

Contenidos

Tema 7 | Análisis Semántico

7.1 Función del analizador semántico.

7.2 Tabla de Símbolos.

7.2.1 Función.

7.2.2 Características.

7.2.3 Contenido.

7.2.4 Operaciones.

7.2.5 Estructura.

7.3 Gramática con atributos.

7.3.1 Métodos de evaluación de atributos.

7.3.2 Tipos de gramáticas con atributos.

7.4 Comprobaciones semánticas

7.4.1 Comprobación de tipos.

7.4.2 Tópicos en la comprobación de tipos.

7.5 Introducción de acciones semánticas en YACC.

7.5.1 Acciones en mitad de las reglas.

7.5.2 Recursividad.

7.5.3 Asignación de tipos a los elementos de la pila.

7.5.4 Ejemplos.

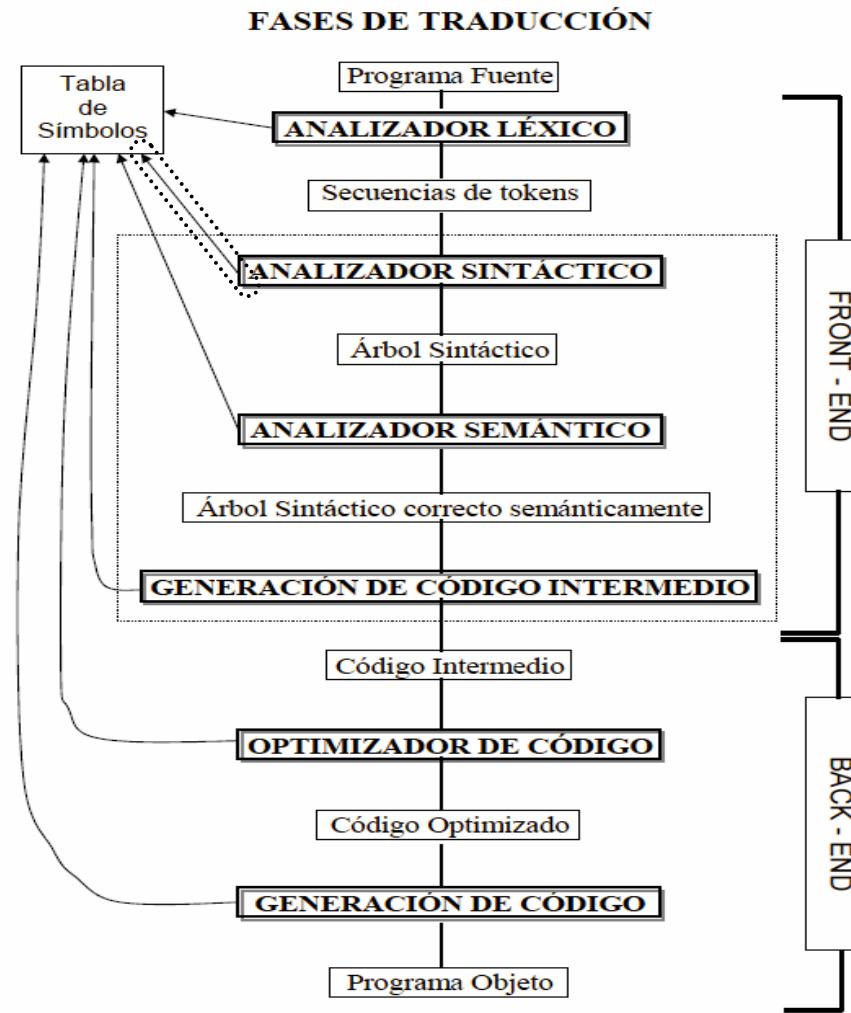
Bibliografía básica

20-Feb-2011

[Aho90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Compiladores. Principios, técnicas y herramientas. Addison-Wesley
Iberoamericana 1990.

[Levi92] J.R. Levine, T. Manson, D. Brown
Lex & Yacc. O'Reilly & Associates, Inc. 1992.

[Benn90] J.P. Bennet
Introduction to compiling techniques: A first course using ANSI C, LEX and YACC. Mc Graw-Hill 1990.



7.1 Función del analizador semántico

- Desde el punto del **analizador sintáctico**, lo único que se necesita identificar es que determinadas palabras (pertenecientes a un componente sintáctico) deben aparecer precedidas o seguidas por otras palabras.
- El **analizador semántico** debe:
 - Revisar el texto fuente para tratar de encontrar errores semánticos.
 - Reúne información de tipos necesaria para fases posteriores de análisis.
 - Verifica que cada operador tenga operandos permitidos (números enteros, números reales, matrices, cadenas, etc.)
 - Realiza otras comprobaciones semánticas

7.2.1 Función de la Tabla de Símbolos

- Un compilador usa la Tabla de Símbolos para llevar un registro de la información sobre el ámbito de los nombres.
- Se examina la Tabla de Símbolos cada vez que se encuentra un nombre en el texto fuente.
- Un mecanismo de la Tabla de símbolos debe permitir añadir entradas nuevas y encontrar las entradas existentes eficientemente.

7.2.2 Características de la Tabla de Símbolos

- Es una estructura volátil y dinámica.
- Simplifica el análisis sintáctico.
- Ayuda en la comprobaciones semánticas.
- Ayuda en la generación de código.

7.2.3 Contenido de la Tabla de Símbolos

- Esencialmente la información que aparece en la tabla de símbolos es de dos tipos:
 - El propio **símbolo**
 - **Atributos** necesarios para definir el símbolo a nivel semántico y de generación de código (variables, subprogramas, etiquetas, ...)
- Los atributos requeridos para cada símbolo dependen a nivel general:
 - Del tipo de **gestión de memoria** (dirección, ...)
 - Si el lenguaje está, o no, estructurado en **bloques**
 - Si el símbolo es, o no, **parámetro** de un procedimiento o función

7.2.4 Operaciones con la Tabla de Símbolos

1. El analizador de léxico deberá

- Crear la tabla de símbolos parcialmente.
- Insertar los símbolo detectados en la Tabla de Símbolos.
- Señalar la línea del programa fuente en donde aparecen.

2. El analizador semántico deberá

- Añadir los tipos, si procede, a los símbolos que aparecen en la tabla de símbolos.

7.2.4 Operaciones con la Tabla de Símbolos

- CREAR
- INSERTAR
- CONSULTAR
- MODIFICAR (añadir atributos nuevos)
- CREAR SUBTABLAS

7.2.4 Operaciones con la Tabla de Símbolos

Cuando y como se usan estas operaciones depende del tipo de lenguaje

- **Declaración de variables de forma explícita**
 - Declaraciones: sólo INSERTAR
 - Referencia: sólo CONSULTAR
- **Declaración de variables de forma implícita**
 - CONSULTAR si no está ya incluida
 - INSERTAR, en caso contrario
- **Lenguajes con estructura de bloques**
 - CREAR SUBTABLAS

7.2.5 Estructura la Tabla de Símbolos

La distribución de la información de la Tabla de Símbolos dependerá de las características del lenguaje y de las restricciones establecidas para los símbolos.

Campo dedicado para el símbolo

- *Formato fijo:*

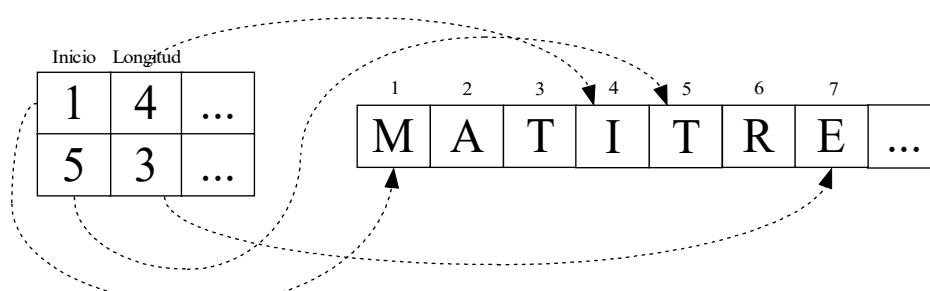
- Apropiado cuando se establece límite en el número de caracteres que forman los símbolos y, además, sea pequeño.
- En este caso sólo se dispone de un área fija en la tabla para almacenar el símbolo.

7.2.5 Estructura de la Tabla de Símbolos

Campo dedicado para el símbolo

- *Formato variable:*

- Se dispone de la Tabla de Símbolos y de un área auxiliar en donde se introducen los símbolos de modo consecutivo. En la Tabla de Símbolos se sustituye el campo dedicado para el nombre del símbolo por un puntero al área auxiliar y un entero que indica la longitud del mismo.



7.2.5 Estructura de la Tabla de Símbolos

Campo Dirección

- **Lenguajes SIN estructura de bloques.** Se asignan direcciones consecutivas según el orden en el que aparecen declaradas.
- **Lenguajes CON estructura de bloques.** Para cada bloque se asigna una subtabla; la dirección será consecutiva para cada bloque.

Se necesitan dos campos:

Nº Bloque	Dirección Bloque
-----------	------------------

Se introduce este campo en la Tabla de Símbolos cuando se declara. Se utiliza este atributo en la fase de generación de código.

7.2.5 Estructura de la Tabla de Símbolos

Campo Tipo

- Se introduce cuando se identifica una declaración explícita o implícita de una variable.
- Se utiliza para determinar la memoria de almacenamiento y para la comprobación de tipos.

