin{array}{ll} (S \Rightarrow^* x) \wedge (x \in (\mathcal{T} \cup \{\lambda\})^*) & \Leftrightarrow \\ (S \Rightarrow^* x) \wedge \left(x \in \left(\mathcal{T} \cup \mathcal{N}^0\right)^*\right) & \end{array}$$

De aquí se deduce que

$$\begin{array}{ll} (S \Rightarrow^* x) \wedge (x \in (\mathcal{T} \cup \mathcal{N}^*)^*) & \Leftrightarrow \\ (S \Rightarrow^* x) \wedge (x \in (\mathcal{T}^* \cup \mathcal{N}^*)^*) & \Leftrightarrow \\ (S \Rightarrow^* x) \wedge (x \in (\mathcal{T} \cup \mathcal{N})^*) & \end{array}$$

Por tanto, $x \in \mathrm{D}(\mathcal{G}) \Rightarrow \mathrm{L}(\mathcal{G}) \subseteq \mathrm{D}(\mathcal{G})$.

La representación gráfica de las distintas derivaciones posibles para generar una tira se denomina **árbol sintáctico** :

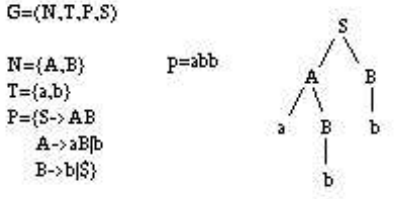


Figura 2.1: Ejemplo de árbol sintáctico.

Una gramática se dice **ambigua** si para alguna palabra del lenguaje que genera existe más de un árbol sintáctico. No existe el concepto de lenguaje ambiguo, lo que es ambiguo es la gramática.

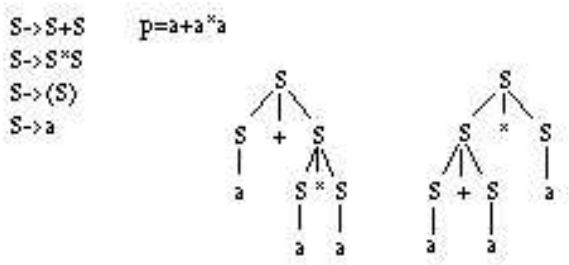


Figura 2.2: Ejemplo de gramática ambigua.

En este último caso, el compilador debería elegir uno de los dos árboles posibles. Ésta será una situación que deberá evitarse, puesto que ello ralentizaría la compilación, además de obligar a que el compilador conozca la precedencia de los operadores. La forma de solucionarlo es definir gramáticas más complejas (con más restricciones).

Otros problemas de las gramáticas de contexto libre que conducen a situaciones similares son la **recursividad por la izquierda** y la existencia de reglas con **factores repetidos por la izquierda**.

Cuando construyamos un compilador, buscaremos, pues, mecanismos deterministas y de poco coste computacional. Usaremos gramáticas de contexto libre y una serie de algoritmos que detectan las anomalías que pueden producir ambigüedad (es decir, que las transforman en gramáticas regulares):

1. **Generación del lenguaje vacío**, $\mathbf{L}(\mathcal{G}) = \varnothing$.
2. **Símbolos inútiles** (no accesibles o no terminables).
3. **Reglas unitarias**.
4. **Reglas** ε o λ.
5. **Ciclos**.

A continuación veremos una serie de algoritmos que realizan estos objetivos. Obtendremos de este modo **gramáticas limpias**, esto es, sin reglas-λ y sin ciclos, es decir, sin reglas unitarias (o *reglas cadena*) ni símbolos inútiles.

Su orden de aplicación sobre una gramática es importante, así que debe respetarse el que seguiremos.

Algoritmo que detecta $\mathbf{L}(\mathcal{G}) = \varnothing$

```
begin
  viejo=0
  nuevo={A / (A->t) e P, t e T*}
  vhile (nuevo<>viejo) do begin
    viejo=nuevo
    nuevo=viejo U {B / (B->a) e P, a e (T U viejo)*}
  end
  if (S e nuevo) then
    vacio=‘‘NO’’
  else
    vacio=‘‘SI’’
end
```

Algoritmo que elimina símbolos inútiles

Dado x, si $S \Rightarrow^* \alpha x \beta \Rightarrow^* t, t \in \mathcal{T}^*$, entonces x no es inútil (es accesible, puesto que $S \Rightarrow^* \alpha x$, y es terminable, ya que $x\beta \Rightarrow^* t$).

```
/* PRIMERA FASE, ELIMINACION DE NO TERMINABLES */

begin
  viejo=0
  nuevo={A / (A->t) e P, t e T*}
  while (nuevo<>viejo) do begin
    viejo=nuevo
    nuevo=viejo U {B / (B->a), a e (T U viejo)*}
  end
  N’=nuevo
  P’={p e P / p:(A->b), b e (T* U N’)}
end

/* SEGUNDA FASE, ELIMINACION DE NO DERIVABLES */

begin
  viejo={S}
  nuevo=viejo U {x / (S->axb) e P}
  while (nuevo<>viejo) do begin
    viejo=nuevo
    nuevo=viejo U {Y / (A->aYb) e P, A e viejo}
  end
  N’=N V nuevo
  T’=T V nuevo
end
```

Como ya hemos dicho, el orden en que aplicamos estos algoritmos es importante. Véase sino el siguiente ejemplo:

Figura 2.3: Relevancia en el orden de los algoritmos de simplificación.

Algoritmo que elimina reglas λ

Una **regla lambda** es aquélla que es del tipo A $\rightarrow \lambda$; en ese caso se dice también que A es *anulable*. Intentaremos evitar las gramáticas que contengan reglas de este tipo, eliminando sus λ-producciones si las tienen, salvo la **metanoción** (regla $S \rightarrow \lambda$), de la que no se puede prescindir porque λ forma parte del lenguaje.

```
begin
  viejo=0
  nuevo={A / (A->l) e P}
  while (nuevo<>viejo) do begin
    viejo=nuevo
    nuevo=viejo U {B / (B->a), a e viejo*}
  end
  N'=N-nuevo
  P'=Para cada prod. A->a0B0a1B1...anBn e P donde ai no
     anulable y Bi anulable, se añade a P'A->a0X0a1X1...anXn
     con Xi=Bi o l sin añadir A->l. Si S es anulable se añade
     S'->S|l y se añade S' a N
end
```

Algoritmo que elimina reglas unitarias

Una **regla unitaria** es de la forma $\mathbf{A} \rightarrow \mathbf{B}$ con $\mathbf{A}, \mathbf{B} \in \mathcal{N}$. No aportan nada y por ello deben eliminarse.

```
begin
  viejo=0
  nuevo=A;   /* debe hacerse para cada A de N */
  while (nuevo<>viejo) do begin
    viejo=nuevo
    nuevo=viejo U {C / (B->C) e P, B e viejo}
  end
  NA=nuevo
end
P'=Si (B->a) e P y no es regla unitaria, añadir (A->a) a P'
   para todas las A para las que B no esté en NA
```

Eliminación de ciclos

En los casos en los que la aplicación de los algoritmos anteriores no haya solucionado este problema, se aplican conversiones a **forma normal** (Chomsky, Greibach).

Recursividad

Dada una regla de producción $A \rightarrow \alpha A \beta$, hay varios tipos de recursividad posibles:

Por la izquierda Se da cuando $\alpha = \lambda$, $A \in \mathcal{N}$ y $\beta \in (\mathcal{N} \bigcup \mathcal{T})^*$. Es el tipo de recursividad no deseable y que querremos eliminar[1].

Por la derecha Se da cuando $\beta = \lambda$, $A \in \mathcal{N}$ y $\alpha \in (\mathcal{N} \bigcup \mathcal{T})^*$. Este tipo de recursividad es usual y no da problemas.

Autoimbricada Se da cuando $\alpha, \beta \neq \lambda$ y $A \in \mathcal{N}$. Tampoco da problemas.

Para *eliminar la recursividad por la izquierda* lo que haremos será:

```
Si tenemos A->Aa1|Aa2|...|Aap|b1|b2|...|bq
  podemos escribirlo como
    A->Aai   1<=i<=p
    A->bj    1<=j<=q
Para quitar la recursividad por la izquierda escribimos
  A->biA'       1<=i<=q, A' nuevo simbolo e Te
  A'->ajA'|l    1<=j<=p
```

Reglas con factores repetidos por la izquierda

Las reglas del tipo $A \rightarrow \alpha B | \alpha C$ también pueden darnos problemas a la hora de enfrentar un análisis, de modo que intentaremos reescribirlas. Hacerlo es tan sencillo como sustituirlas por: $A \rightarrow \alpha A'$ y $A' \rightarrow B|C$ con A' un nuevo símbolo no terminal que se añade a $\mathcal{T}_e$.

[1] La razón es que el análisis lo hacemos de izquierda a derecha.

2.2. Gramáticas y Autómatas

Se establece la siguiente *correspondencia*:

Gramáticas tipo 0	$\Leftrightarrow$	Máquinas de Turing
Gramáticas tipo 1	$\Leftrightarrow$	Autómatas bidireccionales
Gramáticas tipo 2	$\Leftrightarrow$	Autómatas con pila
Gramáticas tipo 3	$\Leftrightarrow$	Autómatas finitos

Cuadro 2.1: Correspondencia gramáticas-autómatas.

2.2.1. Autómatas

Autómatas finitos

Un **autómata finito** queda definido por $\mathbf{AF} = (\mathcal{Q}, \mathcal{T}_e, \delta, q_0, \mathcal{F})$, donde

Q conjunto de estados

$\mathbf{T}_e$ alfabeto de entrada

δ función de transición

$\mathbf{q}_0$ estado inicial

F conjunto de estados finales

y donde la función de transición δ es de la forma

$$\begin{array}{rcl} \delta : \mathcal{Q} \times \mathcal{T}_e & \longrightarrow & \mathrm{P}(\mathcal{Q}) \\ (q_i, x_j) & \longrightarrow & \{q_k\} \end{array}$$

Llamamos **configuración** de un autómata al par *(estado actual, conjunto de caracteres que quedan por leer)*: **(q,w)**.

CONFIGURACIÓN INICIAL: (q_0, t)
CONFIGURACIÓN FINAL u OBJETIVO: $(q_f, \lambda), \quad q_f \in \mathcal{F}$

Distinguimos dos tipos de autómatas finitos:

- **AFD** (*deterministas*)
- **AFND** (*no deterministas*)

Llamamos **transición** a $(\mathbf{q_i}, \mathbf{aw}) \vdash (\mathbf{q_k}, \mathbf{w}), \quad \mathbf{q_f} \in \delta(\mathbf{q_i}, \mathbf{a})$, esto es, a leer un carácter de la tira (cinta) de entrada y modificar el estado interno de la máquina. Se puede usar la notación $\vdash^*$ para indicar una transición con n pasos, $n \geqslant 0$, o bien $\vdash^+$ para indicar $n > 0$.

El **lenguaje generado** por el AF se define:

$$\mathbf{L}(\mathbf{AF}) = \{t/t \in \mathcal{T}_e^*, (q_0, t) \vdash^* (q_i, \lambda), q_i \in \mathcal{F}\}$$

En cuanto a la **representación gráfica**:

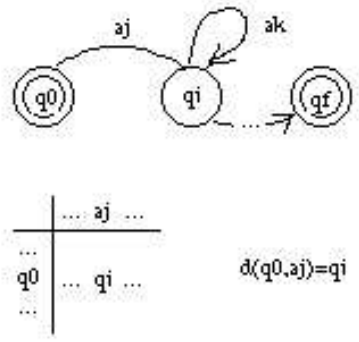


Figura 2.4: Representaciones de un autómata.

Autómatas con pila

Un **autómata con pila** se define $\mathbf{AP} = (\mathcal{Q}, \mathcal{T}_e, \mathcal{T}_p, \delta, q_0, z_0, \mathcal{F})$, con

- $\mathbf{Q}$ conjunto de estados
- $\mathbf{T}_e$ alfabeto de entrada[2]
- $\mathbf{T}_p$ alfabeto de la pila
- δ función de transición
- $\mathbf{q}_0$ estado inicial
- $\mathbf{z}_0$ configuración inicial de la pila
- $\mathbf{F}$ estados finales

donde la función de transición es

$$\begin{array}{rcl} \delta : \mathcal{Q} \times (\mathcal{T}_e \bigcup \{\lambda\}) \times \mathcal{T}_p & \longrightarrow & \mathrm{P}(\mathcal{Q} \times \mathcal{T}_p^*) \\ (q_i, a_j, \beta) & \longrightarrow & \{(q_k, \beta')\} \end{array}$$

[2]Equivale al conjunto de terminales en las gramáticas.

Se incluye incluye λ en el conjunto origen de la función para poder hacer transiciones sin leer, simplemente con el contenido de la pila, y de nuevo el conjunto destino es un *partes de*, contemplando la posibilidad de que sea determinista (**APD**) o no determinista (**APND**).

La **configuración**: $(\mathbf{q}, \mathbf{w}, \alpha)$ con $q \in \mathcal{Q}$ (estado actual), $w \in \mathcal{T}_e{}^*$ (lo que queda por leer) y $\alpha \in \mathcal{T}_p{}^*$ (contenido actual de la pila).

CONFIGURACIÓN INICIAL: (q_0, t, z_0)
CONFIGURACIÓN FINAL u OBJETIVO: $(q_f, \lambda, \alpha), \quad q_f \in \mathcal{F}$[3]

Definimos **movimiento** o **transición**: $(\mathbf{q_i}, \mathbf{aw}, \mathbf{z}\alpha) \vdash (\mathbf{q_k}, \mathbf{w}, \beta\alpha)$; también se admite la notación $\vdash^*$ y $\vdash^+$.

El **lenguaje** que determina un AP:

$$\mathbf{L}\,(\mathrm{AP}) = \{t/t \in \mathcal{T}_e{}^*, (q_0, t, z_0) \vdash^* (q_f, \lambda, \alpha)\,, q_f \in \mathcal{F}\}$$

Otra definición puede ser:

$$\mathbf{L}\,(\mathrm{AP}) = \{t/t \in \mathcal{T}_e{}^*, (q_0, t, z_0) \vdash^* (q_f, \lambda, \lambda)\,, q_f \in \mathcal{F}\}$$

Los que cumplen esta última restricción añadida se denominan **autómatas reconocedores por agotamiento de pila**.

2.3. Traductores

Traductores finitos

Un **traductor finito** se define $\mathbf{TF} = (\mathcal{Q}, \mathcal{T}_e, \mathcal{T}_s, \delta, q_0, \mathcal{F})$, con:

- **Q** conjunto de estados
- $\mathbf{T}_e$ alfabeto de entrada
- $\mathbf{T}_s$ alfabeto de salida
- δ función de transición
- $\mathbf{q}_0$ estado inicial
- **F** conjunto de estados finales

donde la función de transición es

$$\begin{array}{rcl} \delta : \mathcal{Q} \times \{\mathcal{T}_e \bigcup \{\lambda\}\} & \longrightarrow & \mathrm{P}\,(\mathcal{Q} \times \mathcal{T}_s{}^*) \\ (q_i, a_j) & \longrightarrow & \{(q_j, \beta)\} \end{array}$$

[3]La pila puede quedar con caracteres espúreos.

La **configuración** (estado del autómata en un punto determinado de la ejecución) se representa $(\mathbf{q}, \mathbf{t}, \mathbf{s})$ con $q \in \mathcal{Q}$ (estado actual), $t \in \mathcal{T}_e{}^*$ (lo que queda por leer) y $s \in \mathcal{T}_s{}^*$ (lo que ha salido hasta el momento).

Un **movimiento** es $(\mathbf{q}, \mathbf{aw}, \mathbf{s}) \vdash (\mathbf{q'}, \mathbf{w}, \mathbf{sz})$ cuando $\delta(q, a) = (q', z)$, con $q, q' \in \mathcal{Q}$, $w \in \mathcal{T}_e{}^*$ y $z \in \mathcal{T}_s{}^*$.

Y llamamos **conjunto de traducción** a:

$$\mathbf{Tr}(\mathrm{TF}) = \{(t, s); t \in \mathcal{T}_e{}^*, s \in \mathcal{T}_s{}^* / (q_0, t, \lambda) \vdash^* (q_t, \lambda, s), q_t \in \mathcal{F}\}$$

Traductores con pila

Un **traductor con pila** es generado por un APD (del mismo modo que un AF genera un TF) y queda definido por $\mathbf{TP} = (\mathcal{Q}, \mathcal{T}_e, \mathcal{T}_s, \mathcal{T}_p, \delta, q_0, z_0, \mathcal{F})$, con

- $\mathbf{Q}$ conjunto de estados
- $\mathbf{T}_e$ alfabeto de entrada
- $\mathbf{T}_s$ alfabeto de salida
- $\mathbf{T}_p$ alfabeto de pila
- δ función de transición
- $\mathbf{q}_0$ estado inicial
- $\mathbf{z}_0$ configuración inicial de la pila
- $\mathbf{F}$ conjunto de estados finales

donde la función de transición es

$$\begin{array}{rcl} \delta : \mathcal{Q} \times (\mathcal{T}_e \bigcup \{\lambda\}) \times \mathcal{T}_p & \longrightarrow & \mathrm{P}(\mathcal{Q} \times \mathcal{T}_p{}^* \times \mathcal{T}_s{}^*) \\ (q_j, \alpha_i, \beta_k) & \longrightarrow & \{(q_j, \beta_k, s_p)\} \end{array}$$

Su **configuración**: $(\mathbf{q}, \mathbf{t}, \alpha, \mathbf{s})$ con $q \in \mathcal{Q}$ (estado actual), $t \in \mathcal{T}_e{}^*$ (lo que queda por leer de la tira de entrada), $\alpha \in \mathcal{T}_p{}^*$ (contenido de la pila) y $s \in \mathcal{T}_s{}^*$ (lo que ha salido hasta ahora).

CONFIGURACIÓN INICIAL: (q_0, t, z_0, λ)
CONFIGURACIÓN FINAL u OBJETIVO: $(q_f, \lambda, z_0, s), \quad q_f \in \mathcal{F}$

Un **movimiento** (transición entre dos configuraciones): $(\mathbf{q_i}, \mathbf{ax}, \mathbf{Z}\gamma, \mathbf{y}) \vdash (\mathbf{q_j}, \mathbf{x}, \alpha\gamma, \mathbf{yz})$ si $(q_j, \alpha, z) \in (q_i, a, Z)$[4], admitiéndose también la notación $\vdash^*$ y $\vdash^+$.

