Capítulo 6

Tabla de símbolos

6.1 Visión general

También llamada «tabla de nombres» o «tabla de identificadores», se trata sencillamente de una estructura de datos de alto rendimiento que almacena toda la información necesaria sobre los identificadores de usuario. Tiene dos funciones principales:

- Efectuar chequeos semánticos.
- Generar código.

Además, esta estructura permanece en memoria sólo en tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración. Los intérpretes también suelen mantener la tabla de símbolos en memoria durante la ejecución, ya que ésta se produce simultáneamente con la traducción.

La tabla de símbolos almacena la información que en cada momento se necesita sobre las variables del programa; información tal como: nombre, tipo, dirección de localización en memoria, tamaño, etc. Una adecuada y eficaz gestión de la tabla de símbolos es muy importante, ya que su manipulación consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica.

La tabla de símbolos también sirve para guardar información referente a los tipos de datos creados por el usuario, los tipos enumerados y, en general, cualquier identificador creado por el usuario. En estos casos, el desarrollador puede optar por mezclar las distintas clases de identificadores en una sola tabla, o bien disponer de varias tablas, donde cada una de ellas albergará una clase distinta de identificadores: tabla de variables, tabla de tipos de datos, tabla de funciones de usuario, etc. En lo que sigue nos vamos a centrar principalmente en las variables de usuario.

6.2 Información sobre los identificadores de usuario

La información que el desarrollador decida almacenar en esta tabla dependerá de las características concretas del traductor que esté desarrollando. Entre esta información puede incluirse:

- Nombre del elemento. El nombre o identificador puede almacenarse limitando o no la longitud del mismo. Si se almacena con límite empleando un tamaño máximo fijo para cada nombre, se puede aumentar la velocidad de creación y manipulación, pero a costa de limitar la longitud de los nombres en unos

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

casos y desperdiciar espacio en la mayoría. El método alternativo consiste en habilitar la memoria que necesitemos en cada caso para guardar el nombre. En C esto es fácil con el tipo char *; si hacemos el compilador en Modula-2, por ejemplo, habría que usar el tipo ADDRESS. En el caso de Java, esto no entraña dificultad alguna gracias al tipo String. Las búsquedas en la tabla de símbolos suelen hacerse por este nombre; por ello, y para agilizar al máximo esta operación, la tabla de símbolos suele ser una tabla de dispersión (*hash*) en la que las operaciones de búsqueda e inserción poseen un coste aproximadamente constante (≈ O(1)).

- Tipo del elemento. Cuando se almacenan variables, resulta fundamental conocer el tipo de datos a que pertenece cada una de ellas, tanto si es primitivo como si no, con objeto de poder controlar que el uso que se hace de tales variables es coherente con el tipo con que fueron declaradas. El capítulo 7 está exclusivamente dedicado a esta gestión.

- Dirección de memoria en que se almacenará su valor en tiempo de ejecución. Esta dirección es necesaria, porque las instrucciones que referencian a una variable deben saber donde encontrar el valor de esa variable en tiempo de ejecución con objeto de poder generar código máquina, tanto si se trata de variables globales como de locales. En lenguajes que no permiten la recursión, las direcciones se van asignando secuencialmente a medida que se hacen las declaraciones. En lenguajes con estructuras de bloques, la dirección se da con respecto al comienzo del área de memoria asignada a ese bloque (función o procedimiento) en concreto.

- Valor del elemento. Cuando se trabaja con intérpretes sencillos, y dado que en un intérprete se solapan los tiempos de compilación y ejecución, puede resultar más fácil gestionar las variables si almacenamos sus valores en la tabla de símbolos. En un compilador, no obstante, la tabla de símbolos no almacena nunca el valor.

- Número de dimensiones. Si la variable a almacenar es un array, también pueden almacenarse sus dimensiones. Aunque esta información puede extraerse de la estructura de tipos, se puede indicar explícitamente para un control más eficiente.

