Capítulo 7

Gestión de tipos

7.1 Visión general

Un compilador debe comprobar si el programa fuente sigue tanto las convenciones sintácticas como las semánticas que describen al lenguaje fuente. Esta comprobación garantiza la detección y posterior comunicación al programador de ciertos errores de programación que facilitan la depuración pero que no pueden ser gestionados desde el punto de vista exclusivamente sintáctico. De este este tipo de errores, los más comunes se derivan de la utilización de tipos de datos, ya sean proporcionados por el propio lenguaje (tipos primitivos), o creados por el programador en base a construcciones sintácticas proporcionadas por el lenguaje (tipos de usuario). Básicamente, los tipos de datos sirven precisamente para que el programador declare con qué intención va a utilizar cada variable (para guardar un número entero, un número real, una cadena de caracteres, etc.), de manera que el compilador pueda detectar automáticamente en qué situaciones ha cometido un error el programador (por ejemplo, si ha intentado sumar un número entero y una cadena de caracteres).

Con la gestión de tipos se asegura que el tipo de una construcción sintáctica coincida con el previsto en su contexto. Por ejemplo, el operador aritmético predefinido **mod** en Modula-2 exige operandos numéricos enteros, de modo que mediante la gestión de tipos debe asegurarse de que dichos operandos sean de tipo entero. De igual manera, la gestión de tipos debe asegurarse de que la desreferenciación se aplique sólo a un puntero, de que la indización se haga sólo sobre una matriz, de que a una función definida por el usuario se aplique la cantidad y tipo de argumentos correctos, etc.

La información recogida por el gestor de tipos también es útil a la hora de generar código. Por ejemplo, los operadores aritméticos normalmente se aplican tanto a enteros como a reales, y tal vez a otros tipos, y se debe examinar el contexto de cada uno de ellos para determinar el espacio en memoria necesario para almacenar cálculos intermedios. Se dice que un símbolo o función está **sobrecargado** cuando representa diferentes operaciones en diferentes contextos; por ejemplo, la suma (expresada por el símbolo "+") suele estar sobrecargada para los números enteros y para los reales. La sobrecarga puede ir acompañada de una conversión de tipos, donde el compilador proporciona el operador para convertir un operando en el tipo esperado por el contexto. Considérense por ejemplo expresiones como "x + i" donde **x** es de tipo real e **i** es de tipo entero. Como la representación de enteros y reales es distinta dentro de un computador, y se utilizan instrucciones de máquina distintas para las operaciones sobre enteros y reales, puede que el compilador tenga que convertir primero uno de los operadores para garantizar que ambos operandos son del mismo tipo cuando tenga

lugar la suma. La gramática de la figura 7.1 genera expresiones formadas por constantes enteras y reales a las que se aplica el operador aritmético de la suma + . Cuando se suman dos enteros el resultado es entero, y si no es real. De esta manera, cada subexpresión tiene asociado un tipo y un espacio de almacenamiento dependiente de éste, al igual que tendrá asociado un valor en tiempo de ejecución, tal y como se ilustra en el árbol de la derecha de esta figura.

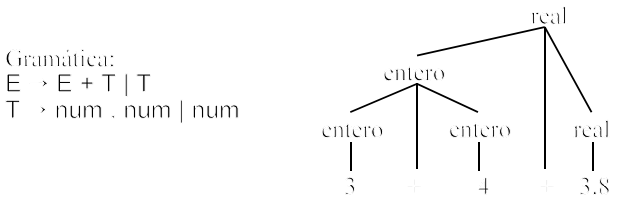


Figure 1 Ejemplo de cálculo del tipo de cada subexpresión. La suma de enteros produce un entero, pero si interviene un valor real, entonces el resultado será un real

Una noción diferente de la sobrecarga es la de **polimorfismo**, que aparece en los lenguajes orientados a objetos. En estos lenguajes existe una jerarquía de herencia en la que los tipos más especializados heredan de los tipos más generales, de manera que los tipos especializados pueden incorporar características de las que carecen los generales, e incluso pueden cambiar el funcionamiento de ciertos métodos de las clases ancestras en lo que se llama **reescritura de funciones**. En estos lenguajes, una función es polimórfica cuando tiene parámetros de tipo general, pero puede ejecutarse con argumentos de tipos especializados de la misma línea hereditaria, es decir, descendientes del tipo general, desencadenando en cada caso una acción diferente por medio de la **vinculación dinámica**. Según la vinculación dinámica, en tiempo de ejecución se detecta el tipo del parámetro real y se ejecutan sus métodos propios, en lugar de los del padre, si éstos están reescritos. Una cosa es que nuestro lenguaje tenga algunas funciones predefinidas sobrecargadas, y otra cosa que se permita la definición de funciones sobrecargadas. En este último caso, hay que solucionar de alguna forma los conflictos en la tabla de símbolos, ya que un mismo identificador de nombre de función tiene asociados diferentes parámetros y cuerpos.

7.2 Compatibilidad nominal, estructural y funcional

Cuando se va a construir un nuevo lenguaje de programación, resulta fundamental establecer un criterio riguroso y bien documentado de compatibilidad entre tipos, de tal manera que los programadores que vayan a utilizar ese lenguaje sepan a qué atenerse.

Entendemos que dos tipos **A** y **B** son compatibles cuando, dadas dos expresiones $\mathbf{e_a}$ y $\mathbf{e_b}$ de los tipos **A** y **B** respectivamente, pueden usarse indistintamente en un contexto determinado. Existen muchos contextos diferentes, de los cuales el más

usual es la asignación, de forma que si la variable $\mathbf{v_a}$ es de tipo **A** y se permite una asignación de la forma $\mathbf{v_a}$ **:=** $\mathbf{e_b}$, diremos que **A** y **B** son compatibles para la asignación. Otro contexto viene dado por las operaciones aritméticas, como la suma: si un lenguaje admite expresiones de la forma $\mathbf{e_a}$ **+** $\mathbf{e_b}$, diremos que **A** y **B** son compatibles para la suma. En este último caso, los tipos **A** y **B** podrían representar, por ejemplo, los tipos **int** y **float** de C.

Básicamente se pueden establecer tres criterios de compatibilidad entre tipos: nominal, estructural y funcional. La compatibilidad nominal es la más estricta, esto es, la que impone las condiciones más fuerte para decidir cuándo dos variables se pueden asignar entre sí. La compatibilidad funcional es la más relajada, mientras que la estructural se halla en un nivel intermedio. Por supuesto, el desarrollador de compiladores puede establecer otros criterios que estén a medio camino entre cualesquiera de estas compatibilidades.

Para explicar la diferencia entre estos criterios usaremos los siguientes dos tipos definidos en C:

```
typedef struct{
        char nombre[20];
        char apellidos[35];
        int edad;
} empleado;
empleado e1, e2;
```

y

```
typedef struct{
        char nombre[20];
        char apellidos[35];
        int edad;
} cliente;
cliente c1, c2;
```

Los tipos **cliente** y **empleado** tienen diferente nombre, o sea, son diferentes desde el punto de vista nominal. Sin embargo, su contenido o estructura es la misma, luego son equivalentes desde el punto de vista estructural. De esta forma, un lenguaje con compatibilidad de tipos a nivel nominal impide hacer asignaciones como **e1=c1**, pero permite hacer **e1=e2**, ya que **e1** y **e2** pertenecen exactamente al mismo tipo (puesto que no puede haber dos tipos con el mismo nombre). Un lenguaje con compatibilidad de tipos estructural permite hacer ambas asignaciones, puesto que sólo requiere que la estructura del l-valor y del r-valor sea idéntica lo cual es cierto para los tipos **empleado** y **cliente**.

La compatibilidad nominal tiene fuertes implicaciones cuando se declaran variables de tipos anónimos. Siguiendo con el ejemplo, se tendría:

```
struct{
        char nombre[20];
        char apellidos[35];
```

```
        int edad;
    } e1, e2;
```

y

```
    struct{
        char nombre[20];
        char apellidos[35];
        int edad;
    } c1, c2;
```

donde los tipos de las variables **e1**, **e2**, **c1** y **c2** carece de nombre. Por regla general, en estas situaciones, un lenguaje con compatibilidad de tipos nominal crea un tipo con un nombre interno para cada declaración anónima, lo que en el ejemplo se traduce en la creación de dos tipos, uno al que pertenecerán **e1** y **e2** (y que, por tanto, serán compatibles nominalmente), y otro al que pertenecerán **c1** y **c2** (que también serán compatibles nominalmente).

Por otro lado, la compatibilidad funcional ni siquiera requiere que las expresiones posean la misma estructura. Dos expresiones $\mathbf{e_a}$ y $\mathbf{e_b}$ de tipos **A** y **B** respectivamente son funcionalmente compatibles si **A** y **B** poseen diferente estructura pero pueden usarse indistintamente en algún contexto. El caso más usual consiste en poder sumar valores enteros y decimales. Por regla general, los enteros se suelen representar en complemento a dos, mientras que los números decimales se representan en coma flotante. Aunque se utilice el mismo número de octetos para representar a ambos, su estructura es esencialmente diferente, pero la mayoría de lenguajes de programación permite realizar operaciones aritméticas entre ellos. En estos lenguajes existe una compatibilidad funcional entre el tipo entero y el decimal.

La compatibilidad funcional requiere que el compilador detecte las situaciones en que se mezclan correctamente expresiones de tipos con diferente estructura, al objeto de generar código que realice la conversión del tipo de la expresión menos precisa a la de mayor precisión. Por ejemplo, al sumar enteros con decimales, los enteros se deben transformar a decimales y, sólo entonces, realizar la operación. A esta operación automática se la denomina **promoción** (del inglés *promotion*).

En los lenguajes orientados a objeto también aparece la compatibilidad por herencia: si una clase **B** hereda de otra **A** (**B** es un **A**), por regla general puede emplearse un objeto de clase **B** en cualquier situación en la que se espere un objeto de tipo **A**, ya que **B** es un **A** y por tanto cumple con todos los requisitos funcionales para suplantarlo.

7.3 Gestión de tipos primitivos

En este punto se estudiará la gestión de una calculadora básica, cuando en ésta se incluye el tipo cadena de caracteres, además del tipo entero con el que ya se ha trabajado en capítulos anteriores.

7.3.1 Gramática de partida

La gramática a reconocer permitirá gestionar valores de tipo entero y de tipo cadena de caracteres. Las cadenas se pueden concatenar mediante sobrecarga del operador "+" que, recordemos, también permite sumar enteros. Mediante dos funciones que forman parte del lenguaje se permitirá la conversión entre ambos tipos. Para evitar perdernos en detalles innecesarios, se han eliminado algunas operaciones aritméticas y la asignación múltiple. Además, para ilustrar que el retorno de carro no siempre ha de ser considerado como un separador, en la gramática que se propone se ha utilizado como terminador, haciendo las funciones del punto y coma de gramáticas de ejemplos anteriores. La gramática expresada en formato Yacc es la siguiente:

```
prog    :   prog asig '\n'
        |   prog IMPRIMIR expr '\n'
        |   prog error '\n'
        |   /* Épsilon */
        ;
asig    :   ID ASIG expr
        ;
expr    :   expr '+' expr
        |   expr '*' expr
        |   A_ENTERO '('expr ')'
        |   A_CADENA '(' expr ')'
        |   ID
        |   NUMERO
        |   CADENA
        ;
```

La misma gramática para JavaCC queda:

```
void gramatica():{}{
    ( sentFinalizada() )*
}
void sentFinalizada():{}{
        <IMPRIMIR> expr() <RETORNODECARRO>
    |   <ID><ASIG> expr() <RETORNODECARRO>
    |   /* Gestión de error */
}
void expr():{}{
    term() ( "+" term() )*
}
void term():{}{
    fact() ( "*" fact() )*
}
void fact():{}{
        <ID>
    |   <NUMERO>
    |   <CADENA>
    |   <A_CADENA> "(" expr() ")"
    |   <A_ENTERO> "(" expr() ")"
}
```

El cometido semántico de cada construcción es:

- Es posible asignar el valor de una expresión a una variable. No es posible hacer asignaciones múltiples: **ID ASIG expr**.
- Se puede trabajar con valores de dos tipos: cadenas de caracteres y números enteros.
- La gramática no posee una zona donde el programador pueda declarar el tipo de cada variable, sino que éste vendrá definido por el de la primera asignación válida que se le haga. Es decir, si la primera vez que aparece la variable x se le asigna el valor 65, entonces el tipo de x es entero. El tipo de una variable no puede cambiar durante la ejecución de un programa.
- Es posible visualizar el valor de una expresión: **IMPRIMIR expr**. Esta sentencia permite, por tanto, visualizar tanto valores enteros como cadenas de caracteres, por lo que si la consideramos desde el punto de vista funcional, podemos decir que **IMPRIMIR** es una función polimórfica.
- Con los enteros vamos a poder hacer sumas y multiplicaciones. En cambio con las cadenas no tiene demasiado sentido hacer ninguna de estas operaciones, aunque utilizaremos el símbolo de la suma para permitir concatenaciones de cadenas. No es posible sumar o concatenar enteros y cadenas.
- Es posible realizar conversiones de tipos mediante funciones propias del lenguaje de programación. Éstas son diferentes a las funciones preconstruidas (built-ins), que se encuentran almacenadas en librerías pero cuyos nombres no forman realmente parte de la sintaxis del lenguaje. Estas funciones son:
  - **A_CADENA(expr)**: tiene como parámetro una expresión de tipo entero y nos devuelve como resultado la cadena de texto que la representa.
  - **A_ENTERO(expr)**: tiene como parámetro una expresión de tipo carácter que representa un número y nos devuelve como resultado su valor entero.