No tiene por qué haber relación en tamaño entre la entrada y el string de salida.

[4]En lugar de $\in$ se pondría un $=$ en el caso determinista.

Conjunto de traducción:

$$\mathbf{Tr}\,(\mathbf{TP}) = \{(t,s)\,((q_0,t,z_0,\lambda) \vdash^* (q_f,\lambda,\alpha,s) \quad \text{sii} \quad q_f \in \mathcal{F}\}$$

Por **agotamiento de pila**:

$$\mathbf{Tr}\,(\mathbf{TP}) = \{(t,s)\,((q_0,t,z_0,\lambda) \vdash^* (q_f,\lambda,\lambda,s) \quad \text{sii} \quad q_f \in \mathcal{F}\}$$

2.3.1. Esquemas de traducción

Los **esquemas de traducción** son una notación. Tienen la siguiente estructura: EDT $= (\mathcal{N}, \mathcal{T}_\mathbf{e}, \mathcal{T}_\mathbf{s}, \mathcal{R}, \mathcal{S})$, con

N conjunto de símbolos no terminales

T$_e$ alfabeto de entrada

T$_s$ alfabeto de salida

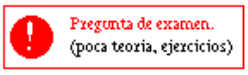
R reglas de traducción

S axioma

donde las *reglas de traducción* son de la forma

$$\mathcal{R} : \mathrm{A} \longrightarrow \alpha\beta \quad \text{con} \quad \begin{cases} \mathrm{A} \in \mathcal{N} \\ \alpha \in (\mathcal{N} \bigcup \mathcal{T}_e)^* \\ \beta \in (\mathcal{N} \bigcup \mathcal{T}_s)^* \end{cases}$$

Los esquemas de traducción son una extensión del concepto de gramática. Cumplen que $\mathcal{N} \bigcap (\mathcal{T}_e \bigcup \mathcal{T}_s) = \varnothing$.



Formas de traducción

Sean un par de tiras de caracteres (t,s) con $t \in \mathcal{T}_e^*$ y $s \in \mathcal{T}_s^*$; entonces:

1. $(\mathbf{S},\mathbf{S})$ es una **forma de traducción**, donde S es el axioma
2. Si $(\gamma\mathbf{A}\mathbf{w}, \rho\mathbf{A}\psi)$ es una forma de traducción y existe $A \to \alpha,\beta \in \mathcal{R}$, entonces $(\gamma\mathbf{A}\mathbf{w}, \rho\mathbf{A}\psi) \to (\gamma\alpha\mathbf{w}, \rho\beta\psi)$ es una forma de traducción. Puede usarse la notación $\to^*$ o $\to^+$.

El **conjunto de traducción de un EDT** se escribe:

$$\mathbf{Tr}\,(\mathrm{EDT}) = \{(t,s) \,/\, (S,S) \to^* (t,s)\,, t \in \mathcal{T}_e^*, s \in \mathcal{T}_s^*\}$$

Capítulo 3

Análisis Léxico. Scanners

Un **analizador léxico** o **scanner** es una entidad cuya tarea consiste en:

Figura 3.1: Objetivo de un *scanner*.

Ahora bien, los **scanners** no sólo hacen esto, sino que, por ejemplo, *reformatean texto* (eliminan espacios y líneas en blanco, comentarios, números de sentencia, números de línea, etc.) e incluso llevan a cabo *tareas semánticas* mediante el manejo de tablas como las que ya conocemos para la representación de autómatas. El problema que hay en el manejo de las tablas es un problema de memoria debido al espacio que ocupan. Para paliar esto, se usan técnicas de hashing y algoritmos como el de la *tabla compacta*, que intenta guardar sólo las casillas con información y no las vacías[1].

Los **analizadores léxicos** identifican con los **AFD** y, por tanto, con las **gramáticas regulares**. Se puden expresar o describir por medio de una **expresión regular**.

Existen diferentes formas de construir un **analizador léxico** o **scanner** (ver figura 3.2, página 22).

El primer método, señalado en la mencionada figura como *método manual* y que veremos en ejercicios, es fácil, pero también laborioso cuando la gramática se complica mínimamente. El tercer método, que veremos en prácticas, es sencillo y cómodo de utilizar.

A continuación, nos centraremos en el segundo de los métodos, el que será exigido en ejercicios en el examen de la asignatura.

[1]Esto puede repercutir en la complicación de la parte semántica.

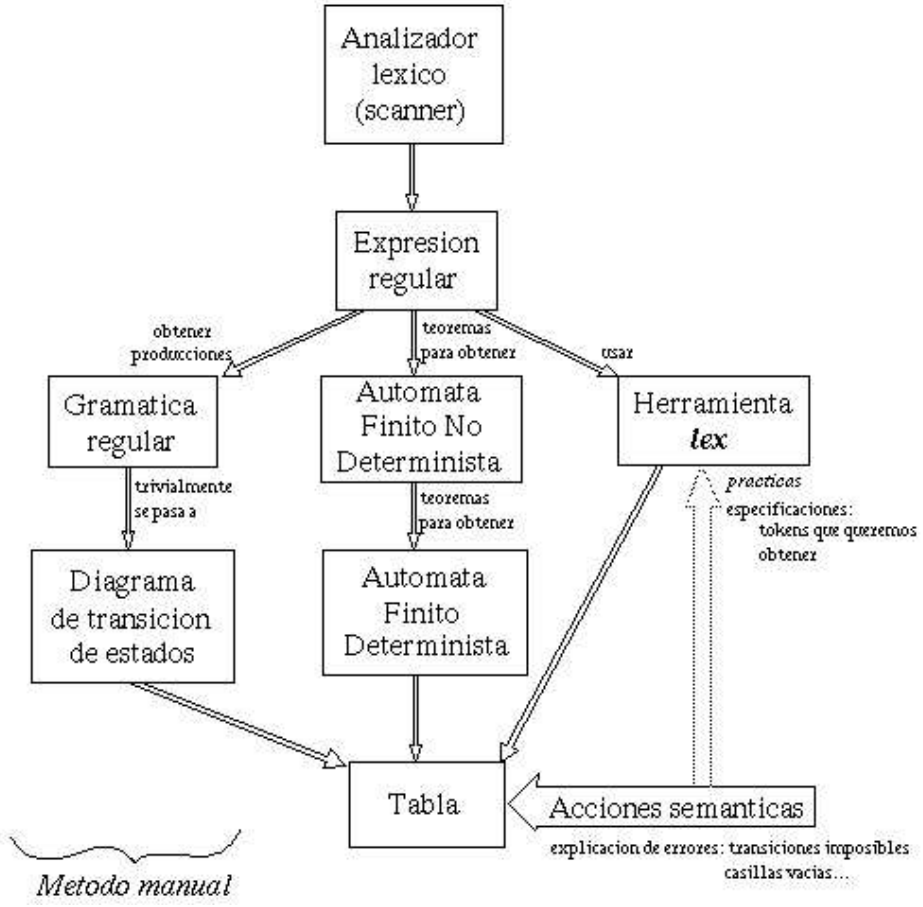


Figura 3.2: Maneras de construir un *scanner*.

3.1. Construcción de scanners mediante AFs

3.1.1. Expresiones Regulares. Propiedades

En primer lugar, haremos un recordatorio de las propiedades de las *expresiones regulares*, ya conocidas por nosotros:

Dadas s, t dos *expresiones regulares*:

Distributiva

$$s(r|t) = sr|st$$
$$(s|r)t = st|rt$$

Idempotencia

$$s^{**} = s^*$$

Conmutativa

$$r|t = t|r$$

Asociativa

$$(sr)t = s(rt)$$

Elemento Neutro

$$\varepsilon r = r\varepsilon = r$$
$$(r|\varepsilon)^* = r^*$$

3.1.2. Algoritmo (método) de Thompson

El algoritmo que describiremos en esta sección permite el paso de una expresión regular a un AFDN. Es un método gráfico que se basa en la concatenación de *cajas* (componentes) a partir de la definición de elementos básicos de dichas *cajas*. Dichos elementos son los que se muestran en la figura 3.3 (página 24); para ejemplos, véanse los ejercicios del tema.

3.1.3. Transformación de un AFND con λ-transiciones en AFD

Antes de ver los pasos que hemos de seguir para llevar a cabo esta transformación, necesitamos manejar algunos conceptos:

- Si s es un estado del AFND, se define su **cerradura**:

$$\mathbf{Cerradura - \Sigma\,(s)} = \left\{s_k \in \text{AFND}/s \rightarrow^{\lambda} s_k\right\}$$

  esto es, el conjunto de estados a los que se puede llegar desde s sólo mediante λ-*transiciones*.

- Si T es un conjunto de estados del AFND, se define a su vez su **cerradura**:

$$\mathbf{Cerradura - \Sigma\,(T)} = \left\{s_k \in \text{AFND}/\exists s_j \in T, s_j \rightarrow^{\lambda} s_k\right\}$$

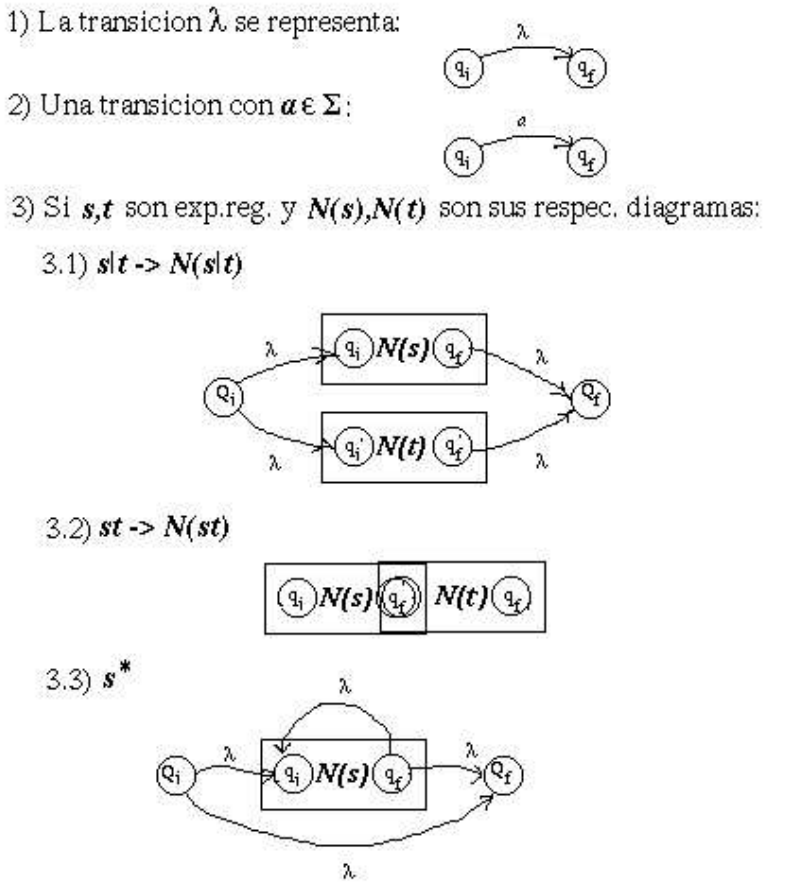


Figura 3.3: Componentes básicos del método de Thompson.

- Si T es un conjunto de estados del AFND y a es uno de sus símbolos terminales, se define una **función de movimiento**:

$$\mathbf{Mueve}\,(\mathbf{T},\mathbf{a}) = \{s_k \in \text{AFND}/\exists sj \in T, s_j \rightarrow^a s_k\}$$

A grandes rasgos, los pasos que se siguen son los siguientes:

1. Se calcula un nuevo estado A, la cerradura-Σ del estado inicial del AFND.
2. Se calcula la función de movimiento para A y todo a_i símbolo terminal del AFND.
3. Se calcula la cerradura-Σ del resultado del paso anterior, para todos los terminales, originándose nuevos estados $B, C, \ldots$
4. Calculamos la función de movimiento para todo a_i símbolo terminal del AFND y los diferentes estados generados en el paso anterior.

El proceso continúa y termina cuando ya no se generan más nuevos estados. El algoritmo se formalizaría:

```
calcular cerradura-E(0)=Estado A
incluir A en NuevosEstados /* estados del AFD */
mientras (no estan todos los T de NuevosEstados marcados) hacer
  marcar T
  para cada a_i e T_e hacer
    U=cerradura-E(Mueve(T,a_i))
    si (U no esta en el conjunto T)
      se añade U a T
    transicion[T,a_i]=U
  fin-para
fin-mientras
```

Capítulo 4

Análisis Sintáctico. Parsers

El **análisis sintáctico** consiste en encontrar u obtener el *árbol de derivación* que tenga como raíz el axioma de una gramática dada.

4.1. Tipos de *parsing*

Hay diferentes **tipos de *parsing***, como se indica en la siguiente figura:

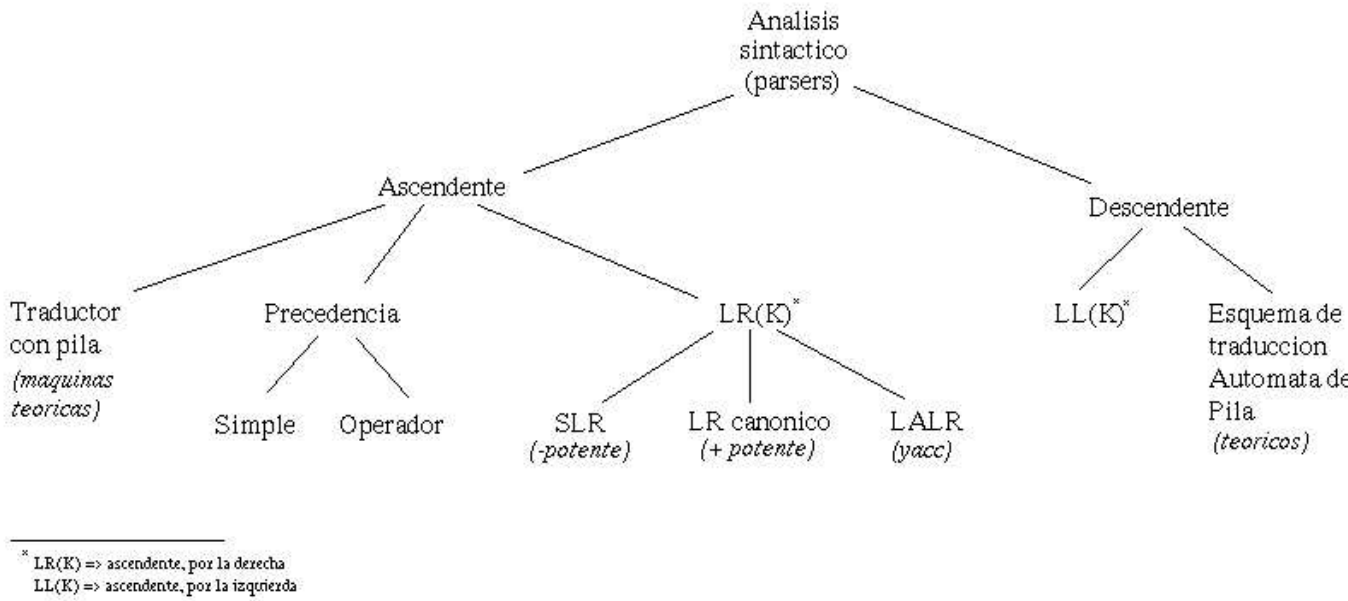


Figura 4.1: Tipos de *parsing*.

Otra clasificación:

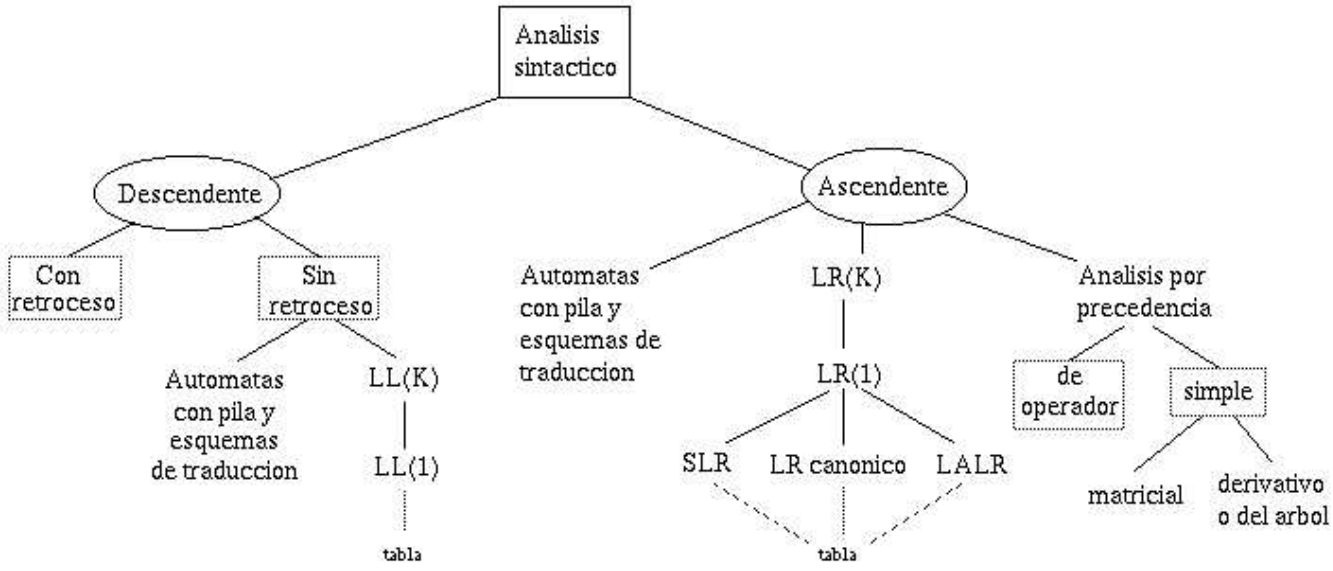


Figura 4.2: Tipos de *parsing*, otra clasificación.