- Tipos de los parámetros formales. Si el identificador a almacenar pertenece a una función o procedimiento, es necesario almacenar los tipos de los parámetros formales para controlar que toda invocación a esta función sea hecha con parámetros reales coherentes. El tipo de retorno también se almacena como tipo del elemento.

- Otra información. Con objeto de obtener resúmenes estadísticos e información varia, puede resultar interesante almacenar otros datos: números de línea en los que se ha usado un identificador, número de línea en que se declaró, tamaño del registro de activación (ver apartado 9.3.2), si es una variable

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

global o local, en qué función fue declarada si es local, etc.

6.3 Consideraciones sobre la tabla de símbolos

La tabla de símbolos puede inicilizarse con cierta información útil, que puede almacenarse en una única estructura o en varias:

Constantes: PI, E, NUMERO_AVOGADRO, etc.

Funciones de librería: EXP, LOG, SQRT, etc.

Palabras reservadas. Algunos analizadores lexicográficos no reconocen directamente las palabras reservadas, sino que sólo reconocen identificadores de usuario. Una vez tomado uno de la entrada, lo buscan en la tabla de palabras reservadas por si coincide con alguna; si se encuentra, devuelven al analizador sintáctico el *token* asociado en la tabla; si no, lo devuelven como identificador de verdad. Esto facilita el trabajo al lexicográfico, es más, dado que esta tabla es invariable, puede almacenarse en una tabla de dispersión perfecta (aquélla en la que todas las búsquedas son exactamente de O(1)) constituyendo así una alternativa más eficiente a PCLex o JavaCC tal y como lo hemos visto.

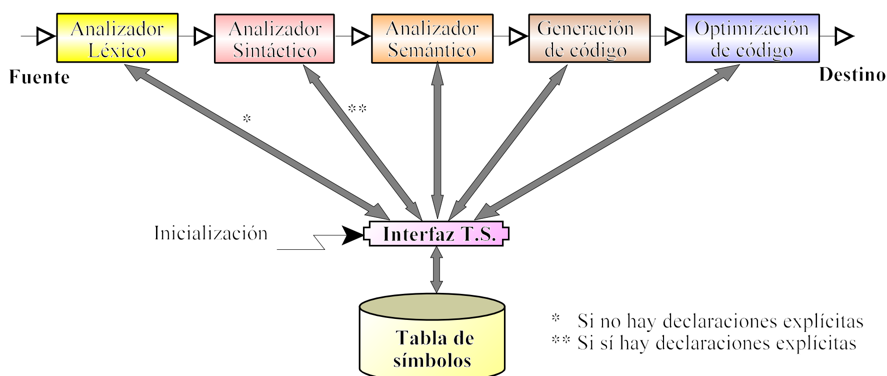

Figure 1 Accesos a la tabla de símbolos por parte de las distintas fases de un compilador

En general, conforme avanza la etapa de análisis y van apareciendo nuevas declaraciones de identificadores, el analizador léxico o el analizador sintáctico según la estrategia que sigamos, insertará nuevas entradas en la tabla de símbolos, evitando siempre la existencia de entradas repetidas. El analizador semántico efectúa las comprobaciones sensibles al contexto gracias a la tabla de símbolos y, ya en la etapa de síntesis, el generador de código intermedio usa las direcciones de memoria asociadas a cada identificador para generar un programa equivalente al de entrada. Por regla general el optimizador de código no necesita hacer uso de ella, aunque nada impide que pueda accederla.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Aunque la eficiencia en los accesos y manipulación de la tabla de símbolos son primordiales para hacer un buen compilador, suele ser conveniente realizar un desarrollo progresivo del mismo. Así, en las fases tempranas de la construcción del compilador, dada su complejidad es mejor incluir una tabla de símbolos lo más sencilla posible (por ejemplo, basada en una lista simplemente encadenada) para evitar errores que se difundan por el resto de fases del compilador. Una vez que las distintas fases funcionen correctamente puede concentrarse todo el esfuerzo en optimizar la gestión de la tabla de símbolos. Para que este proceso no influya en las fases ya desarrolladas del compilador es necesario dotar a la tabla de símbolos de una interfaz claramente definida e invariable desde el principio: en la optimización de la tabla de símbolos se modifica su implementación, pero se mantiene su interfaz. Todo este proceso puede verse en la figura 6.1.