7.3.2 Pasos de construcción

Una vez que se tiene clara la gramática que se va a usar, es conveniente seguir los pasos que se indican a continuación a fin de culminar la construcción del traductor. En estos pasos suele ser común tener que volver atrás con el fin de reorganizar la gramática para adaptar sus construcciones a los requerimientos de los metacompiladores.

7.3.2.1 Propuesta de un ejemplo

Con objeto de proseguir los pasos de construcción, suele ser buena idea plantear un ejemplo de entrada al traductor y la salida que se desea obtener. La entrada aparece en negro, la salida en azul y los mensajes de error en rojo:

❶ b := “Camarada Trotski”
❷ a :=7
❸ c :=12 + 3

❹ PRINT c + 5
20
❺ PRINT b
Camarada Trotski
❻ PRINT b+7
No se pueden sumar cadenas y enteros.
❼ PRINT b+", fundador de la URSS."
Camarada Trotski, fundador de la URSS.
❽ PRINT A_ENTERO("23")*4
92

7.3.2.2 Definición de la tabla de símbolos

En este paso se define la estructura de la tabla de símbolos, si es que esta es necesaria para el problema, así como su interfaz.

En este caso sí necesitaremos la tabla de símbolos, ya que en ella se almacenarán los valores de los identificadores. También será necesario guardar el tipo de cada variable con objeto de controlar que su utilización sea coherente con las intenciones de su declaración (implícita en la primera asignación). De esta forma, el tipo de una variable se guardará en la tabla de símbolos mediante un carácter con el siguiente significado:

- 'c': tipo cadena de caracteres.
- 'e': tipo entero.
- 'i': tipo indefinido. Este tipo se usará en caso de que en la primera asignación se le intente dar a una variable el valor resultante de una expresión errónea.

Parece claro que, dado que existen variables de tipos diferentes, una entrada de la tabla de símbolos debe ser capaz de guardar tanto una cadena como un entero, en función del tipo de la variable que representa. En lenguaje C esto se soluciona mediante una estructura **union**, mientras que en Java se tendrá un manejador a **Object** que, en función del tipo de la variable, apuntará a un **Integer** o a un **String**. La figura 7.2 muestra gráficamente la estructura de la tabla de símbolos.

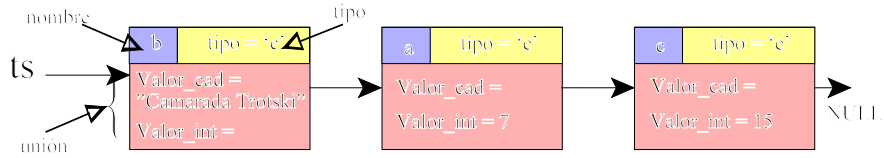


Figure 2 Situación de la tabla de símbolos tras introducir en la calculadora las sentencias del punto 7.3.2.1

7.3.2.3 Asignación de atributos

El siguiente paso consiste en estudiar qué atributo se le tiene que asociar a los terminales y no terminales.

7.3.2.3.1 Atributos de terminales

Con respecto a los terminales **NUMERO** y **CADENA**, dado que representan constantes enteras y de cadena de caracteres, lo más sensato es asignarles como atributos las constantes a que representan. En el caso de **NUMERO** se realizará una conversión del lexema leído al número entero correspondiente, dado que nos interesa operar aritméticamente con él. En cuanto a la cadena de caracteres, habrá que eliminar las comillas de apertura y de cierre ya que son sólo delimitadores que no forman parte del literal en sí.

Por otro lado, y como ya se ha comentado, dado que estamos definiendo un lenguaje sin zona de declaraciones explícitas, será el analizador léxico quien inserte los identificadores en la tabla de símbolos, con objeto de descargar al sintáctico de algunas tareas. El atributo del terminal **ID** es, por tanto, una entrada de dicha tabla. Básicamente, el analizador lexicográfico, cuando se encuentra con un identificador, lo busca en la tabla; si está asocia su entrada como atributo al **ID** encontrado y retorna al sintáctico; si no está crea una nueva entrada en la tabla y la retorna al sintáctico. El problema que se plantea aquí es: ¿con qué se rellena una nueva entrada en la tabla de símbolos? Analicemos por ejemplo la primera sentencia del apartado 7.3.2.1:

b = “Camarada Trotski”

Cuando el analizador léxico encuentra el identificador **b**, lo busca en la tabla de símbolos (que inicialmente está vacía) y, al no encontrarlo, crea un nuevo nodo en ésta. Este nodo tendrá como nombre a “b”, pero ¿cuál es su tipo? El tipo de **b** depende del valor que se le dé en la primera asignación (en este caso una cadena de caracteres), pero el problema es que el analizador léxico no ve más allá del lexema en el que se encuentra en este momento, “b”, por lo que aún es incapaz de conocer el tipo de la expresión que hay detrás del terminal de asignación. El tipo de **b** está implícito en el momento en que se hace su primera asignación, pero el momento en que el analizador léxico realiza la inserción en la tabla de símbolos está antes del momento en que se conoce el tipo de la expresión asignada. La solución es que el analizador sintáctico será el que le ponga el tipo cuando se concluya la primera asignación; mientras tanto, el lexicográfico indicará un tipo transitorio indefinido: “i”. La figura 7.3 muestra el contenido de la tabla de símbolos en el preciso instante en que el analizador léxico retorna al sintáctico la entrada de la tabla de símbolos correspondiente a la **c** de la sentencia ❸.

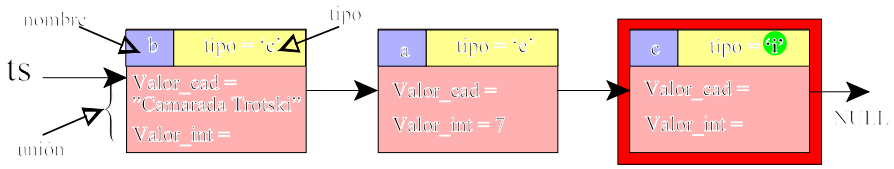


Figure 3 La entrada de la tabla marcada en rojo acaba de ser insertada por el analizador léxico que, como desconoce el tipo de la variable recién insertada **c**, la pone como indefinida: **i**

7.3.2.3.2 Atributos de no terminales

Hasta ahora, en nuestro ejemplo de la calculadora, el no terminal **expr** tenía como atributo el valor entero a que representaba. Sin embargo, en el caso que ahora nos ocupa, el concepto de expresión denotado por el no terminal **expr** sirve para representar expresiones tanto enteras como de cadena de caracteres. Esta dualidad queda bien implementada mediante el concepto de **union** en C, de manera parecida a como se ha hecho en los nodos de la tabla de símbolos. De esta manera, el atributo de **expr** es un registro con un campo para indicar su tipo y una **union** para indicar su valor, ya sea entero o de cadena. En el caso de Java, la solución es aún más sencilla ya que el atributo puede ser un simple **Object**; si dicho **Object** es **instanceof Integer** el tipo será entero; si el **Object** es **instanceof String** el tipo será cadena de caracteres; y si el **null** el tipo será indefinido. En cualquier caso, en aras de una mayor legibilidad, también es posible asociar un objeto con dos campos, tipo y valor, como en la solución en C. La figura 7.4 muestra un ejemplo de los atributos asociados cuando se reconoce la sentencia ❸ del apartado 7.3.2.1. En este caso en concreto, el atributo de **expr** debe coincidir conceptualmente con parte de la estructura de un nodo de la tabla de símbolos, tal y como se muestra en la figura 7.5.

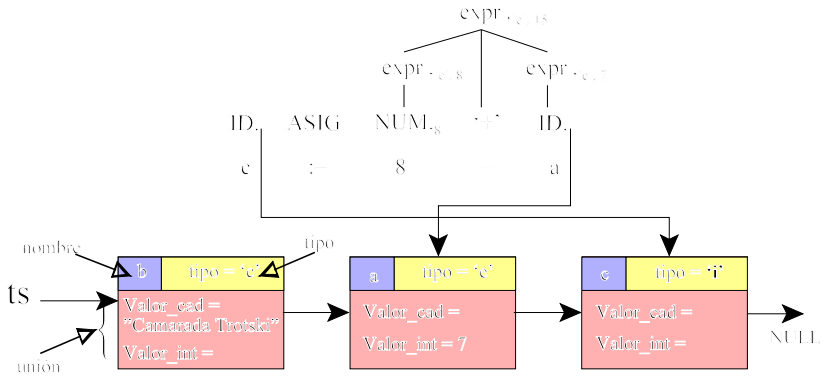


Figure 5 Reconocimiento de una sentencia de asignación y atributos asociados a los terminales y no terminales del árbol sintáctico

El atributo asociado a una expresión permite manipular los errores semánticos. Por ejemplo, una sentencia de la forma:

c = 8 + “hola”

es sintácticamente correcta ya que existe un árbol sintáctico que la reconoce, pero es

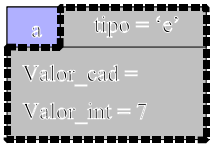


Figure 4 El atributo de una expresión es un tipo y un valor (entero o de cadena en función del tipo), lo que coincide con parte de un nodo de la tabla de símbolos (zona sombreada)

incorrecta semánticamente, ya que no vamos a permitir mezclar números y cadenas en una suma. Por tanto, la expresión resultante en el árbol sintáctico tendrá como tipo el valor ‘i’ de indefinido, tal y como se muestra en el árbol de la figura 7.6.

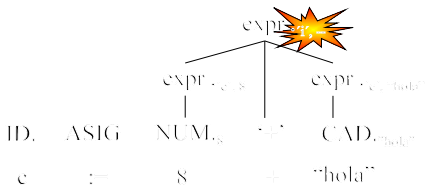


Figure 6 Se produce un error semántico porque no se puede sumar un entero y una cadena. La expresión resultante es, por tanto, indefinida

7.3.2.4 Acciones semánticas

En este apartado se estudiarán las acciones semánticas desde dos puntos de vista:

- ✎ Los controles de compatibilidad de tipos.
- ✎ Las acciones a realizar en caso de que se satisfagan las restricciones de tipos.

Por regla general, el estudio de las acciones semánticas suele hacerse comenzando por las reglas por las que primero se reduce en un árbol sintáctico cualquiera, ya que éstas suelen ser las más sencillas, y así se aborda el problema en el mismo orden en el que se acabarán ejecutando las acciones semánticas. Las reglas:

```
expr    :    NUMERO
        |    CADENA
```

son las más simples ya que la acción semántica asociada tan sólo debe asignar el tipo a **expr** y copiar el atributo del *token* como valor de **expr**.

La regla:

```
expr    :    expr ‘*’ expr
```

debe controlar que las dos expresiones sean de tipo entero. Si lo son, el tipo de la **expr** resultante también es entero y el valor será el producto de las expresiones del consecuente. En caso contrario, resultará una expresión de tipo indefinido.

La regla:

```
expr    :    expr ‘+’ expr
```

es un poco más complicada, ya que el operador ‘+’ está sobrecargado y con él puede tanto sumarse enteros como concatenarse cadenas de caracteres. Por tanto, la acción semántica debe comprobar que los tipos de las expresiones del antecedente sean iguales. Si son enteras se produce una expresión entera suma de las otras dos. Si son cadenas de caracteres se produce una expresión que es la concatenación de las otras dos. En cualquier otro caso se produce una expresión indefinida.