Campo Nº de dimensiones / Nº de parámetros

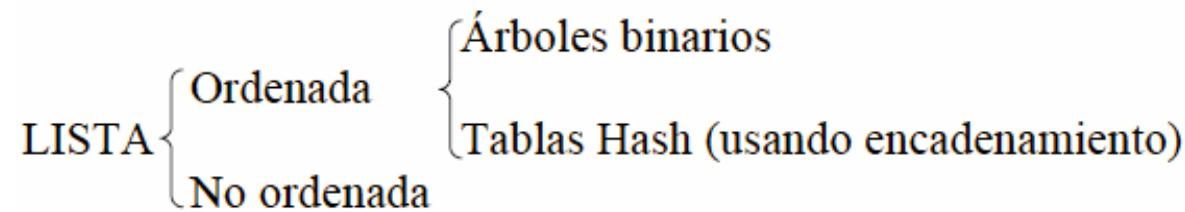
- La realiza el analizador semántico y sirve para delimitar el tamaño de memoria necesaria para representar el símbolo.
- Ejemplo: el tamaño o dimensión de un array; si se trata de una función o procedimiento, el número de argumentos que posee y sus tipos para la reserva de memoria.

Campo Lista Cruzada de Referencia

Campo Puntero de Orden

7.2.5 Estructura de la Tabla de Símbolos

- Organización en lenguajes SIN estructura de bloques



7.2.5 Estructura de la Tabla de Símbolos

- **Organización en lenguajes CON estructura de bloques**

- La tabla se estructura como una PILA cuyos elementos son las subtablas asociadas con los bloques.
- Además, se añade una tabla ÍNDICE que apunta al inicio de cada subtabla, indicando el comienzo y el final de cada bloque.
- Para el manejo de la estructura tipo pila se añaden dos operaciones:
 - **Set**: Para marcar el comienzo de cada bloque
 - **Reset**: Marca el fin de bloque

Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

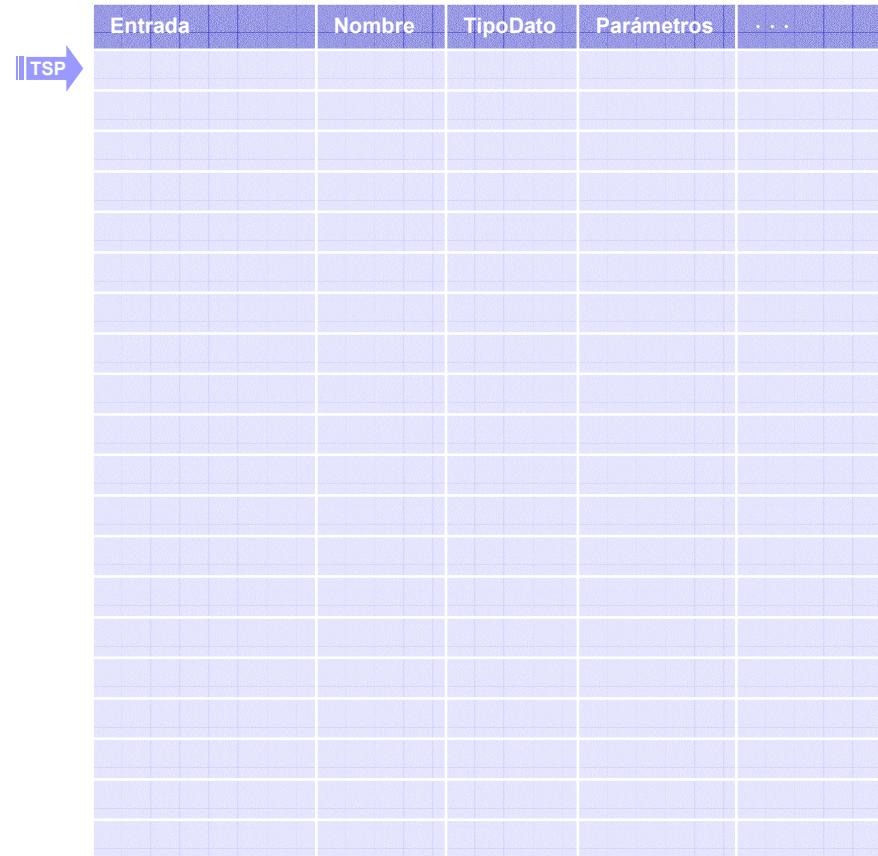
    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
    }

    scanf ("Lee datos", fa, fb, fc);
    ia= funcion1 (ib, ic);
}

```



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

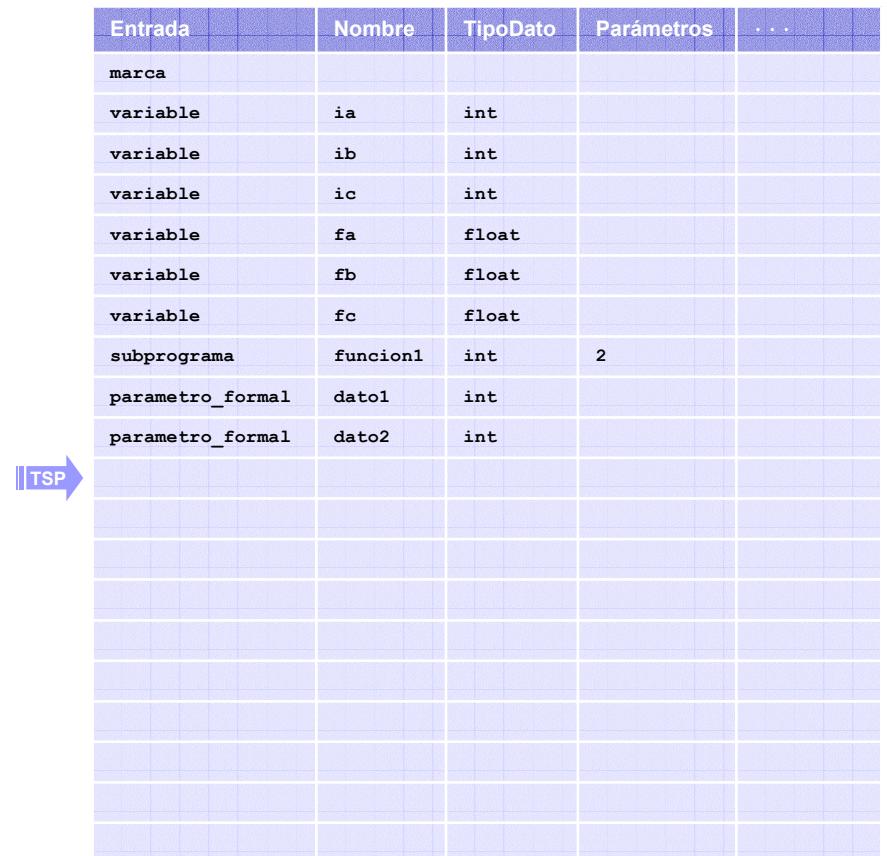
main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
        scanf ("Lee datos", fa, fb, fc);
        ia= funcion1 (ib, ic);
    }
}

```



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
        scanf ("Lee datos", fa, fb, fc);
        ia= funcion1 (ib, ic);
    }
}

```

Entrada	Nombre	TipoDatos	Parámetros	...
marca				
variable	ia	int		
variable	ib	int		
variable	ic	int		
variable	fa	float		
variable	fb	float		
variable	fc	float		
subprograma	funcion1	int	2	
parametro_formal	dato1	int		
parametro_formal	dato2	int		
marca				
variable	ca	char		
variable	cb	char		
subprograma	funcion2	char	2	
parametro_formal	dato1	char		
parametro_formal	dato2	char		



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
        scanf ("Lee datos", fa, fb, fc);
        ia= funcion1 (ib, ic);
    }
}

```

Entrada	Nombre	TipoDatos	Parámetros	...
marca				
variable	ia	int		
variable	ib	int		
variable	ic	int		
variable	fa	float		
variable	fb	float		
variable	fc	float		
subprograma	funcion1	int	2	
parametro_formal	dato1	int		
parametro_formal	dato2	int		
marca				
variable	ca	char		
variable	cb	char		
subprograma	funcion2	char	2	
parametro_formal	dato1	char		
parametro_formal	dato2	char		
marca				
variable	ca	char		



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
        scanf ("Lee datos", fa, fb, fc);
        ia= funcion1 (ib, ic);
    }
}

```

Entrada	Nombre	TipoDatos	Parámetros	...
marca				
variable	ia	int		
variable	ib	int		
variable	ic	int		
variable	fa	float		
variable	fb	float		
variable	fc	float		
subprograma	funcion1	int	2	
parametro_formal	dato1	int		
parametro_formal	dato2	int		
marca				
variable	ca	char		
variable	cb	char		
subprograma	funcion2	char	2	
parametro_formal	dato1	char		
parametro_formal	dato2	char		
marca				
variable	ca	char		

TSP

Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

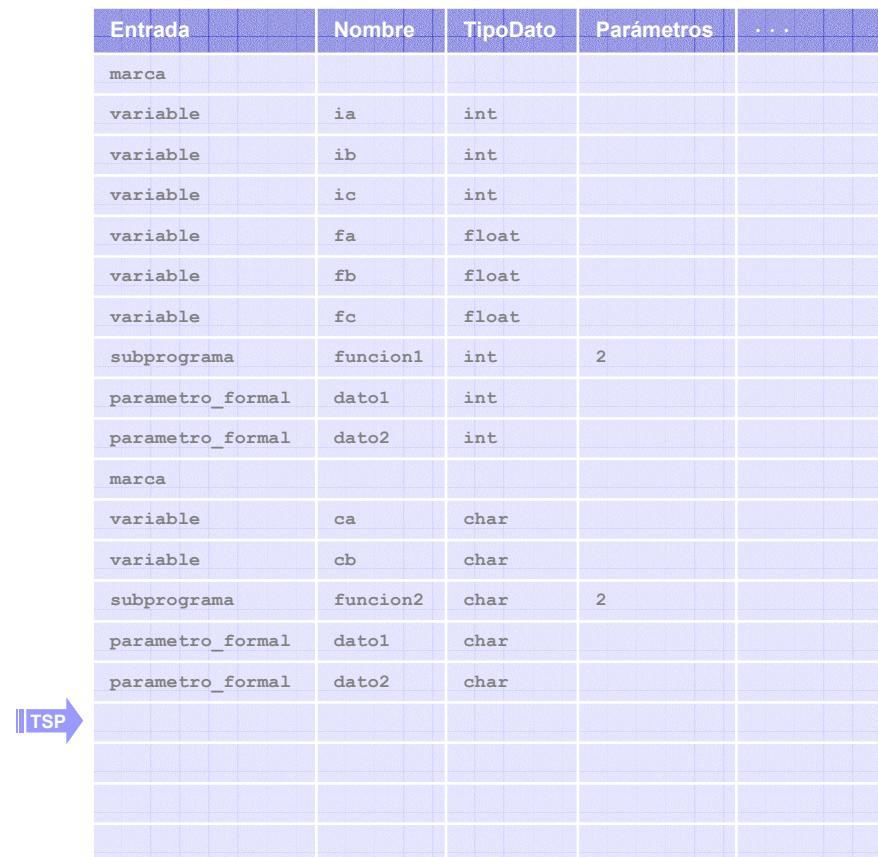
```
main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }

        scanf ("Lee datos", fa, fb, fc);
        ia= funcion1 (ib, ic);
    }
}
```



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
        scanf ("Lee datos", fa, fb, fc);
        ia= funcion1 (ib, ic);
    }
}