El **parsing descendente** intenta construir el árbol sintáctico partiendo del *axioma* (raíz) desplegando hojas hasta llegar a la frase que tengamos. Por su parte, el **parsing ascendente** parte de los *tokens* (hojas, palabras que ha detectado el *scanner*) para intentar llegar al axioma (raíz).

La gran mayoría de los compiladores modernos son analizadores sintácticos ascendentes, que suelen basarse en *pivotes*[1]. De ellos, los más potentes son los tipo **LR(K)**, donde la **K** representa el número de tokens que el analizador necesita para saber qué pivote aplicar a cada paso. Por supuesto, lo más interesante son los LR(1), de los cuales se conocen los tres tipos que se mencionan: SLR (el más sencillo pero menos potente, impone más restricciones a sus gramáticas), LR canónico (el más potente pero también más complejo) y el LALR (un punto intermedio entre los dos anteriores y en el que se basan herramientas como `yacc` o `bison`). Como siempre, el objetivo será la consecución de la tabla.

Otro tipo de analizadores sintácticos ascendentes son los que utilizan la filosofía de **análisis por precedencia**. Históricamente fueron los que primero se usaron, aunque hoy no tengan demasiada utilidad.

Los analizadores sintácticos descendentes, por su parte, se clasifican en analizadores **con retroceso**, en los que se parte del axioma, se hace una exploración y si resulta fallida se vuelve hacia atrás (*backtracking*) y se reintenta hasta que se consigue o bien se agotan todas las posibilidades, y **sin retroceso**, en los que el algoritmo es más *predictivo* y conoce con certeza en cada caso qué rama del árbol expandir. Entre estos últimos se encuentran los **LL(K)**, donde de nuevo la **K** indica el número de tokens que el parser

[1]Conjunto de tokens que forma la parte derecha de una regla de traducción.

necesita conocer para saber por dónde ir; también aquí lo interesante es que K=1, de modo que estudiaremos los LL(1) y cómo obtener la tabla a partir de ellos.

Por último, mencionar que tanto entre los parsers descendentes como entre los ascendentes encontramos autómatas de pila y esquemas de traducción, pero son más bien máquinas teóricas que aplicaciones reales.

Para cada método de los que estudiaremos, veremos:

√ Condiciones que impone.

√ Definición de las gramáticas que admite[2].

√ Construcción de las tablas.

√ Funcionamiento.

4.2. Análisis por precedencia

4.2.1. Precedencia Simple

Es un método de análisis sintáctico ascendente.

Definición:

Entre dos tokens dados T_1 y T_2, se establece una **relación de precedencia** mediante la cual uno de ellos tendrá *mayor*, *igual* o *menor precedencia* que el otro:

$$\begin{array}{lcl} T_1 & > & T_2 \\ T_1 & = & T_2 \\ T_1 & < & T_2 \end{array}$$

Dada una gramática, una tabla de precedencia (donde se indiquen las precedencias relativas entre cada par de símbolos) y una expresión, todos los símbolos que queden entre dos signos "<" y ">" con signos de igualdad entre ellos se dice que forman un **pivote**. En la gramática ha de existir, entonces, alguna regla del tipo $A \rightarrow pivote$[3]. Renombrando trozos de la expresión de esta manera, se puede construir desde la base hacia la raíz el árbol de derivación correspondiente[4].

Las casillas en blanco de la tabla de precedencia son *errores* e indican que los dos símbolos no pueden relacionarse (no hay una relación de precendencia definida entre ellos) por medio de la gramática en cuestión.

[2]Que serán siempre *de contexto libre*, las adecuadas para manejar estructuras de frase y ya no palabras.
[3]Recuérdese que ahora los terminales y no terminales no representan símbolos, sino palabras, tokens.
[4]Véanse ejemplos del tema.

Método derivativo o del árbol

Es un método manual aplicable en caso de gramáticas sencillas, pero no recomendable en cuanto éstas se complican un poco, puesto que es sencillo olvidarse de algún caso. Ello es así porque consiste simplemente en intentar extraer la tabla de precedencia a partir simplemente de analizar "a ojo" las producciones de la gramática, hasta agotar todas las posibles derivaciones[5].

Método matricial

Para ver cómo obtener la tabla de precedencia mediante este método necesitamos previamente definir:

Relación de equivalencia (R) Cumple lo siguiente:

1. ***Reflexividad***, $x\mathrm{R}x, \forall x \in \mathrm{R}$.
2. ***Simetría***, $x\mathrm{R}y \Rightarrow y\mathrm{R}x, \forall x, y \in \mathrm{R}$.
3. ***Transitividad***, $x\mathrm{R}y, y\mathrm{R}z \Rightarrow x\mathrm{R}z, \forall x, y, z \in \mathrm{R}$.

Relación universal Se define:

$$\mathbf{U} = A \times A = \{(x, y) / x \in A, y \in A\}$$

Relación traspuesta (de una dada) Se representa R' o R^T:

$$\mathbf{R^T} = \{(y, x) / x\mathrm{R}y\}$$

Relación complementaria (a una relación dada):

$$\mathbf{R}^{\sim} = \{(x, y) /^{\sim} (x\mathrm{R}y)\}$$

Relaciones binarias Dadas dos relaciones R y S:

$$\begin{array}{rcl} \mathbf{R + S} & = & \{(x, y)/(x\mathrm{R}y) \vee (x\mathrm{S}y)\} \\ \mathbf{R \times S} & = & \{(x, y)/(x\mathrm{R}y) \wedge (x\mathrm{S}y)\} \\ \mathbf{R \cdot S} & = & \{(x, y)/\exists z, (x\mathrm{R}z) \wedge (z\mathrm{S}y)\} \end{array}$$

El método se describe de la siguiente manera:

$\surd$ Dada una gramática $\mathcal{G} = \{\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S}\}$:

1. Se dice que $\mathbf{x} < \mathbf{y}$ si y sólo si $A \to \alpha x B \beta \in \mathcal{P}$ y $B \Rightarrow^+ y\gamma$.
2. Se dice que $\mathbf{x} \pm \mathbf{y}$ si y sólo si $A \to \alpha x y \in \mathcal{P}$.
3. Se dice que $\mathbf{x} > \mathbf{y}$ si y sólo si $A \to A \alpha C y \beta \in \mathcal{P}$ y $C \Rightarrow^+ \gamma x$.

Dada una misma tabla de precedencia simple, pueden obtenerse diferentes gramáticas[6].

[5] Ver ejemplos del tema.

[6] Una pregunta de examen puede ser, dada una tabla de precedencia simple, intentar obtener su gramática.

Más definiciones:

Relación Primero Diremos que A primero X si y sólo si $\exists A \Rightarrow X\alpha\beta$.

Relación Primero$^+$ Diremos que A primero$^+ X$ si y sólo si $\exists A \Rightarrow^+ X\alpha\beta$.

Relación Último Diremos que A ultimo X si y sólo si $\exists A \Rightarrow \alpha\beta X$.

Relación Último$^+$ Diremos que A ultimo$^+ X$ si y sólo si $\exists A \Rightarrow^+ \alpha\beta X$.

Relación Dentro Diremos que A dentro X si y sólo si $\exists A \Rightarrow \alpha X\beta$.

Relación Dentro$^+$ Diremos que A dentro$^+ X$ si y sólo si $\exists A \Rightarrow^+ \alpha X\beta$.

Algoritmo de Warshall

```
/* permite hallar PRIMERO+ a partir de PRIMERO */

   B=A;   // copia de la matriz de partida
   I=1;
3: repeat
     if (B(I,J)=1) then
       for K=1 to N do
         if (A(J,K)=1) then
           A(I,K)=1;
   until todos los valores de J;
   I=I+1;
   if (I<=N) then
     goto 3;
   else
     stop;
```

Cuadro 4.1: Algoritmo de Warshall.

Las relaciones se obtienen:

$\surd$ $(\pm)$, observando las reglas de la gramática.

$\surd$ $(<) = (\pm)(\text{primero}^+)$.

$\surd$ $(>) = (\text{ultimo}^+)^{\mathrm{T}}(\pm)(I + \text{primero}^+)$, donde I es la matriz identidad (todo ceros con unos en la diagonal).

4.2.2. Precedencia Operador

Puede considerarse un caso particular de *precedencia simple* (podría, por tanto, resolverse utilizando matrices o el método del árbol); es un método de análisis sintáctico ascendente en el que se buscan relaciones entre los símbolos.

La diferencia es que, dado que la precedencia se define entre los operadores, no es necesario tener en cuenta los identificadores (que no tienen precedencia, son indistinguibles entre sí), no son necesarios para determinar si la expresión está bien escrita.

Construcción de la tabla

Los pasos son los siguientes:

i) Si el operador θ_1 tiene *mayor precedencia* que el operador θ_2, escribiremos

$$\theta_1 > \theta_2$$
$$\theta_2 < \theta_1$$

ii) Si θ_1 y θ_2 son operadores de igual precedencia, anotaremos

$$\left.\begin{array}{l}\theta_1 > \theta_2 \\ \theta_2 > \theta_1\end{array}\right\} \text{ si son asociativos por la izquierda}$$

$$\left.\begin{array}{l}\theta_1 < \theta_2 \\ \theta_2 < \theta_1\end{array}\right\} \text{ si son asociativos por la derecha}$$

iii) Además de todo ello, se cumple siempre que:

a) Los *identificadores* tienen *mayor* precedencia que cualquier operador.

b) Los *paréntesis* tienen *mayor* precedencia que cualquier otro símbolo.

c) El símbolo $ (indicador de principio/fin de cadena) tiene *menor* precedencia que cualquier otro símbolo.

Mayor precedencia	^ E	asociatividad por la derecha
	* /	asociatividad por la izquierda
Menor precedencia	+ -	asociatividad por la izquierda

Cuadro 4.2: Precedencia usual entre los operadores matemáticos.

4.3. Análisis sintáctico LL(K)

4.3.1. Análisis LL(1)

Es un **método predictivo**, **no recursivo**. Suele dar buenos resultados en aplicaciones pequeñas (shells, compiladores de pequeños lenguajes, etc) donde *precedencia simple* no resulta suficiente.

El método de funcionamiento es independiente del lenguaje. Al variar éste sólo variará la tabla. Como ya hemos comentado alguna vez, las casillas en blanco de la tabla suponen acciones semánticas, errores. Veremos en seguida su estructura y cómo se construye.

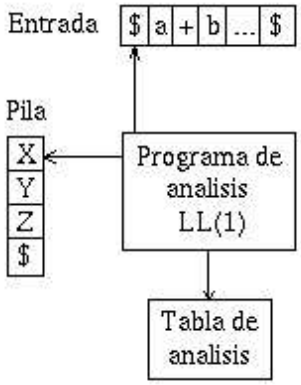


Figura 4.3: Estructura de un parser LL(1).

Funcionamiento

Los pasos que sigue un analizador sintáctico de este tipo son:

Se lee la entrada y la cima de la pila. Puede ocurrir:

1. Si $X = a = \$$ (donde X es el elemento de la cima de la pila, a el primer elemento del buffer de entrada y \$ el *carácter nulo*), se llega al FIN DEL ANÁLISIS.
2. Si $X = a \neq \$$, el parser saca X de la pila y mueve el apuntador de entrada al siguiente elemento del buffer.
3. Si X es *no terminal*, el parser consulta la tabla de análisis (que llamaremos M) en la posición (X, a). Hay dos posibilidades:
   a) Si $M(X, a) = \varnothing$, entonces se ha producido un ERROR.
   b) Si $M(X, a) = (X \rightarrow uwz)$, se continúa el análisis.

El pseudocódigo de la tabla 4.3.1 (página 34) realiza lo que acabamos de explicar[7].

Construcción de la tabla de análisis

Definiremos antes de nada dos operaciones:

1. **Primero**(X), que calcula el primer terminal de una cadena de caracteres, de la siguiente manera:
   a) Si $X \in \mathcal{T}$[8], entonces PRIMERO(X)=$\{X\}$.
   b) Si $X \rightarrow \varepsilon$, entonces $\varepsilon \in$ PRIMERO(X).

[7] Ver ejemplos en los ejercicios del tema.
[8] Hablamos siempre refiriéndonos a la gramática que acepta el parser, $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$.

```
entrada: cadena w y tabla de analisis sintactico M
salida: si w pertenece al lenguaje que acepta el parser,
        una derivacion por la izquierda de w, si no error
{
  apuntar æ al primer símbolo de w;
  do
  {
    sea X el simbolo de la cima de la pila y
        a el simbolo apuntado por æ;
    if (X es terminal o $)
      if (X=a)
        extraer X de la pila y avanzar æ;
      else
        return (error);
    else
      if (M[X,a]=X->Y1Y2...Yk)
      {
        extraer X de la pila;
        meter Yk,Yk-1...Y1 en la pila;
        emitir la produccion X->Y1Y2...Yk;
      }
      else
        return (error);
  } while (X no sea $); /* pila vacia */
}
```

Cuadro 4.3: Algoritmo de análisis LL(1).

   c) Si $X \in \mathcal{N}$ y $X \to Y_1 Y_2 \dots Y_n$, entonces $a \in$ PRIMERO(X) y $\exists i / a \in$ PRIMERO(Y_i) y $\varepsilon \in$ PRIMERO(Y_j) $\forall j < i$.

   Las tres reglas se aplican sobre la cadena en cada caso.

2. **Siguiente**(A) –con $A \in \mathcal{N}$–, que calcula, dado un no terminal A, el primer terminal que está a continuación de A:

   a) Se cumple que \$ $\in$ SIGUIENTE(S), donde S es el símbolo inicial, la metanoción, el axioma.

   b) Si $A \to \alpha B \beta$, entonces $\{$PRIMERO$(\beta) - \{\varepsilon\}\} \subset$ SIGUIENTE(B).

   c) Si $A \to \alpha B \beta$ y $\beta \to^* \varepsilon$ o $A \to \alpha B$, entonces SIGUIENTE$(A) \subset$ SIGUIENTE(B).

   De nuevo, esas tres reglas no son excluyentes.

Una vez que se han calculado los conjuntos PRIMERO y SIGUIENTE, las reglas para construir la tabla son:

1. Para toda producción $A \to \alpha \in \mathcal{P}$, aplicar los dos pasos siguientes.
2. $\forall a \in$ PRIMERO(α), $M[A, a] = A \to \alpha$.
3. Si $\varepsilon \in$ PRIMERO(α), entonces $M[A, b] = A \to \alpha \forall b \in$ SIGUIENTE(A).

Como ya ha sido comentado, las casillas en blanco supondrán errores y darán lugar a tratamientos semánticos, mientras que si en una casilla aparece más de una regla, la conclusión es que *la gramática no es LL(1)* (no es **determinista**).

Comprobación de si una gramática es LL(1)

Dada una gramática libre del contexto $\mathcal{G} = \{\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S}\}$, ¿cómo podemos saber si es una gramática LL(1) o no sin tener que construir la tabla? La cuestión es tan sencilla como realizar las siguientes comprobaciones $\forall A \to \alpha | \beta$:

1. PRIMERO$(\alpha) \cap$ PRIMERO$(\beta) = \varnothing$.
2. Sólo α ó β pueden derivar ε (nunca ambos).
3. Si $\beta \Rightarrow^* \varepsilon$, debe darse que PRIMERO$(\alpha) \cap$ SIGUIENTE$(A) = \varnothing$.

4.4. Análisis sintáctico LR(K)

En esta sección analizaremos una técnica eficiente de *análisis sintáctico ascendente* que se puede utilizar para analizar una clase más amplia de gramáticas independientes del contexto, denominada **análisis sintáctico LR(K)**. Este tipo de analizadores recibe este nombre por lo siguiente: la L hace referencia a qe el examen de la entrada se realiza de *izquierda a derecha*; la R se debe a que la reconstrucciones en orden inverso nos llevan a derivaciones por la derecha y la K, como ya imaginamos, es por el número de símbolos

de anticipación que se necesitan para una correcta toma de decisiones.

Después de estudiar el modo de operar de un analizador sintáctico LR, introduciremos *tres técnicas para construir la tabla de análisis sintáctico LR* para una gramática dada:

- √ **SLR** (LR *Sencillo*), el más fácil pero también el menos potente (puede que no consiga construir la tabla para algunas gramáticas).
- √ **LALR** (LR con *examen por anticipado*), un método "intermedio", que funciona con la mayoría de las gramáticas de los lenguajes de programación y que usan herramientas como `yacc`, que veremos.
- √ **LR canónico**, el más poderoso y costoso de los tres. Veremos modos de simplificarlo para obtener alguno de los dos anteriores.

4.4.1. Análisis LR(1)

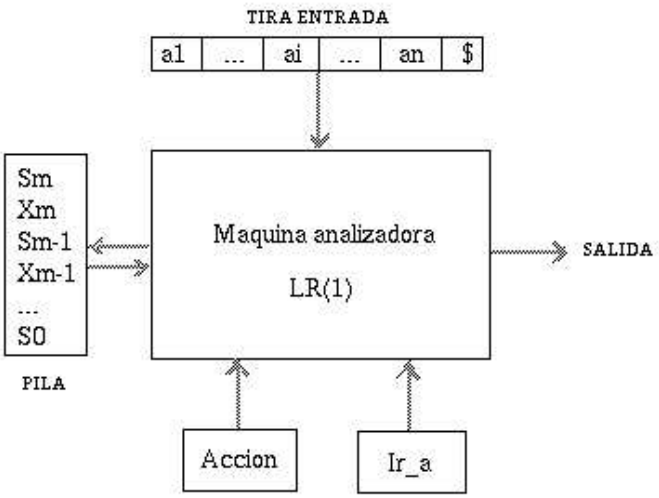


Figura 4.4: Modelo de un analizador sintáctico LR(1).

Este es el esquema de un analizador sintáctico LR(1): se apoya en una tabla de análisis sintáctico con dos partes, *acción* e *ir_a*. De un tipo de LR(1) a otro sólo cambian las tablas. El programa analizador lee caracteres del buffer de entrada de uno en uno, utiliza una pila para almacenar una cadena donde X_i son símbolos gramaticales y S_i son *estados*.