Las reglas:

```
expr    :    A_ENTERO ‘(’expr ‘)’
```

```
        |   A_CADENA '(' expr ')'
```

toman, respectivamente, expresiones de tipo cadena y entero y devuelven enteros y cadenas, también respectivamente. En cualquier otro caso se produce una expresión indefinida. La función **A_CADENA()** convierte el valor de su parámetro en una ristra de caracteres que son dígitos. La función **A_ENTERO()** convierte su parámetro en un número, siempre y cuándo la cadena consista en una secuencia de dígitos; en caso contrario también se genera una expresión indefinida.

La regla:

```
expr    :   ID
```

es trivial, ya que el analizador sintáctico recibe del léxico la entrada de la tabla de símbolos en la que reside la variable. Por tanto, dado que el atributo de una expresión comparte parte de la estructura de un nodo de la tabla de símbolos, en esta acción semántica lo único que hay que hacer es copiar dicho trozo (ver figura 7.5).

La semántica de la asignación dada por la regla:

```
asig    :   ID ASIG expr
```

es un poco más compleja. Recordemos que el tipo de una variable viene dado por el tipo de lo que se le asigna por primera vez. Por ejemplo, si en laprimera asignación a la variable **a** se le asigna el valor 7, entonces su tipo es entero durante toda la ejecución del programa; si se le asigna "Ana", entonces será de tipo cadena. Por tanto, la asignación debe:

1. Si el tipo del l-valor es indefinido, se le asigna el valor y el tipo del r-valor, sea éste el que sea.
2. Si el tipo del l-valor está definido, entonces:
   2.1. Si coinciden los tipos del l-valor y del r-valor entonces se asigna al l-valor el valor del r-valor.
   2.2. Si no coinciden los tipos, se emite un mensaje de error y el l-valor se deja intacto.

En esta explicación queda implícito qué hacer cuando se intenta asignar a una variable una expresión indefinida: si la variable aún es indefinida se deja tal cual; si no, la variable no pasa a valer indefinido, sino que se queda como está. Si ante una asignación indefinida se nos ocurriera poner la variable en estado indefinido, la secuencia:

```
a = 7              ✔ Funciona
a = 7 + "hola"     ✗ Falla (y cambia el tipo de a a indefinido)
a = "mal"          ✔ Funciona (pero no debería)
```

cambiaría el valor y el tipo de **a**, lo cual es incorrecto con respecto a la semántica de nuestro lenguaje.

Una vez asignado el tipo a una variable éste siempre es el mismo; si durante la ejecución se le asigna una expresión errónea, el tipo de la variable seguirá siendo el mismo y su valor tampoco cambiará. Este comportamiento podría modificarse añadiendo una variable *booleana* a la tabla de símbolos que me diga si el valor de la variable es conocido o no, de tal manera que una asignación errónea pondría esta

variable a *false*, tal y como se ilustra en la figura 7.7.

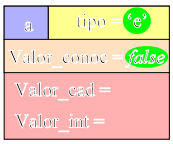


Figure 7 Inclusión de un campo que indica que, a pesar de conocerse el tipo de la variable, su valor se desconoce debido a que la última asignación fue errónea

7.3.3 Solución con Lex/Yacc

La solución en C parte de una tabla de símbolos con la estructura propuesta en los apartados anteriores, y con la misma interfaz que la que se introdujo en el apartado 6.4.2 y que se muestra a continuación en el fichero **TabSimb.c**:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef struct _simbolo {
4        struct _simbolo * sig;
5        char nombre[20];
6        char tipo;
7        union {
8              char valor_cad[100];
9              int valor_int;
10       } info;
11 } simbolo;
12
13 void insertar(simbolo ** p_t, simbolo * s) {
14       s->sig = (*p_t);
15       (*p_t) = s;
16 }
17
18 simbolo * buscar(simbolo * t, char nombre[20]) {
19       while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
20             t = t->sig;
21       return (t);
22 }
23 void imprimir(simbolo * t) {
24       while (t != NULL) {
25             printf("%s:%c: ", t->nombre, t->tipo);
26             if(t->tipo == 'c') printf("%s\n", t->info.valor_cad);
27             else if(t->tipo == 'e') printf("%d\n", t->info.valor_int);
28             else printf("Indefinido\n");
29             t = t->sig;
30       }
31 }
```

El programa Lex funciona de manera parecida al de la calculadora del capítulo 6, sólo que ahora hemos de reconocer también literales de tipo cadena de caracteres que se retornan a través del *token* **CADENA**. Como atributo se le asocia el lexema sin

comillas. Además, cuando se reconoce un identificador de usuario por primera vez, se debe introducir en la tabla de símbolos indicando que su tipo es indefinido, de manera que el sintáctico pueda asignarle el tipo correcto cuando éste se conozca a través del tipo y valor de la primera expresión que se le asigne. Además, el retorno de carro no se ignora sino que se le devuelve al analizador sintáctico ya que forma parte de nuestra gramática. Por último, nótese que el *token* **IMPRIMIR** se corresponde con el lexema "PRINT", puesto que nada obliga a que los *tokens* deban llamarse igual que las palabras reservadas que representan. El programa **TipSimpl.lex** queda:

```
1  %%
2  [0-9]+    {
3                  yylval.numero=atoi(yytext);
4                  return NUMERO;
5            }
6  \"[^\"]*\"  {
7                  strcpy(yylval.cadena, yytext+1);
8                  yylval.cadena[yyleng-2] = 0;
9                  return CADENA;
10           }
11 "PRINT"            { return IMPRIMIR;}
12 "A_ENTERO"        { return A_ENTERO;}
13 "A_CADENA"        { return A_CADENA;}
14 ":="               { return ASIG; }
15 [a-zA-Z][a-zA-Z0-9]*   {
16                              if (strlen(yytext)>19) yytext[19] = 0;
17                              yylval.ptr_simbolo = buscar(t,yytext);
18                              if(yylval.ptr_simbolo == NULL) {
19                                   yylval.ptr_simbolo = (simbolo *)malloc(sizeof(simbolo));
20                                   strcpy(yylval.ptr_simbolo->nombre,yytext);
21                                   yylval.ptr_simbolo->tipo='i';
22                                   insertar(&t,yylval.ptr_simbolo);
23                              }
24                              return ID;
25                         }
26 [ \t]+      {;}
27 .|\n        { return yytext[0]; }
```

La línea 16 ha sido incluida para evitar errores graves en C cuando el programador utilice nombres de variables de más de 19 caracteres.

Por último, el programa Yacc realiza exactamente las acciones comentadas en el apartado 7.3.2.4. El control que exige que la cadena que se convierta en entero deba estar formada sólo por dígitos (con un guión opcional al principio para representar los negativos) se realiza mediante la función **esNumero()** definida en la sección de código C del área de definiciones. Además, se ha prestado especial cuidado en que los errores semánticos de tipos no produzcan más que un sólo mensaje por pantalla. Todos nos hemos enfrentado alguna vez a un compilador que, ante un simple error en una línea de código, proporciona tres o cuatro mensajes de error. Estas cadenas de errores dejan perplejo al programador, de manera que el constructor de compiladores debe

evitarlas en la medida de sus posibilidades. De esta forma, ante una entrada como “c := 4 * “p” * “k” * 5“, cuyo árbol se presenta en la figura 7.8.a) sólo se muestra un mensaje de error. Ello se debe a que los mensajes de error al sumar o multiplicar se muestran sólo cuando cada parámetro es correcto por sí sólo, pero operado con el adyacente produce un error. No obstante, una entrada como “c := 4 + “p” + “k” * 5“, cuyo árbol se presenta en la figura 7.8.b), posee dos errores de tipos: 1) la suma de 4 más “p”; y 2) el producto de “k” por 5; nuestro intérprete informa de los dos, ya que se producen en ramas diferentes del árbol.

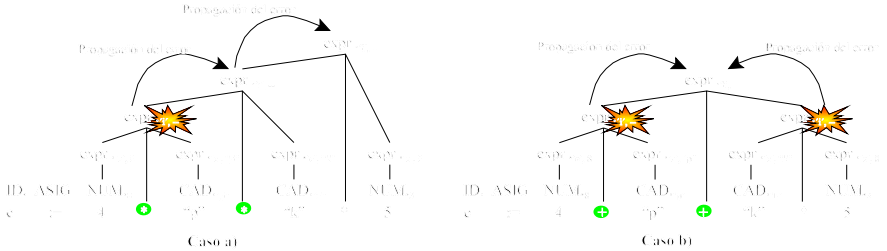


Figure 8 Gestión de errores de tipos

El fichero **TipSimpy.yac** es:

```
%{
#include "TabSimb.c"
typedef struct {
	char tipo;
	union {
		char valor_cad[100];
		int valor_int;
	} info;
} expresion;
int esNumero(char * s){
	int i=0;
	if (s[i] == '-') i++;
	for(; s[i] != 0; i++)
		if ((s[i]<'0') || (s[i]>'9')) return 0;
	return 1;
}
simbolo * t = NULL;
%}

%union{
	char cadena[100];
	int numero;
	simbolo * ptr_simbolo;
	expresion valor;
}
%token <cadena> CADENA
```

```
%token <numero> NUMERO
%token <ptr_simbolo> ID
%token IMPRIMIR ASIG A_ENTERO A_CADENA
%type <valor> expr
%start prog
%left '+'
%left '*'
%%
prog    :   /* Épsilon */
        |   prog asig '\n'
        |   prog IMPRIMIR expr '\n'{
                    if ($3.tipo == 'e')
                      printf("%d\n",$3.info.valor_int);
                    else if ($3.tipo == 'c')
                      printf("%s\n",$3.info.valor_cad);
                    else
                      printf("Indefinido.\n");
                }
        |   prog error '\n'     { yyerrok; }
        ;
asig    :   ID ASIG expr {
                    if($1->tipo == 'i')
                        $1->tipo = $3.tipo;
                    if ($3.tipo == 'i')
                        printf("Asignacion no efectuada.\n");
                    else
                        if ($3.tipo != $1->tipo)
                            printf("Asignacion de tipos incompatibles.\n");
                        else if($1->tipo =='e')
                                $1->info.valor_int = $3.info.valor_int;
                        else
                                strcpy($1->info.valor_cad, $3.info.valor_cad);
                }
    ;
expr    :   expr '+' expr {
                    if(($1.tipo == 'c') && ($3.tipo == 'c')) {
                        $$.tipo = 'c';
                        sprintf($$.info.valor_cad,
                                "%s%s", $1.info.valor_cad, $3.info.valor_cad);
                    } else if(($1.tipo == 'e') && ($3.tipo == 'e')) {
                        $$.tipo = 'e';
                        $$.info.valor_int = $1.info.valor_int + $3.info.valor_int;
                    } else {
                        $$.tipo = 'i';
                        if (($1.tipo != 'i') && ($3.tipo != 'i'))
                            printf("No se pueden sumar cadenas y enteros.\n");
                    }
                }
        |   expr '*' expr  {
```

```
76                          if(($1.tipo == 'c') || ($3.tipo == 'c')) {
77                              $$.tipo = 'i';
78                              printf("Una cadena no se puede multiplicar.\n");
79                          } else if(($1.tipo == 'e') || ($3.tipo == 'e')) {
80                              $$.tipo = 'e';
81                              $$.info.valor_int = $1.info.valor_int * $3.info.valor_int;
82                          } else
83                              $$.tipo = 'i';
84                      }
85              |       A_ENTERO '(' expr ')'{
86                          if ($3.tipo != 'c') {
87                              $$.tipo = 'i';
88                              printf("Error de conversión. Se requiere una cadena.\n");
89                          } else if (esNumero($3.info.valor_cad)){
90                              $$.tipo = 'e';
91                              $$.info.valor_int= atoi($3.info.valor_cad);
92                          } else{
93                              $$.tipo = 'i';
94                              printf("La cadena a convertir sólo puede tener dígitos.\n");
95                          }
96                      }
97              |       A_CADENA '(' expr ')'       {
98                          if ($3.tipo != 'e') {
99                              $$.tipo = 'i';
100                             printf("Error de conversión. Se requiere un entero.\n");
101                         } else {
102                             $$.tipo = 'c';
103                             itoa($3.info.valor_int, $$.info.valor_cad, 10);
104                         };
105                     }
106             |       ID  {
107                         $$.tipo = $1->tipo;
108                         if ($$.tipo == 'i')
109                             printf("Tipo de %s no definido.\n",$1->nombre);
110                         else
111                             if ($$.tipo == 'e')
112                                 $$.info.valor_int = $1->info.valor_int;
113                             else
114                                 strcpy($$.info.valor_cad, $1->info.valor_cad);
115                     }
116             |       NUMERO    {
117                         $$.tipo = 'e';
118                         $$.info.valor_int = $1;
119                     }
120         | CADENA    {
121                         $$.tipo = 'c';
122                         strcpy($$.info.valor_cad, $1);
123                     }
124             ;
```