```

Entrada	Nombre	TipoDatos	Parámetros	...
marca				
variable	ia	int		
variable	ib	int		
variable	ic	int		
variable	fa	float		
variable	fb	float		
variable	fc	float		
subprograma	funcion1	int	2	
parametro_formal	dato1	int		
parametro_formal	dato2	int		
marca				
variable	ca	char		
variable	cb	char		
subprograma	funcion2	char	2	
parametro_formal	dato1	char		
parametro_formal	dato2	char		
marca				
variable	aux	char		



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
    }

    scanf ("Lee datos", fa, fb, fc);
    ia= funcion1 (ib, ic);
}

```

Entrada	Nombre	TipoDatos	Parámetros	...
marca				
variable	ia	int		
variable	ib	int		
variable	ic	int		
variable	fa	float		
variable	fb	float		
variable	fc	float		
subprograma	funcion1	int	2	
parametro_formal	dato1	int		
parametro_formal	dato2	int		
marca				
variable	ca	char		
variable	cb	char		
subprograma	funcion2	char	2	
parametro_formal	dato1	char		
parametro_formal	dato2	char		
marca				
variable	aux	char		

TSP

Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
    }

    scanf ("Lee datos", fa, fb, fc);
    ia= funcion1 (ib, ic);
}

```

Entrada	Nombre	TipoDatos	Parámetros	...
marca				
variable	ia	int		
variable	ib	int		
variable	ic	int		
variable	fa	float		
variable	fb	float		
variable	fc	float		
subprograma	funcion1	int	2	
parametro_formal	dato1	int		
parametro_formal	dato2	int		
marca				
variable	ca	char		
variable	cb	char		
subprograma	funcion2	char	2	
parametro_formal	dato1	char		
parametro_formal	dato2	char		

TSP

Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

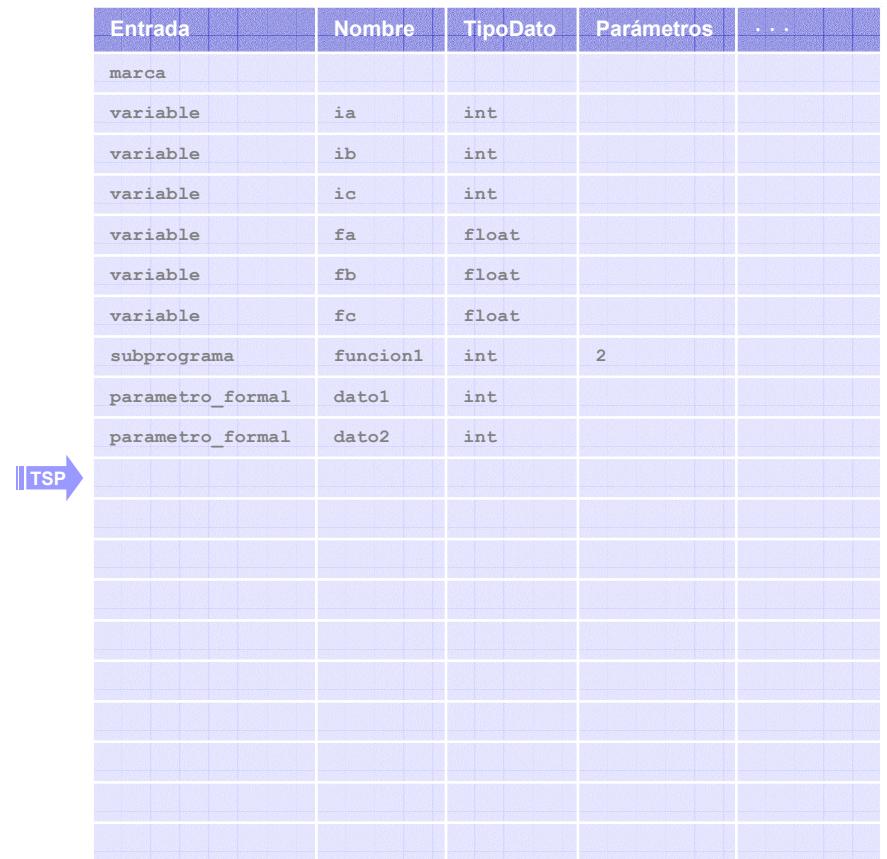
    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
    }

    scanf ("Lee datos", &fa, &fb, &fc) ;
    ia= funcion1 (ib, ic) ;
}

```



Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```
main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
    }

    scanf ("Lee datos", fa, fb, fc);
    ia= funcion1 (ib, ic);
}
```

Ejemplo 7.1: Evolución de la Tabla de Símbolos durante el análisis

```

main ()
{
    int ia, ib, ic ;
    float fa, fb, fc ;

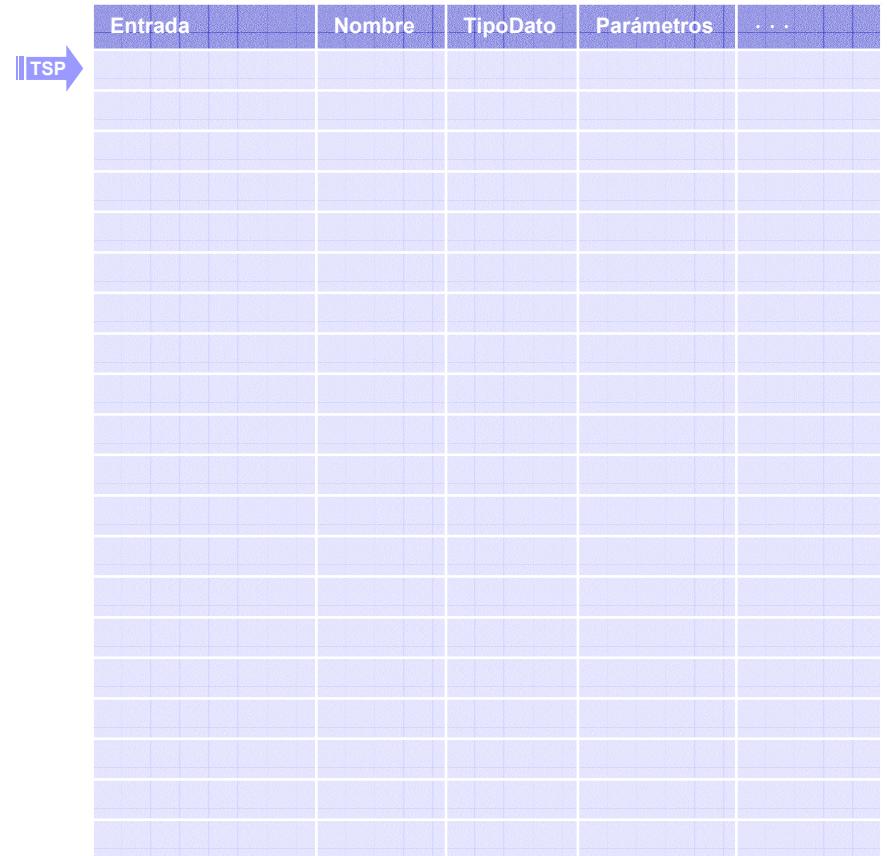
    int funcion1 (int dato1, int dato2)
    {
        char ca, cb ;

        char funcion2 (char dato1, char dato2)
        {
            char ca ;
            ca= dato1 ;
            if (ca==dato2)
                dato2= dato1 ;
            else
                dato2= ca ;
        }

        ca= funcion2 ('a', '&');
        while (ca!='&')
        {
            char aux ;
            aux= funcion2 ('$', '&');
        }
    }

    scanf ("Lee datos", fa, fb, fc);
    ia= funcion1 (ib, ic);
}

```



7.3 Gramática con atributos

Una **gramática con atributos** es una gramática en la que:

- A los **símbolos** (terminales y no terminales) se les puede asociar **atributos** (que deberán tomar valores en un dominio predeterminado).
- A las **producciones de la gramática** se les puede asociar **reglas de evaluación** de los atributos asociados con los símbolos que aparecen en la producción.
- Las reglas de evaluación asociadas con una producción son **ejecutadas** cuando la producción sea **aplicada** en el proceso de análisis.

7.3 Gramática con atributos

Sea $G = (T, N, P, S)$ una gramática libre del contexto y $C = \{c_1, c_2, \dots, c_n\}$ el conjunto de atributos asociados con los símbolos de la gramática G .