Cada símbolo de estado resume la información contenida debajo de él en la pila, y se usan la combinación del símbolo de estado en la cima de la pila y el símbolo en curso de la entrada para indexar la tabla de análisis sintáctico y determinar la decisión de desplazamiento a reducción del analizador.

La tabla de análisis sintáctico, como decimos, consta de dos partes: la función *acción*, que indica una acción del analizador (indica *qué hacer*), y la función *ir_a*, que indica las transiciones entre estados. La primera se usa siempre, en cada uno de los pasos, mientras que la segunda se usa sólo si tiene lugar una reducción.

Funcionamiento

1. Tomamos S_m de la cima de la pila y a_i de la entrada actual.
2. Se consulta la entrada $accion[S_m, a_i]$ de la tabla de acciones del analizador, pudiendo obtenerse uno de los siguientes valores:
   a) Desplazar S_p, donde S_p es un estado.
   b) Reducir por una producción gramatical $A \to \beta$. En este caso se consulta también la entrada $ir\_a[S_{m-r}, A]$ (donde r es la longitud de β) de la tabla de transiciones para obtener un nuevo estado S_q que se añade a la cima de la pila.
   c) Aceptar.
   d) Error.
3. Repetir hasta terminar la tira de entrada, aceptarla o llegar a un error.

Una **configuración** de un analizador sintáctico LR(1) es un par cuyo primer componente es el contenido de la pila y el segundo la entrada que aún está sin procesar:

$$(S_0X_1S_1X_2S_2 \dots X_mS_m, a_ia_{i+1} \dots a_n\$)$$

El siguiente movimiento del analizador se determina leyendo a_i, el símbolo de la entrada en curso, y S_m, el estado del tope de la pila, y consultado después la entrada $accion[S_m, a_i]$ de la tabla de acciones del analizador. Las configuraciones obtenidas después de cada uno de los cuatro tipos de movimiento son las siguientes:

- Si $accion[S_m, a_i]$ =desplazar S_p, el analizador ejecuta un movimiento de desplazamiento entrando en la configuración

  $$(S_0X_1S_1X_2S_2 \dots X_mS_ma_iS_p, a_{i+1} \dots a_n\S)$$

  (el analizador ha desplazado a la pila el símbolo de la entrada en curso y el siguiente estado, y a_{i+1} se convierte en el símbolo de entrada actual).

- Si $accion[S_m, a_i]$ =reducir $A \to \beta$, entonces el analizador ejecuta un movimiento de reducción, entrando en la configuración

  $$(S_0X_1S_1X_2S_2 \dots X_{m-r}S_{m-r}AS_q, a_ia_{i+1} \dots a_n\$)$$

  donde $S_q = ir\_a[S_{m-r}, A]$ y r es la longitud de β, el lado derecho de la producción (el analizador ha extraído $2r$ símbolos de la pila, r de estados y r de símbolos de la gramática, exponiendo el estado S_{m-r}, luego ha introducido A, el lado izquierdo de la producción, y S_q, la entrada de $ir\_a[S_{m-r}, A]$, en la pila; no se ha modificado la entrada en curso). La secuencia de símbolos gramaticales extraídos de la pila, $X_{m-r+1} \dots X_m$, concuerda siempre con β, el lado derecho de la producción con que se efectúa la reducción.

- Si $accion[S_m, a_i]$ =aceptar, el análisis sintáctico ha terminado con éxito.

- Si $accion[S_m, a_i]$ =error, el analizador sintáctico ha descubierto un error.

Construcción de tablas de análisis sintáctico SLR

Se llama **elemento de análisis sintáctico LR(0)** (o, simplemente, *elemento*) de una gramática $\mathcal{G}$ a una producción de dicha gramática que contiene un punto en alguna posición del lado derecho. Por tanto, la producción $A \to XYZ$ produce los cuatro elementos:

$$A \to .XYZ$$
$$A \to X.YZ$$
$$A \to XY.Z$$
$$A \to XYZ.$$

La producción $A \to \varepsilon$ sólo genera el elemento $A \to \cdot$.

Una serie de conjuntos de elementos LR(0) se denomina **colección canónica LR(0)** y proporciona la base para construir analizadores sintácticos SLR. Para construir la *colección canónica LR(0)* para una gramática se define una *gramática aumentada* y dos funciones, *cerradura* e *ir_a*:

Gramática aumentada Dada una gramática $\mathcal{G}$, la *gramática aumentada* $\mathcal{G}'$ resulta de añadir a $\mathcal{G}$ la regla $S' \to S$, donde S es el axioma de la gramática.

Cerradura Dado un subconjunto I de elementos LR(0):

* Todo elemento de I está en $cerradura(I)$.
* $\forall A \to \alpha \cdot B\beta \in cerradura(I)$, si $\exists B \to \gamma \in \mathcal{P}$, entonces $B \to .\gamma \in cerradura(I)$.
* Se aplica lo anterior hasta que $cerradura(I)$ no varía.

Intuitivamente, si $A \to \alpha \cdot B\beta$ está en $cerradura(I)$ indica que, en algún momento del proceso de análisis sintáctico, se cree posible ver a continuación una cadena derivable de B como entrada. Si $B \to \gamma$ es una producción, también se espera ver una subcadena derivable de γ en este punto. Por esta razón, se incluye $B \to \cdot\gamma$ en $cerradura(I)$.

Se puede calcular la función *cerradura* como se muestra en la tabla 4.4.1.

Obsérvese que si se añade una producción de B a la cerradura de I con el punto en el extremo izquierdo, entonces todas las producciones de B se añadirán de manera similar a la cerradura. De hecho, se pueden dividir todos los conjuntos de los elementos que interesan en dos clases:

1. *Elementos nucleares*, todos los elementos cuyos puntos no estén en el extremo izquierdo, y el elemento inicial $S' \to \cdot S$.
2. *Elementos no nucleares*, todos los elementos cuyos puntos están en el extremo izquierdo.

Operación ir_a La segunda función útil es $ir\_a(I, X)$, donde I es un conjunto de elementos y X es un símbolo de la gramática ($X \in (\mathcal{N} \cup \mathcal{T})$). Se define como la cerradura del conjunto de todos los elementos $A \to \alpha X \cdot \beta$ tales que $A \to \alpha \cdot X\beta$ esté en I.

```
funcion cerradura(I)
begin
  J=I;
  repeat
    for cada elemento A->a.Bb en J y cada produccion
        B->l de G tal que B->.l no este en J do
          añadir B->.l a J;
  until no se puedan añadir mas elementos a J;
  return J;
end
```

Cuadro 4.4: Algoritmo de cálculo de *cerradura*.

Definidos estos conceptos, podemos ver el **algoritmo para la obtención de la *colección canónica LR(0)*** de un conjunto de elementos LR(0) para una gramática aumentada $\mathcal{G}'$. Esta *colección* nos permitirá obtener el autómata a partir de la gramática.

```
procedimiento elementos_LR0 (G)
begin
  C={cerradura(S'->.S)};     /* gramática aumentada */
  repeat
    for cada conjunto de elementos I en C y cada simbolo
        gramatical X de G tal que ir_a(I,X) no esté vacío y
        no esté en C do
          añadir ir_a(I,X) a C;
  until no se puedan añadir más conjuntos de elementos a C;
end
```

Cuadro 4.5: Construcción de la colección canónica LR(0).

Diremos que un elemento $A \to \beta_1\beta_2$ es un **elemento válido** para un prefijo variable $\alpha\beta_1$ si existe una derivación $S' \Rightarrow \alpha A\omega \Rightarrow^* \alpha\beta_1\beta_2\omega$. En general, un *elemento* será *válido* para muchos prefijos.

Vistos estos conceptos, los **pasos de construcción de la tabla** son:

1. Dada una gramática G, se aumenta para producir G'.
2. Se construye la colección canónica LR(0):

$$C = \{I_0, I_1, \ldots, I_n\}$$

3. Los estados se obtienen a partir de los I_i (siendo I_k el estado k).

4. Las casillas se rellenan:
   a) Si $A \rightarrow \alpha.a\beta \in I_i$, e $Ir\_a(I_i, a) = I_j$, entonces se hace $accion[i, a] = desplazar \quad j$.
   b) Si $A \rightarrow \alpha. \in I_i$, entonces se hace $accion[i, a] = reducir \quad A \rightarrow \alpha \quad \forall a \in siguiente(A)$.
   c) Si $S' \rightarrow S. \in I_i$, entonces se hace $accion[i, \$] = aceptar$.

   Si las acciones anteriores generan acciones contradictorias (aparece más de una acción en alguna casilla de alguna de las tablas) se dice que la *gramática no es SLR.*
5. Si $Ir\_a(I_i, A) = I_j$, siendo A un no terminal, entonces se hace $Ir\_a[i, A] = j$.
6. Todas las entradas no definidas por las reglas anteriores son consideradas $error$.

Aunque no veremos los criterios de decisión, es importante saber que el método SLR no funciona para cualquier gramática. El problema es que, como hemos visto, SLR no tiene en cuenta más que aquélla parte de la cadena de entrada que ya ha leído para tomar decisiones, es decir, no es capaz de mirar un elemento más allá. A continuación veremos el método LR-canónico, que sí lo hace.

Construcción de tablas de análisis sintáctico LR-canónico

El **método LR-canónico** es un método de construcción de tablas de análisis sintáctico más general, más potente, pero también más costoso desde el punto de vista del diseño.

El *LR-canónico* añade información adicional al estado redefiniendo los elementos de análisis para incluir un símbolo terminal como segundo componente, de forma que pasan a ser de la forma $[A \rightarrow \alpha.\beta, a]$ en lugar de $[A \rightarrow \alpha.\beta]$ como en SLR. Este nuevo elemento que se tiene en cuenta para la predicción, recibe el nombre de **símbolo de anticipación**, y:

- no tiene efecto si $\beta \neq \varepsilon$
- si $\beta = \varepsilon$, el elemento $[A \rightarrow \alpha., a]$ pide una reducción por $A \rightarrow \alpha$ sólo si el siguiente símbolo de la entrada es a

De este modo, podemos tener varios casos, varias reducciones con diferentes no terminales a, algo que proporciona su potencia al LR-canónico, frente al SLR.

El método para construir la colección de conjuntos de elementos válidos ahora es fundamentalmente el mismo que la forma en que se construye la colección canónica de conjuntos de elementos del análisis sintáctico LR(0); sólo hay que modificar los dos procedimientos $Cerradura$ e Ir_a.

```
function cerradura(I)
begin
  repeat
    for cada elemento [A->s.Bb, a] en I,
        cada produccion B->r en G'
      y cada terminal b de Primero(ba) tal que
        [B->.r,b] no esta en I, do
          añadir [B->.r, b] a I;
  until no se puedan añadir mas elementos a I;
  return (I);
end
```

Cuadro 4.6: Algoritmo de cálculo de *cerradura* para LR-canónico.

```
function Ir_a(I,X)
begin
  J=conjunto de los elementos [A->sX.b,a] tal que
    [A->s.Xb,a] esta en I;
  return (cerradura(J));
end
```

Cuadro 4.7: Algoritmo de cálculo de *Ir_a* para LR-canónico.

```
procedure elementos(G')   /* gramatica aumentada */
begin
  C=cerradura([S'->.S,$]);
  repeat
    for cada conjunto de elementos I en C
      y cada simbolo X tal que Ir_a(I,X) no sea vacio
      y no este en C, do
        añadir Ir_a(I,X) a C;
  until no se puedan añadir mas conjuntos de elementos a C;
end
```

Cuadro 4.8: Construcción del conjunto de los elementos de análisis LR-canónico.

Una vez que tenemos el autómata finito (su diagrama de transición de estados), los pasos para la **construcción de la tabla de análisis sintáctico** son:

1. Dada una gramática G, se aumenta para producir G'.
2. Se construye $C = \{I_0, I_1, \ldots, I_n\}$, la colección de conjuntos de elementos LR(1) para G'.
3. El estado i se obtiene a partir de I_i.
4. Las casillas se rellenan:
   a) Si $[A \to \alpha.a\beta, b] \in I_i$ e $Ir\_a(I_i, a) = I_j$, entonces se hace $accion[i, a] = desplazar \quad j$, para todo terminal a..
   b) Si $[A \to \alpha., a] \in I_i$, entonces se hace $accion[i, a] = reducir \quad A \to \alpha$ (con $A \neq S'$).
   c) Si $[S' \to S., \$] \in I_i$, entonces se hace $accion[i, \$] = aceptar$.

   Si las reglas anteriores producen un conflicto, se dice que la gramática no es LR(1).
5. Si $Ir\_a(I_i, A) = I_j$, entonces se hace $Ir\_a[i, A] = j$.
6. Todas las entradas no definidas por las reglas anteriores se consideran $error$.

Construcción de tablas de análisis sintáctico LALR

Veremos el **método LALR** como una "consecuencia" del LR-canónico, como una simplificación. Es decir, dada una gramática, aplicado el método LR-canónico y obtenida la tabla de análisis sintáctico (o simplemente el autómata finito), construiremos a partir de ello un LALR realizando una *compresión de estados*.

El LR-canónico es un método de análisis muy potente, pero que produce muchos estados, de los cuales, en gran mayoría de ocasiones podemos hacer una *compresión* de la siguiente manera: analizando los estados del autómata generado por el método LR-canónico, comprobaremos que el **corazón** (*core*, producción) de los elementos LR(1) es el mismo en varios estados, generalmente dos (aunque pueden ser más), que sólo varían en el elemento de anticipación (*lookahead*). El método LALR lo único que hace es reunir esos pares (conjuntos) de estados en uno solo, agrupando los elementos de anticipación a la derecha.

Evidentemente, la fusión de los estados debe ir acompañada por la compatibilidad de transiciones entre los estados que se fusionan, es decir, para fusionar dos estados no sólo es necesario que el corazón de todas las reglas de ambos sean iguales (no vale tampoco que uno sea subconjunto del otro), sino que además estados fusionados deben transicionar a estados fusionados. Si ello no ocurriera así, diríamos que la gramática no es LALR.

La obtención de la **tabla de análisis sintáctico LALR** a partir del nuevo autómata simplificado (comprimido) es sencilla, consiste únicamente en traducir las modificaciones a las celdas de la misma. Si la gramática realmente es LALR, no debería haber problema alguno.

Capítulo 5

Análisis Semántico
Traducción dirigida por la sintaxis

En el presente tema veremos el último de los tipos de análisis que mencionamos en la introducción de la asignatura (figura 1.1, página 7): el **análisis semántico**.

Ya hemos visto el *análisis léxico* (scanners, que detectaban las palabras y comprobaban si estaban bien formadas), el *análisis sintáctico* (parsers, que estudiaban la corrección en la estructuras de las frases) y nos falta, por tanto, la parte del análisis que juzga el *sentido* de las construcciones que han sido previamente validadas por scanners y parsers.

La encarnación más simple de esta tarea es la **comprobación de tipos**. Veremos, pues, cómo se añaden reglas semánticas de este tipo a las estructuras que ya hemos estudiado (será, al fin y al cabo, añadir más información a las reglas que ya manejamos).

El mecanismo típico es trasladar las comprobaciones pertinentes a los nodos del árbol sintáctico que hemos aprendido a construir en el capítulo anterior (asignando un conjunto de rutinas a los mismos, que se ocuparán, entre otras cosas, de la *comprobación de tipos*, *gestión de memoria*, *generación de código objeto*...). Al situar en este punto la clave de la acción también será más fácil, como veremos, la **generación de código**. Además, la tarea se verá simplificada también porque usaremos la misma sintaxis para la semántica que para la generación de código.

En la figura 5.1 (página 44) se muestra un esquema que relaciona los temas principales que veremos en breve en el seno de la **traducción dirigida por la sintaxis**. Esta parte de la disciplina del análisis semántico se apoya en conceptos de grafos, particularmente de grafos acíclicos dirigidos, y es una alternativa al diseño más formal de esquemas de traducción. En *análisis de tipos* haremos referencia tanto a los tipos básicos (`integer`, `real`, `char`,...) como a los constructores de tipos, operadores que, trabajando con tipos básicos, construyen otros tipos (`records`, `arrays`, `pointers`,...). En cuanto a la *generación de código*, veremos cómo la definición de reglas en este caso estará íntimamente relacionado con la arquitectura subyacente y definiremos nuestra "máquina teórica", para trabajar sobre ella.

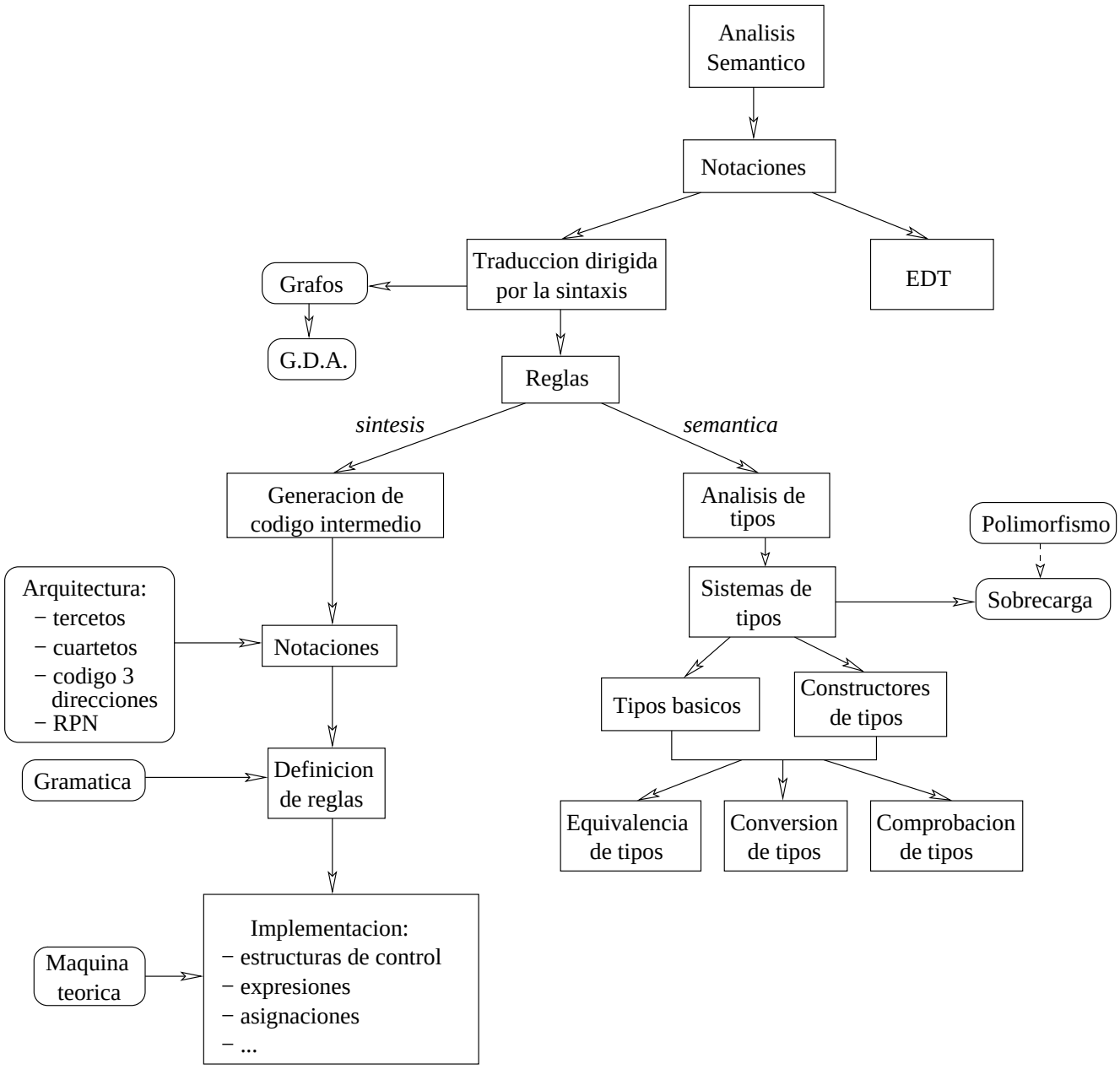


Figura 5.1: Esquema de análisis semántico.