En esta figura se menciona una distinción importante entre la construcción de compiladores para lenguajes que obligan al programador a declarar todas las variables (C, Modula-2, Pascal, Java, etc.), y lenguajes en los que las variables se pueden usar directamente sin necesidad de declararlas (BASIC, Logo, etc.). Cuando un lenguaje posee área de declaraciones y área de sentencias, la aparición de un identificador tiene una semántica bien diferente según el área en que se encuentre:

- Si aparece en el área de declaraciones, entonces hay que insertar el identificador en la tabla de símbolos, junto con la información que de él se conozca. Si el identificador ya se encontraba en la tabla, entonces se ha producido una redeclaración y debe emitirse un mensaje de error semántico.
- Si aparece en el área de sentencias, entonces hay que buscar el identificador en la tabla de símbolos para controlar que el uso que de él se está haciendo sea coherente con su tipo. Si el identificador no se encuentra en la tabla, entonces es que no ha sido previamente declarado, lo que suele considerarse un error semántico del que hay que informar al programador.

En estos casos, el analizador lexicográfico, cuando se encuentra un identificador desconoce en qué área lo ha encontrado, por lo que no resulta fácil incluir una acción léxica que discrimine entre realizar una inserción o una búsqueda; así, deberá ser el analizador sintáctico quien haga estas operaciones. Así, el atributo del *token* identificador suele ser su propio nombre.

Sin embargo, hay unos pocos lenguajes de programación en los que no es necesario declarar las variables, bien porque sólo existe un único tipo de datos (como en nuestro ejemplo de la calculadora donde todo se consideran valores enteros) o bien porque el propio nombre de la variable sirve para discernir su tipo (en BASIC las variables que acaban en “$” son de tipo cadena de caracteres). En estos casos, el propio analizador lexicográfico puede insertar el identificador en la tabla de símbolos la primera vez que aparezca; antes de insertarlo realiza una operación de búsqueda por si ya existía. Y sea cual sea el caso, se obtiene la entrada o índice de la tabla en la que se encontraba o se ha incluido; dicho índice será el valor que el analizador léxico pasa al sintáctico como atributo de cada identificador.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Por último, es importante recalcar que la tabla de símbolos contiene información útil para poder compilar, y por tanto sólo existe en tiempo de compilación, y no de ejecución. Sin embargo, en un intérprete dado que la compilación y la ejecución se producen a la vez, la tabla de símbolos permanece en memoria todo el tiempo.

6.4 Ejemplo: una calculadora con variables

Llegados a este punto, vamos a aumentar las posibilidades de la calculadora propuesta en epígrafes anteriores, incluyendo la posibilidad de manipular variables. Para simplificar el diseño no incluiremos un área de declaraciones de variables, sino que éstas se podrán utilizar directamente inicializándose a 0, puesto que tan sólo disponemos del tipo entero.

Para introducir las variables en la calculadora necesitamos una nueva construcción sintáctica con la que poder darle valores: se trata de la asignación. Además, continuaremos con la sentencia de evaluación y visualización de expresiones a la que antepondremos la palabra reservada **PRINT**. En cualquier expresión podrá intervenir una variable que se evaluará al valor que tenga en ese momento. Así, ante la entrada:

❶ a := 7 * 3;
❷ b := 3 * a;
❸ a := a + b;

se desea almacenar en la variable **a** el valor 21, en la **b** el valor 63 y luego modificar **a** para que contenga el valor 84. Para conseguir nuestros propósitos utilizaremos una tabla de símbolos en la que almacenaremos tan sólo el nombre de cada variable, así como su valor ya que estamos tratando con un intérprete. La tabla de símbolos tendrá

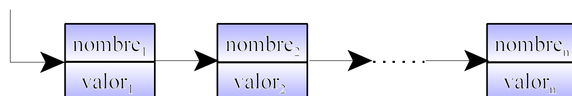

Figure 2 Una de las tablas de símbolos más simples (y más ineficientes) que se pueden construir

una estructura de lista no ordenada simplemente encadenada (ver figura 6.2), ya que la claridad en nuestro desarrollo prima sobre la eficiencia. En los desarrollos con Java haremos uso de la clase **HashMap**.