Dada una regla de evaluación $b = f(c_1, \dots, c_k)$ asociada con la producción $A \rightarrow \alpha \in P$

- **Atributo heredado:** Si b está asociado con algún símbolo de α
- **Atributo sintetizado:** Si b está asociado con el símbolo no terminal A

7.3 Gramática con atributos. Atributos sintetizados

Evaluación

- Las reglas de evaluación de los atributos sintetizados se suelen ejecutar cuando se realizan **reducciones en el análisis sintáctico ascendente**.

Requisitos para la evaluación

- Las reglas de evaluación de los atributos sintetizados deben definirse en función de los atributos asociados con los símbolos gramaticales a su derecha.

Gramática S-Atribuida

- Una gramática es S-Atribuida cuando todos sus atributos son sintetizados.

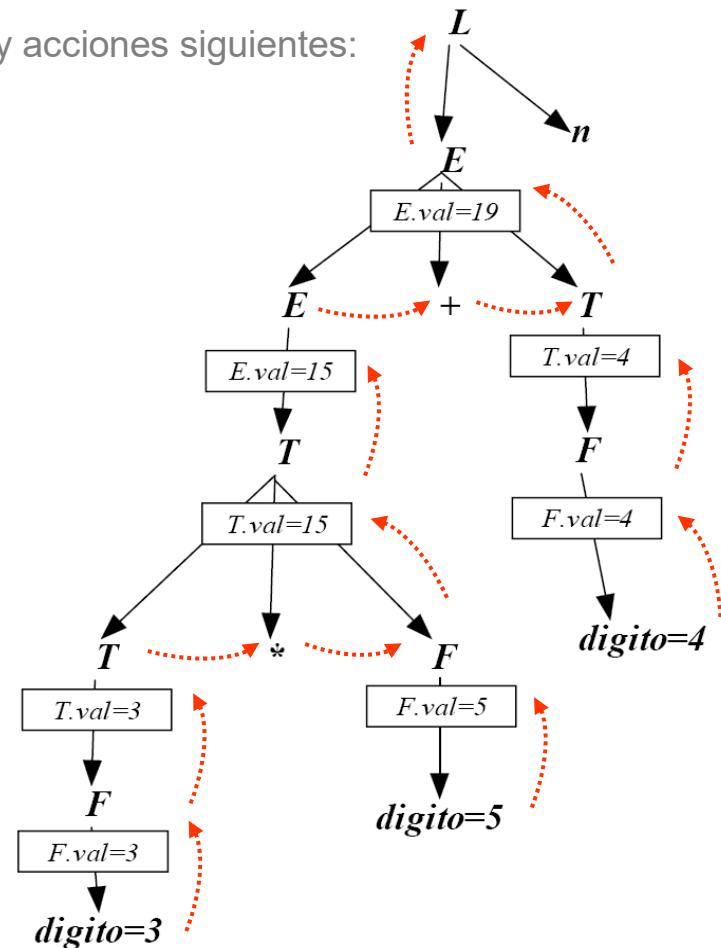
7.3 Gramática con atributos. Atributos sintetizados

Ejemplo 7.2: Gramática S-Atribuida. Sea el conjunto de producciones y acciones siguientes:

ACCIONES

$L \rightarrow E$	{ print ($E_1.val$) }
$E \rightarrow E + T$	{ $E_0.val = E_1.val + T_3.val$ }
$E \rightarrow T$	{ $E_0.val = T_1.val$ }
$T \rightarrow T * F$	{ $T_0.val = T_1.val * F_3.val$ }
$T \rightarrow F$	{ $T_0.val = F_1.val$ }
$F \rightarrow (E)$	{ $F_0.val = E_2.val$ }
$F \rightarrow \text{digito}$	{ $F_0.val = \text{digito}$ }

La evaluación de la expresión: $3 * 5 + 4n$



7.3 Gramática con atributos. Atributos heredados

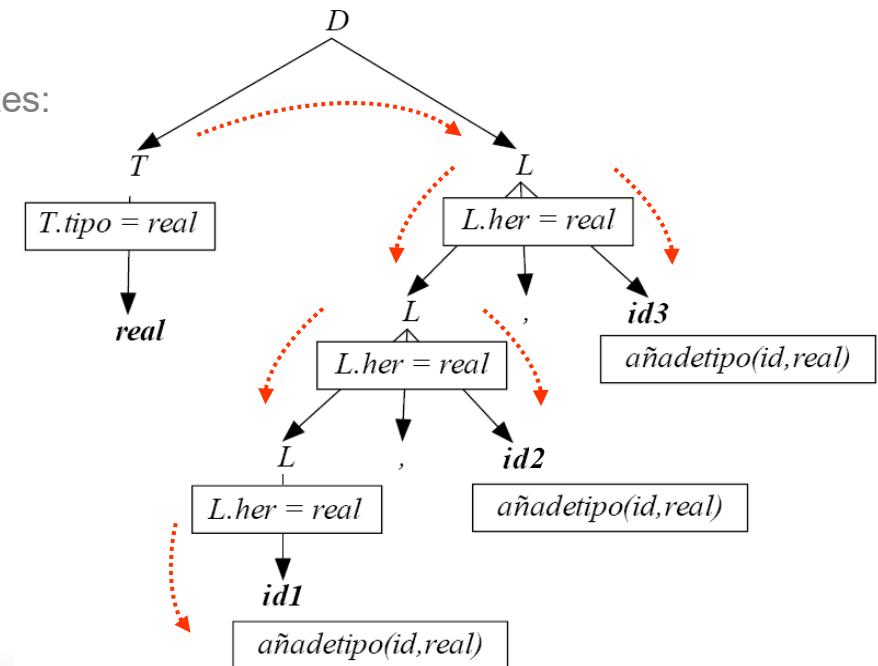
La evaluación de un **atributo heredado** depende de los atributos asociados con los símbolos precedentes en la derivación.

Los atributos heredados se suelen evaluar cuando se realizan **derivaciones en el análisis sintáctico descendente**.

Ejemplo 7.3: Sea el conjunto de producciones y acciones siguientes:

Producciones	Reglas
$D \rightarrow TL$	$L_2.in = T_1.tipo;$
$T \rightarrow \text{int}$	$T_0.tipo = \text{entero};$
$T \rightarrow \text{real}$	$T_0.tipo = \text{real};$
$L \rightarrow L, id$	$L_1.in = L_0.in;$ $\text{añadetipo}(id, L_0.in);$ $\text{añadetipo}(id, L_0.in);$
$L \rightarrow id$	

Real $id1, id2, id3$



7.3.1 Métodos de evaluación de los atributos

Métodos de evaluación de las reglas semánticas:

- **Árbol de análisis:** para cada entrada se construye el árbol sintáctico (grafos acíclicos).
- **Basado en las reglas semánticas:** Dependiendo de las reglas semánticas (atributos heredados o sintetizados) se establece el orden de evaluación.
- **Dirigido por sintaxis:** El orden de evaluación es impuesto por la estrategia de análisis.

7.3.2 Tipos de gramáticas con atributos

Gramáticas S-Atribuidas

Una gramática es S-Atribuida cuando todos sus atributos son sintetizados.

La estructura de la pila se adecua para que cada símbolo de la gramática disponga de sus atributos asociados.

La evaluación de los atributos se suele realizar cuando se **reduce en el análisis sintáctico ascendente**.

0
5	X	X.x, X.y,...X.z
6	Y	Y.u,...,Y.s
7	Z	Z.a,...,Z.c

7.3.2 Tipos de gramáticas con atributos

Gramáticas L-Atribuidas

Sea la producción $A \rightarrow X_1 X_2 \cdots X_n$. Una gramática es L-Atribuida si todos los atributos heredados asociados con X_j , $1 \leq j \leq n$, sólo dependen de:

1. Los atributos asociados con los símbolos $X_1 X_2 \cdots X_{j-1}$
2. Los atributos asociados con A

TODA GRAMÁTICA L-ATRIBUIDA PUEDE SER EVALUDA MEDIANTE ANÁLISIS ASCENDENTE.

7.3.2 Tipos de gramáticas con atributos

Esquemas de Traducción

Son gramáticas con atributos en que las acciones semánticas se insertan en las producciones entre { }

El lugar donde se insertan las acciones semánticas indica el lugar donde se pueden evaluar los atributos:

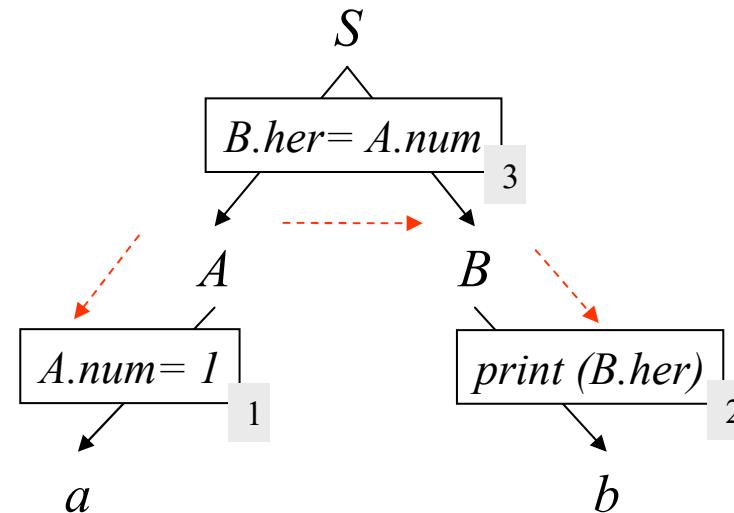
1. Un atributo **sintetizado** del símbolo no terminal a la izquierda de una producción sólo se puede evaluar cuando se han evaluado los atributos de los que depende. La acción usualmente se sitúa al final de la producción, si bien, también puede aparecer entre símbolos.
2. Una acción no debe referir un atributo **sintetizado** de un símbolo que esté en la derecha de la producción.
3. Un atributo **heredado** de un símbolo del lado derecho de la producción se debe evaluar en un acción que aparezca antes que el símbolo.