5.1. Notación

Para describir **acciones semánticas** utilizaremos dos elementos:

- **reglas**
- **atributos**

de tal manera que, en la *traducción dirigida por la sintaxis*, dada una producción $A \to \alpha$ tendremos un conjunto de reglas semánticas $b = f(c_1, c_2, \ldots, c_n)$ donde c_i son los atributos.

Cada nodo de evaluación en un árbol sintáctico (nodos correspondientes a partes izquierdas de reglas de producción de la gramática) tendrá asociada una f (función de evaluación) que aplicará sobre sus propios c_i y sobre c_j relativos o pertenecientes a sus hijos directos. Este proceso, que se da típicamente en la evaluación ascendente de izquierda a derecha, se denomina **arrastre de información**.

5.2. Sistemas de Tipos

5.2.1. Tipos básicos

Un **sistema de tipos** es el conjunto de *expresiones de tipos* que tenemos a disposición en el lenguaje. A su vez, una **expresión de tipo** es, bien un *tipo básico*, o bien el resultado de aplicar un *constructor de tipos* a un tipo básico.

Nosotros consideraremos:

Tipos básicos los siguientes:

- √ `boolean`
- √ `char`
- √ `integer`
- √ `real`

como en la mayoría de los lenguajes, y además:

- √ `error_tipo`
- √ `void`

Constructores de tipos los siguientes:

- √ Matrices, que representaremos `array(I,T)` donde `I` es el tipo de los índices y `T` el de las variables que contendrá el array.
- √ Productos, que se definen: si T_1 y T_2 son expresiones de tipos, entonces $T_1 \times T_2$ es otra expresión de tipo y se denomina *producto*.
- √ Registros, consecuencia de la definición de productos, pues no son más que "productos con nombres".
- √ Apuntadores o punteros, que representaremos `pointer(T)` donde `T` es el tipo a que apuntan.
- √ Funciones, que representaremos $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_p$, donde $T_1 \times T_2 \times \ldots \times T_n$ es el *dominio* de la función (de modo que los argumentos de una función son un producto) y T_p el tipo del rango.

Además, claro está, pueden construirse "constructores de constructores" (punteros a punteros, punteros a funciones, arrays, de punteros,...). Esto dará lugar a los *problemas de equivalencia* que veremos más adelante.

Existen dos tipos de *atributos*:

- ▷ *heredados*, cuyo valor se calcula a partir de los valores de los atributos de los hermanos y del padre del nodo al que pertenecen
- ▷ *sintetizados*, cuyo valor se calcula a partir de los valores de los atributos de los hijos del nodo al que pertenecen

En traducción dirigida por la sintaxis usaremos únicamente atributos sintetizados, razón por la que en alguna bibliografía puede encontrarse la denominación *traducción (definición) dirigida por la sintaxis con atributos sintetizados.*

En adelante, usaremos la siguiente gramática como referencia en explicaciones, ejemplos y ejercicios:

```
P -> D;E
D -> D;D | id:T
T -> char | integer | array[num] of T | ^T
E -> literal | numero | id | E mod E | E[E] | ^E
S -> id:=E
S -> if E then S1
S -> while E do S1
S -> S1;S2
```

Cuadro 5.1: Gramática de referencia.

Veamos a continuación algunas reglas semánticas :

↪ La comprobación de tipos matriciales:

$$E \rightarrow E^1[E^2]$$

requerirá comprobar que E^2 es de tipo entero[1] y asignar al tipo de E (no terminal de arrastre de información) el tipo de los elementos del array:

```
E.tipo = if (E2.tipo=integer) and
            (E1.tipo=array[S,T]) then T
         else
            error_tipo;
```

[1]Supondremos que el lenguaje sólo permite que los índices de los arrays sean enteros.

↪ La comprobación de operaciones

$$E \rightarrow E^1 \ mod \ E^2$$

(donde *mod* juega el mismo papel que cualquier otra operación),

```
E.tipo = if (E1.tipo=integer) and
            (E2.tipo=integer) then integer
         else
            error_tipo;
```

↪ La comprobación de tipos en proposiciones del lenguaje como

$$S \rightarrow \ if \ E \ then \ S^1$$
$$S \rightarrow \ while \ E \ do \ S^1$$

es tan sencilla como

```
S.tipo = if (E.tipo=boolean) then S1.tipo
         else error_tipo;
```

↪ La comprobación de la concatenación de proposiciones,

$$S \rightarrow S1; S2$$

se haría:

```
S.tipo = if (S1.tipo=void) and
            (S2.tipo=void) then void
         else
            error_tipo;
```

↪ La comprobación de tipos asociada a la asignación

$$S \rightarrow id = E$$

es de las más simples:

```
S.tipo = if (id.tipo=E.tipo) then void
         else error_tipo;
```

↪ La comprobación de tipos en expresiones o variables en sí

$$E \rightarrow literal$$
$$E \rightarrow numero$$
$$E \rightarrow int$$
$$E \rightarrow real$$
$$E \rightarrow \ldots$$

no tiene ningún secreto,

```
E.tipo = literal | numero | integer | real | ...
```

↪ La comprobación de tipos relativa a funciones, por último,

$$E \rightarrow E^1(E^2)$$

se resuelve

```
E.tipo = if (E2.tipo=S) and
            (E1.tipo=(fun S->T)) then T
         else
            error_tipo;
```

5.2.2. Equivalencia de expresiones de tipo

Se consideran dos tipos de equivalencia para las *expresiones de tipo*:

Equivalencia estructural Se dice que dos expresiones de tipo tienen *equivalencia estructural* cuando son del mismo tipo básico o son resultado de aplicar los mismos constructores en el mismo orden a los mismos tipos básicos (o incluso expresiones de tipo).

Equivalencia por nombre Se dice que dos expresiones de tipo tienen *equivalencia por nombre* cuando son del mismo "nombre" de tipo, es decir, son expresiones de tipo *declarado* idéntico.

Una forma de realizar la comprobación de equivalencia de tipos es mediante la **codificación de expresiones de tipo**, esto es, asignar un código binario a cada constructor de tipo y cada tipo básico, de suerte que cada expresión pueda codificarse mediante una secuencia de ceros y unos que represente los códigos de los constructores y tipos básicos que se han utilizado para su composición. La comprobación de la equivalencia de dos expresiones de tipo es tan trivial como comparar bit a bit sus respectivas codificaciones.

Esta estrategia tiene la ventaja de que ahorra memoria (no se necesita tanto arrastre de información) y, puesto que la comparación binaria puede realizarse en la propia ALU, es más rápida. Sin embargo, podría suceder que dos tipos compuestos distintos tuviesen como representación la misma secuencia de bits (ya que no se reflejan, por ejemplo, tamaños de matrices o argumentos de funciones).

```
function equivest(s,t):boolean;
begin
   if (s y t son del mismo tipo basico) then
     return true
   else if (s=array(s1,s2) and t=array(t1,t2)) then
     return (equivest(s1,t1) and equivest(s2,t2))
   else if (s=s1xs2 and t=t1xt2) then
     return (equivest(s1,t1) and equivest(s2,t2))
   else if (s=pointer(s1) and t=pointer(t1)) then
     return equivest(s1,t1)
   else if ...
     ...
   else
     return false
end
```

Cuadro 5.2: Algoritmo de comprobación de equivalencia estructural.

5.2.3. Conversores de tipo

Supóngase la situación:

$$a = b \times c$$

donde a y c son reales mientras que b es un entero. Restringiéndonos a lo que hemos visto hasta ahora, está claro que esto fallaría, mas nuestra experiencia nos dice que es algo perfectamente viable ¿cómo es posible, entonces? Mediante una **conversión de tipo**.

Esta *conversión*, cuando es implícita, hecha automáticamente por el compilador se denomina **coerción**, mientras que cuando es explícitamente indicada por el programador es lo que conocemos como **cast**.

5.2.4. Sobrecarga

Se dice que un *operador* está **sobrecargado** cuando en función de las variables con las que trabaja tiene un significado u otro. La *sobrecarga* puede afectar también a los *símbolos*, en cuyo caso puede crear problemas de ambigüedad (por ejemplo, si se sobrecargan los paréntesis para referenciar matrices), motivo por el que se recomienda evitarlo.

Asimismo, también podemos enfrentarnos a la *sobrecarga* de *funciones*, que recibe el nombre de **polimorfismo**.

En ese caso, ante una construcción

$$E \rightarrow E^1(E^2)$$

Producción	*Regla Semántica*
$E \rightarrow numero$	`E.tipo = integer`
$E \rightarrow numero \cdot numero$	`E.tipo = real`
$E \rightarrow id$	`E.tipo = buscaTDS(id.entrada)`
$E \rightarrow E^1 op E^2$	`E.tipo = if ((E1.tipo=integer) and (E2.tipo=integer)) then integer else if ((E1.tipo=integer) and (E2.tipo=real)) then real else if ((E1.tipo=real) and (E2.tipo=integer)) then real else if ((E1.tipo=real) and (E2.tipo=real)) then real else error_tipo`

Cuadro 5.3: Comprobación de tipos para coerción de entero a real.

se admite que la expresión de tipo E^2 no sea única. Se trabaja entonces con un conjunto de expresiones de tipo posibles (E^2 pasa a ser un conjunto) con la repercusión y modificaciones que ello introduce en las reglas semánticas que hemos visto:

```
E.tipo = { t / existen s en E2.tipos, s->t en E1.tipos }
```

Capítulo 6

Generación de Código Intermedio

Tras el análisis semántico llega el momento de generar lo que denominamos **código intermedio**, que posteriormente será optimizado antes de dar lugar al definitivo **código objeto**.

El motivo por el que se realiza este paso *intermedio*, la razón por la que se genera *código intermedio* es evitar tener que construir un compilador desde cero para todas y cada una de las diferentes arquitecturas. Además, ello va a reportar otros beneficios, ya que nos permitirá optimizar mejor, ya que el proceso de optimización trabajará sobre este mismo código intermedio. Por último, también favorece la reutilización.

6.1. Notaciones

No existe un "lenguaje intermedio universal". Los más conocidos y usados, que veremos, son:

- Notación Polaca Inversa (RPN)
- Cuartetos
- Tercetos, Tercetos indirectos
- Lenguajes a tres direcciones
- P-código

Interesa tener *código basado en 3 direcciones*, del estilo

$$a = b \text{ op } c$$

(dos operandos y el resultado) ya que facilitará la generación de código objeto y también su depuración.

6.1.1. Notación Polaca Inversa (RPN)

Es la primera notación de código intermedio que se utilizó. Con una pila es posible construir un generador de código muy sencillo, y es en lo que se basó esta idea.

La **notación polaca inversa**, también llamada *postfija* representa las jerarquías en forma de árbol de modo que cada nodo aparece inmediatamente después de sus hijos. El formato postfijo es fácil de traducir y pasar a código máquina con ayuda de una pila: se colocan los operandos y después los operadores.

```
OPERADORES            + - / * mod div
FUNCIONES             sqrt sin cos tan abs ord char
BLOQUES DE CODIGO     bloque ... finbloque
ENTRADA/SALIDA        read write
MATRICES              decl sub
ASIGNACION            :=
BIFURCACIONES
  SALTO INCONDICIONAL b
  SALTO SI POSITIVO   bp
  SALTO SI NEGATIVO   bn
  SALTO SI CERO       bc
```

Cuadro 6.1: Notación Polaca Inversa (RPN).

En RPN los paréntesis se ignoran y la traducción de una expresión a esta notación sigue las directrices que se muestran a continuación:

Producción	*Reglas Semánticas*	
$S \rightarrow E$		
$E \rightarrow T$		
$E \rightarrow E + T$	$P(i) =' +'$	$i = i + 1$
$E \rightarrow E - T$	$P(i) =' -'$	$i = i - 1$
$E \rightarrow -T$	$P(i) =$ @	$i = i + 1$[a]
$T \rightarrow F$		
$T \rightarrow T * F$	$P(i) =' *'$	$i = i - 1$
$T \rightarrow T/F$	$P(i) =' /'$	$i = i - 1$
$F \rightarrow id$	$P(i) = buscarTDS(i)$	$i = i - 1$
$F \rightarrow (E)$		

[a] Arrastre de información, @ es un símbolo de control.

Cuadro 6.2: Traducción a RPN.

donde la semántica se asocia a cada nodo.

Hay dos formas de evaluar un árbol sintáctico de una expresión: mediante esquemas de traducción (no lo veremos) o mediante traducción dirigida por la sintaxis (lo que estamos viendo). Se hace de abajo a arriba y de izquierda a derecha, de manera determinista, obteniendo, por diferentes técnicas que veremos en este tema, el resultado en código intermedio.

6.1.2. Cuartetos

La notación de **cuartetos** (también llamados *cuádruplos* en alguna bibliografía, como [4]) tiene la siguiente forma:

$$(< operador >, < operando1 >, < operando2 >, < resultado >)$$

es decir, A/B, por ejemplo, sería representado como $(/, A, B, T_1)$ donde T_1 es un registro temporal.

Los símbolos usados en esta notación son:

```
OPERACIONES ARITMETICAS          + - * / ^
                                 mod abs sqrt sin cos
ASIGNACION X:=E                  (:=, E, , X)
ENTRADA/SALIDA                   (read, , , X) (write, , , X)
SALTO ABSOLUTO INCONDICIONAL     (JP, n, , )
                                      n = posicion absoluta a saltar
SALTO RELATIVO INCONDICIONAL     (JR, n, , )
                                      n = posicion relativa actual
SALTO INCONDICIONAL
  SALTO SI IGUAL A CERO          (JZ, n, E, )
  SALTO SI MAYOR QUE CERO        (JGZ, n, E, )
  SALTO SI MENOR QUE CERO        (JLZ, n, E, )
  SALTO SI IGUALES               (JE, n, X1, X2)
  SALTO SI MAYOR                 (JG, n, X1, X2)
  SALTO SI MENOR                 (JL, n, X1, X2)
```

Cuadro 6.3: Notación de cuartetos.

6.1.3. Tercetos

La notación de **tercetos** (también llamados *triples* en alguna bibliografía, como [4]) se diferencia de la de *cuartetos* en que el resultado queda implícito en el propio *terceto*, siendo su formato:

$$(< operador >, < operando1 >, < operando2 >)$$

de manera que para hacer $a + b + c$ tendríamos en primer lugar el terceto $(+, a, b)$ etiquetado como $[x]$ y seguidamente el terceto $(+, [x], c)$, referenciando al anterior.

La ventaja de los *tercetos* es que ocupan mucha menos memoria en variables temporales. Por otra parte, la recolocación del código en optimización es más compleja porque hay que mantener las referencias correctamente. Para enfrentar este problema surgen los **tercetos indirectos**, que son *tercetos* junto con los que se guarda un ***vector secuencia*** que marca el orden de ejecución tras la recolocación, de manera que no es necesario reescribir los tercetos -sus referencias- en caso de reordenamiento. Además, si los *tercetos* se repiten sólo es necesario repetirlos en el *vector secuencia* y no "físicamente".

Por su estructura, los tercetos carecen de instrucciones `JE`, `JG` y `JL`, aunque pueden ser "simuladas"; el resto de la notación es gemela a la de cuartetos (tabla 6.3).

6.1.4. Código a tres direcciones

Esta última notación que veremos, llamada **código a tres direcciones**, es en cierto modo una generalización de todas las anteriores. Es una notación más, que recibe su nombre por razones obvias debidas a su estructura,

$$ti = tj \text{ op } tk$$

y que, quizás por su mayor claridad (recuerda al pseudocódigo), se usa con mayor frecuencia en los lenguajes intermedios y también, como haremos nosotros, en optimización de código.

Los símbolos que utiliza son los mismos que en *cuartetos* y *tercetos*, con la salvedad de que para los saltos se utiliza la palabra reservada `goto`.

6.2. Máquina Objeto

Para seguir concretando el el proceso de generación de código objeto, describiremos a continuación una máquina secuencial Von Neumann, que utilizaremos como base de aquí en adelante. Supondremos una CPU con una serie de operaciones (instrucciones) y una forma de mover datos entre la memoria y los registros.

Esta **máquina objeto** tendrá:

- ▷ n registros $R_0, R_1, \ldots R_{n-1}$
- ▷ instrucciones de la forma

$$\text{operacion } fuente, destino$$

con las palabras reservadas clásicas

```
ADD
SUB
MUL
DIV
CALL   (llamada a funciones)
GOTO   (saltos)
MOVE   (accesos a memoria: cargas/almacenamientos)
```

Los *modos de direccionamiento* que consideraremos serán los presentes en la tabla 6.4.