La figura 6.3 muestra cómo se debe actualizar la tabla de símbolos para reflejar las asignaciones propuestas. Así, en la sentencia ❶ una vez evaluada la expresión 7*3 al valor 21 se inspeccionará la tabla de símbolos en busca del identificador **a**. Dado que éste no está, el analizador léxico lo creará inicializándolo a 0; a continuación la regla asociada a la asignación modificará la entrada de la **a** para

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

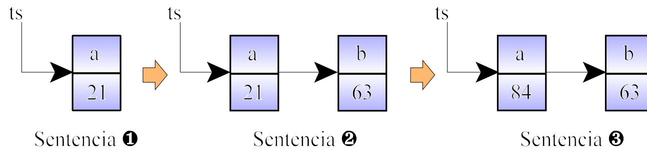

Figure 3 Situación de la tabla de símbolos tras ejecutar algunas sentencias de asignación

asignarle el valor 21. En la sentencia ❷, la evaluación de la expresión requiere buscar **a** en la tabla de símbolos para recoger su valor -21- para multiplicarlo por 3. El resultado -63- se almacena en la entrada asociada a la **b** de igual forma que se hizo para la variable **a** en la sentencia ❶. Por último, la sentencia ❸ evalúa una nueva expresión en la que intervienen **a** y **b**, asignando su suma -84- a la entrada de **a** en la tabla de símbolos.

6.4.1 Interfaz de la tabla de símbolos

Como ya se ha comentado, la interfaz de la tabla de símbolos debe quedar clara desde el principio de manera que cualquier modificación en la implementación de la tabla de símbolos no tenga repercusión en las fases del compilador ya desarrolladas. Las operaciones básicas que debe poseer son:

- **crear()**: crea una tabla vacía.
- **insertar(simbolo)**: añade a la tabla el símbolo dado.
- **buscar(nombre)**: devuelve el símbolo cuyo nombre coincide con el parámetro. Si el símbolo no existe devuelve null.
- **imprimir()**: a efectos informativos, visualiza por la salida estándar la lista de variables almacenadas en la tabla de símbolos junto con sus valores asociados.

6.4.2 Solución con Lex/Yacc

Asumimos que no hay área de declaraciones en nuestro lenguaje, por lo que la inserción de los símbolos en la tabla se debe hacer desde una acción léxica. Como atributo del *token* ID se usará un puntero a su entrada en la tabla de símbolos.

Las acciones semánticas usarán este puntero para acceder al valor de cada variable: si el identificador está a la izquierda del *token* de asignación, entonces se machacará el valor; y si forma parte de una expresión, ésta se evaluará al valor de la variable.

Por otro lado, el atributo del *token* NUM sigue siendo de tipo **int**. Dado que se necesitan atributos de tipos distintos (para NUM y para ID), habra que declarar un **%union** en Yacc, de la forma:

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
%union {
        int numero;
        simbolo * ptrSimbolo;
}
```

y los terminales y no terminales de la forma:

```
%token <numero> NUM
%token <ptrSimbolo> ID
%type <numero> expr
```

Para finalizar, podemos optar por permitir asignaciones múltiples o no. Un ejemplo de asignación múltiple es:

a := b := c := 16;

que asigna el valor 16 a las variables **c**, **b** y **a** en este orden. Para hacer esto, consideraremos que las asignaciones vienen dadas por las siguientes reglas:

```
asig    :       ID ASIG expr
        |       ID ASIG asig
        ;
```

La figura 6.4 ilustra el árbol sintáctico que reconoce este ejemplo. Como puede verse en él, el no terminal **asig** también debe tener asociado como atributo el campo **numero** del **%union**, con objeto de ir propagando el valor de la expresión y poder realizar las asignaciones a medida que se asciende en el árbol sintáctico.

La tabla de símbolos viene dada por el código siguiente, almacenado en el fichero **TabSimb.c**:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 typedef struct _simbolo {
4         struct _simbolo * sig;
5         char nombre [20];
6
```