7.3.2 Tipos de gramáticas con atributos

Esquemas de Traducción

Ejemplo 7.4: Gramática con atributos que no cumple la condición 3 para atributos heredados para la secuencia de entrada *ab*.

$$\begin{array}{ll} S \rightarrow A B & \{ B.\text{her} = A.\text{num} \} \\ A \rightarrow a & \{ A.\text{num} = 1 \} \\ A \rightarrow c & \{ A.\text{num} = 0 \} \\ B \rightarrow b & \{ \text{print}(B.\text{her}) \} \\ B \rightarrow d & \end{array}$$



La acción “`print(B.her)`” se ejecutaría, bajo estas condiciones, antes de que se conociese el valor del atributo, es decir, cuando es fijado.

7.4 Comprobaciones semánticas

Comprobaciones **estáticas**:

Sintácticas y semánticas.

Comprobaciones **dinámicas**:

Realizadas en tiempo de ejecución.

7.4 Comprobaciones semánticas

Comprobaciones de:

- **Tipo:** Verificación del tipo de los operandos en las expresiones.
- **Flujo de control:** Verificación los puntos del programa de salida y entrada del control (ejemplo: **Break** en C).
- **Unicidad:** Verificación de la presencia de símbolos de forma única (ejemplo: declarar un símbolo una sola vez).
- **Relación de nombres:** Un mismo nombre puede aparecer más de una vez (ejemplo: nombre de bloque en Ada).

7.4 Comprobaciones semánticas

EXPRESIONES DE TIPOS

- **Tipos básicos** (tipos simples) son expresiones de tipo (ejemplo: boolean, char, integer, real, ...)
- Una expresión de tipos puede ser nombrada por un **nombre**. Un nombre de tipos es una expresión de tipos.

En una expresión de tipo pueden aparecer **variables de tipos**.

7.4 Comprobaciones semánticas

CONSTRUCTORES DE TIPOS

- **Matrices** (Arrays)

Array (I, T) : Expresión de tipo que representa el tipo de una matriz con elementos de tipo T y conjunto de índices I .

En Pascal:

```
VAR A: array [1..10] of integer;
```

Le asocia la expresión tipo: array (1..10, integer) a A

- **Producto:** Sea T_1 y T_2 expresiones de tipo, entonces $T_1 \times T_2$ (producto cartesiano) es también una expresión de tipo.

7.4 Comprobaciones semánticas

CONSTRUCTORES DE TIPOS

- **Registros:** Es como **Producto** pero con nombre.

Record ((dirección × integer) × (lexema × array(1..15,char)))	
PASCAL	Lenguaje C
<pre>type file=record direccion : integer ; lexema : array[1..15] of char ; end ;</pre>	<pre>typedef struct { int direccion ; char lexema[15] ; } file ;</pre>

7.4 Comprobaciones semánticas

CONSTRUCTORES DE TIPOS

- **Apuntadores:** Sea T una expresión de tipo, entonces `pointer(T)` es una expresión de tipo. `pointer(FILA) : apunta a un objeto tipo FILA`

↓
`VAR p : ^FILA`

- **Funciones:** Como una función matemática. Sean D y R dos expresiones de tipos entonces:

$D \rightarrow R$ es otra expresión de tipo

Ejemplo:

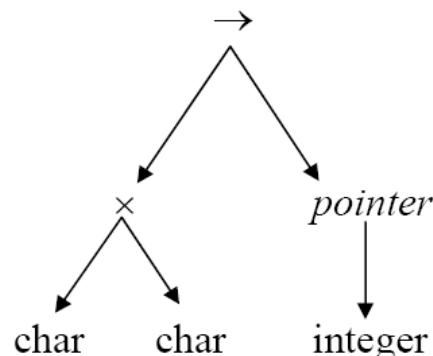
`char × char → pointer(integer)`
↓
`function f(a, b : char) : ^integer;`

7.4 Comprobaciones semánticas

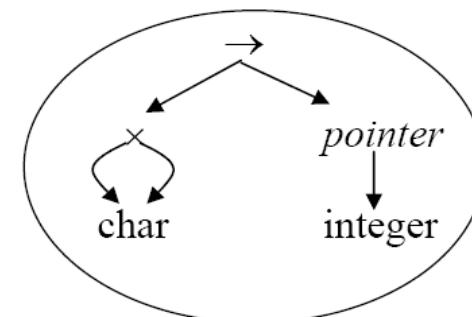
REPRESENTACIÓN DE LAS EXPRESIONES DE TIPOS

- Mediante grafo dirigido acíclico (GDA)

Ejemplo 7.5: Representación de la expresión de tipo $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$



hojas interiores: constructor de expresiones
o expresiones de tipos.
hojas finales: tipos básicos.



7.4 Comprobaciones semánticas

REPRESENTACIÓN DE LAS EXPRESIONES DE TIPOS

- Mediante **mapa de bits** (usado por D. M. Ritchie para el compilador de C). Se parte de los constructores básicos:

Codificación

$\text{pointer}(T)$: Expresión Puntero $\rightarrow 01$
 $\text{return}(T)$: Expresión función $\rightarrow 11$
 $\text{array}(T)$: Expresión array $\rightarrow 10$

Tipos básicos	Codificación
Boolean	$\rightarrow 0000$
Char	$\rightarrow 0001$
Integer	$\rightarrow 0010$
Real	$\rightarrow 0011$

Expresiones de Tipo	Mapa de Bit
char	000000 0001
$\text{return}(\text{char})$	000011 0001
$\text{pointer}(\text{return}(\text{char}))$	000111 0001
$\text{array}(\text{pointer}(\text{return}(\text{char})))$	100111 0001

La principal ventaja es que se ahorra espacio.

Dos secuencias de bits distintas representan expresiones de tipos distintos.

Tipos distintos podrían tener la misma secuencia de bits.

7.4 Comprobaciones semánticas

EQUIVALENCIA DE TIPOS

- **Equivalencia estructural:** Dos expresiones de tipos son equivalentes estructuralmente si responden a un mismo tipo básico o están formadas en base de aplicar el mismo constructor de tipos sobre expresiones equivalentes estructuralmente o sobre los mismos tipos básicos.

Modo de determinar si dos expresiones de tipos son equivalentes:

- Aplicando GDA
- Algoritmo de análisis de los constructores

7.4 Comprobaciones semánticas

EQUIVALENCIA DE TIPOS

Algoritmo de equivalencia estructural:

```
function equivest (s, t) : boolean ;
begin
    if s y t son el mismo tipo_básico 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 = s1×s2 and t = t1×t2 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 s = s1→s2 and t = t1→t2 then
        return equivest (s1,t1) and equivest (s2,t2)
    else
        return false ;
end;
```

7.4 Comprobaciones semánticas

EQUIVALENCIA DE TIPOS

- **Equivalencia de nombre:** Cuando se da nombre a las expresiones de tipos éstas son nombradas, entonces dos expresiones de tipos son **equivalentes de nombre** si y solo si ambas expresiones son idénticas.

Si aparecen expresiones de tipos a las que se ha dado nombre, también ha de ser reconsiderado el concepto de equivalencia estructural.

Dos expresiones de tipos son equivalentes estructuralmente si sustituidos los nombres de tipos por sus correspondientes expresiones de tipos, resultan dos nuevas expresiones de tipos que son equivalentes estructuralmente.

7.4.1 Comprobación de tipos

COMPROBADOR DE TIPOS ELEMENTAL

Un comprobador de tipos ha de disponer de:

- Asignación de tipos.
- Comprobador de tipos en las expresiones.
- Comprobador de tipos en las proposiciones o sentencias.
- Comprobador de tipos de las funciones.

Básicamente se advierten dos tareas:

- De asignación.
- De evaluación y comprobación.

7.4.1 Comprobación de tipos

Ejemplo 7.7: Sea un lenguaje sencillo que obedece a una gramática con el conjunto de producciones siguientes.

$$\begin{aligned} P &\rightarrow D ; E ; S \\ D &\rightarrow D ; D \\ D &\rightarrow id : T | \epsilon \\ T &\rightarrow \text{char} | \text{integer} | \text{array [num] of } T | ^T | T \rightarrow T \\ E &\rightarrow \text{literal} | \text{num} | id | E \text{ mod } E | E[E] | E^{\wedge} | E(E) | \epsilon \\ S &\rightarrow id := E | \text{if } E \text{ then } S | \text{while } E \text{ do } S | S ; S \end{aligned}$$

Las comprobaciones semánticas vendrán referidas a:

- (a) Asignación de tipos.
- (b) Comprobación de tipos en expresiones.
- (c) Comprobación de tipos en proposiciones.
- (d) Comprobación de tipos en funciones.

7.4.1 Comprobación de tipos

Ejemplo 7.7 (cont.)

(a) Asignación de tipos:

$P \rightarrow D ; E ; S$

$D \rightarrow D ; D$

$D \rightarrow id : T$

{ añadetipo(id.entrada, T.tipo); }

$T \rightarrow char$

{ T.tipo := char ; }

$T \rightarrow integer$

{ T.tipo := integer ; }

$T \rightarrow ^T_1$

{ T.tipo := pointer (T.tipo) ; }

$T \rightarrow array[num] of T_1$

{ T.tipo := array (1..num.val, T1.tipo) ; }

$T \rightarrow T_1 \rightarrow T_2$

{ T.tipo := T1.tipo → T2.tipo; }

7.4.1 Comprobación de tipos

Ejemplo 7.7 (cont.)

(b) Comprobación de tipos en expresiones:

$E \rightarrow \text{literal}$	{ $E.\text{tipo} := \text{char} ;$ }
$E \rightarrow \text{num}$	{ $E.\text{tipo} := \text{integer} ;$ }
$E \rightarrow \text{id}$	{ $E.\text{tipo} := \text{busca} (\text{id.entrada}) ;$ }
$E \rightarrow E_1 \text{ mod } E_2$	{ $E.\text{tipo} := \text{if } E_1.\text{tipo}=\text{integer} \text{ and } E_2.\text{tipo}=\text{integer}$ then integer else error_tipo; }
$E \rightarrow E_1[E_2]$	{ $E.\text{tipo} := \text{if } E_2.\text{tipo}=\text{integer} \text{ and } E_1.\text{tipo}=\text{array } (s, t)$ then t else error_tipo; }
$E \rightarrow E_1^{\wedge}$	{ $E.\text{tipo} := \text{if } E_1.\text{tipo}=\text{pointer } (t)$ then t else error_tipo ; }

7.4.1 Comprobación de tipos

Ejemplo 7.7 (cont.)

(c) Comprobación de tipos en proposiciones:

$S \rightarrow id := E$

```
{ id.tipo := busca (id.entrada);  
  S.tipo := if id.tipo=E.tipo  
            then vacio  
            else error_tipo; }
```

$S \rightarrow \text{if } E \text{ then } S_1$

```
{ S.tipo := if E.tipo=boolean  
      then S1.tipo  
      else error_tipo; }
```

$S \rightarrow \text{while } E \text{ do } S_1$

```
{ S.tipo := if E.tipo=boolean  
      then S1.tipo  
      else error_tipo; }
```

$S \rightarrow S_1 ; S_2$

```
{ S.tipo := if S1.tipo=vacio and S2.tipo=vacio  
      then vacio  
      else error_tipo; }
```

7.4.1 Comprobación de tipos

Ejemplo 7.7 (cont.)

(d) Comprobación de tipos en funciones: En la fase de declaraciones ha de ser definido el tipo.

$E \rightarrow E_1(E_2)$

{ $E.\text{tipo} := \text{if } E_2.\text{tipo} = s \text{ and } E_1.\text{tipo} = s \rightarrow t$
 then t
 else error_tipo; }

7.4.1 Comprobación de tipos

Ejemplo 7.8: Dadas las sentencias válidas para la gramática definida en el ejemplo anterior.

```
a : integer ;
b : char ;
a := b mod 18 ;
```

El árbol sintáctico para las sentencias de declaración es:

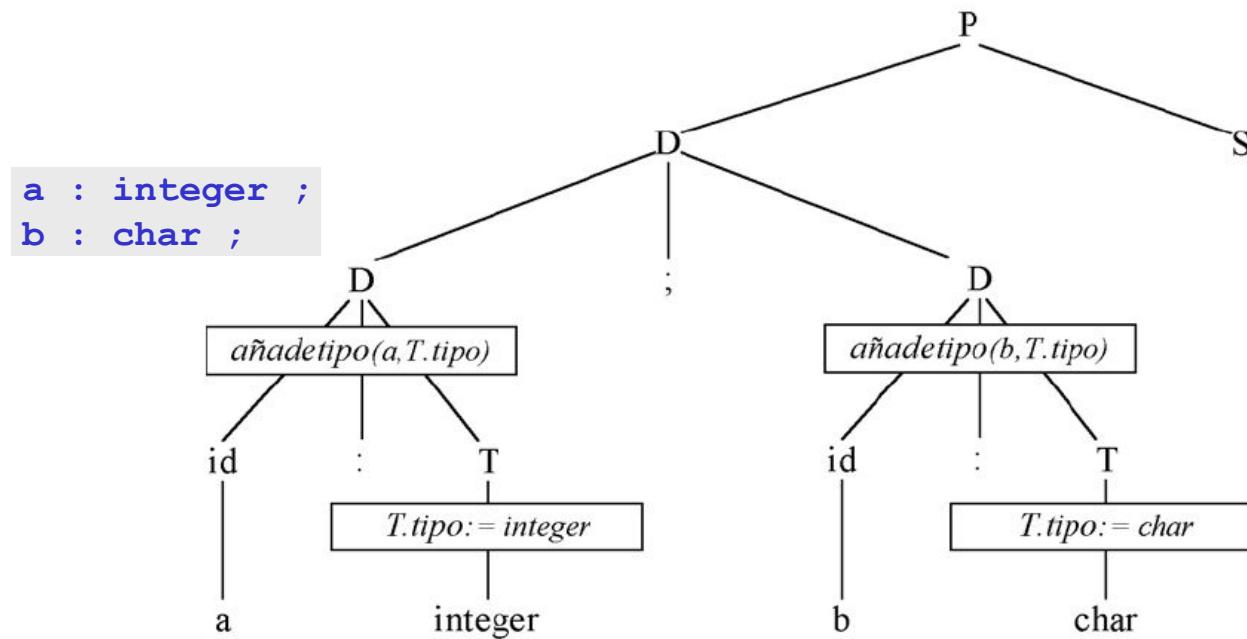


Tabla de símbolos	
...	...
a	integer
b	char
...	...

7.4.1 Comprobación de tipos

Ejemplo 7.8 (cont.)

De igual modo, el árbol sintáctico para la sentencia de asignación es:

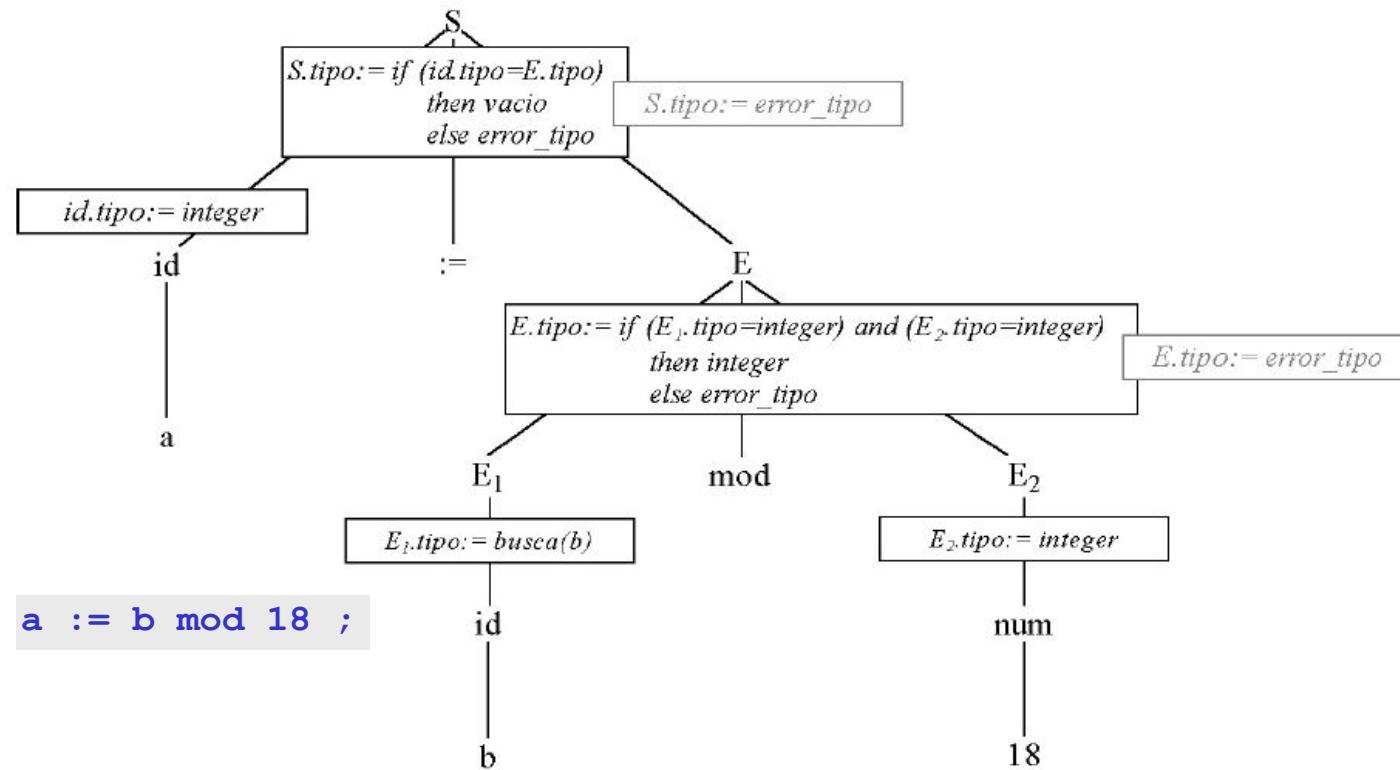


Tabla de símbolos	
...	...
a	integer
b	char
...	...

7.4.2 Tópicos en la comprobación de tipos

CONVERSIÓN DE TIPOS (coerción)

- **Implícita:** Se realiza automáticamente por medio del compilador.
- **Explícita:** El programador indica expresamente la conversión.

7.4.2 Tópicos en la comprobación de tipos

Ejemplo 7.9: A partir de una gramática que admite tipos real y entero, se muestra la conversión de ambos ante una operación binaria.

$E \rightarrow \text{ConstEntera}$

{ $E.\text{tipo} := \text{integer} ;$ }

$E \rightarrow \text{ConstReal}$

{ $E.\text{tipo} := \text{real} ;$ }

$E \rightarrow \text{id}$

{ $E.\text{tipo} := \text{busca} (\text{id.entrada}) ;$ }

$E \rightarrow E_1 \text{ op } E_2$

{ $E.\text{tipo} :=$ if $E_1.\text{tipo}=\text{integer}$ and $E_2.\text{tipo}=\text{integer}$
 then integer
 else if $E_1.\text{tipo}=\text{integer}$ and $E_2.\text{tipo}=\text{real}$
 then real
 else if $E_1.\text{tipo}=\text{real}$ and $E_2.\text{tipo}=\text{integer}$
 then real
 else if $E_1.\text{tipo}=\text{real}$ and $E_2.\text{tipo}=\text{real}$
 then real
 else error_tipo ; }

7.4.2 Tópicos en la comprobación de tipos

SOBRECARGA DE FUNCIONES Y OPERADORES

- **+,-**: Son símbolos sobrecargados ya que su significado dependerá del contexto (unarios y binarios).
- **(,)** : Pueden estar sobrecargados (Lenguajes como ADA y FORTRAN), ya que se usan como referencia de elementos de matrices y como funciones. La sobrecarga se resuelve, en algunos casos, en base a los tipos de las expresiones en donde aparecen (estudio del contexto).

7.5 Introducción de acciones semánticas en YACC

A cada regla de la gramática, el usuario puede asociar una serie de **acciones** (acciones semánticas), que se ejecutarán una vez **reconocida la regla sintáctica**, es decir, cuando se aplique la reducción de la misma en la fase de análisis sintáctico ascendente.

El formato es el siguiente:

```
A : CUERPO { sentencia/s }
```

Ejemplo 7.10: Una regla gramatical simple que incluye una acción semántica simple.

```
A : B C { printf ("Un mensaje\n") ;  
          a= 35 ;  
      }  
;
```

7.5 Introducción de acciones semánticas en YACC

Seudovariables: Son variables que emplea el YACC, por indicación del usuario, para pasar información de una regla sintáctica a otra y que pueden aparecer en las acciones semánticas.

- **$\$\$$** : Es la seudovariable asociada al símbolo no terminal de la izquierda de una regla gramatical.
- **$\$n$** : Es la seudovariable asociada al **símbolo** que ocupa el lugar **n-ésimo** dentro de la parte derecha de una regla gramatical (donde **n** es un entero mayor que 0).

Ejemplo 7.11: Una regla gramatical simple que incluye una acción semántica con utilización de seudovariables.

```
A      : B C D          { $$ = $1 * $3 ; } ;
Expr : `(` Expr `)'    { $$ = $2 ; } ;
```

7.5.1 Acciones en mitad de las reglas gramaticales

YACC permite introducir acciones al final de una regla o entre sus símbolos:

- a) A : B C D { acción } ;
- b) A : B { acción1 } C D { acción2 } ;

En este último caso, se transforma la regla de la siguiente forma:

```
 $$1 : { acción1 } ;
A : B $$1 C D { acción2 } ;
```

Ejemplo 7.12: Especificación de una gramática incluyendo acciones semánticas.

```
A : B { $$ = 1 } C D { x = $2 ; y = $3 } ;
pasa a ser:
```

```
 $$1 : { $$ = 1 } ;
A : B $$1 C D { x = $2; y = $3 } ;
```

Ejemplo 7.13: Especificación BYACC de una calculadora según la siguiente gramática:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \text{numero}$$

Especificación LEX

```
numero [0-9]+
%%
{numero}      return (NUMERO);
"-"          return (MINUS);
"+"
"*"
"/"
"("
")"
";"
["\n\t"]+
.
printf ("\nCarácter erróneo: %s\n", yytext);
%%
```

Especificación YACC

```
%token NUMERO
%left PLUS MINUS
%left MULT DIV
%right UNARIO

%start linea
%%
linea : Exp { printf ("Valor actual: %d\n", $1); }
          PCOMA linea
          |
;

Exp : EXP PLUS Exp { $$ = $1 + $3; }
     | EXP MINUS Exp { $$ = $1 - $3; }
     | EXP MULT Exp { $$ = $1 * $3; }
     | EXP DIV Exp { $$ = $1 / $3; }
     | PIZQ Exp PDCH { $$ = $2; }
     | MINUS Exp %prec UNARIO { $$ = -$2; }
     | NUMERO { sscanf (yytext, "%d", &$$); }
;

%%

#include "lexyy.c"
#include "msj.err".c

void yyerror (char *msj)
{ fprintf (stderr, msj); }

int main ()
{ yyparse(); return (0); }
```

Ejecuciones del ejemplo anterior:

```
(10-12)*25;
Valor actual: -50
3+4 ;
Valor actual: 7
(-12+(10+2))/4 ;
Valor actual: 0
```

Ejecuciones del ejemplo anterior introduciendo errores en las expresiones:

```
(10-12)25;
Error (5) esperaba PLUS MINUS MULT DIV PCOMA
```

```
3+;
Error (8) esperaba NUMERO PIZQ MINUS
```

```
(-12+(10+2)/4;
Error (6) esperaba PDCH PLUS MINUS MULT DIV
```

7.5.2 Recursividad

Las acciones de un símbolo S pueden referirse a valores obtenidos por acciones asociadas a símbolos que se encuentren en el contexto izquierdo de S. para ello utilizaremos **seudovariables** del tipo **$\$n$** , donde **n** represente un valor **negativo o cero**.

Ejemplo 7.13: Dada la siguiente gramática, asociaremos a Y un atributo cuyo valor sea el número de símbolos que hay a su izquierda.

$$\begin{aligned} S &\rightarrow Z W Y \quad | \quad Z R Y \\ &Z \rightarrow c d \quad | \quad d \\ &W \rightarrow W_1 a \quad | \quad d \\ &R \rightarrow b \\ &Y \rightarrow e \end{aligned}$$

$S \rightarrow Z W Y$	$\{ Y.i = Z.num + W.num ; \}$
$ \quad Z R Y$	$\{ Y.i = Z.num + R.num ; \}$
$Z \rightarrow c d$	$\{ Z.num = 2 ; \}$
$ \quad d$	$\{ Z.num = 1 ; \}$
$W \rightarrow W_1 a$	$\{ W.num = W_1.num + 1 ; \}$
$ \quad d$	$\{ W.num = 1 ; \}$
$R \rightarrow b$	$\{ R.num = 1 ; \}$
$Y \rightarrow e$	

Ejemplo 7.13 (cont.): La especificación YACC queda de la siguiente forma:

```
%token A B C D E

%%
S : Z W Y { printf ("Numero de simbolos: %d\n", $3); }
| Z R Y { printf ("Numero de simbolos: %d\n", $3); }
;
Z : C D { $$ = 2; }
| D { $$ = 1; }
;
W : W A { $$ = $1 + 1; }
| D { $$ = 1; }
;
R : B { $$ = 1; }
;
Y : E { $$ = $0 + $-1; }
```

\$0 indica la seudovariable del primer símbolo a la izquierda de donde aparezca el símbolo no terminal Y.
\$-1 indica la seudovariable del segundo símbolo a la izquierda de donde aparezca el símbolo no terminal Y.

7.5.3 Asignación de tipos a los elementos de la pila

Por defecto, los elementos de la pila son de tipo entero.

```
#ifndef YYSTYPE
typedef int YYSTYPE
#endif
```

Es posible cambiar el tipo de los elementos de dos formas:

- Asignar el tipo directamente a **YYSTYPE**.
- Declarar una **unión** y asignar a cada **símbolo de la gramática** el tipo deseado.

La declaración se incluye en la especificación YACC dentro de la zona de declaraciones con el siguiente formato:

```
%union {
    tipo_1 nombre_tipo_1
    tipo_2 nombre_tipo_2
    ...
    tipo_n nombre_tipo_n
}
```

7.5.3 Asignación de tipos a los elementos de la pila

(cont.)

La asignación de tipo a los distintos símbolos de la gramática también se realizará en la zona de declaraciones de la siguiente forma.

- Símbolos no terminales:

```
%type <nombre_tipo_i> simbNT_1, simbNT_2, ..., simbNT_n
```

- Símbolos terminales:

```
%token <nombre_tipo_i> simbT_1, simbT_2, ..., simbT_n
```

o también:

```
%left <nombre_tipo_i> simbT_1, simbT_2, ..., simbT_n  
%right <nombre_tipo_i> simbT_1, simbT_2, ..., simbT_n
```

Ejemplo 7.14: Evaluador de expresiones.

linea : exp ; linea | ;

exp : exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| '(' exp ')' '
| '-' exp
| ENTERO
| REAL
;

Atributos necesarios para la comprobación de tipos y la evaluación de la expresión, de acuerdo con la gramática anterior:

exp	exp.val	Enteros o reales
	exp.tipo	{entero, real, error}

ENTERO	ENTERO.tipo	entero
	ENTERO.valor	Enteros

REAL	REAL.tipo	real
	REAL.valor	Reales

Ejemplo 7.14 (cont.): La especificación de gramática con atributos quedaría de la siguiente forma:

```

línea : exp { if(exp.tipo = error)
    then IMPRIMIR ("Error de tipos en la linea")
    else IMPRIMIR (exp.val);
}

';' linea | ;

exp   : exp1 '+' exp2 if(exp1.tipo = exp2.tipo)
        then exp.tipo = exp1.tipo
        else exp.tipo = error
        exp.val= exp1.val + exp2.val

    | exp1 '-' exp2 if(exp1.tipo = exp2.tipo)
        then exp.tipo = exp1.tipo
        else exp.tipo = error
        exp.val= exp1.val - exp2.val

    | exp1 '*' exp2 if(exp1.tipo = exp2.tipo)
        then exp.tipo = exp1.tipo
        else exp.tipo = error
        exp.val= exp1.val * exp2.val

```

exp ₁ '/' exp ₂ if(exp ₁ .tipo = exp ₂ .tipo)	
then exp.tipo = exp ₁ .tipo	
else exp.tipo = error	
exp.val= exp ₁ .val / exp ₂ .val	
'(' exp ₁ ')'	exp.tipo = exp ₁ .tipo
	exp.val = exp ₁ .val
'-' exp ₁	exp.tipo = exp ₁ .tipo
	exp.val = - exp ₁ .val
ENTERO	exp.tipo = ENTERO.TIPO;
	exp.val = ENTERO.val
REAL	exp.tipo = REAL.tipo
	exp.val = REAL.val
;	

Ejemplo 7.14 (cont.): La especificación YACC de la gramática con atributos quedaría de la siguiente forma:

```
%{  
/* Tipo del atributo val para exp */  
  
typedef union valor {  
    int entero;  
    float real;  
};  
  
/* Tipo de los atributos de exp. Tipo para exp.tipo y valor  
para exp.val */  
  
typedef struct intervalo {  
    int tipo;  
    union valor val;  
};  
  
/* simbolo tipo para los simbolos terminales  
atributo tipo para los simbolos no terminales */  
  
%union {  
    int simbolo;  
    struct intervalo atributo;  
}
```

```
/* Asignación de los tipo a los simbolos de la gramatica */  
%token <simbolo> ENTERO REAL  
%type <atributo> exp  
%left '+' '-'  
%left '*' '/'  
%right UNARIO  
  
%Start linea  
  
%
```

Ejemplo 7.14 (cont.): La especificación YACC de la gramática con atributos quedaría de la siguiente forma:

```
linea : exp {
    switch ($1.tipo)
    {
        case 1: printf(" El valor de la linea actual
                        es:%d\n", $1.val.entero);
        break;
        case 2: printf(" El valor de la linea actual
                        es:%f\n", $1.val.real);
        break;
        case 3: printf(" Error en los tipos de la
                        expresión");
        break;
    }
}

';' linea | ;

exp : exp '+' exp { if ($1.tipo==$3.tipo && $1.tipo==1)
    {
        $$ .val.entero=$1.val.entero+$3.val.entero;
        $$ .tipo=1;
    }
    else {
        if ($1.tipo == $3.tipo && $1.tipo == 2)
        {
            $$ .val.real = $1.val.real + $3.val.real;
            $$ .tipo=2;
        }
        else $$ .tipo=3;
    }
}
```

Ejemplo 7.14 (cont.): La especificación YACC de la gramática con atributos quedaría de la siguiente forma:

```
| exp '-' exp { if ($1.tipo == $3.tipo && $1.tipo == 1)
| {
|     $$.val.entero=$1.val.entero - $3.val.entero;
|     $$.tipo=1;
| }
| else {
|     if ($1.tipo == $3.tipo && $1.tipo == 2)
|     {
|         $$.val.real = $1.val.real - $3.val.real;
|         $$.tipo = 2;
|     }
|     else $$.tipo=3;
| }
|
| exp '*' exp { if ($1.tipo == $3.tipo && $1.tipo == 1)
| {
|     $$.val.entero=$1.val.entero*$3.val.entero;
|     $$.tipo = 1;
| }
| else {
|     if ($1.tipo == $3.tipo && $1.tipo == 2)
|     {
|         $$.val.real = $1.val.real * $3.val.real;
|         $$.tipo = 2;
|     }
|     else $$.tipo = 3;
| }
```

Ejemplo 7.14 (cont.): La especificación YACC de la gramática con atributos quedaría de la siguiente forma:

```
| exp '/' exp { if ($1.tipo == $3.tipo && $1.tipo == 1)
{   $$ .val.entero=$1.val.entero / $3.val.entero;
$$ .tipo = 1;
}
else {
if ($1.tipo == $3.tipo && $1.tipo == 2)
{
$$ .val.real = $1.val.real / $3.val.real;
$$ .tipo = 2;
}
else $$ .tipo = 3;
}
}

| '(' exp ')' { if ($2.tipo == 1)
$$ .val.entero=$2.val.entero;
else
$$ .val.real = $2.val.real;
$$ .tipo = $2.tipo;
}

| '-' exp %prec UNARIO
{ if ($2.tipo==1)
$$ .val.entero=- $2.val.entero;
else $$ .val.entero == $2.val.entero;
$$ .tipo = $2.tipo;
}

| ENTERO { $$ .tipo = 1;
sscanf(yytext,"%d",&$$ .val.entero);
}

| REAL { $$ .tipo = 2;
sscanf(yytext,"%E",&$$ .val.real);
}

;

%
```

Ejemplo 7.15: Revisión del ejemplo 7.14 (evaluador de expresiones) incluyendo 26 variables de la a a la z, tratamiento de errores de división por cero y operador asignación (permite asignaciones múltiples en una sola línea).

```
#{
double mem[26]; /* Memoria para las variables 'a'...'z'*/
#include <math.h>
#include <stdio.h>
void yyerror(char*);
void error_en_float(void);
}

union {
    double valor; /* valor actual */
    int indice; /* indice de mem[] */
}

ttoken <valor> NUM
ttoken <indice> VAR
ttype <valor> exp
tright '='
tleft '-' '+'
tleft '*' '/'
tleft NEG /* Para darle precedencia al menos unario. */
tright '^'

%%
input : /* vacio */
| input linea
;

linea : '\n'
| exp '\n' { printf("\t%.10g\n", $1); }
| error '\n' { yyerror("Linea sin datos\n"); }
;

exp : NUM { $$ = $1; }
| VAR { $$ = mem[$1]; }
| VAR '=' exp { $$ = mem[$1] = $3; }
| exp '+' exp { $$ = $1 + $3; }
| exp '-' exp { $$ = $1 - $3; }
| exp '*' exp { $$ = $1 * $3; }
| exp '/' exp {
        if ($3==0.0)
            yyerror("División por cero");
        $$=$1 /$3;
    }
| '-' exp tprec NEG { $$ = -$2; }
| exp '^' exp { $$ = pow ($1, $3); }
| '(' exp ')' { $$ = $2; }
;
%%
```

```
#include "yylex.c"

#include <signal.h>
#include <setjmp.h>
jmp_buf inicio;

/* función principal */

void main(void)
{
    printf("\nCalculadora 2: Escriba una expresión
aritmética y pulse <intro>");
    printf("\nOperadores permitidos: +,-,*,/,^,-
unario, y ()");
    printf("\nNuevas opciones:");
    printf("\n\tPermite uso de variables
desde 'a' a 'z'");
    printf("\n\tComprueba la división por cero");
    printf("\n\tRecupera errores de punto flotante
en ejecución");
    printf("\nPulse <control>-C (EOF) para salir\n");
    setjmp(inicio);
    signal(SIGFPE, error_en_float);
    yyparse();
}

/* funciones de error */

void yyerror (char *s)
{
    printf ("%s\n", s);
    longjmp(inicio,0);
}

void error_en_float(void)
{
    yyerror("Error trabajando con números reales");
}
```

Ejemplo 7.16: Especificación Lex Yacc para un lenguaje sin estructura de bloques con gestión de tabla de símbolos.

ANALIZADOR LÉXICO

```
%{
/* 
 * Analizador de Léxico.
 */
%}

blanco [ \t]
digito [0-9]
letra [a-zA-Z]

%t
{blanco}+ ;
\n
entero return (TENTERO);
real return (TREAL);
inicio return (INICIO);
fin return (FIN);

(letra)((letra)|(digito))* ( yyval.lexema= strdup(yytext);
                            return (ID);
                            )

(digito)+ (digito)+"."(digito)*
"+"
"-"
"*"
"/"
"="
"["
"]"
";"
","
""

(
printf ("\nError en linea %d.\n"
        "Carácter desconocido %s\n",
        yylineno, yytext);
)
```

ANÁLISIS SINTÁCTICO Y SEMÁNTICO

```
%{
#include <stdio.h>
#include <string.h>

/* Los ATRIBUTOS usados son el tipo de los elementos y el lexema */
typedef enum {entero, real, desconocido, no_asignado} dtipo ;

typedef struct {
    int token ;           /* Código del token */
    dtipo tipo ;         /* Tipo del token */
    char lexema ;        /* Nombre del token (lexema) */
} atributos;

#define YYSTYPE atributos

/*********************DEFINICIÓN DE LOS ELEMENTOS DE LA TABLA DE SÍMBOLOS****/
/* DEFINICIÓN DE LOS ELEMENTOS DE LA TABLA DE SÍMBOLOS */
/*********************DEFINICIÓN DE LOS ELEMENTOS DE LA TABLA DE SÍMBOLOS */

#define MAX_TS 900

typedef struct entrada_ts {
    char nombre ;          /* nombre del identificador */
    dtipo tipo_ts ;        /* Tipo del identificador */
} ;

struct entrada_ts TS[MAX_TS] ;

int i=0 ; /* Índice de la tabla de simbolos */
```

Ejemplo 7.16: (cont.)

Continuación

```
%{
/* ACCIONES SOBRE LA TABLA DE SÍMBOLOS */
/*
 * INTRODUCE un identificador en la tabla de símbolos
 * con el tipo "no_asignado"
 */

void introTS (char *identificador)
{
    int j= 0;
    int e= 0;

    while ((j<=(i-1)) && (i!=0))
    {
        if (!strcmp(TS[j].nombre, identificador))
        {
            e = 1;
            printf ("\nError en la linea %d. Identificador
duplicado: %s\n", yylineno, identificador);
            break ;
        }
        else j++ ;
    }
    if (e==0)
    {
        TS[i].nombre = strdup (identificador);
        TS[i].tipo_ts = no_asignado ;
        i++ ;
    }
}
```

```
/*
 * BUSCA un identificador en la TS y nos devuelve el tipo.
 * Si no existe, el tipo es "desconocido".
 */

dtipo buscaTS (char *identificador)
{
    int j= 0;
    int e= 0;
    dtipo tip ;

    while ((j<=(i-1)) && (i!=0))
    {
        if (!strcmp(TS[j].nombre, identificador))
        {
            tip= TS[j].tipo_ts ;
            e= 1 ;
            break ;
        }
        else j++ ;
    }
    if (e==0)
    {
        printf ("\nError en la linea %d. Identificador
no declarado %s\n", yylineno, identificador);
        tip= desconocido;
    }
    return tip;
}
```