```
%%
#include "TipSimpl.c"
void main(){
      yyparse();
      imprimir(t);
}
void yyerror(char * s){
      printf("%s\n",s);
}
```

7.3.4 Solución con JFlex/Cup

La solución con estas dos herramientas obedece a los mismos mecanismos que la anterior, con Lex/Yacc. Quizás la diferencia más notable sea el tratamiento del tipo de una expresión, o de una variable. En Java aprovechamos la clase **Object** para decir que el atributo de una expresión será un **Object** que apuntará a un **Integer**, a un **String** o a **null** en caso de que el valor sea indefinido. La función **tipo()** declarada en el ámbito de las acciones semánticas de Cup produce el carácter 'e', 'c' o 'i' en función de aquéllo a lo que apunte realmente un atributo.

Por otro lado, en la función **A_ENTERO()** no necesitamos una función en Java que nos diga si la cadena que se le pasa como parámetro tiene formato de número o no, ya que ello puede conocerse mediante la captura de la excepción **NumberFormatException** en el momento de realizar la conversión.

Así, el fichero **TablaSimbolos.java** es:

```
import java.util.*;
class Simbolo{
      String nombre;
      Object valor;
      public Simbolo(String nombre, Object valor){
          this.nombre = nombre;
          this.valor = valor;
      }
}
public class TablaSimbolos{
      HashMap t;
      public TablaSimbolos(){
          t = new HashMap();
      }
      public Simbolo insertar(String nombre){
          Simbolo s = new Simbolo(nombre, null);
          t.put(nombre, s);
          return s;
      }
      public Simbolo buscar(String nombre){
          return (Simbolo)(t.get(nombre));
      }
      public void imprimir(){
```

```
24              Iterator it = t.values().iterator();
25              while(it.hasNext()){
26                   Simbolo s = (Simbolo)it.next();
27                   System.out.print(s.nombre + ": ");
28                   if (s.valor == null) System.out.print("i: Indefinido");
29                   else if (s.valor instanceof Integer) System.out.print("e: "+s.valor.toString());
30                   else if (s.valor instanceof String) System.out.print("c: "+s.valor);
31              }
32         }
33 }
```

El fichero **TipSimb.jflex** queda:

```
 1 import java_cup.runtime.*;
 2 import java.io.*;
 3 %%
 4 %{
 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.POR); }
18 "("        { return new Symbol(sym.LPAREN); }
19 ")"        { return new Symbol(sym.RPAREN); }
20 "\n"       { return new Symbol(sym.RETORNODECARRO); }
21 ":="       { return new Symbol(sym.ASIG); }
22 \"[^\"]*\"  { return new Symbol(sym.CADENA,yytext().substring(1,yytext().length()-1));}
23 [:digit:]+ { return new Symbol(sym.NUMERO, new Integer(yytext())); }
24 "PRINT"                  { return new Symbol(sym.IMPRIMIR); }
25 "A_CADENA"              { return new Symbol(sym.A_CADENA); }
26 "A_ENTERO"              { return new Symbol(sym.A_ENTERO); }
27 [:jletter:][:jletterdigit:]*  {
28                                   Simbolo s;
29                                   if ((s = tabla.buscar(yytext())) == null)
30                                        s = tabla.insertar(yytext());
31                                   return new Symbol(sym.ID, s);
32                              }
33 [ \t\r]+   {;}
34 .          { System.out.println("Error léxico."+yytext()+"-"); }
```

Y por último, **TipSimp.cup** es:

```
1 import java_cup.runtime.*;
2 import java.io.*;
```

```
parser code {:
    static TablaSimbolos tabla = new TablaSimbolos();
    public static void main(String[] arg){
        parser parserObj = new parser();
        Yylex miAnalizadorLexico =
                    new Yylex(new InputStreamReader(System.in), tabla);
        parserObj.setScanner(miAnalizadorLexico);
        try{
            parserObj.parse();
            tabla.imprimir();
        }catch(Exception x){
            x.printStackTrace();
            System.out.println("Error fatal.\n");
        }
    }
:};
action code {:
    private static char tipo(Object o){
        if (o == null) return 'i';
        else if (o instanceof Integer) return 'e';
        else return 'c';
    }
:}
terminal RETORNODECARRO, MAS, POR;
terminal IMPRIMIR, ASIG, LPAREN, RPAREN, A_ENTERO, A_CADENA;
terminal Simbolo ID;
terminal Integer NUMERO;
terminal String CADENA;
non terminal asig, prog;
non terminal Object expr;
precedence left MAS;
precedence left POR;
/* Gramática */
prog    ::=  /* Épsilon */
             |   prog asig RETORNODECARRO
             |   prog IMPRIMIR expr:e RETORNODECARRO {:
                        if (tipo(e) == 'i')
                          System.out.println("Indefinido.");
                        else
                          System.out.println(e.toString());
                    :}
             | prog error RETORNODECARRO
             ;
asig    ::= ID:s ASIG expr:e {:
                if (tipo(e) == 'i')
                    System.out.println("Asignacion no efectuada.");
                else
                    if ((s.valor == null) || (tipo(s.valor) == tipo(e)))
                        s.valor = e;
```

```
52                          else
53                              System.err.println("Asignacion de tipos incompatibles.");
54                      :}
55          ;
56  expr    ::=     expr:e1 MAS expr:e2 {:
57                  if((tipo(e1) == 'c') && (tipo(e2) == 'c'))
58                      RESULT = e1.toString() + e2.toString();
59                  else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
60                      RESULT = new Integer(     ((Integer)e1).intValue()
61                                                + ((Integer)e2).intValue());
62                  else {
63                      RESULT = null;
64                      if ((tipo(e1) != 'i') && (tipo(e2) != 'i'))
65                          System.err.println("No se pueden sumar cadenas y enteros.");
66                  }
67          :}
68          |       expr:e1 POR expr:e2 {:
69                          if((tipo(e1) == 'c') || (tipo(e2) == 'c')) {
70                              RESULT = null;
71                              System.err.println("Una cadena no se puede multiplicar.");
72                          } else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
73                              RESULT = new Integer(     ((Integer)e1).intValue()
74                                                        * ((Integer)e2).intValue());
75                          else
76                              RESULT = null;
77                      :}
78          | A_ENTERO LPAREN expr:e RPAREN {:
79                  if (tipo(e) != 'c') {
80                      RESULT = null;
81                      System.err.println("Error de conversión. Se requiere una cadena.");
82                  } else try {
83                      RESULT = new Integer(Integer.parseInt(e.toString()));
84                  } catch (Exception x){
85                      RESULT = null;
86                      System.err.println("La cadena a convertir sólo puede tener dígitos.");
87                  }
88          :}
89          | A_CADENA LPAREN expr:e RPAREN {:
90                          if (tipo(e) != 'e') {
91                              RESULT = null;
92                              System.err.println("Error de conversión. Se requiere un entero.");
93                          } else
94                              RESULT = e.toString();
95                      :}
96          | ID:s      {:
97                          RESULT = s.valor;
98                          if (tipo(s.valor) == 'i')
99                              System.err.println("Tipo de "+ s.nombre +" no definido.");
100                 :}
```

```
101         | NUMERO:n      {: RESULT = n; :}
102         | CADENA:c      {: RESULT = c; :}
103         ;
```

7.3.5 Solución con JavaCC

Construir la solución en JavaCC una vez solucionado el problema con JFlex/Cup resulta muy sencillo. Básicamente se han seguido los siguientes pasos:

- Crear la clase contenedora con la función **main()**.
- Reconocer los *tokens* necesarios.
- Incluir la gramática del punto 7.3.1.
- Incluir variables en las reglas para almacenar los atributos retornados por los no terminales.
- Las acciones semánticas a incluir son las mismas que en JFlex/Cup; en este caso hemos optado por incluir estas acciones en funciones propias de la clase **Calculadora**, sólo que cambiando cada asignación a **RESULT** de la forma "RESULT = dato;" por "return dato;" y haciendo que ésta sea la última sentencia de cada acción. El extraer las acciones en funciones aparte aumenta la legibilidad y claridad del código, aunque se pierde la localidad espacial de las reglas y sus acciones.

El programa **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          private static char tipo(Object o){
10             if (o == null) return 'i';
11             else if (o instanceof Integer) return 'e';
12             else return 'c';
13         }
14         private static void usarIMPRIMIR(Object e){
15             if (tipo(e) == 'i')
16               System.out.println("Indefinido.");
17             else
18               System.out.println(e.toString());
19         }
20         private static void usarASIG(Simbolo s, Object e){
21             if (tipo(e) == 'i')
22                 System.out.println("Asignacion no efectuada.");
23             else
24                 if ((s.valor == null) || (tipo(s.valor) == tipo(e)))
```

```
25                          s.valor = e;
26                      else
27                          System.err.println("Asignacion de tipos incompatibles.");
28          }
29          private static Object usarMAS(Object e1, Object e2){
30              if((tipo(e1) == 'c') && (tipo(e2) == 'c'))
31                  return e1.toString() + e2.toString();
32              else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
33                  return new Integer(((Integer)e1).intValue() + ((Integer)e2).intValue());
34              else {
35                  if ((tipo(e1) != 'i') && (tipo(e2) != 'i'))
36                      System.err.println("No se pueden sumar cadenas y enteros.");
37                  return null;
38              }
39          }
40          private static Object usarPOR(Object e1, Object e2){
41              if((tipo(e1) == 'c') || (tipo(e2) == 'c')) {
42                  System.err.println("Una cadena no se puede multiplicar.");
43                  return null;
44              } else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
45                  return new Integer(((Integer)e1).intValue() * ((Integer)e2).intValue());
46              else
47                  return null;
48          }
49          private static Object usarID(Simbolo s){
50              if (tipo(s.valor) == 'i')
51                  System.err.println("Tipo de "+ s.nombre +" no definido.");
52              return s.valor;
53          }
54          private static Object usarA_CADENA(Object e){
55              if (tipo(e) != 'e') {
56                  System.err.println("Error de conversión. Se requiere un entero.");
57                  return null;
58              } else
59                  return e.toString();
60          }
61          private static Object usarA_ENTERO(Object e){
62              if (tipo(e) != 'c') {
63                  System.err.println("Error de conversión. Se requiere una cadena.");
64                  return null;
65              } else try {
66                  return new Integer(Integer.parseInt(e.toString()));
67              } catch (Exception x){
68                  System.err.println("La cadena a convertir sólo puede tener dígitos.");
69                  return null;
70              }
71          }
72      }
73  PARSER_END(Calculadora)
```

```
SKIP : {
        " "
    |   "\t"
    |   "\r"
}
TOKEN [IGNORE_CASE] :
{
        <IMPRIMIR: "PRINT">
    |   <A_CADENA: "A_CADENA">
    |   <A_ENTERO: "A_ENTERO">
    |   <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
    |   <NUMERO: (["0"-"9"])+>
    |   <CADENA: "\""(~["\""])*"\"">
    |   <RETORNODECARRO: "\n">
}
/*
gramatica ::= ( sentFinalizada )*
*/
void gramatica():{}{
    (sentFinalizada())*
}
/*
sentFinalizada ::= IMPRIMIR expr '\n' | ID ASIG expr '\n' | error '\n'
*/
void sentFinalizada():{
    Simbolo s;
    Object e;
}{ try {
        <IMPRIMIR> e=expr() <RETORNODECARRO>  { usarIMPRIMIR(e); }
    |   s=id() ":=" e=expr() <RETORNODECARRO>     { usarASIG(s, e); }
    }catch(ParseException x){
        System.out.println(x.toString());
        Token t;
        do {
            t = getNextToken();
        } while (t.kind != RETORNODECARRO);
    }
}
/*
expr ::= term ('+' term)*
*/
Object expr():{
    Object t1, t2;
}{
    t1=term() ( "+" t2=term() { t1=usarMAS(t1, t2); } )* { return t1; }
}
/*
term ::= fact ('*' fact)*
*/
```

```
123 Object term():{
124     Object f1, f2;
125 }{
126     f1=fact() ( "*" f2=fact() { f1=usarPOR(f1, f2); } )*     { return f1; }
127 }
128 /*
129 fact ::= ID | NUMERO | CADENA | A_CADENA '(' expr ')' | A_ENTERO '(' expr ')'
130 */
131 Object fact():{
132     Simbolo s;
133     int i;
134     String c;
135     Object e;
136 }{
137         s=id()          { return usarID(s); }
138     |   i=numero()      { return new Integer(i); }
139     |   c=cadena()      { return c; }
140     |   <A_CADENA> "(" e=expr() ")"   { return usarA_CADENA(e); }
141     |   <A_ENTERO> "(" e=expr() ")"   { return usarA_ENTERO(e); }
142 }
143 Simbolo id():{}{
144     <ID>    {
145                 Simbolo s;
146                 if ((s = tabla.buscar(token.image)) == null)
147                     s = tabla.insertar(token.image);
148                 return s;
149             }
150 }
151 int numero():{}{
152     <NUMERO> { return Integer.parseInt(token.image); }
153 }
154 String cadena():{}{
155     <CADENA> { return token.image.substring(1, token.image.length() - 1); }
156 }
157 SKIP : {
158     <ILEGAL: (~[])>    { System.out.println("Carácter: "+image+" no esperado.");}
159 }
```

7.4 Gestión de tipos complejos

Una vez estudiados los tipos primitivos, en este apartado se estudiarán las características sintácticas y semánticas de un compilador que permite al programador construir tipos de datos complejos, punteros, *arrays*, etc.

Cuando un lenguaje de programación permite usar tipos complejos, debe suministrar construcciones sintácticas tanto para construir nuevos tipos como para referenciarlos. Algunos ejemplos de **constructores de tipo** son:

- POINTER TO. Para crear punteros.
- RECORD OF. Para crear registros.

- ARRAY OF. Para crear tablas y matrices.
- PROCEDURE RETURNS. Para crear funciones que devuelven un resultado.

Éstos no son realmente tipos sino constructores de tipo ya que, aunque permiten construir tipos complejos, no tienen sentido por sí solos y deben ser aplicados sobre un tipo base que puede ser, a su vez, otro constructor de tipo o bien un tipo primitivo. Los constructores de tipo son, por tanto, recursivos. El caso del registro es más especial porque puede aplicarse al producto cartesiano de varios tipos.

Todo esto nos lleva a dos diferencias fundamentales con respecto a la gestión de tipos primitivos que se hizo en el apartado anterior. En primer lugar, el lenguaje a definir requiere una zona de declaraciones donde enumerar las variables de un programa e indicar su tipo. En otras palabras, el tipo de una variable ya no vendrá dado por su primera asignación sino que éste deberá declararse explícitamente. Y en segundo lugar, dado que un tipo complejo puede ser arbitrariamente largo, éste no podrá codificarse con una sola letra (‘e’ para los enteros, ‘c’ para las cadenas, etc.) sino que habrá que recurrir a algún otro método.

Por otro lado, cuando se usa una variable declarada de cualquiera de los tipos anteriores, es necesario utilizar un **modificador de tipo** para referenciar los componentes. Por ejemplo:

- ^. Colocado al lado de un puntero, representa aquéllo a lo que se apunta.
- .campo. Colocado al lado de un registro, representa uno de sus campos.
- [nº]. Colocado al lado de un array, representa uno de sus elementos.
- (). Colocado al lado de una función, invoca a ésta y representa el valor de retono.

Como puede observarse, cada constructor de tipo tiene su propio modificador de tipo y éstos no son intercambiables, es decir, el “^” sólo puede aplicarse a punteros y a nada más, el “.” sólo puede aplicarse a registros, etc. Esto nos da una tabla que relaciona biunívocamente cada constructor de tipo con su modificador:

Constructores de tipos	Modificadores de tipos
POINTER TO	^
RECORD OF	.campo
ARRAY OF	[nº]
PROCEDURE RETURNS	()

También es importante darse cuenta de que un modificador de tipo no tiene porqué aplicarse exclusivamente a una variable, sino que puede aplicarse a una expresión que tenga, a su vez, otro modificador de tipo, p.ej.: “ptr^.nombre” se refiere al campo **nombre** del registro apuntado por **ptr**, y el modificador “.” se ha aplicado a la expresión **ptr^** y no a una variable directamente.

7.4.1 Objetivos y propuesta de un ejemplo

En los siguientes puntos se creará la gestión de los siguientes tipos complejos:

- Punteros
- Arrays de dimensiones desconocidas.
- Funciones sin parámetros.

El hecho de desconocer las dimensiones de los *arrays* y de obviar los parámetros de las funciones nos permitirá centrarnos exclusivamente en la gestión de tipos. Por otro lado, admitiremos los tipos básicos:

- Lógico o booleano.
- Entero.
- Real.
- Carácter.

En total tenemos cuatro tipos primitivos y tres constructores de tipos. Es importante observar que por cada tipo primitivo es necesario suministrar al programador algún mecanismo con el que pueda definir constantes de cada uno de esos tipos: constantes *booleanas*, enteras, reales y de carácter.

En lo que sigue construiremos la gestión de tipos de un compilador, lo que quiere decir que dejaremos a un lado el ejemplo de la calculadora y nos centraremos en reconocer declaraciones y expresiones válidas. Para comprobar que nuestro compilador está funcionando bien, cada vez que el programador introduzca una expresión se nos deberá mostrar un mensaje donde, en lenguaje natural, se informe del tipo de dicha expresión. Lo siguiente es un ejemplo de entrada:

```
a: POINTER TO BOOLEAN;
a^;
Es un boolean.
a;
Es un puntero a un boolean.
beta : POINTER TO ARRAY[] OF POINTER TO PROCEDURE(): INTEGER;
beta^;
Es un array de un puntero a una función que devuelve un entero.
beta[2];
Esperaba un array.
beta();
Esperaba una función.
true;
Es un boolean.
```

7.4.2 Pasos de construcción

Una vez propuesto el ejemplo, es necesario definir una gramática que reconozca las distintas cláusulas, y proponer los atributos que debe tener cada terminal y no terminal para, en base a acciones semánticas, culminar con nuestro objetivo de controlar los tipos de datos compuestos.

7.4.2.1 Gramática de partida

La gramática que se va a utilizar para el reconocimiento de las sentencias del ejemplo anterior es:

```
prog    :       /* Épsilon */
        |       prog decl ';'
        |       prog expr ';'
        |       prog error ';'
        |
        ;
decl    :       ID ',' decl
        |       ID ':' tipo
        ;
tipo    :       INTEGER
        |       REAL
        |       CHAR
        |       BOOLEAN
        |       POINTER TO tipo
        |       ARRAY '[' ']' OF tipo
        |       PROCEDURE '(' ')' ':' tipo
        ;
expr    :       ID
        |       NUM
        |       NUMREAL
        |       CARACTER
        |       CTELOGICA
        |       expr '^'
        |       expr'['NUM']'
        |       expr'(')'
        ;
```

Esta gramática expresada en notación EBNF de JavaCC queda:

```
prog():{}{
        ( bloque() )*
}
bloque():{}{
                LOOKAHEAD(2)
                decl() <PUNTOYCOMA>
        |       expr() <PUNTOYCOMA>
        |       /* Gestión del error */
}
decl():{}{
                LOOKAHEAD(2)
                <ID> ":" tipo()
        |       <ID> "," decl()
}
tipo():{}{
        (       <POINTER> <TO>
        |       <ARRAY> "[" "]" <OF>
        |       <PROCEDURE> "(" ")" ":"
        )* (
                <INTEGER>
        |       <REAL>
        |       <CHAR>
```

```
        |   <BOOLEAN>
        )
}
expr():{}{
        (   <ID>    (
                        "^"
                    |   "[" <NUM> "]"
                    |   "(" ")"
                    )*
        |   <NUM>
        |   <NUMREAL>
        |   <CARACTER>
        |   <CTELOGICA>
        )
}
```

en la que puede observarse que se ha utilizado recursión a derecha para facilitar el proceso de propagación de atributos, como se verá posteriormente. Además, esta gramática no permite construcciones de la forma 8^, mientras que la dada a través de reglas de producción sí; por tanto, en una se detectará como error sintáctico y en otra como error semántico de tipos.

7.4.2.2 Gestión de atributos

La gestión de los atributos de los símbolos terminales no reviste demasiada complejidad. Los *tokens* que representan constantes no necesitan atributo alguno puesto que nuestro objetivo no es procesar valores ni generar código equivalente. Por otro lado, como nuestro lenguaje define un mecanismo para hacer explícitas las declaraciones de variables, lo mejor es que el analizador léxico proporcione al sintáctico los nombres de las variables para que éste las inserte o las busque en la tabla de símbolos según éstas aparezcan en una declaración o no.

Un problema más complejo consiste en cómo identificar el tipo de una variable compleja (ya que hemos visto que no es posible hacerlo mediante una única letra). En lo que sigue, supongamos que el usuario define la siguiente variable:

```
alfa: POINTER TO ARRAY [23] OF POINTER TO PROCEDURE(): ARRAY
      [2] OF INTEGER;
```

Indiscutiblemente vamos a necesitar una tabla de símbolos donde almacenar el nombre de cada variable y su tipo. Además, el tipo hay que almacenarlo de alguna forma inteligente que luego nos sirva para detectar errores y que permita saber qué modificador de datos se puede aplicar y cuáles no. La forma en que se decida guardar el tipo de una variable también nos debe servir para guardar el tipo de una expresión una vez se haya aplicado a una variable una secuencia de modificadores de tipo válidos.

La solución va a consistir en codificar cada tipo o constructor de tipo con una letra, incluyendo el tipo indefinido que representaremos con una 'u' de *undefined*:

Tipos y constructores	Letra-código
POINTER TO	p
ARRAY [] OF	a
PROCEDURE() :	f
BOOLEAN	b
INTEGER	i
REAL	r
CHAR	c
INDEFINIDO	u

Y un tipo completo se representará mediante una pila en la que se guardará una secuencia de estas letras. La figura 7.9 muestra las letras asociadas al tipo de la

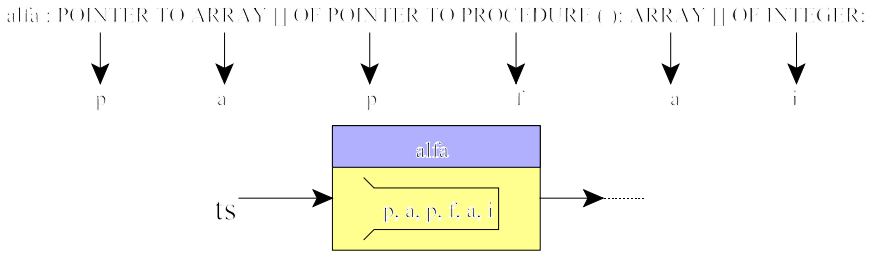


Figure 9 Codificación de un tipo complejo y estructura de la tabla de símbolos

variable **alfa** y la estructura de esta variable en la tabla de símbolos.

La pila se ha construido mirando el tipo de derecha a izquierda, lo que asegura que en la cima de la pila se tenga el tipo principal de **alfa**: **alfa** es, ante todo, un puntero (representado por la 'p'). Es por este motivo que en el punto se construyó una gramática recursiva a la derecha ya que las reducciones se harán en el orden en que se deben introducir elementos en la pila.

Además, si una expresión tiene asociado como atributo una pila de tipos resulta muy sencillo averiguar qué modificador se le puede aplicar:

- ^, si la cima es 'p'.
- [NUM], si la cima es 'a'.
- (), si la cima es 'f'.

Y no sólo eso resulta fácil, también resulta fácil calcular la pila de tipos de una expresión tras aplicarle un modificador de tipo. Por ejemplo, si la variable **gamma** es de tipo POINTER TO **X** entonces **gamma^** es de tipo **X**, esto es, basta con quitar la cima de la pila de tipos asociada a **gamma** para obtener la pila de tipos de la expresión **gamma^**.

7.4.2.3 Implementación de una pila de tipos

La pila de caracteres que se propone puede implementarse de muy diversas maneras. En nuestro caso, un *array* hará las veces de pila de forma que la posición 0 almacenará el número de elementos y éstos se colocarán de la posición 1 en adelante, correspondiendo la posición más alta a la cima de la pila. Con 20 posiciones será suficiente. La estructura de la pila de tipos de **alfa** se muestra en la figura 7.10.

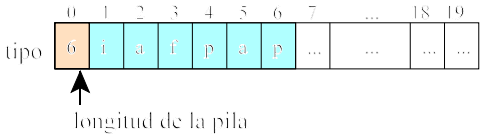


Figure 10 Estructura de almacenamiento del tipo de la variable **alfa**

Las operaciones de apilar, desapilar y cima son muy sencillas con esta estructura. Supongamos que la variable **tipo** es **char[20]**, entonces:

- **apilar(char x)**. Consiste en incrementar tipo[0] y luego meter x en tipo[tipo[0]]; en una sola instrucción: tipo[++tipo[0]] = x;
- **desapilar()**. Consiste únicamente en decrementar la longitud de tipo[0], quedando tipo[0]--;
- **char cima()**. La posición 0 nos da la posición en que se encuentra la cima, luego en una sola instrucción: return tipo[tipo[0]];

Aunque éste será el método que usaremos, es evidente que se está desaprovechando memoria al almacenar los tipos. Ello no debe preocuparnos dadas las cantidades de memoria que se manejan hoy día. En caso de que la memoria fuese un problema habría que utilizar algún tipo de codificación basada en bits. Dado que en total tenemos 8 letras que representar, nos bastaría con 3 bits para representar cada una de ellas, como se muestra en la siguiente tabla:

Tipos y constructores	Código de bits
POINTER TO	p - 100
ARRAY [] OF	a - 101
PROCEDURE() :	f - 110
BOOLEAN	b - 111
INTEGER	i - 001
REAL	r - 010
CHAR	c - 011
INDEFINIDO	u - 000

Además, suponiendo que el espacio de almacenamiento sobrante se rellena con 0s y dado que el código **000** se ha asociado al tipo indefinido, es posible no almacenar el tamaño de la pila sino que para encontrar la cima bastará con buscar la

primera secuencia de tres bits diferente de 000. Si esta no existe es que el tipo es indefinido.

Según esta codificación, los códigos **111**, **001**, **010**, **011** y **000** sólo pueden encontrarse en la base de la pila; es más, los códigos **100**, **101** y **110** nunca podrán estar en la base dado que requieren otro tipo por debajo sobre el que aplicarse. Este hecho nos permite utilizar los mismos códigos para representar tanto tipos primitivos como constructores de tipo: si se halla en la base de la pila es que se refiere a un tipo primitivo y en caso contrario lo que se referencia es un constructor. Si además representamos los tipos mediante un puntero a entero, el tipo indefinido vendría dado por un puntero a **null**, con lo que la codificación sería:

Tipos y constructores	Código de bits
POINTER TO	p - 01
ARRAY [] OF	a - 10
PROCEDURE() :	f - 11
BOOLEAN	b - 00
INTEGER	i - 01
REAL	r - 10
CHAR	c - 11

donde las líneas con el mismo color tienen asociado el mismo código de bits. Nótese cómo, de nuevo, no es posible usar el código **00** para representar a un constructor de tipo siempre que no se quiera guardar la longitud del tipo y los bits no utilizados se rellenen a 0.

No puede existir confusión, ya que lo que está en la base de la pila sólo puede ser un tipo primitivo; en la figura 7.11 se muestra cómo quedaría el tipo de **alfa** codificado con un entero de 16 bits.

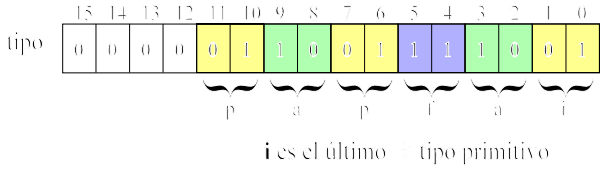


Figure 11 Codificación en base a bits

7.4.2.4 Acciones semánticas

Para las acciones semánticas vamos a centrarnos en la gramática dada a través de reglas de producción. En el lenguaje que estamos definiendo, existe zona de declaraciones y zona de utilización de variables (aunque pueden entremezclarse) por lo tanto es el analizador sintáctico quien inserta los identificadores en la tabla de símbolos en el momento en que se declaran.

Como puede observarse en las reglas de **prog**, existe una regla épsilon que nos permite entrar por ella justo al comienzo del análisis, como paso base de la recursión de las reglas de **prog** sobre sí misma.

En las reglas de **decl** y de **tipo**, hacemos la gramática recursiva a la derecha. Así, las reglas de **tipo** permiten construir la pila desde la base hasta la cima, ejecutando tan sólo operaciones **apilar**. Por otro lado, las reglas **decl** permiten asignar el tipo a cada uno de los identificadores declarados, también de derecha a izquierda, tal y como se ilustra en la figura 7.12. Como puede apreciarse en ella, cuando el analizador sintáctico se encuentra un tipo primitivo construye una pila con un sólo carácter (el correspondiente al tipo encontrado) y lo asigna como atributo del no terminal **tipo**. A medida que se reduce a nuevos no terminales **tipo** en base a las reglas recursivas a la derecha, se toma el atributo del **tipo** del consecuente como punto de partida para construir el atributo del antecedente; basta con añadir a éste el carácter correspondiente al constructor de tipo de que se trate (marcados en gris en la figura 7.12). Una vez construido el tipo completo, el analizador sintáctico debe reducir por la regla:

```
decl   :      ID ':' tipo
```

de tal manera que debe crearse una entrada en la tabla de símbolos para la variable indicada como atributo de **ID** (**c** según la figura 7.12); el tipo de esta variable vendrá dado por el atributo de **tipo**. Por supuesto es necesario comprobar que no se ha realizado una redeclaración de variables. Dado que es posible realizar declaraciones de varias variables simultáneamente, el resto de variables (**b** y **a** segun la figura 7.12) se reconocen mediante la reducción de la regla:

```
decl   :      ID ',' decl
```

Obviamente el proceso debe ser muy similar al de la regla anterior, esto es, crear una entrada en la tabla de símbolos para la variable del atributo de **ID** y asignarle una pila de tipos. Claro está, para ello es necesario disponer de la pila de tipos como atributo del no terminal **decl**, por lo que dicha pila debe ser propagada en ambas reglas de producción desde el no terminal del consecuente al antecedente, o sea, a **decl**. En conclusión, tanto **tipo** como **decl** tienen como atributo una pila de tipos.

Pasemos ahora a estudiar las reglas de las expresiones. El objetivo es que una expresión tenga también asociada una pila de tipos de manera que, una vez completado el análisis de la expresión completa, su pila de tipos se pase como parámetro a una función que traslade su contenido a una frase en lenguaje natural.

Las expresiones pueden dividirse en tres bloques: 1) primitivas, formadas por una constante de cualquier tipo; 2) simples, formadas por una variable; y 3) complejas, formadas por una expresión simple con modificadores de tipo a su derecha.

Las expresiones del tipo 1 tienen asociada una pila de tipos con un único carácter, que se corresponderá con el tipo del literal reconocido:

- ‘i’ para NUM.
- ‘r’ para NUMREAL.
- ‘c’ para CARACTER.
- ‘b’ para CTELOGICA.

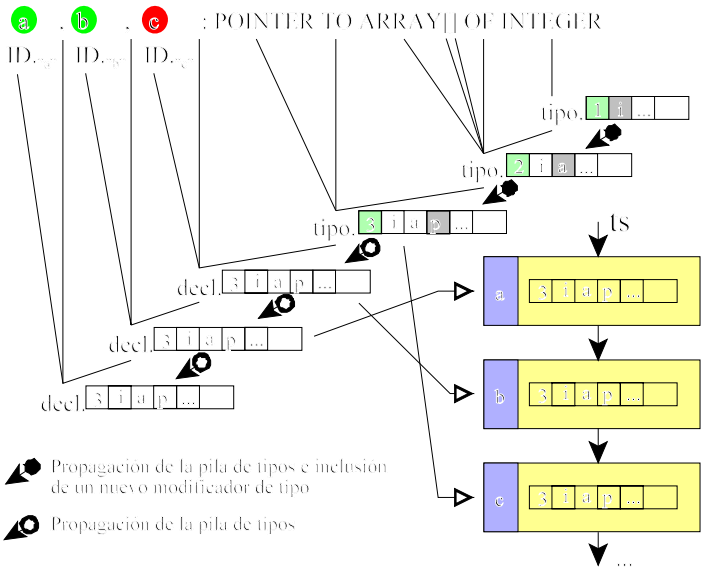


Figure 12 Construcción de una pila-tipo y asignación a cada variable en la tabla de símbolos

Las expresiones del tipo 2 tienen una pila de tipos exactamente igual que la de la variable a que representan. Si la variable no ha sido declarada entonces la pila de tipos tendrá únicamente una 'u' que representa al tipo indefinido (no se puede usar la letra 'i' porque se confundiría con la 'i' de **INTEGER**).

Las expresiones del tipo 3 parten de una expresión (que tiene una pila de tipos asociada) junto con un modificador de tipo. En este caso hay que controlar que el tipo principal de la expresión (representado por la cima de su pila de tipos), es compatible con el modificador de tipo, tal y como se vio al final del apartado 7.4.2.2. Por ejemplo, si encontramos "expr^", "expr[NUM]" o "expr()" hay que controlar que el tipo principal de alfa sea 'p', 'a' o 'f' respectivamente. Suponiendo que la variable **a** ha sido declarada tal y como aparece en la figura 7.12, la figura muestra cómo varía la pila de tipos durante el reconocimiento de la expresión "a^[3]".

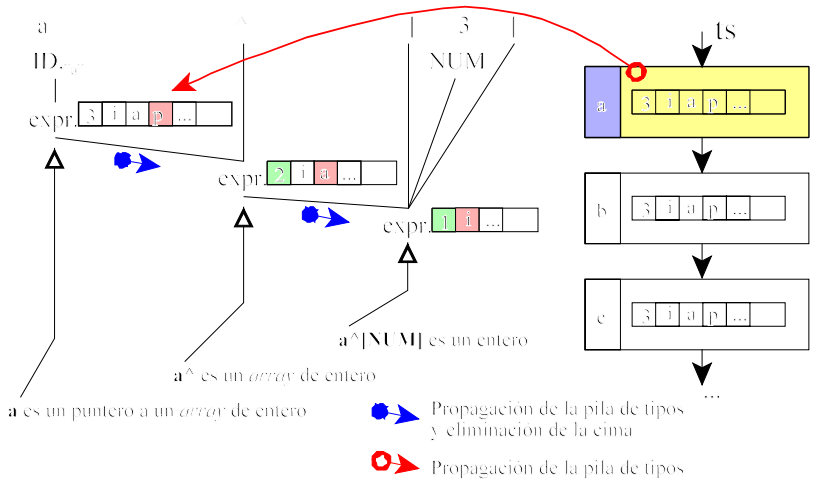
En el momento en que reduce por la regla:

```
expr    :    ID
```

se busca en la tabla de símbolos a la variable que viene dada como atributo de **ID**; una vez encontrada, el atributo de **expr** será una copia de la pila de tipos de dicha variable. A medida que se van aplicando las reglas:

```
expr    :    expr '^'
        |    expr'['NUM']'
        |    expr'('')'
        ;
```

se va comprobando que el modificador de la regla aplicada sea compatible con el tipo de la cima de la pila de la expresión a que se aplica. Si es compatible, entonces el tipo de la expresión resultante (el antecedente) será el mismo que la del consecuente pero sin la cima. Si no se debe emitir un mensaje de error y la expresión resultante será de



tipo indefinido.

En resumen, los constructores de tipo construyen la pila metiendo caracteres en ella, mientras que los modificadores de tipo destruyen la pila sacando caracteres por la cima.

7.4.3 Solución con Lex/Yacc

Lo más interesante de esta solución radica en la implementación del tipo de datos pila y de sus operaciones, especialmente de los controles de los límites del *array* que se han incluido. Estas operaciones obedecen básicamente al esquema propuesto en el apartado 7.4.2.3, y se codifican entre las líneas 9 y 32 del fichero **TabSimb.c** que se presenta a continuación:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define TAM_PILA 21
4  #define TAM_NOMBRE 20
5  /*
6       El caracter 0 indica el número de posiciones ocupadas en
7       el resto de la cadena
8  */
9  typedef char pila_tipo[TAM_PILA];
10 void crearPila(pila_tipo tipo, char x){ tipo[0]=1; tipo[1]=x; tipo[TAM_PILA-1]=0; }
11 void insertarPila(pila_tipo tipo, char x){
12      if (tipo[0] < TAM_PILA-2) tipo[++tipo[0]]=x;
13      else printf("Tipo demasiado complejo.\n");
14 }
15 void eliminarPila(pila_tipo tipo){ if (tipo[0] > 0) tipo[0]--; }
16 char cimaPila(pila_tipo tipo) { return tipo[tipo[0]]; }
17 void copiarPila(pila_tipo destino, pila_tipo origen) { strcpy(destino, origen); }
18 void verPila(pila_tipo tipo) {
19      unsigned char cont;
20      printf("El tipo es ");
21      for(cont = tipo[0]; cont>0; cont--)
22      switch(tipo[cont]) {
23           case('i'): { printf("un entero.\n"); break; }
24           case('r'): { printf("un real.\n"); break; }
25           case('b'): { printf("un booleano.\n"); break; }
26           case('c'): { printf("un caracter.\n"); break; }
27           case('p'): { printf("un puntero a "); break; }
28           case('a'): { printf("un array de "); break; }
29           case('f'): { printf("una funcion que devuelve "); break; }
30           case('u'): { printf("indefinido.\n"); break; }
31      };
32 }
33 /* Definición de la tabla de símbolos */
34 typedef struct _simbolo {
35      struct _simbolo * sig;
36      char nombre[TAM_NOMBRE];
```

```
37      pila_tipo tipo;
38  } simbolo;
39  void insertar(simbolo ** p_t, char nombre[TAM_NOMBRE], pila_tipo tipo) {
40      simbolo * s = (simbolo *) malloc(sizeof(simbolo));
41      strcpy(s->nombre, nombre);
42      strcpy(s->tipo, tipo);
43      s->sig = (*p_t);
44      (*p_t) = s;
45  }
46  simbolo * buscar(simbolo * t, char nombre[TAM_NOMBRE]) {
47      while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
48          t = t->sig;
49      return (t);
50  };
51  void imprimir(simbolo * t) {
52      while (t != NULL) {
53              printf("%s. ", t->nombre);
54              verPila(t->tipo);
55              t = t->sig;
56          }
57  }
```

Como se deduce de la función **verPila()**, convertir una pila de tipos en una frase en lenguaje natural que la describa es tan sencillo como concatenar trozos de texto prefijados para cada letra o código de tipo.

El fichero **TipCompl.lex** realiza el análisis léxico, prestando cuidado de que el programador no escriba identificadores demasiado largos. En caso de que un identificador tenga más de 19 caracteres, el propio analizador lo trunca informando de ello en un mensaje por pantalla. Es muy importante informar de este truncamiento, ya que puede haber identificadores muy largos en los que coincidan sus primeros 19 caracteres, en cuyo caso nuestro compilador entenderá que se tratan del mismo identificador. El programa Lex es:

```
1  %%
2  [0-9]+          {return NUM;}
3  [0-9]+\.[0-9]+  {return NUMREAL;}
4  \'.\'           {return CARACTER;}
5  "TRUE"  |
6  "FALSE"         {return CTELOGICA;}
7  "INTEGER"       {return INTEGER;}
8  "REAL"          {return REAL;}
9  "CHAR"          {return CHAR;}
10 "BOOLEAN"       {return BOOLEAN;}
11 "POINTER"           {return POINTER;}
12 "TO"                {return TO;}
13 "ARRAY"             {return ARRAY;}
14 "OF"                {return OF;}
15 "PROCEDURE"     {return    PROCEDURE;}
16 [a-zA-Z_][a-zA-Z0-9_]*      {
```

```
17                          if (strlen(yytext) >= TAM_NOMBRE) {
18                              printf("Variable %s truncada a ", yytext);
19                              yytext[TAM_NOMBRE-1] = 0;
20                              printf("%s.\n", yytext);
21                          }
22                          strcpy(yylval.nombre, yytext);
23                          return ID;
24                      }
25 [ \t\n]+          {;}
26 .                 {return yytext[0];}
```

El programa Yacc que soluciona el analizador sintáctico y semántico se mantiene relativamente reducido al haberse extraído la funcionalidad de la pila de tipos y de la tabla de símbolos en funciones aparte en el fichero **TabSimb.c**. El fichero **TipCompy.yac** es:

```
 1 %{
 2 #include "TabSimb.c"
 3 simbolo * tabla=NULL;
 4 %}
 5 %union {
 6       pila_tipo tipo;
 7       char nombre[20];
 8 }
 9 %token INTEGER REAL CHAR BOOLEAN POINTER TO
10 %token ARRAY OF PROCEDURE
11 %token NUM CARACTER NUMREAL CTELOGICA
12 %token <nombre> ID
13 %type <tipo> tipo expr decl
14 %start prog
15 %%
16 prog      :     /*´Epsilon */
17           |     prog decl ';'        {verPila($2);}
18           |     prog expr ';'        {verPila($2);}
19           |     prog error ';'       {yyerrok;}
20           ;
21 decl      :     ID ',' decl {
22                                       copiarPila($$,$3);
23                                       if(buscar(tabla, $1)!= NULL)
24                                           printf("%s redeclarada.\n",$1);
25                                       else
26                                           insertar(&tabla, $1, $3);
27                                   }
28           |  ID ':' tipo {
29                                       copiarPila($$,$3);
30                                       if(buscar(tabla, $1)!= NULL)
31                                           printf("%s redeclarada.\n",$1);
32                                       else
33                                           insertar(&tabla, $1, $3);
34                                   }
```

```
35      ;
36 tipo :       INTEGER     { crearPila($$, 'i'); }
37         |    REAL        { crearPila($$, 'r'); }
38         |    CHAR        { crearPila($$, 'c'); }
39         |    BOOLEAN     { crearPila($$, 'b'); }
40         |    POINTER TO tipo {
41                                 copiarPila($$,$3);
42                                 insertarPila($$, 'p');
43                             }
44         |    ARRAY '['  ']' OF tipo {
45                                 copiarPila($$,$5);
46                                 insertarPila($$, 'a');
47                             }
48         |    PROCEDURE '('')'':'tipo {
49                                 copiarPila($$,$5);
50                                 insertarPila($$, 'f');
51                             }
52         ;
53 expr   :    ID                 {
54                                     if(buscar(tabla, $1)==NULL) {
55                                         crearPila($$, 'u');
56                                         printf("%s no declarada.\n",$1);
57                                     } else
58                                         copiarPila($$,buscar(tabla,$1)->tipo);
59                                 }
60         |    NUM                { crearPila($$, 'i'); }
61         |    NUMREAL            { crearPila($$, 'r'); }
62         |    CARACTER           { crearPila($$, 'c'); }
63         |    CTELOGICA          { crearPila($$, 'b'); }
64         |    expr '^'           {
65                                     if(cimaPila($1) != 'p') {
66                                         crearPila($$, 'u');
67                                         printf("Esperaba un puntero.\n");
68                                     } else {
69                                         copiarPila($$, $1);
70                                         eliminarPila($$);
71                                     }
72                                 }
73         |    expr'['NUM']'      {
74                                     if(cimaPila($1) != 'a') {
75                                         crearPila($$, 'u');
76                                         printf("Esperaba un array.\n");
77                                     } else {
78                                         copiarPila($$, $1);
79                                         eliminarPila($$);
80                                     }
81                                 }
82         |    expr'(' ')'        {
83                                     if(cimaPila($1) != 'f') {
```

```
                                        crearPila($$, 'u');
                                        printf("Esperaba una funcion.\n");
                                    } else {
                                        copiarPila($$, $1);
                                        eliminarPila($$);
                                    }
                                }
        ;
%%
#include "TipCompl.c"
void main() {
    yyparse();
    imprimir(tabla);
}
void yyerror(char * s) {
    printf("%s\n",s);
}
```

7.4.4 Solución con JFlex/Cup

La solución con Java pasa por crear un símbolo formado por un nombre de tipo **String** y una pila de caracteres de tipo **Stack**. Además, lo más sensato es introducir en esta misma clase la función que genera la cadena de texto que describe en lenguaje natural la pila de tipos. La clase **Simbolo** queda:

```
class Simbolo{
    String nombre;
    Stack tipo;
    public Simbolo(String nombre, Stack tipo){
        this.nombre = nombre;
        this.tipo = tipo;
    }
    public static String tipoToString(Stack st){
        String retorno = "";
        for(int cont = st.size()-1; cont>=0; cont--)
        switch(((Character)st.elementAt(cont)).charValue()) {
            case('i'): { retorno += "un entero."; break; }
            case('r'): { retorno += "un real."; break; }
            case('b'): { retorno += "un booleano."; break; }
            case('c'): { retorno += "un caracter."; break; }
            case('p'): { retorno += "un puntero a "; break; }
            case('a'): { retorno += "un array de "; break; }
            case('f'): { retorno += "una funcion que devuelve "; break; }
            case('u'): { retorno += "indefinido."; break; }
        };
        return retorno;
    }
}
```

La tabla de símbolos es muy parecida a la de ejercicios anteriores, excepto por

el hecho de que la función **insertar()** también acepta un objeto de tipo **Stack**. Esto se debe a que en el momento de la inserción de un símbolo también se conoce su tipo, dado que es Cup quien realiza las inserciones. Así, la clase **TablaSimbolos** queda:

```
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, Stack st){
8             Simbolo s = new Simbolo(nombre, st);
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 + ": " + Simbolo.tipoToString(s.tipo));
20            }
21       }
22 }
```

El programa en JFlex se limita a reconocer los *tokens* necesarios, de manera análoga a como se hacía en Lex. Lo único interesante viene dado por las líneas 5 y 6 del siguiente código; la declaración de las líneas 10 y 11 hace que en la clase **Yylex** se creen variables que llevan la cuenta del número de línea y de columna donde comienza el *token* actual. Dado que estas variables son privadas es necesario crear las funciones de las líneas para exteriorizarlas. El código de **TipCompl.jflex** es:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  %%
4  %{
5        public int linea(){ return yyline+1; }
6        public int columna(){ return yycolumn+1; }
7  %}
8  %unicode
9  %cup
10 %line
11 %column
12 %%
13 [:digit:]+                  { return new Symbol(sym.NUM); }
14 [:digit:]+\.[:digit:]+ { return new Symbol(sym.NUMREAL); }
15 \'.\'                       { return new Symbol(sym.CARACTER); }
16 "TRUE"  |
17 "FALSE"                    { return new Symbol(sym.CTELOGICA); }
```

```
18 "INTEGER"        { return new Symbol(sym.INTEGER); }
19 "REAL"           { return new Symbol(sym.REAL); }
20 "CHAR"           { return new Symbol(sym.CHAR); }
21 "BOOLEAN"        { return new Symbol(sym.BOOLEAN); }
22 "POINTER"        { return new Symbol(sym.POINTER); }
23 "TO"             { return new Symbol(sym.TO); }
24 "ARRAY"          { return new Symbol(sym.ARRAY); }
25 "OF"             { return new Symbol(sym.OF); }
26 "PROCEDURE"   { return   new Symbol(sym.PROCEDURE); }
27 "^"   { return new Symbol(sym.CIRCUN); }
28 "["   { return new Symbol(sym.LCORCH); }
29 "]"   { return new Symbol(sym.RCORCH); }
30 "("   { return new Symbol(sym.LPAREN); }
31 ")"   { return new Symbol(sym.RPAREN); }
32 ":"   { return new Symbol(sym.DOSPUN); }
33 ";"   { return new Symbol(sym.PUNTOYCOMA); }
34 ","   { return new Symbol(sym.COMA); }
35 [:jletter:][:jletterdigit:]*  { return new Symbol(sym.ID, yytext()); }
36 [ \t\n\r]+        {;}
37 .            { System.out.println("Error léxico: '"+yytext()+"'."); }
```

Por último, el programa en Cup realiza algunas declaraciones importantes en el área de código del analizador sintáctico. En concreto, crea como estática la tabla de símbolos, de manera que sea accesible desde las acciones semánticas mediante la referencia **parser.tabla**. También se crea un analizador léxico **miAnalizadorLexico** de tipo **Yylex** como variable de instancia; esto se hace con el objetivo de poder referenciar las funciones **linea()** y **columna()** del analizador léxico en un mensaje de error personalizado. Los mensajes de error se pueden personalizar reescribiendo la función **syntax_error()** que toma como parámetro el *token* actual que ha provocado el error. En nuestro caso se ha reescrito esta función en las líneas 20 a 23. En éstas se llama, a su vez, a la función **report_error()** que se encarga de sacar un mensaje por la salida de error (**System.err**) y que, en caso necesario, también puede ser reescrita por el desarrollador. De hecho, la salida de error se ha utilizado en todos los mensajes de error semántico que se visualizan: “variable no declarada”, “variable redeclarada”, “esperaba un puntero”, etc.

Por otro lado, y como ya se ha indicado, se ha utilizado un objeto de tipo **Stack** para almacenar la pila de tipos. Nótese cómo, por regla general, se utiliza compartición estructural, esto es, no se generan copias de la pila de tipos. Por ejemplo, durante una declaración, la pila de tipos se construye al encontrar un tipo primitivo y a medida que se encuentran constructores de tipos se van añadiendo elementos a la pila original, sin cear copias de la misma (se usa asignación pura y no la función **clone()**). El único caso en el que es necesario trabajar con una copia de una pila de tipos es cuando se recupera el tipo de un identificador como expresión base sobre la que aplicar modificadores posteriores (véase la línea 82); si no se hiciera una copia **clone()** en este caso, cada vez que se aplicara un modificador de tipo se estaría cambiando el tipo del identificador (y de todos los que se declararon junto a él).

El fichero **TipCompl.cup** queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  import java.util.*;
4  parser code {:
5        static TablaSimbolos tabla = new TablaSimbolos();
6        Yylex miAnalizadorLexico;
7        public static void main(String[] arg){
8              parser parserObj = new parser();
9              parserObj.miAnalizadorLexico =
10                                 new Yylex(new InputStreamReader(System.in));
11             parserObj.setScanner(parserObj.miAnalizadorLexico);
12             try{
13                   parserObj.parse();
14                   tabla.imprimir();
15             }catch(Exception x){
16                   x.printStackTrace();
17                   System.err.println("Error fatal.");
18             }
19       }
20       public void syntax_error(Symbol cur_token){
21             report_error("Error de sintaxis: linea "+miAnalizadorLexico.linea()+
22                                       ", columna "+miAnalizadorLexico.columna(), null);
23       }
24 :};
25 terminal INTEGER, REAL, CHAR, BOOLEAN, POINTER, TO;
26 terminal ARRAY, OF, PROCEDURE;
27 terminal NUM, CARACTER, NUMREAL, CTELOGICA;
28 terminal RPAREN, LPAREN, RCORCH, LCORCH, CIRCUN;
29 terminal DOSPUN, PUNTOYCOMA, COMA;
30 terminal String ID;
31 non terminal Stack tipo, expr, decl;
32 non terminal prog;
33 /* Gramática */
34 start with prog;
35 prog      ::=  /*´Epsilon */
36             |        prog decl:d PUNTOYCOMA
37                                 {: System.out.println(Simbolo.tipoToString(d)); :}
38             |        prog expr:e PUNTOYCOMA
39                                 {: System.out.println(Simbolo.tipoToString(e)); :}
40             |        prog error PUNTOYCOMA
41                                 {: ; :}
42             ;
43 decl      ::=  ID:id COMA decl:t {:
44                                 RESULT = t;
45                                 if(parser.tabla.buscar(id)!= null)
46                                       System.err.println(id + " redeclarada.");
47                                 else
48                                       parser.tabla.insertar(id, t);
```

```
49                                      :}
50          |   ID:id DOSPUN tipo:t {:
51                                          RESULT = t;
52                                          if(parser.tabla.buscar(id)!= null)
53                                              System.err.println(id + " redeclarada.");
54                                          else
55                                              parser.tabla.insertar(id, t);
56                                      :}
57      ;
58  tipo ::=  INTEGER    {: RESULT = new Stack(); RESULT.push(new Character('i')); :}
59      |   REAL        {: RESULT = new Stack(); RESULT.push(new Character('r')); :}
60      |   CHAR        {: RESULT =new Stack(); RESULT.push(new Character('c')); :}
61      |   BOOLEAN     {: RESULT =new Stack(); RESULT.push(new Character('b')); :}
62      |   POINTER TO tipo:t     {:
63                                          RESULT = t;
64                                          RESULT.push(new Character('p'));
65                                      :}
66      |   ARRAY LCORCH RCORCH OF tipo:t {:
67                                          RESULT = t;
68                                          RESULT.push(new Character('a'));
69                                      :}
70      |   PROCEDURE RPAREN LPAREN DOSPUN tipo:t {:
71                                          RESULT = t;
72                                          RESULT.push(new Character('f'));
73                                      :}
74      ;
75  expr    ::=  ID:id      {:
76                              Simbolo s;
77                              if ((s = parser.tabla.buscar(id)) == null) {
78                                  RESULT = new Stack();
79                                  RESULT.push(new Character('u'));
80                                  System.err.println(id + " no declarada.");
81                              } else
82                                  RESULT = (Stack)s.tipo.clone();
83                          :}
84      |   NUM     {: RESULT = new Stack(); RESULT.push(new Character('i')); :}
85      |   NUMREAL    {: RESULT = new Stack(); RESULT.push(new Character('r')); :}
86      |   CARACTER  {: RESULT =new Stack(); RESULT.push(new Character('c')); :}
87      |   CTELOGICA{: RESULT = new Stack(); RESULT.push(new Character('b')); :}
88      |   expr:t CIRCUN     {:
89                          if(((Character)t.peek()).charValue() != 'p') {
90                              RESULT = new Stack(); RESULT.push(new Character('u'));
91                              if(((Character)t.peek()).charValue() != 'u')
92                                  System.err.println("Esperaba un puntero.");
93                          } else {
94                              RESULT = t;
95                              RESULT.pop();
96                          }
97                      :}
```

```
 98      |   expr:t LCORCH NUM RCORCH {:
 99                  if(((Character)t.peek()).charValue() != 'a') {
100                      RESULT = new Stack(); RESULT.push(new Character('u'));
101                      if(((Character)t.peek()).charValue() != 'u')
102                          System.err.println("Esperaba un array.");
103                  } else {
104                      RESULT = t;
105                      RESULT.pop();
106                  }
107              :}
108  |       expr:t LPAREN RPAREN {:
109                  if(((Character)t.peek()).charValue() != 'p') {
110                      RESULT = new Stack(); RESULT.push(new Character('u'));
111                      if(((Character)t.peek()).charValue() != 'u')
112                          System.err.println("Esperaba una función.");
113                  } else {
114                      RESULT = t;
115                      RESULT.pop();
116                  }
117              :}
118      ;
```

7.4.5 Solución con JavaCC

La solución con JavaCC utiliza acciones semánticas muy parecidas a las de la solución con JFlex y Cup. De hecho, las clases **TablaSimbolos** y **Simbolo** son exactamente iguales. Por otro lado, la pila de tipos se construye invertida en la regla de **tipo**, por lo que es necesario darle la vuelta antes de devolverla en las líneas 117 a 121.

Aunque se ha utilizado un objeto de tipo **Stack** para almacenar los caracteres que representan los tipos, también podría haberse usado un *array* de **char**, de la misma manera en que se hizo en la solución con Lex y Yacc. En tal caso, no habría sido necesario controlar los límites del *array* ya que, en caso de que éstos se violaran, se elevaría una excepción que se capturaría en el **try-catch** de las líneas 87 a 92 como si se tratase de cualquier error sintáctico. El fichero **TiposComplejos.jj** queda:.

```
 1  PARSER_BEGIN(TiposComplejos)
 2      import java.util.*;
 3      public class TiposComplejos{
 4          static TablaSimbolos tabla = new TablaSimbolos();
 5          public static void main(String args[]) throws ParseException {
 6          new TiposComplejos(System.in).prog();
 7              tabla.imprimir();
 8          }
 9          private static Stack declaracion(String id, Stack t){
10              if(tabla.buscar(id)!= null)
11                  System.err.println(id + " redeclarada.");
12              else
13                  tabla.insertar(id, t);
14              return t;
```

```
15          }
16          private static Stack invertir(Stack in){
17               Stack out=new Stack();
18               while(!in.empty())
19                    out.push(in.pop());
20               return out;
21          }
22          private static Stack utilizacion(String id){
23               Simbolo s;
24               if ((s = tabla.buscar(id)) == null) {
25                    System.err.println(id + " no declarada.");
26                    return nuevaPila('u');
27               } else
28                    return (Stack)s.tipo.clone();
29          }
30          private static Stack nuevaPila(char c){
31               Stack st = new Stack();
32               st.push(new Character(c));
33               return st;
34          }
35          private static Stack comprobar(Stack st, String mensaje, char c){
36               if(((Character)st.peek()).charValue() != c) {
37                    if(((Character)st.peek()).charValue() != 'u')
38                         System.err.println("Esperaba "+mensaje+".");
39                    return nuevaPila('u');
40               } else {
41                    st.pop();
42                    return st;
43               }
44          }
45     }
46 PARSER_END(TiposComplejos)
47 SKIP : {
48          " "
49     |    "\t"
50     |    "\n"
51     |    "\r"
52 }
53 TOKEN [IGNORE_CASE] :
54 {
55          <POINTER: "POINTER">
56     |    <TO: "TO">
57     |    <ARRAY: "ARRAY">
58     |    <OF: "OF">
59     |    <PROCEDURE: "PROCEDURE">
60     |    <INTEGER: "INTEGER">
61     |    <REAL: "REAL">
62     |    <CHAR: "CHAR">
63     |    <BOOLEAN: "BOOLEAN">
```

```
64        |     <NUM: (["0"-"9"])+>
65        |     <NUMREAL: (["0"-"9"])+"."(["0"-"9"])+>
66        |     <CARACTER: ("\'"~["\'"])>
67        |     <CTELOGICA: ("TRUE"|"FALSE")>
68        |     <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
69        |     <PUNTOYCOMA: ";">
70  }
71  /*
72  prog ::= ( bloque )*
73  */
74  void prog():{}{
75        ( bloque() )*
76  }
77  /*
78  bloque ::= decl ';' | expr ';' | error ';'
79  */
80  void bloque():{
81        Stack st;
82  }{
83        try {
84              LOOKAHEAD(2)
85              st=decl() <PUNTOYCOMA> { System.out.println(Simbolo.tipoToString(st)); }
86        |     st=expr() <PUNTOYCOMA> { System.out.println(Simbolo.tipoToString(st)); }
87        }catch(ParseException x){
88              System.err.println(x.toString());
89              Token t;
90              do {
91                    t = getNextToken();
92              } while (t.kind != PUNTOYCOMA);
93        }
94  }
95  /*
96  decl ::= ID ( ',' ID )* ':' tipo
97  */
98  Stack decl():{
99        String nombre;
100       Stack t;
101 }{
102             LOOKAHEAD(2)
103             nombre=id() ":" t=tipo() { return declaracion(nombre, t); }
104       |     nombre=id() "," t=decl() { return declaracion(nombre, t); }
105 }
106 /*
107 tipo ::= ( POINTER TO | ARRAY '[' ']' OF | PROCEDURE '(' ')' ':' )*
108                                                 ( INTEGER | REAL | CHAR | BOOLEAN )
109 */
110 Stack tipo():{
111       Stack st = new Stack();
112 }{
```

```
113       (     <POINTER> <TO>          { st.push(new Character('p')); }
114       |     <ARRAY> "[" "]" <OF>     { st.push(new Character('a')); }
115       |     <PROCEDURE> "(" ")" ":"  { st.push(new Character('f')); }
116       )* (
117             <INTEGER>               { st.push(new Character('i')); return invertir(st); }
118       |     <REAL>                  { st.push(new Character('r')); return invertir(st); }
119       |     <CHAR>                  { st.push(new Character('c')); return invertir(st); }
120       |     <BOOLEAN>               { st.push(new Character('b')); return invertir(st); }
121       )
122 }
123 /*
124 expr ::= NUM | NUMREAL | CARACTER | CTELOGICA | ID ( '^' | '[' NUM ']' | '(' ')' )*
125 */
126 Stack expr():{
127       String nombre;
128       Stack st;
129 }{
130       (     nombre=id() { st = utilizacion(nombre); }
131                   (
132                         "^"                 { st = comprobar(st, "un puntero", 'p'); }
133                   |     "[" <NUM> "]"       { st = comprobar(st, "un array", 'a'); }
134                   |     "(" ")"             { st = comprobar(st, "una función", 'f'); }
135                   )*          { return st; }
136       |     <NUM>                 { return nuevaPila('e'); }
137       |     <NUMREAL>             { return nuevaPila('r'); }
138       |     <CARACTER>            { return nuevaPila('c'); }
139       |     <CTELOGICA>           { return nuevaPila('b'); }
140       )
141 }
142 String id():{}{
143       <ID>     { return token.image; }
144 }
145 SKIP : {
146       <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
147 }
```