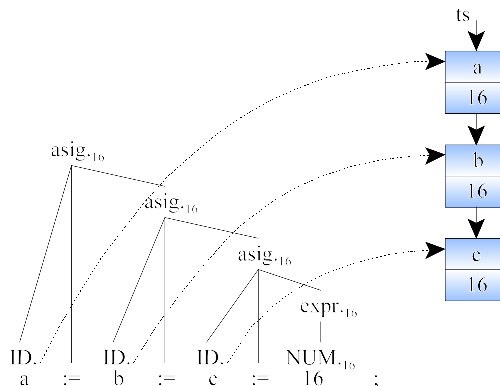

Figure 4 Árbol sintáctico para reconocer una asignación múltiple

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
BY NC SA

```
 7          int valor;
 8 } simbolo;
 9 simbolo * crear() {
10          return NULL;
11 };
12 void insertar(simbolo **pT, simbolo *s) {
13          s->sig = (*pT);
14          (*pT) = s;
15 };
16 simbolo * buscar(simbolo * t, char nombre[20]){
17          while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
18                   t = t->sig;
19          return (t);
20 };
21 void imprimir(simbolo * t) {
22          while (t != NULL) {
23                   printf("%s\n", t->nombre);
24                   t = t->sig;
25          }
26 };
```

El programa Lex resulta sumamente sencillo. Tan sólo debe reconocer los lexemas e insertar en la tabla de símbolos la primera vez que aparezca cada identificador. El siguiente fichero **Calcul.lex** ilustra cómo hacerlo:

```
 1 %%
 2 [0-9]+                {
 3                            yylval.numero = atoi(yytext);
 4                            return NUMERO;
 5                       }
 6 ":="                  { return ASIG; }
 7 "PRINT"               { return PRINT; }
 8 [a-zA-Z][a-zA-Z0-9]*   {
 9                                 yylval.ptrSimbolo = buscar(t,yytext);
10                                 if (yylval.ptrSimbolo == NULL) {
11                                      yylval.ptrSimbolo=(simbolo *) malloc(sizeof(simbolo));
12                                      strcpy(yylval.ptrSimbolo->nombre, yytext);
13                                      yylval.ptrSimbolo->valor=0;
14                                      insertar(&t, yylval.ptrSimbolo);
15                                 }
16                                 return ID;
17                        }
18 [b̸\t\n]+               { ; }
19 .                      { return yytext[0]; }
```

El programa Yacc tan sólo difiere de la calculadora tradicional en los accesos a la tabla de símbolos para obtener y alterar el valor de cada variable. El siguiente código se supone almacenado en el fichero **Calcuy.yac**:

```
1 %{
2 #include "TabSimb.c"
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
3  simbolo * t;
4  %}
5  %union {
6          int numero;
7          simbolo * ptrSimbolo;
8  }
9  %token <numero> NUMERO
10 %token <ptrSimbolo> ID
11 %token  ASIG PRINT
12 %type <numero> expr asig
13 %left '+' '-'
14 %left '*' '/'
15 %right MENOS_UNARIO
16 %%
17 prog    :       prog asig  ';'          { printf("Asignacion(es) efectuada(s).\n"); }
18         |       prog PRINT expr  ';'    { printf("%d\n",$3); }
19         |       prog error ';'          { yyerrok; }
20         |       /* Épsilon */
21         ;
22 asig    :       ID ASIG expr    {
23                                         $$ = $3;
24                                         $1->valor = $3;
25                                 }
26         |       ID ASIG asig    {
27                                         $$ = $3;
28                                         $1->valor = $3;
29                                 }
30         ;
31 expr    :       expr '+' expr   {$$ = $1 + $3;}
32         |       expr '-' expr   {$$ = $1 - $3;}
33         |       expr '*' expr   {$$ = $1 * $3;}
34         |       expr '/' expr   {$$ = $1 / $3;}
35         |       '(' expr ')'    {$$ = $2;}
36         |       '-' expr   %prec MENOS_UNARIO  {$$ = - $2;}
37         |       ID              {$$ = $1->valor; }
38         |       NUMERO          {$$ = $1;}
39         ;
40 %%
41 #include "Calcul.c"
42 #include "errorlib.c"
43 void main()
44 {       t = crear();
45         yyparse ();
46         imprimir(t);
47 }
```