Modo	*Forma*	*Dirección*	*Coste*
absoluto	M	M[a]	1
registro	R	R[b]	0
indexado	$C(R)$	C+contenido(R)[c]	1
registro indirecto	$*R$	contenido(R)	1
indexado indirecto	$*C(R)$	contenido(C+contenido(R))	2
literal	$\#C$	C	1

[a] M es una posición de memoria principal.
[b] R es un registro de la CPU.
[c] C es una constante.

Cuadro 6.4: Modos de direccionamiento

6.2.1. Método de las Cajas

Es una manera de modelizar las operaciones entre dos operandos. Si analizamos de manera sencilla[1] las posibles instrucciones que manejamos, veremos que casi todas ellas son muy similares. Todas siguen el siguiente esquema:

```
MOV          operandoA, registroI
MOV          operandoB, registroJ
OPERACION_X  registroI, registroJ
MOV          registroJ, temporalK
```

De manera que si tomamos esta estructura como "construcción base" a la que denominamos *caja*, el código al final es un conjunto de *cajas*[2].

[1] Sin tener en cuenta cuestiones como si los registros están previamente ocupados, etc.
[2] El paso de este "ensamblador" después de "encajonado" a código máquina es trivial: la operación se codifica según un convenio, se buscan las constantes en la TDS,...

El problema es que el código resultante de aplicar al pie de la letra estas traducciones es un código muy malo, entendiendo por *bondad del código* una combinación entre su rapidez y la memoria que ocupa y/o utiliza, puesto que es un código en absoluto optimizado.

El código que genera un cuarteto

$$(< operacion >, < operando1 >, < operando2 >, < destino >)$$

(y, por supuesto, también un terceto) es también una caja:

```
MOV           operandoA, registroI
MOV           operandoB, registroJ
OPERACION_X   registroI, registroJ
MOV           registroJ, destino
```

La generación de código a tres direcciones también se hace por semántica. Los intérpretes clásicos (puros) usaban RPN ya que es más intuitivo (ya contamos con una pila).

Una forma sencilla de optimización que se puede hacer en tiempo de compilación es mantener una tabla de localización del almacenamiento de los operandos, para evitar la carga sucesiva del mismo operador varias veces, igual que la generación de variables temporales lógicamente innecesarias.

6.2.2. Generación y Representación de saltos

Si sabemos generar saltos, sabemos generar todas las instrucciones de control (`if`, `for`, `do`, `while`). Son necesarias dos "pasadas" por el código para hacer eso: en la primera se generan etiquetas relativas, que se sustituyen en la segunda, una vez que ya se conoce la dirección base.

6.2.3. Acceso a elementos de matrices

Se puede acceder rápidamente a los elementos de una matriz si se guardan en un bloque de posiciones consecutivas. Si el ancho de cada elemento de la matriz es a, entonces el i-ésimo elemento de la matriz A comienza en la posición

$$base + (i - inf) \times a$$

donde inf es el límite inferior de los subíndices y $base$ es la dirección relativa de la posición de memoria asignada a la matriz (es decir, es $A[inf]$).

La expresión anterior se puede evaluar parcialmente durante la compilación si se escribe como

$$i \times a + (base - inf \times a)$$

ya que la subexpresión $base - inf \times a$ se puede evaluar cuando aparece la declaración de la matriz y ser guardada en entrada de la TDS de A.

Esto se puede extender a los cálculos de direcciones de elementos de matrices multidimensionales. Una matriz bidimensional se almacena generalmente en una de dos formas, bien en *forma de filas* (fila a fila), bien en *forma de columnas* (columna a columna). En el caso de una matriz bidimensional almacenada por filas, la dirección relativa de $A[i, j]$ se puede calcular mediante

$$base + ((i - inf_i) \times n_j + j - inf_j) \times a$$

donde inf_i e inf_j son los límites inferiores de los valores de i y j, y n_j es el número de valores que puede tomar j. Es decir, si sup_j fuese el límite superior para el valor de j, entonces $n_j = sup_j - inf_j + 1$.

Suponiendo que i y j son los únicos valores que no se conocen durante la compilación, la expresión anterior se puede reescribir como

$$((i \times n_j) + j) \times a + (base - (inf_i \times n_j) + inf_j) \times a)$$

donde el último término puede determinarse durante la misma.

Generalizando, si tenemos una matriz de n dimensiones

$$M[inf_1 \ldots sup_1, inf_2 \ldots sup_2, \ldots, inf_n \ldots sup_n]$$

donde $inf_1, inf_2, \ldots, inf_n$ son los límites inferiores de cada dimensión, $sup_1, sup_2, \ldots, sup_n$ son los límites superiores de cada dimensión y n es el número de dimensiones de la matriz, dando por sentado que $sup_i \geq inf_i$ el número de elementos de cada dimensión es de nuevo $e_i = sup_i - inf_i + 1$ y el acceso a un elemento cualquiera $M[j_1, j_2, \ldots, j_n]$ tendrá la dirección

$$\begin{array}{lllll} base & + & (j_1 - i_1) \times e_2 \times e_3 \times \ldots \times e_n & + \\ & + & (j_2 - i_2) \times e_3 \times e_4 \times \ldots \times e_n & + \\ & + & \ldots & + \\ & + & (j_{n-1} - i_{n-1}) \times e_n & + \\ & + & (j_n - i_n) & \end{array}$$

Por último, en caso de tener una matriz *triangular* de la que se almacenen sólo los elementos por debajo de la diagonal, el acceso a cada uno de ellos $[i, j]$ sería $\frac{i + (j-1) \times j}{2}$.

6.2.4. Representación de strings

Pero no sólo la representación de matrices es un problema, sino que también lo es la de las cadenas de caracteres. Las soluciones históricas que se han adoptado al respecto son:

- Definición de un tamaño máximo para cualquier string y reservar un banco de memoria "troceado" al efecto. Un ejemplo de uso de esta estrategia es `Fortran`.
- Mantenimiento de un registro con el par (nombre, longitud del string) y un puntero a la dirección de inicio, todo ello almacenado en la TDS. Esta estrategia se debe a McKeeman, Horning y Wortman, data de 1970 y es usada por lenguajes como `Pascal`.

Capítulo 7

Optimización de Código

Técnicas de generación de código como el *método de las cajas* que hemos visto son, como ya se comentó en su momento, extremadamente ineficientes. Por ello, la **optimización de código** puede verse incluso como parte de la propia generación.

Las técnicas de optimización se aplican a 3 niveles:

1. ***Optimización en código fuente***, realizada normalmente por el programador, cada vez se usa menos debido al gran desarrollo hardware actual.
2. ***Optimización en código intermedio***, la más importante de las tres, es la que realizan los compiladores gracias a la aplicación de técnicas de análisis de flujo de datos.
3. ***Optimización en código objeto***, muy orientada a la máquina y muy dependiente de ésta, tiene problemas de portabilidad.

7.1. Técnicas de optimización en código fuente

7.1.1. Reducción simple o Reducción de operaciones

Consiste en evitar incluir en el código fuente operaciones que pueden ser realizadas previamente, por ejemplo, en lugar de poner

```
s = 2 * PI
```

se pondría

```
s = 6.283185
```

Esto es importante sobre todo si expresiones como la anterior se incluyen en lazos, subrutinas, etc. En ocasiones, incluso es preferible generar previamente varias variables que se utilicen en diferentes iteraciones que operar en cada interación.

7.1.2. Reacondicionamiento o Reordenamiento de instrucciones

Consiste en alterar el orden de secuenciación de las instrucciones (sin que varíe su semántica, claro está) a fin de conseguir tanto la generación de menos instrucciones de código intermedio como de código objeto, como el uso de un menor número de registros y variables temporales.

7.1.3. Eliminación de redundancias

Debe evitarse siempre la realización de cálculos repetitivos. La optimización siempre centra su especial atención en cuerpos críticos como los lazos, donde es frecuente encontrar expresiones similares a:

```
M[i][j][k] = M[i x bi x di][j x bj x dj][k x bk x dk] + incremento
```

donde toda la primera parte[1] supone una redundancia y podría extraerse.

7.1.4. Reducción de potencias

Ante una operación como

```
s = a**2
```

siempre es preferible utilizar

```
s = a*a
```

ya que no hay ningún ensamblador que contenga instrucciones de potenciación, mientras que todos incluyen el producto puesto que es una operación que ya las propias ALUs realizan.

7.1.5. Reordenación de expresiones: Algoritmo de Nakata

El **algoritmo de Nakata y Anderson** data de 1964 y su objetivo es marcar una pauta para la reordenación de expresiones:

```
PASO-1. Construir el árbol de la expresión a reordenar.
PASO-2. Etiquetado del árbol:
        PASO-2.1. IF n es hoja THEN
                     IF n es hijo más a la derecha de su padre THEN
                        etiqueta(n) = 1
                     ELSE
```

[1] En lenguajes que manejen las matrices como punteros, como `C` y algunos bloques de `Fortran`.

```
                    etiqueta(n) = 0
                ELSE BEGIN
                   SEAN n1, n2,..., nk hijos ordenados
                        /* por etiqueta */
                   THEN
                      etiqueta(n) = max(etiqueta(ni) + i - 1)
                END
PASO-3. Una vez etiquetado el árbol, para generar la expresión
        se parte de la raíz y se desciende por la rama del subárbol
        cuya etiqueta contenga el valor máximo.
        En caso de igualdad, se opta por la subrama de la derecha,
        salvo si la operación no es conmutativa.
```

7.1.6. Extracción de invariantes

Llamamos expresiones **invariantes** de lazo a aquéllas que pueden sacarse fuera de los bucles ya que su valor no sufre variación durante las iteraciones. Así, en lugar de

```
i = 1..N
  a[i] = f(c1,c2,...,cn)
otro i
```

se persigue construir preferiblemente

```
t = f(c1,c2,...,cn)
i = 1..N
  a[i] = t
otro i
```

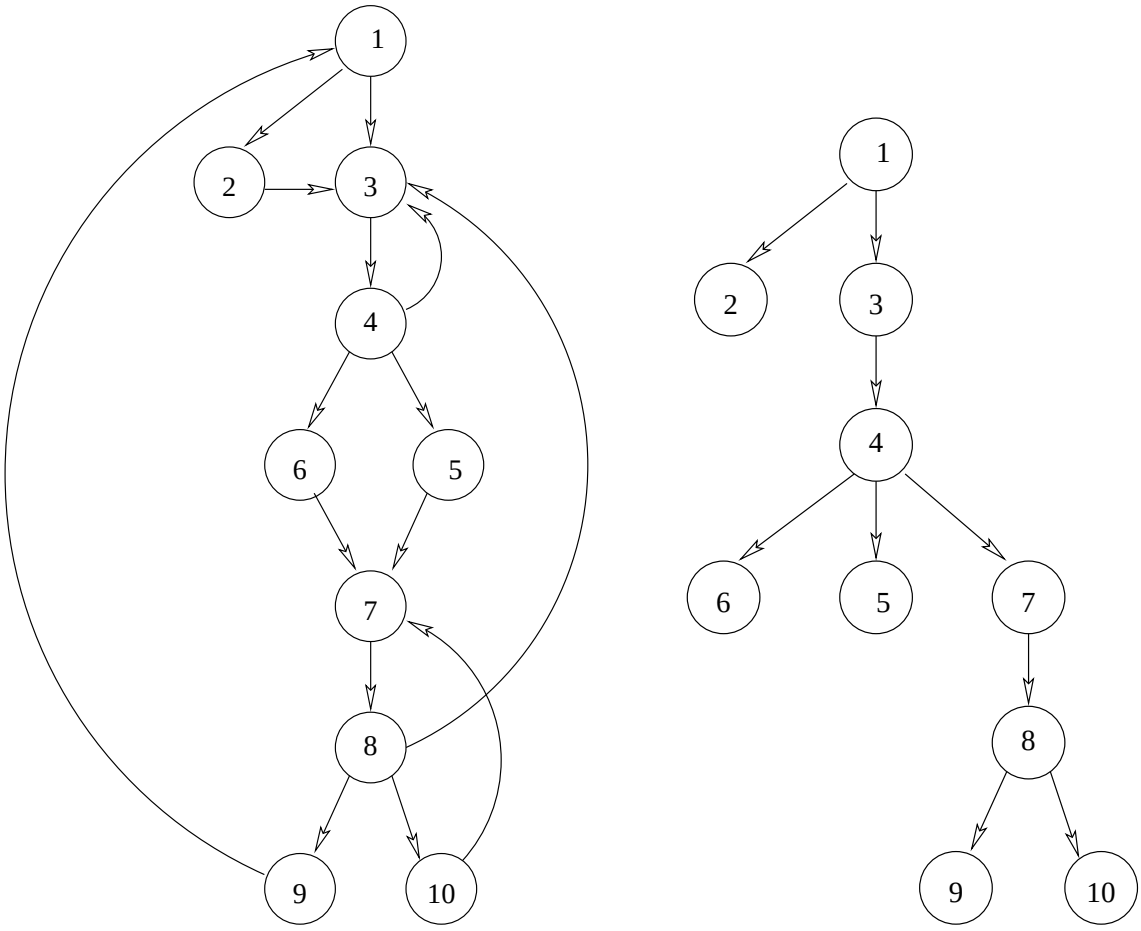
7.2. Análisis Global del Flujo de Datos

7.2.1. Detección de lazos en los grafos de flujo

Decimos que un nodo d **domina** a un nodo n si todo *camino* desde el nodo inicial a n pasa obligatoriamente por d.

Así, por ejemplo, según la figura 7.1 (página 62):

```
1  domina  TODOS
2  domina  2
3  domina  TODOS - {1,2}
4  domina  TODOS - {1,2,3}
```



(a) Grafo de Flujo (b) Árbol de Dominación

Figura 7.1: Grafo de Flujo y Árbol de Dominación.

```
 5  domina  5
 6  domina  6
 7  domina  7,8,9,10
 8  domina  8,9,10
 9  domina  9
10  domina  10
```

En el análisis de código, los nodos se identifican con bloques de instrucciones.

Propiedades de los lazos

Los lazos tienen una serie de propiedades que debemos considerar:

- Un lazo tendrá un sólo punto de entrada, que domina a todos los nodos del lazo.

- Debe existir al menos una forma de iterar, es decir, hay al menos un camino hacia el encabezamiento.
- Para localizar lazos el método es buscar aristas cuyas cabezas dominen a las colas.

7.3. Análisis de Flujo

El **análisis de flujo** consiste en recopilar información de variables, expresiones, etc., por ejemplo dónde se definen, hasta dónde llegan, dónde se modifican,... a través del árbol de análisis sintáctico.

Para ello se usan *ecuaciones*, y la ecuación base es:

$$sal[S] = gen[S] \cup (ent[S] - desact[S])$$

donde

S	es un bloque de código
$sal[S]$	representa las variables que salen activas tras la ejecución de S
$gen[S]$	las variables que toman valor en S
$ent[S]$	las variables que llegan activas a S
$desact[S]$	las variables que son redefinidas en S

Claro está que, cuando hablamos de variables, también podemos hablar de expresiones e incluso punteros y llamadas a funciones. Sin embargo, nos ceñiremos a las primeras por simplicidad.

El tipo de análisis de flujo que se realice (hay varias estrategias diferentes) marcan su diferencia en la definición del cálculo de los conjuntos sal, gen, ent y $desact$.

En el análisis de bloques de código, se establecen *puntos* entre sentencias y también entre cada par de bloques. Un **camino** de un punto p_1 a un punto p_n es una secuencia de puntos $p_1, p_2, p_3, \ldots, p_n$ de forma que p_i es el punto que precede a una proposición y p_{i+1} es el que la sigue, o bien p_i es el final de un bloque y p_{i+1} es el comienzo del siguiente.

7.3.1. Definiciones de Alcance

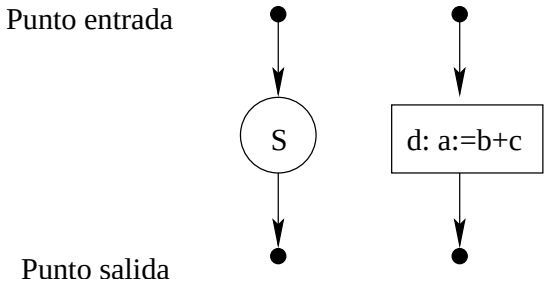
Llamamos **definición** de una variable x a una proposición que asigna un valor a x ó que puede asignarlo. En el primer caso se concreta y se habla de *definición no ambigua*.

Llamamos **alcance** de una definición d a un punto p si existe un camino desde el siguiente punto a d hasta p sin que la definición se desactive. Una definición se *desactiva* si durante el camino a otro punto se define de nuevo.

Veremos a continuación los valores que pueden tomar en un bloque de código los conjuntos sal, gen, ent y $desact$.

Bloque único

Para el caso más sencillo, el de un bloque único de código, tenemos que:

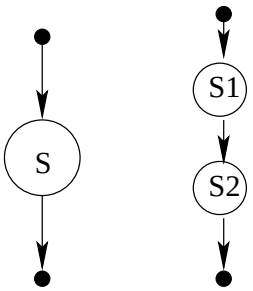


$$\begin{array}{lcl} gen[S] & = & \{d\} \\ desact[S] & = & D_a - \{d\} \\ sal[S] & = & gen[S] \cup (ent[S] - desact[S]) \end{array}$$

donde D_a representa todas las definiciones de la variable a del programa.

Secuencia

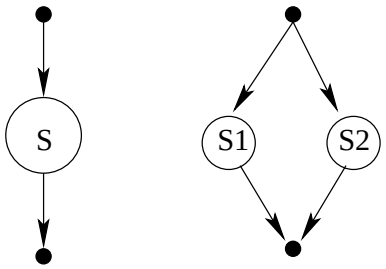
Dada una secuencia de bloques de instrucciones:



$$\begin{array}{lcl} gen[S] & = & gen[S_2] \cup (gen[S_1] - desact[S_2]) \\ desact[S] & = & desact[S_2] \cup (desact[S_1] - gen[S_2]) \\ ent[S_1] & = & ent[S] \\ ent[S_2] & = & sal[S_1] \\ sal[S] & = & sal[S_2] \end{array}$$

Bifurcación

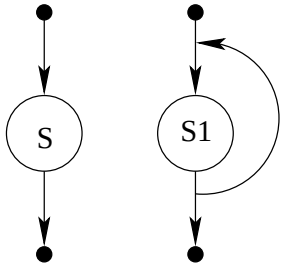
El análisis de sentencias de control tipo `if` y algunos saltos es dependiente del tipo de análisis que se esté realizando. Nosotros consideraremos que ambas ramas se activan.



$$
\begin{array}{lcl}
gen[S] & = & gen[S_1] \cup gen[S_2] \\
desact[S] & = & desact[S_1] \cap desact[S_2] \\
ent[S_1] & = & ent[S] \\
ent[S_2] & = & ent[S] \\
sal[S] & = & sal[S_1] \cup sal[S_2]
\end{array}
$$

Bucle

Por último, el análisis de bucles o lazos, no encierra ningún secreto:



$$
\begin{array}{lcl}
gen[S] & = & gen[S_1] \\
desact[S] & = & desact[S_1] \\
ent[S_1] & = & ent[S] \cup sal[S_1] \quad = \quad ent[S] \cup gen[S_1] \\
sal[S] & = & sal[S_1]
\end{array}
$$

DEMOSTRACIÓN.-

Partimos de

$$\begin{aligned} ent[S_1] &= ent[S] \cup sal[S_1] \\ sal[S_1] &= gen[S_1] \cup (ent[S_1] - desact[S_1]) \end{aligned}$$

que escribiremos, por comodidad,

$$\begin{aligned} E_1 &= E \cup X_1 \\ X_1 &= G_1 \cup (E_1 - D_1) \end{aligned}$$

donde E, G_1 y D_1 son "constantes".
Procederemos por inducción:
Iteración 1

$$\begin{aligned} E_1^0 &= E \\ X_1^0 &= G_1 \cup (E - D_1) \end{aligned}$$

Iteración 2

$$\begin{aligned} E_1^1 &= E \cup X_1^0 = E \cup [G_1 \cup (E - D_1)] = E \cup G_1 \\ X_1^1 &= G_1 \cup (E_1^1 - D_1) = G_1 \cup [(E \cup G_1) - D_1] = G_1 \cup (E - D_1) \end{aligned}$$

(y así sucesivamente; no necesitamos seguir porque ya hemos obtenido $E = E \cup G_1$, que es lo que pretendíamos)

Dado un árbol sintáctico, su recorrido hacia arriba arrastrando atributos sintetizados permite obtener *gen* y *desact*, mientras que un posterior recorrido en profundidad hacia abajo teniendo en cuenta atributos heredados (del padre y/o hermanos) permite obtener *ent* y *sal*.

7.3.2. Notación Vectorial

El análisis de flujo de datos, como ya hemos mencionado, suele acometerse por bloques preferentemente a por instrucciones. Se definen entonces *vectores* que representan lo que se genera/desactiva:

$$V = (a_1 a_2 \dots a_n)$$

con n =número de definiciones del programa. En el bloque S_i, la componente $a_i = 1$ indica que se ha generado o desactivado (según el vector que se considere) la definición i.

Las ecuaciones que se aplican no varían, de modo que simplemente recordemos que, entre vectores binarios:

$$\begin{array}{lllll} A & - & B & = & V_A \wedge \neg V_B \\ A & \cup & B & = & V_A \vee V_B \\ A & \cap & B & = & V_A \wedge V_B \end{array}$$

7.3.3. Solución Iterativa para las ecuaciones de Flujo de Datos

Definiciones de alcance (alcance de definiciones)

Definimos un **bloque** como una proposición de asignación o una cascada de proposiciones de asignación.

Partimos de que se ha calculado *genera* y *desactiva* de cada bloque para todos los bloques del programa, y de que:

$$\begin{aligned} ent[B] &= \bigcup_{predecesores} sal[P] \qquad P \in predecesores \\ sal[B] &= gen[B] \cup (ent[B] - desact[B]) \end{aligned}$$

y bajo estas condiciones el algoritmo es:

```
Entrada: gen[B] y desact[B] para cada B
Salida : ent[B] y sal[B]    para cada B

ent[Bi] := 0;
para cada Bi hacemos
  sal[Bi] := gen[Bi];
  cambio  := true;
  mientras (cambio) hacemos
    cambio := false;
    para cada Bi hacemos
      ent[Bi] := Up sal[P];  /* P predecesores */
      salant  := sal[Bi];
      sal[Bi] := gen[Bi] U (ent[Bi] - desact[Bi]);
      si (sal[Bi] <> salant) entonces
        cambio := true;
    fin-para
  fin-mientras
fin-para
```

7.3.4. Análisis de Expresiones Disponibles

Decimos que una expresión $x + y$[2] está **disponible** en un punto P si todo camino, no necesariamente sin lazos, desde el nodo inicial hasta P evalúa $x + y$ y después de la última evaluación antes de P no hay asignación ni de x ni de y.

Decimos que un bloque B **desactiva** una expresión $x + y$ si asigna un valor a x o a y.

Decimos que un bloque B **genera** una expresión $x + y$ si evalúa $x + y$ y no genera una redefinición de x ó de y.

[2]El signo "+" no significa aquí la operación de suma, sino cualquier operación en general.

Expresaremos:

$$\begin{array}{lcl} e\_gen[B_i] & = & \text{expresión que genera el bloque } B_i \\ e\_desact[B_i] & = & \text{expresión que desactiva el bloque } B_i \end{array}$$

$$\begin{array}{lcl} U & = & \text{universal} \\ \varnothing & = & \text{vacío} \end{array}$$

Y usaremos las ecuaciones:

$$\begin{array}{lcll} sal[B_i] & = & e\_gen[B_i] \cup (ent[B_i] - e\_desact[B_i]) & \\ ent[B_i] & = & \bigcap_p sal[P_i] & B_i \neq B_{inicial} \\ ent[B_i] & = & \varnothing & B_i = B_{inicial} \end{array}$$

Como se puede observar, el análisis anterior daba todo el alcance posible, sin embargo ahora queremos ser muy conservadores para poder usar el valor de la expresión (en lugar de etiquetar las asignaciones, como antes, ahora se etiquetan las expresiones). El algoritmo es:

```
Entrada: e_gen[Bi] y e_desact[Bi] para todo Bi
Salida : ent[Bi]   y sal[Bi]      para todo Bi

ent[Bi] := 0;
sal[Bi] := e_gen[Bi];
para todo Bi<>B1 hacemos
  sal[Bi] := U - desact[Bi];
cambio := true;
mientras (cambio) hacemos
  cambio := false;
  para cada Bi<>B1 hacemos
    ent[Bi] := ^p sal[P];
    salant  := sal[Bi];
    sal[Bi] := e_gen[Bi] U (ent[Bi] - e_desact[Bi]);
    si (salant <> sal[Bi]) entonces
      cambio := true;
  fin-para
fin-mientras
```

7.3.5. Análisis de Variables Activas

Una variable a es **activa** en un punto P si se puede utilizar el valor de a en algún camino que comience en P.

Este análisis pretende liberar memoria, de modo que *estar activo* significa que se puede (y de hecho, se hace) usar el valor; si no se vuelve a usar, entonces no se está activo. Esta filosofía obliga a hacer el análisis en el código al revés, de abajo hacia arriba.

Llamamos **entrada** al conjunto de variables activas al comienzo de un bloque. Llamamos **salida** al conjunto de variables activas a la salida de un bloque. Llamamos **definidas** al conjunto de variables a las que se les asigna un valor en el bloque. Por último, se llama **uso** al conjunto de variables cuyos valores se utilizan antes de cualquier definición de la variable.

Las ecuaciones usadas en este tipo de análisis son:

$$\begin{aligned} ent[B_i] &= uso[B_i] \cup (sal[B_i] - def[B_i]) \\ sal[B_i] &= \bigcup_s ent[S_i] \qquad S_i \in sucesores \end{aligned}$$

El algoritmo es el siguiente:

```
Entrada: grafo de flujo G y def[Bi], uso[Bi] para todo Bi
Salida : sal[Bi] para todo Bi

para todo Bi hacemos
  sal[Bi] == 0;
para todo Bi hacemos
  ent[Bi] := uso[Bi];
mientras ocurren cambios en los conjuntos ent[Bi] hacemos
  para cada Bi hacemos
    sal[Bi] := Us ent[S];   /* S sucesores */
    ent[Bi] := uso[Bi] U (sal[Bi] - def[Bi]);
  fin-para
fin-mientras
```

Capítulo 8

Errores

En este breve tema comentaremos algunas generalidades acerca de los **errores** en el ámbito que nos ocupa.

8.1. Tipos de errores

Consideraremos la exitencia de varios **tipos de errores**:

- *Errores léxicos*, son aquéllos en los que donde se esperaba una palabra reservada del lenguaje aparece otra (por ejemplo, poner `far` en lugar de `for`).
- *Errores sintácticos*, son aquéllos que provocan que no se encuentre un árbol sintáctico que analice el programa (por ejemplo, cuando falta un paréntesis).
- *Errores semánticos*, aparecen cuando llevamos a cabo acciones inadecuadas sobre elementos incorrectos (por ejemplo, hacer la raíz cuadrada de una variable de tipo `string`).
- *Errores de compilación*, aparecen por limitaciones propias del compilador (algunos no permiten más de 10 bucles anidados, otros no más de 30 procedimientos,...).
- *Errores de ejecución*, se deben a descuidos del programados (situaciones como accesos a matrices fuera de rango, a ficheros no abiertos, ...).

8.2. Recuperación de errores

Según el tipo de error ante el que nos encontremos, serán susceptibles de ser realizadas diferentes medidas de correción. En particular, en compiladores puede ser interesante para nosotros intentar subsanar *errores lexicográficos*, tanto aquéllos que suponen la sustitución de unos elementos por otros, como la inclusión de elementos extraños entre elementos correctos, o la desaparición de algún elemento.

Afrontaremos, pues, la corrección de errores lexicográficos desde estas perspectivas.

8.2.1. Corrección de errores de cambio

Supone admitir palabras del lenguaje con uno o más errores. Se define el **operador de cambio**[1]:

$$C(a) = T - \{a\}$$

de tal modo que

$$\begin{aligned} C^1(x) &= \{y/y \in T^* \wedge y \text{ resultado de 1 cambio en } x\} \\ C^2(x) &= \{y/y \in T^* \wedge y \text{ resultado de 2 cambios en } x\} \\ &\dots \\ C^n(x) &= \{y/y \in T^* \wedge y \text{ resultado de } n \text{ cambios en } x\} \end{aligned}$$

Además, por convenio, si $|x| < n$, entonces $C^n(x) = \varnothing$.

De modo que para corregir errores, es decir, para admitir palabras con algún error durante el proceso de compilación, si tenemos una gramática $G = (N, T, P, S)$, crearemos una gramática $G' = (N, T, P', S)$ en la que las reglas serán

$$P' = P \cup \{\text{reglas de las operaciones de cambio}\}$$

de tal forma que el "nuevo lenguaje" aceptado puede expresarse como

$$L(G') = \{y/y \in T^* \wedge y \in C^n(x) \wedge x \in L(G), n \geq 0\}$$

La corrección de errores se implementa mediante un *esquema de traducción*, dado por $EDT(N, T_e, T_s, P, S)$, definido según la estructura:

$$\begin{array}{lcccll} A & \rightarrow & \alpha_i & , & \alpha_i & \text{(no existen errores)} \\ A & \rightarrow & \mu & , & \alpha_i & (\text{con } \mu = C^1(\alpha_i)) \\ A & \rightarrow & \beta & , & \alpha_i & (\text{con } \beta = C^2(\alpha_i)) \\ & \dots & & & & \\ A & \rightarrow & \delta & , & \alpha_i & (\text{con } \delta = C^n(\alpha_i)) \end{array}$$

8.2.2. Corrección de errores de borrado

Del mismo modo, se define el **operador de borrado**:

$$B(a) = \lambda \text{ ó } \varepsilon \qquad \forall a \in T$$

de tal forma que

$$\begin{aligned} B^n(x) &= \{\varepsilon \text{ si } |x| < n\} \\ &= \{y/y \in T^* \wedge y \text{ resultado de eliminar } n \text{ caracteres en } x\} \end{aligned}$$

El error de borrado es muy fácil de detectar en lenguajes donde se definan todas las palabras de la misma longitud.

[1] Que sólo actúa sobre elementos *terminales*.

8.2.3. Corrección de errores de inclusión

El **operador de inclusión** se define:

$$I(a) = xa + ax \qquad \forall a \in T \quad \forall x \in T$$

de tal forma que si $T = \{a_1, a_2, \ldots, a_n\}$, entonces

$$I(a) = \{aa_1 + aa_2 + \ldots + aa_n + a_1a + a_2a + \ldots a_na\}$$

8.3. Modo pánico

Algunos compiladores, cuando encuentran un error, detienen el análisis. Otros, en cambio, buscan el principio y final de "sección" o "bloque" entre los que se encuentra el error, lo ignoran, y siguen analizando el resto del código. Este modo de comportarse es lo que se conoce como **modo pánico**.

Capítulo 9

La Tabla de Símbolos

Ya hemos hablado en alguna ocasión de las **tablas de símbolos**. Independientemente del modo de almacenamiento, la **tabla de símbolos** o **TDS** es una estructura utilizada por los compiladores para almacenar información sobre variables (nombre, tipo, ámbito,...) y procedimientos (nombre, parámetros de entrada y salida, carácter de los parámetros de entrada —por valor o por referencia—,...).

La mayoría de los compiladores mantiene una TDS por tipo de objeto (una para variables, otra para procedimientos,...), y suelen ser de carácter estático. Su utilidad es patente al principio del proceso de compilación, pues en el paso a código objeto ya dejan de ser necesarias.

Veremos a continuación una serie de formas de implementar las TDS.

9.1. Tipos

9.1.1. TDS lineal

Consiste en ir almacenando la información sobre los objetos según se va obteniendo, linealmente. La búsqueda se realiza, por tanto, de manera secuencial.

9.1.2. TDS ordenada

Se determina un ordenamiento, por ejemplo, el orden alfabétido de identificadores. Una vez definido, la inserción se realiza en el lugar adecuado, reestructurando la TDS en caso necesario, y la búsqueda suele ser *dicotómica*, rápida y fácil de implementar.

9.1.3. TDS árbol binario

La TDS árbol binario guarda una estructura en forma de árbol, tal como su nombre indica y se puede observar en la figura 9.1 (página 76). Pese a que se desperdicia un poco más de espacio adicional en los punteros, es uno de los tipos más utilizados, pues su implementación no es compleja y tanto inserción como búsqueda tienen la misma complejidad.

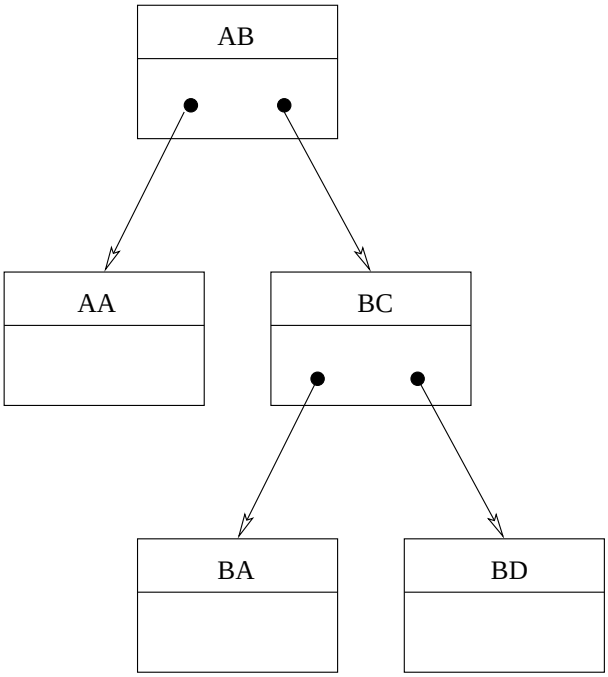


Figura 9.1: TDS en forma de árbol binario.

9.1.4. TDS con hashing

Dado un identificador C con n caracteres, $C = c_1 c_2 \ldots c_n$, la *función de hash* típica es:

$$Hashing(C) = (\alpha_0 + \sum_{i=0}^{n} \alpha_i c_i) \text{ modulo } T$$

donde

α_0	es la dirección o posición inicial, donde empieza en memoria la TDS
α_i	es el valor numérico asignado a c_i
T	es el tamaño de la TDS

Para otro tipo de claves (identificadores) no sólo compuestos por caracteres (como por ejemplo $i2$), podría adoptarse:

$$Hashing(C) = (ord(id[i] * 10) + ord(id[2])) \text{ modulo } T$$

Hashing abierto

Con la mayoría de las funciones de hash existe la posibilidad de *colisiones*. La técnica de hashing abierto establece que en caso de conflicto el valor se introduce en la siguiente posición libre, realizando la exploración linealmente a partir del lugar original. En búsqueda, si el valor que se pretende encontrar no se halla en la posición que le corresponde, se sigue buscando secuencialmente a partir de la misma. En la mayoría de las ocasiones se mantienen punteros de enlace por comodidad y rapidez.

Hashing con overflow

El hashing con overflow es otra variante que pretende solucionar el problema de las colisiones. En este caso, los valores conflictivos van a parar a una segunda tabla, donde se almacenan linealmente enlazados.

Hashing con encadenamiento

El hashing con encadenamiento perfeciona la técnica del hashing con overflow y establece que la primera TDS contenga sólo punteros la TDS de segundo nivel.

9.1.5. TDS enlazadas

Como hemos mencionado al principio, en la tabla de símbolos suelen almacenarse datos relativos a variables y procedimientos. Por lo que respecta a las variables, habíamos comentado que puede incluirse el nombre, tipo,... y también el alcance (bloques del programa en los que la variable está activa). El método de las TDS enlazadas evita la inclusión de este dato en la TDS definiendo una tabla por bloque del programa y relacionándolas entre sí.

Según este método, podemos definir un identificador (variable) en relación a su *amplitud*:

- Un identificador se declara al principio de un bloque[1] y se puede usar dentro de él.
- Un identificador declarado en un bloque puede ser udaso dentro de los bloques interiores a la definición, excepto cuando se hace otra definición de variable con el mismo bloque (en dicho bloque interior).

Este es el método que se usa en la mayoría de los compiladores actuales. La estrategia de búsqueda consiste en comprobar si el identificador está en la tabla del bloque en que se encuentra la referencia, y si no es así, seguir el puntero a la tabla del bloque que contiene al bloque actual y buscar allí. Se procede recursivamente hasta que se encuentra o se llega al bloque principal.

[1] En algunos lenguajes esto no es necesario que se cumpla estrictamente.

Capítulo 10

Representación de la Información y Gestión de Memoria

10.1. Representación de la Información

Aunque ya hemos dado pinceladas al respecto en temas anteriores, volveremos ahora a hablar con un poco más de detalle de la representación de las estructuras de información.

10.1.1. Representación de arrays y matrices

La representación de un array es simple, pues no es más que una colección de n objetos iguales que se ubican contiguos en memoria.

En el caso de las matrices (arrays bidimensionales) $M[b_1 \dots a_1, b_2 \dots a_2]$, tenemos dos funciones de acceso según su almacenamiento:

- *Almacenamiento por filas*:

  $$Dir(M_{ij}) = M + k \cdot [(i - b_1) \times C + (j - b_2)]$$

  donde M es la dirección base, k es el número de bytes por elemento y C es el número de elementos por fila o, lo que es lo mismo, el número de columnas.

- *Almacenamiento por columnas*:

  $$Dir(M_{ij}) = M + k \cdot [(i - b_2) \times F + (i - b_1)]$$

  donde M es la dirección base, k es el número de bytes por elemento y F es el número de elementos por columna o, lo que es lo mismo, el número de filas.

Representación con vectores enlazados

Otra forma de representar matrices es mediante vectores: se tiene un vector de punteros (uno por fila) que apuntan a su vez a los vectores-fila que forman el array bidimensional. Esta representación ocupa un poco más de espacio (el vector de punteros), pero requiere bastantes menos cálculos para acceder:

$$Dir(M_{ij}) = (vector + i - 1) + j - 1$$

donde $(vector + i - 1)$ es el valor del puntero correspondiente (la dirección). Como se puede ver, simplemente hay que hacer sumas y restas.

Representación de matrices de n dimensiones

Dada una matriz:

$$M[i_1 \ldots s_1, i_2 \ldots s_2, \ldots, i_n \ldots s_n]$$

donde

$$\begin{array}{rcl} i_i & = & \text{límites inferiores} \\ s_i & = & \text{límites superiores} \\ n & = & \text{número de dimensiones} \end{array}$$

y, obviamente $i_i \leq s_i$. Si llamamos

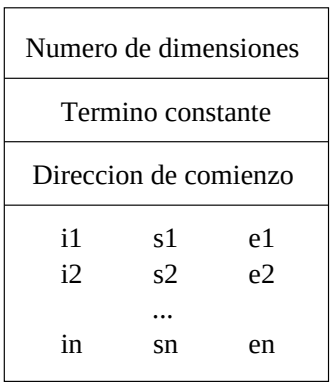
$$\begin{array}{rcl} e_1 & = & s_1 - i_1 + 1 \\ e_2 & = & s_2 - i_2 + 1 \\ & \ldots & \\ e_n & = & s_n - i_n + 1 \end{array}$$

entonces para acceder a $M[j_1, j_2, \ldots, j_n]$ debemos hacer

$$\begin{array}{rcl} M & + & (j_1 - i_1) * e_2 * e_3 * \ldots * e_n \quad + \\ & + & (j_2 - i_2) * e_3 * \ldots * e_n \quad + \\ & + & \ldots \quad + \\ & + & (j_{n-1} + i_{n-1}) * e_n \quad + \\ & + & (j_n + i_n) \end{array}$$

Vector "Dope" o Vector de Información

Para representar este tipo de estructura, veremos por último un método que consiste en almacenar un vector de información con la siguiente forma:



La utilidad de esta estrategia reside en la recopilación de la máxima información posible sobre las matrices, a fin de que no conlleve más cálculos el acceso que la propia operación que se pretendía realizar sobre ellas. Además, esta estructura permite la implementación del dinamismo con arrays bidimensionales de manera sencilla.

10.1.2. Representación de cadenas y tiras de caracteres

Las cadenas suelen almacenarse contiguas en memoria, determinando un carácter que señala el fin de las mismas. Los mayores problemas los presentan las operaciones de manejo de tiras de caracteres (copia, concatenación), ya que las ubicaciones de las cadenas originales en memoria pueden ser distantes y crear una nueva cadena en cada operación supone demasiado moviento de datos y reserva de espacio en el *data block*. Lo que se suele hacer es asociar las cadenas mediante punteros, evitando de esta manera el mencionado tráfico de operacones en memoria. Claro que esto puede derivar en una situación donde la proliferación de cadenas segmentadas y punteros sea demasiado costosa; en ese caso, algunos compiladores incluyen rutinas de reagrupamiento que se disparan automáticamente.

10.1.3. Representación de registros

Los registros (también conocidos como *records*) suelen almacenarse contiguos en memoria. En la TDS se guarda información acerca del número y tipo de sus campos, de modo que el acceso a un determinado dato siempre es fácilmente calculable.

10.2. Gestión de memoria proporcionada por el compilador

Los compiladores suelen proporcinar una serie de rutinas que son incorporadas al código del usuario y que se encargan de gestionar la detección de errores de ejecución (por ejemplo, divisiones por cero), de la gestión de variables locales y globales, del control del tamaño y situación de los objetos en la memoria, etc.

(a) M. principal. (b) M. proceso.

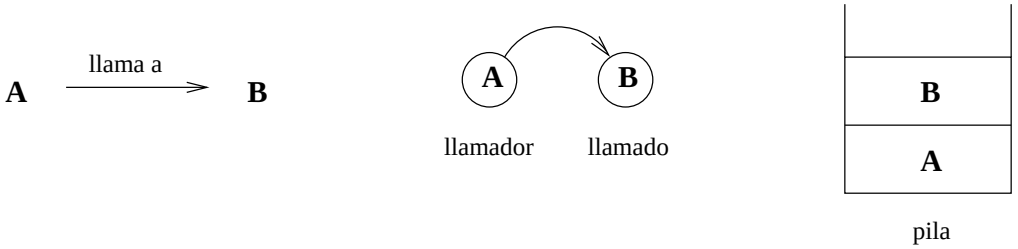
Las variables locales de los procedimientos se colocan en la pila, de forma que mecanismos como la recursión puedan funcionar adecuadamente. De aquí surge el concepto de *registro de activación*, que veremos.

Algunos lenguajes sólo tienen *gestión estática de memoria* (por ejemplo, en `F66` nada puede crecer en tiempo de ejecución y no existe la recursividad), mientras que la gran mayoría tiene *asignación dinámica*.

10.2.1. Registro de activación

Los **registros de activación** son la unidad estructural de la pila y almacenan:

- Datos locales del procedimiento
- Parámetros que se le pasan
- Datos temporales introducidos por el compilador
- Datos sobre el estado de la máquina antes de la ejecución del procedimiento (estado de la CPU: registros, flags,...)
- Puntero al registro de activación anterior



Un *procedimiento* es **recursivo** si se puede activar antes de la finalización de la activación anterior.

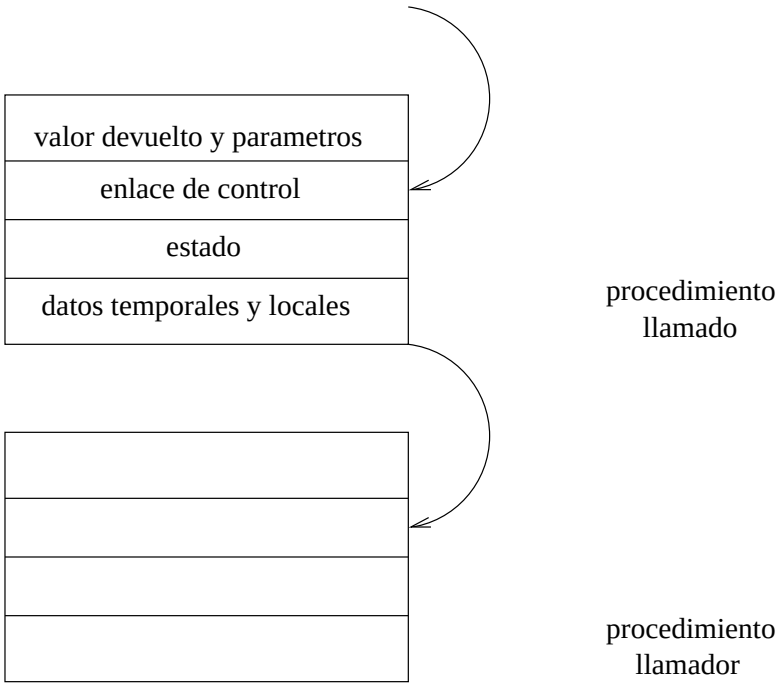
Los *registros de activación* se pueden organizar en forma de árbol, que recibe el nombre de **árbol de activación**. En este árbol, se dice que un nodo A es **padre** de un nodo B si el control fluye de A hacia B. Del mismo modo, se dice que un nodo A está **a la izquierda** de un nodo B si se ejecuta A antes que B.

10.2.2. Secuencia de llamada y retorno

La **secuencia de llamada** consiste en guardar en el registro de activación el estado de la máquina, asignar los valores a las variables locales y comenzar la ejecución del procedimiento.

Por su parte, la **secuencia de retorno** recupera los datos del campo `estado` del registro de activación para restaurar la situación anterior y borra el mencionado registro para continuar la ejecución.

Existen dos tipos de punteros implementados en la estructura de registros de activación: *estáticos* y *dinámicos*. Los estáticos permiten buscar variables a las que tiene acceso



el procedimiento pero que no son locales y los dinámicos son referencias al procedimiento llamador (son los que realmente permiten implementar la pila).

Las variables globales del programa se ubican en la parte de datos estáticos del proceso, de forma que se ahorra el tener que colocarlos repetidamente en los registros de activación y se consigue que estén accesibles para todos los subprocesos.

10.2.3. Subprogramas

Los argumentos con que se llama a un subprograma pueden ser:

- *Parámetros* o *Parámetros ficticios*, variables o identificadores que solamente aparecen en el subprograma que es llamado.
- *Argumentos* o *Parámetros efectivos*, variables o expresiones que aparecen en el procedimiento llamador.

Además, la llamada puede clasificarse:

- *Sólo lectura*: al llamar a una función $f(a)$, en f se trabaja con una copia de a.
- *Sólo escritura*: al llamar a $f(a)$, en f se trabaja con una copia de a cuyo valor inicial no está determinado (puede ser un valor por defecto) y al finalizar se copia su valor al verdadero a, que actúa, por tanto, como un parámetro de devolución.
- *Lectura y escritura*
  - *Por valor o resultado*: al llamar a $f(a)$, en f se lee de a y se escribe en una copia de a, volcándose su valor en el verdadero a al final del procedimiento.

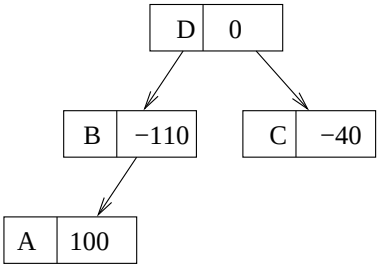
- *Por dirección*: al llamar a $f(a)$, en f se trabaja directamente con a, tanto para lectura como para escritura (se tiene la dirección).
- *Por nombre*: el procedimiento al que se llama no recibe una dirección o un contenido, sino un "literal", de forma que, según su contexto, le dará el valor correspondiente.

10.2.4. Estructuras `COMMON` y `EQUIVALENCE` en `FORTRAN`

En `FORTRAN` se pueden establecer relaciones de equivalencia entre variables con respecto a la posición física que ocupan en un bloque común (bloque que puede crecer dinámicamente). Esto se hace con sentencias del tipo:

```
EQUIVALENCE A, B+100
EQUIVALENCE C, D-40
EQUIVALENCE A, C+30
EQUIVALENCE E, F
```

Cada bloque tiene un *líder*, que se utiliza como raíz para construir el *árbol de equivalencia*.



Esta estructura viene del ensamblador, y los compiladores o lenguajes que la utilizan (como `FORTRAN`) emplean en realidad una variante en la que, en el árbol de equivalencia, el líder está unido mediante enlaces directos a la hoja más a la izquierda y a la hoja más a la derecha, que son respectivamente los extremos inferior y superior del bloque `COMMON`.

10.3. Montadores y Cargadores

Los **montadores** (***linker***) resuelven las referencias externas que existen en los objetos a montar, para lo que utilizan una tabla. Durante el proceso, se detectan errores ("`Unresolved external`") y si no aparecen enlazan todos los objetos en un único objeto entendible por el *cargador*.

Los **cargadores** (***loader***) cargan en memoria un objeto, resolviendo el problema de la reubicación. En los programas, existen *partes reubicables* y *partes no reubicables* (como el código de las interrupciones, por ejemplo). Tras cargarlo en memoria, el *cargador* da el control al programa, tras establecer punteros con valor adecuado (puntero de inicio de programa...), valores de los registros de la CPU, flags, etc.

10.4. Ficheros objeto

Contienen una serie de registros de la forma:

```
<cabecera><texto><checksum>
```

donde la `cabecera` contiene, a su vez, `<longitud><tipo>` y donde el `checksum` representa la comprobación de paridad para el control de errores.

Entre los tipos de registros que puede contener un *fichero objeto* podemos encontrar:

De definición de bloques Contienen:

```
<cabecera><datos_bloque><checksum>
```

donde

```
<cabecera>     = <longitud><tipo_definicion_bloque>
<datos_bloque> = <longitud><nombre_bloque><tipo_bloque>
```

donde el `tipo` del bloque se refiere a si es reubicable o no, por ejemplo.

De referencias externas Contienen:

```
<cabecera><datos_externos><checksum>
```

donde

```
<datos_externos> = <direccion><nombre_bloque_externo>
```

De texto con iteración Se usan para rellenar zonas de memoria con un dato:

```
<cabecera><bloque_iteracion><checksum>
```

donde

```
<bloque_iteracion> = <direccion><contador><contenido>
```

donde se repetirá `contenido contador` veces.

Otros Como registros de *principio de bloque*, *final de bloque*,...

10.5. Intérpretes

Fueron muy usados en un principio debido a que los equipos tenían poca memoria y era preferible tener cargado siempre el *intérprete* y el código fuente a tener un compilador y el código objeto; con el avance de la tecnología, no obstante, fue decayendo posteriormente casi por completo su utilización, frente a la de los compiladores. Actualmente, desde la aparición de `VBASIC` y sobre todo de `java`, están volviendo a resurgir.

Distinguimos dos tipos de *intérpretes*:

Clásicos Son intérpretes que van ejecutando el código fuente línea a línea tal y como está tecleado.

Modernos Son intérpretes que realmente generan un código intermedio sin errores, que es el que es interpretado línea a línea.

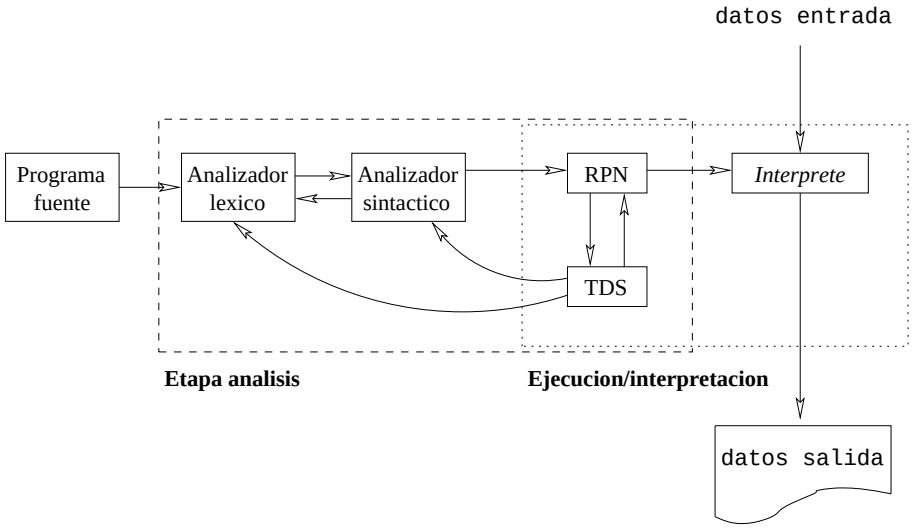


Figura 10.1: Estructura y funcionamiento de un intérprete moderno.

Un ejemplo:

```
/* CODIGO FUENTE */
A := 9;
B := 8;
B := B+A;
WRITE B;
GOTO X;

/* CODIGO RPN */
A 9 := ;;
B 8 := ;;
B B A + := ;;
B WRITE ;;
X GOTO ;;

/* INTERPRETE */
case v[pos] of
  ident, entero : apilar(v[op]); p:=p+1;
  '+'           : sumar PCP + PCP - 1 e introducir el resto en la pila
  ``write''     : representar en pantalla rep
  ``:=''        : evaluar elemento PCP y ponerlo en direccion PCP - 1
end
```

Índice alfabético

Índice de figuras

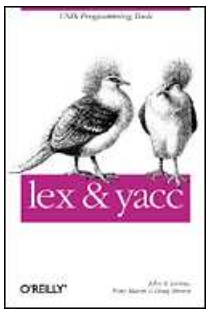
Índice de cuadros

Bibliografía

[1] Bennett, J.P.
Introduction to Compiling Techniques.
Mc-Graw Hill International, 1996.
2^a edición, Series in Software Engineering.

[2] Galán Pascual, C. Sanchís Llorca, F.J.
Compiladores. Teoría y construcción.
Paraninfo S.A., 1988.
Madrid.

[3] Mason, Tony y Levine, John y Brown, Dough.
Lex & Yacc.
O'Reilly, 1992.
2^a edición .



[4] Aho, Alfred V. y Sethi, Ravi y Ullman, Jeffrey D.
Compiladores. Principios, técnicas y herramientas.
Addison Wesley Longman, 1998.
1^a reimpresión .