6.4.3 Solución con JFlex/Cup

Dado que Java incorpora el tipo **HashMap** de manera predefinida, nuestra

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

tabla de símbolos será una clase **Tablasimbolos** que encapsula en su interior una tabla de dispersión y proporciona al exterior tan sólo las operaciones del punto 6.4.1. Como clave de la tabla usaremos un **String** (el nombre del símbolo) y como dato utilizaremos un objeto de tipo **Simbolo** que, a su vez, contiene el nombre y el valor de cada variable. El fichero **Simbolo.java** es:

```
1 class Simbolo{
2         String nombre;
3         Integer valor;
4         public Simbolo(String nombre, Integer valor){
5                 this.nombre = nombre;
6                 this.valor = valor;
7         }
8 }
```

Y **TablaSimbolos.java** quedaría:

```
1 import java.util.*;
2 public class TablaSimbolos{
3         HashMap t;
4         public TablaSimbolos(){
5                 t = new HashMap();
6         }
7         public Simbolo insertar(String nombre){
8                 Simbolo s = new Simbolo(nombre, new Integer(0));
9                 t.put(nombre, s);
10                return s;
11        }
12        public Simbolo buscar(String nombre){
13                return (Simbolo)(t.get(nombre));
14        }
15        public void imprimir(){
16                Iterator it = t.values().iterator();
17                while(it.hasNext()){
18                        Simbolo s = (Simbolo)it.next();
19                        System.out.println(s.nombre + ": "+ s.valor);
20                }
21        }
22 }
```

La tabla de símbolos debería ser vista tanto por el analizador sintáctico como por el léxico, con objeto de poder acceder a sus elementos. Aunque en este ejemplo no es estrictamente necesario, haremos que la tabla de símbolos sea creada por el programa principal como dato propio del analizador sintáctico (ya que estamos en un análisis dirigido por sintaxis) y se pasará al analizador léxico como un parámetro más en el momento de su construcción. De esta manera el fichero **Calcul.jflex** quedaría:

```
1 import java_cup.runtime.*;
2 import java.io.*;
3 %%
4 %{
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
5           private TablaSimbolos tabla;
6           public Yylex(Reader in, TablaSimbolos t){
7                   this(in);
8                   this.tabla = t;
9           }
10  %}
11  %unicode
12  %cup
13  %line
14  %column
15  %%
16  "+"         { return new Symbol(sym.MAS); }
17  "-"         { return new Symbol(sym.MENOS); }
18  "*"         { return new Symbol(sym.POR); }
19  "/"         { return new Symbol(sym.ENTRE); }
20  ";"         { return new Symbol(sym.PUNTOYCOMA); }
21  "("         { return new Symbol(sym.LPAREN); }
22  ")"         { return new Symbol(sym.RPAREN); }
23  ":="        { return new Symbol(sym.ASIG); }
24  "PRINT"             { return new Symbol(sym.PRINT); }
25  [:jletter:][:jletterdigit:]*        {
26                                          Simbolo s;
27                                          if ((s = tabla.buscar(yytext())) == null)
28                                                  s = tabla.insertar(yytext());
29                                          return new Symbol(sym.ID, s);
30                                  }
31  [:digit:]+ { return new Symbol(sym.NUMERO, new Integer(yytext())); }
32  [b̸\t\r\n]+             {;}
33  .           { System.out.println("Error léxico."+yytext()+"-"); }
```

Las acciones sobre la tabla de símbolos se han destacado en el listado del fichero **Calcuy.cup** que sería:

```
1   import java_cup.runtime.*;
2   import java.io.*;
3   parser code {:
4           static TablaSimbolos tabla = new TablaSimbolos();
5           public static void main(String[] arg){
6                   parser parserObj = new parser();
7                   Yylex miAnalizadorLexico =
8                                   new Yylex(new InputStreamReader(System.in), tabla);
9                   parserObj.setScanner(miAnalizadorLexico);
10                  try{
11                          parserObj.parse();
12                          tabla.imprimir();
13                  }catch(Exception x){
14                          x.printStackTrace();
15                          System.out.println("Error fatal.\n");
16                  }
17          }
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
cc BY NC SA

```
18 :};
19 terminal PUNTOYCOMA, MAS, MENOS, POR, ENTRE;
20 terminal UMENOS, LPAREN, RPAREN, PRINT, ASIG;
21 terminal Simbolo ID;
22 terminal Integer NUMERO;
23 non terminal listaExpr;
24 non terminal Integer expr, asig;
25 precedence left MAS, MENOS;
26 precedence left POR, ENTRE;
27 precedence left UMENOS;
28 listaExpr     ::=   listaExpr asig PUNTOYCOMA
29                              {: System.out.println("Asignacion(es) efectuada(s)."); :}
30               |     listaExpr PRINT expr:e PUNTOYCOMA
31                              {: System.out.println("= " + e); :}
32               |     listaExpr error PUNTOYCOMA
33               |     /* Epsilon */
34               ;
35 asig      ::=  ID:s ASIG expr:e   {:
36                                         RESULT = e;
37                                         s.valor = e;
38                                    :}
39               |     ID:s ASIG asig:e  {:
40                                         RESULT = e;
41                                         s.valor = e;
42                                    :}
43               ;
44 expr      ::=  expr:e1 MAS expr:e2
45                              {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
46           |    expr:e1 MENOS expr:e2
47                              {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
48           |    expr:e1 POR expr:e2
49                              {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
50           |    expr:e1 ENTRE expr:e2
51                              {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
52           |    ID:s          {: RESULT = s.valor; :}
53           |    NUMERO:n      {: RESULT = n; :}
54           |    MENOS expr:e      %prec UMENOS
55                              {: RESULT = new Integer(0 - e.intValue()); :}
56           |    LPAREN expr:e RPAREN   {: RESULT = e; :}
57           ;
```

6.4.4 Solución con JavaCC

El proceso que se sigue para solucionar el problema de la calculadora en JavaCC es casi igual al de JFlex/Cup solo que, recordemos, no es posible pasar más atributo del analizador léxico al sintáctico que el propio lexema leído de la entrada. De esta manera, hemos optado por insertar en la tabla de símbolos en la regla asociada al identificador, en lugar de en una acción léxica.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
cc BY NC SA

Lo más reseñable de esta propuesta es la regla asociada a la asignación múltiple:

```
asig    ::=     ( <ID> “:=” )+ expr() <PUNTOYCOMA>
```

Debido a la naturaleza descendente del analizador generado por JavaCC, resulta difícil propagar el valor de **expr** a medida que se construye el árbol, por lo que se ha optado por almacenar en un vector todos los objetos de tipo **Simbolo** a los que hay que asignar la expresión, y realizar esta operación de golpe una vez evaluada ésta. Una alternativa habría sido

```
asig    ::=     <ID> “:=” expr()
        |       <ID> “:=” asig()
```

que solucionaría el problema si **asig** devuelve el valor de la expresión. Nótese cómo esta solución pasa por convertir la gramática en recursiva a la derecha, aunque tiene el problema de ser LL(4) debido a que ambas reglas comienzan por <ID> “:=” y a que tanto **expr** como **asig** pueden comenzar con un <ID>.

Así pues, el programa en JavaCC **Calculadora.jj** es:

```
1  PARSER_BEGIN(Calculadora)
2       import java.util.*;
3       public class Calculadora{
4           static TablaSimbolos tabla = new TablaSimbolos();
5           public static void main(String args[]) throws ParseException {
6           new Calculadora(System.in).gramatica();
7               tabla.imprimir();
8           }
9       }
10 PARSER_END(Calculadora)
11 SKIP : {
12          "␢"
13      |   "\t"
14      |   "\n"
15      |   "\r"
16 }
17 TOKEN [IGNORE_CASE] :
18 {
19          <PRINT: "PRINT">
20      |   <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
21      |   <NUMERO: (["0"-"9"])+>
22      |   <PUNTOYCOMA: ";">
23 }
24 /*
25 gramatica ::= ( sentFinalizada )*
26 */
27 void gramatica():{}{
28      ( sentFinalizada() )*
29 }
30 /*
31 sentFinalizada ::=          PRINT expr ";"
32                         |   ( ID ":=" )+ expr ";"
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
cc BY NC SA

```
33                          |       error ";"
34  */
35  void sentFinalizada():{
36          int resultado;
37          Vector v = new Vector();
38          Simbolo s;
39  }{
40          try{
41                  <PRINT> resultado=expr() <PUNTOYCOMA>
42                                          { System.out.println("="+resultado); }
43          |
44                  ( LOOKAHEAD(2)
45                          s=id() ":=" { v.add(s); } )+ resultado=expr() <PUNTOYCOMA>
46                                          {
47                                                  Integer valor = new Integer(resultado);
48                                                  Iterator it = v.iterator();
49                                                  while(it.hasNext())
50                                                          ((Simbolo)it.next()).valor = valor;
51                                                  System.out.println("Asignacion(es) efectuada(s).");
52                                          }
53          }catch(ParseException x){
54                  System.out.println(x.toString());
55                  Token t;
56                  do {
57                          t = getNextToken();
58                  } while (t.kind != PUNTOYCOMA);
59          }
60  }
61  /*
62  expr ::= term ( ( "+" | "-" ) term )*
63  */
64  int expr():{
65          int     acum1=0,
66                  acum2=0;
67  }{
68          acum1=term()    (               ("+" acum2=term() {acum1+=acum2;} )
69                                  |       ("-" acum2=term() {acum1-=acum2;} )
70                          )*
71          { return acum1; }
72  }
73  /*
74  term ::= fact ( ( "*" | "/" ) fact )*
75  */
76  int term():{
77          int     acum1=0,
78                  acum2=0;
79  }{
80          acum1=fact()    (               ("*" acum2=fact() {acum1*=acum2;} )
81                                  |       ("/" acum2=fact() {acum1/=acum2;} )
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
82                          )*
83          { return acum1; }
84  }
85  /*
86  fact ::= (“-“)* ( ID | NUMERO | "(" expr ")" )
87  */
88  int fact():{
89          int   acum=0,
90                signo=1;
91          Simbolo s;
92  }{
93          ("-" { signo *= -1; } )*
94          (           s=id() { acum = s.valor.intValue(); }
95                |     acum=numero()
96                |     "(" acum=expr() ")"
97          )
98          { return signo*acum; }
99  }
100 Simbolo id():{}{
101         <ID>        {
102                         Simbolo s;
103                         if ((s = tabla.buscar(token.image)) == null)
104                               s = tabla.insertar(token.image);
105                         return s;
106                     }
107 }
108 int numero():{}{
109         <NUMERO>        { return Integer.parseInt(token.image); }
110 }
111 SKIP : {
112         <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
113 }
```

Como se ha comentado anteriormente, la necesidad de realizar todas las asignaciones de golpe en una asignación múltiple se podría evitar introduciendo la regla:

```
/*
asig ::= ( ID “:=” )+ expr
*/
int asig():{
            int resultado;
            Simbolo s;
}{
            (           LOOKAHEAD(4)
                        s=id() “:=” resultado=asig() { s.valor = new Integer(resultado); }
            |           s=id() “:=” resultado=expr() { s.valor = new Integer(resultado); }
            )
            { return resultado; }
}
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
CC BY NC SA

Y la regla

```
id() ":=" )+ expr() <PUNTOYCOMA>
```

se sustituye por

```
asig() <PUNTOYCOMA> { System.out.println("Asignacion(es) efectuada(s)."); }
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike