Tema 6
Gestión de Tipos

Un compilador debe comprobar si el programa fuente sigue tanto las convenciones sintácticas como las semánticas del lenguaje fuente. Esta comprobación garantiza la detección y comunicación de algunas clases de errores de programación.

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

Puede necesitarse la información sobre los tipos, reunida por la gestión de tipos, cuando se genera el código. Por ejemplo, los operadores aritméticos como + normalmente se aplican tanto a enteros como a reales, tal vez a otros tipos, y se debe examinar el contexto de + para determinar el sentido que se pretende dar. Se dice que un símbolo que puede representar diferentes operaciones en diferentes contextos está "sobrecargado". 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.

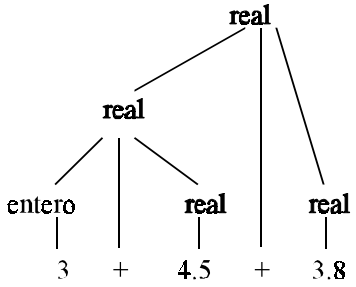
Una noción diferente de la sobrecarga es la de "polimorfismo". Una función es polimórfica cuando puede ejecutarse con argumentos de tipos diferentes, desencadenando en cada caso una acción diferente. Por ejemplo, la suma es un operador polimórfico, pues permite sumar enteros, reales, reales con enteros, y concatenar cadenas o sumar enteros con caracteres. Una cosa es que nuestro lenguaje tenga algunas funciones predefinidas polimórficas, y otra cosa que se permita la definición de funciones polimórficas. Así, hay que solucionar de alguna forma los conflictos en la tabla de símbolos.

Veamos un ejemplo sencillo: Considérese 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 de + para garantizar que ambos operandos sean del mismo tipo cuando tenga lugar la suma. La siguiente gramática 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.

E → E + T | T

T → **num . num** | **num**

Por ejemplo si quisiéramos determinar el tipo de cada subexpresión, obtendríamos el árbol de la figura.
(Este problema está resuelto en la relación de problemas de YACC)



Veamos ahora un ejemplo en el que gestionaremos los tipos Enteros y Caracteres. La gramática a reconocer será la siguiente:

```
prog   : prog asig '\n'
       | prog IMPRIMIR expr '\n'
       |
       ;
asig   : ID ASIG expr
       | ID ASIG asig
       ;

expr   : expr '+' expr
       | expr '*' expr
       | A_ENTERO '('expr ')'
       | A_CARACTER '(' expr ')'
       | ID
       | NUMERO
       | CARACTER
       ;
```

En este ejemplo podremos

* Asignar expresiones a variables; también podremos hacer asignaciones múltiples.

* IMPRIMIR expr.

  Esta expr puede ser de dos tipos: bien entera o bien carácter. Sin embargo la función imprimir es la misma. Si vemos la función imprimir como funcional, podemos decir que IMPRIMIR es de tipo POLIMÓRFICA.

Con los enteros vamos a poder hacer '+' y '*', en cambio con los caracteres no podemos hacer ninguna de estas operaciones.

* Además vamos a permitir también funciones de conversión

$$\left.\begin{matrix}\text{Entero}\\ \text{Caracter}\end{matrix}\right\} \Rightarrow \left\{\begin{matrix}\text{A\_CARACTER}\\ \text{A \_ENTERO}\end{matrix}\right.$$

Podemos decir que estas funciones son funciones predefinidas también llamadas funciones *built-in*.

A_CARACTER(expr) : Tiene como parámetro una expr de tipo entero y nos devuelve como resultado el código ASCII que equivale a dicho entero.

A_ENTERO(expr) : Tiene como parámetro una expr de tipo carácter y nos devuelve como resultado el código ASCII de dicho carácter.

Dado un enunciado tenemos que :

1. Ponernos un ejemplo del tipo de entrada con el que nos vamos a encontrar

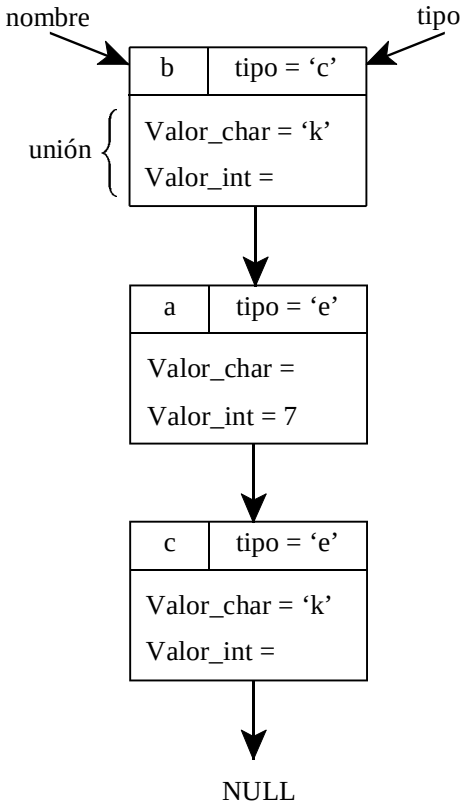
```
b = 'k'
a =7
c =12 + 3
IMPRIMIR c + 5
IMPRIMIR b
```

2. Definir la tabla de símbolos, si es que esta es necesaria para el problema.

En este caso sí necesitaremos la tabla de símbolo, en ella almacenaremos los identificadores. Además, como lo que estamos haciendo es un intérprete, no me basta con guardar el tipo del identificador que lo denotaremos de la siguiente forma:

Tipo ⇕
- 'c' : si es de tipo carácter
- 'e' : si es de tipo entero

sino que tendremos que almacenar el valor de dicho identificador, pero aquí nos encontramos con un problema, unas veces este valor será de tipo entero, y otras veces será de tipo carácter, con lo que deberemos de utilizar una estructura de tipo unión. Veamos ésto gráficamente:



3. El tercer paso es estudiar qué atributo se le tiene que asociar a los no terminales.

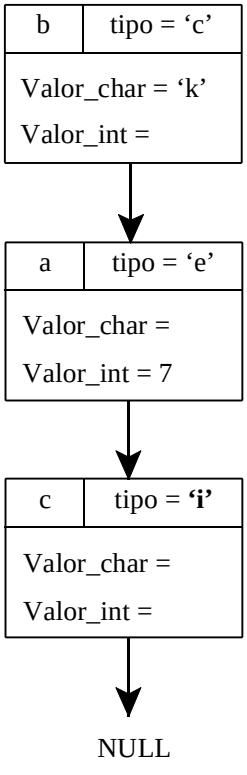
En mi programa no hay declaración de variable, entonces es LEX quien debe insertar en la tabla de símbolos, con objeto de descargar a YACC de más tareas. Si hubiera una zona de declaración en el programa entonces insertaría YACC en la tabla de símbolo.

Hasta ahora todo va bien, pero analicemos que ocurriría si tuviésemos la siguiente entrada

```
b = 'k'
a = 7
c = 8 + a
IMPRIMIR c + 5
IMPRIMIR b
```

Cuando LEX encuentra *c = 8 + a* lo inserta en la tabla de símbolos, pero ¿de qué tipo es la variable *c*?. El tipo de c no se sabe hasta su primera asignación. El tipo está implícito en el momento en que se hace su primera asignación.

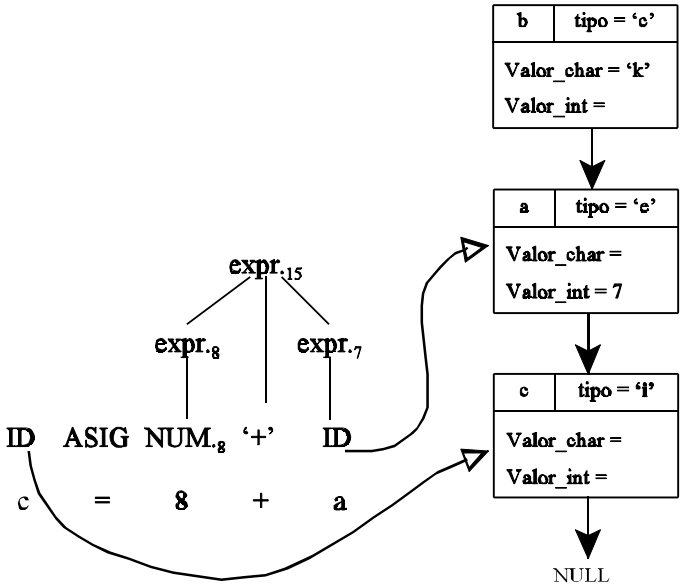
¿Qué tipo le asigna LEX a la variable *c*? No sabe que tipo asignarle, será YACC el que le ponga el tipo cuando se haga la primera asignación. LEX le pondrá el tipo = 'i' (indefinido)



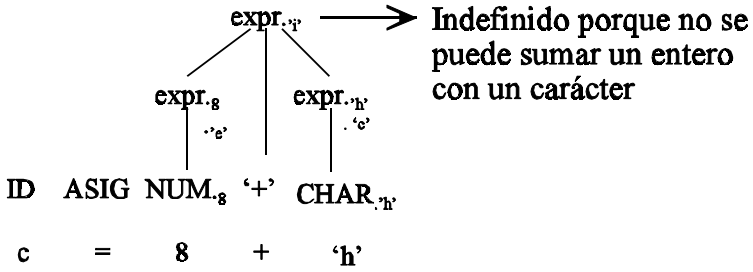
Gestión de Tipos

En cuanto a los atributos asociados a los no terminales:

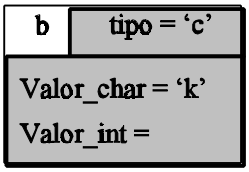
- **CHAR** : tiene asociado un carácter.
- **NUM** tiene asociado un entero.
- **ID** el atributo que devuelve es un puntero a la tabla de símbolo



¿Qué ocurriría en este caso? c = 8 + 'h'. Es decir que ocurriría en el caso en que quisiéramos operar un entero con un carácter.



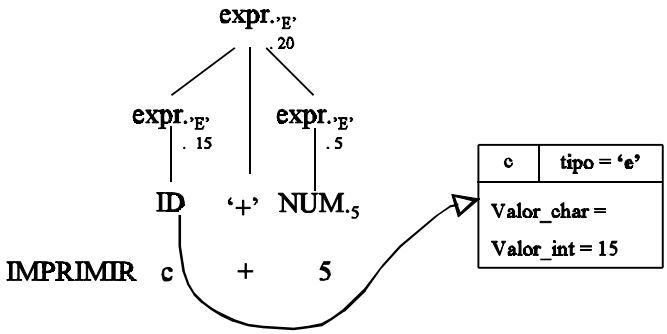
Como vemos en **expr** no solo vamos a guardar el valor, sino que también vamos a guardar el tipo. A una expresión tenemos que asociarle también dos atributos que realmente son lo mismo que la parte sombreada:



En la función **A_CARACTER(expr)** preguntaremos el tipo y:
- Si no es entero nos dará un error
- Si es entero devuelve un carácter

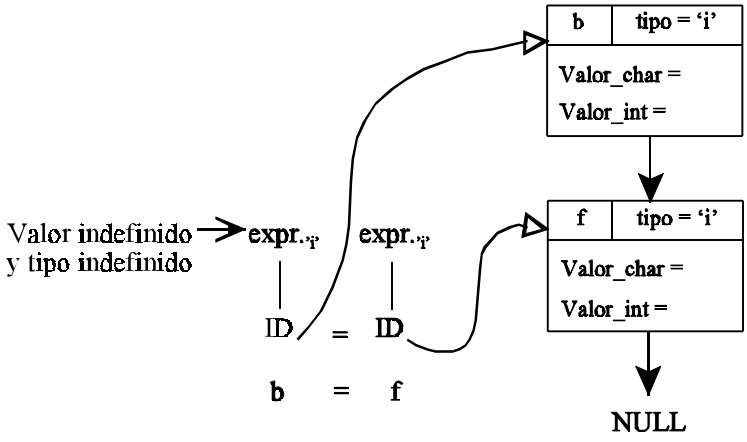
En la función **A_ENTERO(expr)** preguntaremos el tipo y:
- Si no es carácter nos dará un error
- Si es carácter devuelve un entero

Si nos encontramos : IMPRIMIR c + 5



Veamos que ocurre si tenemos un identificador de usuario asignado a otro identificador de usuario

b = f ⇨ LEX mete en la tabla de símbolo *b* con tipo 'i', se encuentra *f* y también la mete con tipo 'i', (como se dijo antes, el tipo de un identificador no se sabe hasta la primera asignación).

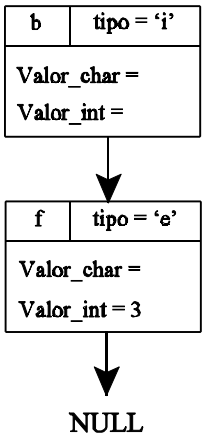


El hecho de que la parte derecha de la asignación tenga tipo indefinido es un error. ¿Qué ocurre con *b*? En *b* no puedo guardar nada: *b* permanece en la tabla de símbolos pero tiene valor y tipo indefinido.

Suponer que la siguiente instrucción es f = 3.

b = f

f = 3

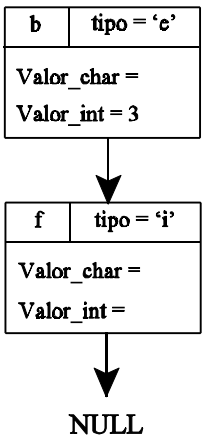


Veamos que ocurriría si a un identificador que ya tiene asignado un tipo se le asignara un valor de otro tipo diferente. Estudiemos la siguiente entrada

b = 3

b = f ⇨ Intento asignarle a *b* un valor desconocido. ¿Que hacemos? Emitimos un error y dejamos *b*, con el tipo que tenía.

b = 'k' ⇨ Error, porque *b* tiene tipo entero, e intento ahora asignarle un tipo carácter.



Como consecuencia, cuando hay un error no se le asigna el tipo 'i' porque sino permitiríamos cosas como

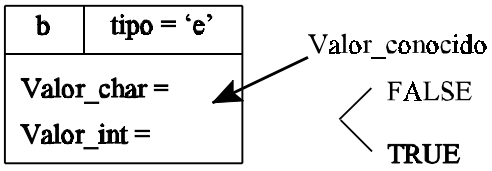
b = 3 ➪ Funciona
b = f ➪ Falla
b = 'k' ➪ ¡Funciona! (Pero no debe)

Un tipo indefinido implica un valor desconocido.

Para solucionar una situación como

b = 3
b = f ⇦ Valor desconocido
b = ‘k’

Una vez asignado el tipo siempre es el mismo, si durante la ejecución ocurre un error, el tipo de la variable seguirá siendo el mismo, pero podemos añadir una variable booleana a la tabla de símbolos que me diga si el valor de la variable es conocido o no.



A continuación se proporciona la solución completa de este problema

NOTA: Siempre que tengamos que estudiar un programa LEX/YACC ya hecho, debemos de estudiarlo de las últimas reglas hacia las primeras reglas, ya que las reglas que se reducen primero son las últimas

Fichero: Tabsimb3.c

```
#include <stdlib.h>
#include <stdio.h>
typedef struct nulo
	{
		struct nulo * sig;
		char nombre [20];
		char tipo;
		union
		  {
			char valor_char;
			int valor_int;
		  }info;
	} simbolo;

simbolo * crear()
{
	return NULL;
};

void insertar(p_t,s)
simbolo **p_t;
simbolo * s;
{
	s->sig = (*p_t);
	(*p_t) = s;
};

simbolo * buscar(t,nombre)
simbolo * t;
char nombre[20];
{
	while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
		t = t->sig;
	return (t);
};
void imprimir(t)
simbolo * t;
{
	while (t != NULL)
		{
			printf("%s\n", t->nombre);
			t = t->sig;
		}
};
```

Código LEX : Ejem3l.lex

```
%%
[0..9]                  {
                                yylval.numero=atoi(yytext);
                                return NUMERO;
                        }
""."""                  {
                                yylval.caracter = yytext[1];
                                return CARACTER;
                        }
"IMPRIMIR"              {       return IMPRIMIR;}
"A_ENTERO"              {       return A_ENTERO;}
"A_CARACTER"            {       return A_CARACTER;}
":="                    {       return ASIG;}
[a-zA-Z][a-zA-Z0-9]*{
                           yylval.ptr_simbolo = buscar(t,yytext);
                           if(yylval.ptr_simbolo == NULL)
                           {
                              yylval.ptr_simbolo = (simbolo *)malloc(sizeof(simbolo));
                              strcpy(yylval.ptr_simbolo->nombre,yytext);
                              yylval.ptr_simbolo->tipo='i';
                              insertar(&t,yylval.ptr_simbolo);
                            }
                           return ID;
                         }
[ \t]+                  {;}
.|\n                    { return yytext[0];}
```

Código YACC : Ejem3y.yac

```
%{
#include "tabsimb3.c"
typedef struct
	{
	char tipo;
	union
		{
			char valor_char;
			int valor_int;
		}info;
	}expresion;

simbolo * t;

%}

%union{
	char caracter;
	int numero;
	simbolo * ptr_simbolo;
	expresion valor;
}

%token <caracter> CARACTER
%token <numero> NUMERO
%token <ptr_simbolo> ID
%token IMPRIMIR ASIG A_ENTERO A_CARACTER
%type <valor> expr asig
%start prog

%right '+'
%right '*'
%%
prog	: prog asig '\n'
	| prog IMPRIMIR expr '\n'{
				if ($3.tipo == 'e')
				  printf("%d\n",$3.info.valor_int);
				else if ($3.tipo == 'c')
				  printf("%c\n",$3.info.valor_char);
				else
				  printf("Indefinido.\n");
				}
	|
	;
```

Código YACC : Ejem3y.yac

```
asig    : ID ASIG expr {
                        $$.tipo = 'i';
                        if($1->tipo == 'i')
                           $1->tipo = $3.tipo;
                        if ($3.tipo == 'i')
                           printf("Asignacion de tipo indefinido.\n");
                        else
                           if ($3.tipo != $1->tipo)
                              printf("Asignacion de tipos incompatibles.\n");
                           else
                            {  $$.tipo = $1->tipo;
                              if($1->tipo =='e')
                               {  $1->info.valor_int = $3.info.valor_int;
                                  $$.info.valor_int = $3.info.valor_int;
                                }
                              else
                              {  $1->info.valor_char = $3.info.valor_char;
                                 $$.info.valor_char = $3.info.valor_char;
                              }
                           }
                  }
        | ID ASIG asig   {
                        $$.tipo = 'i';
                        if($1->tipo == 'i')
                          $1->tipo = $3.tipo;
                        if ($3.tipo == 'i')
                          printf("Asignacion de tipo indefinido.\n");
                        else
                          if ($3.tipo!= $1->tipo)
                                printf("Asignacion de tipos incompatibles.\n");
                          else
                          {
                                $$.tipo = $1->tipo;
                                if($1->tipo =='e')
                                { $1->info.valor_int = $3.info.valor_int;
                                  $$.info.valor_int = $3.info.valor_int;
                                }
                                else
                                {
                                  $1->info.valor_char = $3.info.valor_char;
                                  $$.info.valor_char = $3.info.valor_char;
                                }
                            };
              }
        ;
```

Código YACC : Ejem3y.yac

```
expr    : expr '+' expr  {
                        if(($1.tipo != 'e') || ($3.tipo != 'e'))
                          {
                                $$.tipo = 'i';
                                printf("Error en expresion.\n");
                           }
                        else
                          {
                                 $$.tipo = 'e';
                                 $1.info.valor_int = $1.info.valor_int + $3.info.valor_int;
                          };
                }

        | expr '*' expr  {
                        if(($1.tipo != 'e') || ($3.tipo != 'e'))
                          {
                                $$.tipo = 'i';
                                printf("Error en expresion.\n");
                          }
                        else
                          {
                                 $$.tipo = 'e';
                                 $1.info.valor_int = $1.info.valor_int * $3.info.valor_int;
                          };
                }

        | A_ENTERO '(' expr ')'{
                                if ($3.tipo != 'c')
                                  {
                                        printf("Error de conversión de tipos. \n");
                                        $$.tipo = 'i';
                                  }
                                else
                                  {
                                        $$.info.valor_int= (int)$3.info.valor_char;
                                        $$.tipo = 'e';
                                  };
                        }
```

Código YACC : Ejem3y.yac

```
        | A_CARACTER '(' expr ')'    {
                                       if ($3.tipo != 'e')
                                         {
                                                printf("Error de conversión de tipos. \n");
                                                $$.tipo = 'i';
                                         }
                                       else
                                         {
                                                $$.info.valor_char= (char)$3.info.valor_int;
                                                $$.tipo = 'c';
                                         };
                                       }
        | ID            {
                        $$.tipo = $1->tipo;
                        if ($$.tipo == 'i')
                                printf("Tipo de %s no definido.\n",$1->nombre);
                        else
                        {
                                        if ($$.tipo == 'e')
                                                $$.info.valor_int = $1->info.valor_int;
                                        else
                                                $$.info.valor_char = $1->info.valor_char;
                        };
                        }
        | NUMERO    {
                                $$.tipo = 'e';
                                $$.info.valor_int = $1;
                        }
        | CARACTER          {
                                $$.tipo = 'c';
                                $$.info.valor_char = $1;
                        }
        ;
%%
#include "ejem3l.c"
void main()
{
        t = crear();
        yyparse();
        imprimir(t);
};

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

Sistemas de Tipos

Hasta ahora hemos visto los tipos primitivos (carácter, entero, real). Pero los tipos más interesantes son los tipos estructurados o tipos complejos (array, registros, punteros,...). Para estudiar estos tipos necesitamos estudiar los constructores de tipos.

Constructores de tipos son palabas que extienden el tipo. Por ejemplo

POINTER TO
RECORD OF
ARRAY
PROCEDURE

estos no son tipos, sino constructores de tipos (Se utilizan en la zona de declaraciones). Además estos constructores de tipos son recursivos. El problema que nos encontramos es que el tipo de una variable puede ser todo lo extenso que queramos, por lo que no podremos definir el tipo con una letra (como hicimos en el ejemplo anterior).

Por otro lado tenemos los *modificadores de tipos* (éstos se utilizan en la parte de sentencias). Por regla general los constructores de tipos tienen sus propios modificadores de tipos. Por ejemplo

Constructores de tipos	Modificadores de tipos
POINTER TO _____	^
RECORD OF _____	. __
ARRAY _____	**[ ]**
PROCEDURE _____	**( )**

(No veremos los registros porque el registro actúa sobre el producto cartesiano de todos los tipos posibles.)

¿Qué vamos a permitir en nuestro lenguaje?

- Los constructores de tipos

  - POINTER TO ___
  - ARRAY ___
  - PROCEDURE ___

- Además de los tipos básicos

  - BOOLEAN
  - INTEGER
  - REAL
  - CHAR

Tenemos cuatro tipos primitivos y tres constructores de tipos.

Además de declaraciones de variables de estos tipos, vamos a permitir que el usuario utilice expresiones de la forma

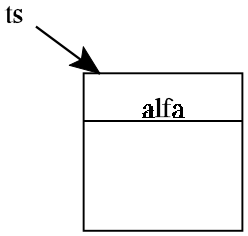
a: POINTER TO BOOLEAN
a^
>Es un boolean
a
> Es un puntero a un boolean

Tenemos el problema de como identificar el tipo de una variable compleja (ya que hemos visto que no es posible hacerlo mediante una letra).

Supongamos que el usuario define la siguiente variable:

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

Vamos a tener una tabla de símbolos donde almacenaremos el nombre de la variable y el tipo.



El tipo tenemos que almacenarlo de forma inteligente ya que luego nos va a servir para detectar errores, y además me permitirá saber qué tipo de modificadores de datos se pueden aplicar:

alfa ^	➪	Correcto
alfa [2]	➪	¡Incorrecto!
alfa ( )	➪	¡Incorrecto!

Para simplificar el problema trabajaremos con procedimientos que no tienen parámetros y siempre devuelven un valor

Cada tipo lo vamos a codificar con una letra

POINTER TO → p
ARRAY → a
PROCEDURE → f

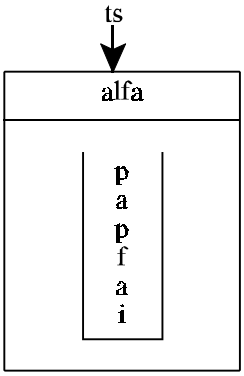
BOOLEAN → b
INTEGER → i
REAL → r
CHAR → c

El problema lo solucionaremos con una pila. De forma que

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

↓	↓	↓	↓	↓	↓
p	a	p	f	a	i

quedaría almacenado de la siguiente forma



La pila la hemos construido mirando el tipo de derecha a izquierda. Para asegurar que en la cima de la pila tengamos el tipo principal de *alfa*. Es decir, en nuestro caso utilizamos una gramática recursiva por la derecha.

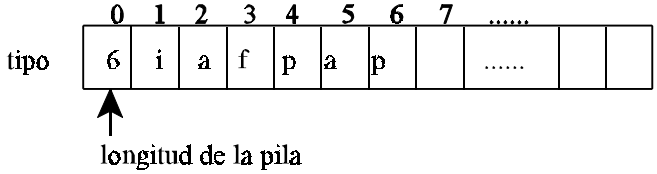
Ahora es muy fácil saber qué modificador de tipo puedo usar

^ si la cima de la pila es “p”
[ ] si la cima de la pila es “a”
( ) si la cima de la pila es “f”

Si alfa es un POINTER TO X, el tipo de alfa^ es X, o lo que es lo mismo, basta con eliminar el tope o cima de la pila para obtener el tipo de X.

¿Cómo almacenaremos la pila en nuestro programa LEX/YACC?

El tope de la pila lo guardaremos en el elemento 0 de un array que nos hará de pila. En el caso anterior, la pila la guardamos en un array de caracteres que será (char tipo [20])



- Si queremos acceder a la cima de la pila:

  1.- Vamos a tipo [0] y miramos la longitud de la pila (6).
  2.- Vamos a tipo [6]

  es decir
  int a = tipo [0];
  cima = tipo [a];

  o lo que es lo mismo cima = tipo[tipo[0]];

- Cuando queramos desapilar un elemento de la pila

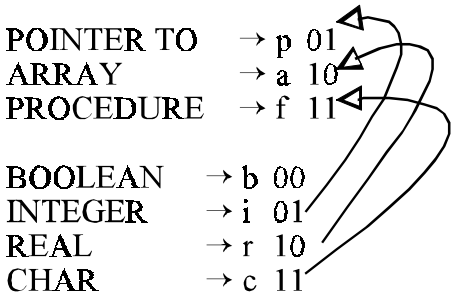
  tipo[0]-- ;

Estamos desaprovechando memoria al almacenar los tipos, pero ello no nos preocupa dadas las cantidades actuales de memoria que se manejan. En caso de que la memoria fuese un problema, podríamos usar algún tipo de codificación.

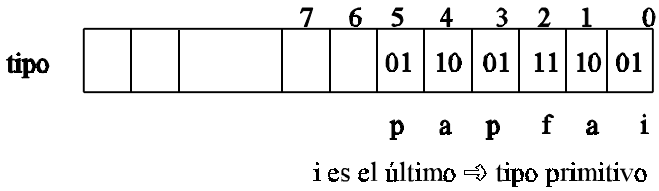
POINTER TO		→ p	100
ARRAY		→ a	101
PROCEDURE		→ f	110
BOOLEAN	→ b	000	
INTEGER	→ i	001	
REAL	→ r	010	
CHAR		→ c	011

3 bits serían suficientes.

También podemos codificar cada constructor de tipo con un par de bits, y no un carácter, con lo cual ahorramos espacio



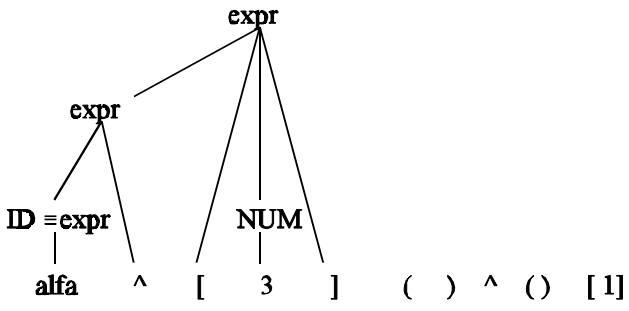
No puede existir confusión, ya que lo que está al final de todo el tipo solo puede ser un tipo primitivo



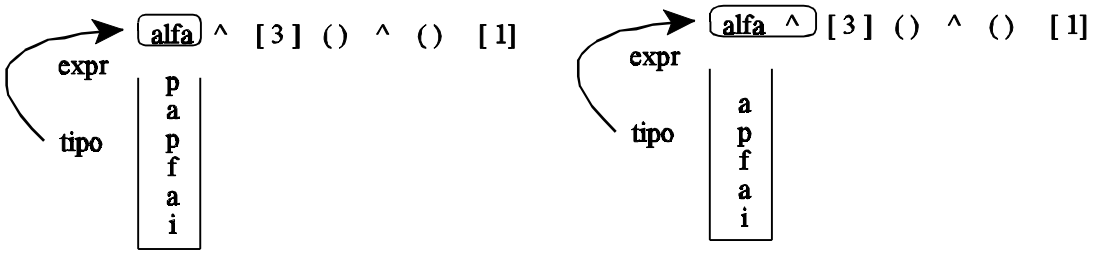
Si encontramos una expr con un modificador de tipo, hay que controlar que el tipo principal de tal expresión, coincide con el modificador de tipo. Por ejemplo, si hacemos alfa^, alfa[num] ó alfa( ) hay que controlar que el tipo principal de alfa sea *p*, *a* ó *f* respectivamente.

En cualquier caso, necesitamos una gramática para poder manejar toda esas expresiones.

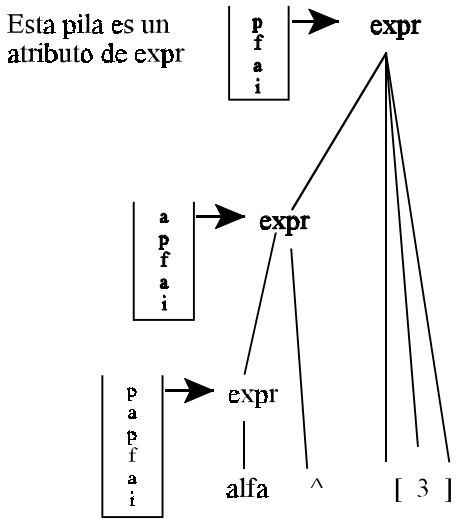
```
expr    : ID
        | expr '^'
        | expr '(' ')'
        | expr '[' NUM ']'
```



La variable *alfa* podemos verla como un *ID* o bien como una *expr*, sea como sea, el tipo de *alfa* es el mismo. Como *ID* tiene un atributo que es un puntero a la tabla de símbolos, donde se guarda el tipo, mientras que como *expr* tiene como atributo una pila.



Ejemplo: vamos a construir el árbol sintáctico de la expresión *alfa ^ [3]*



Los constructores meten en la pila, y los modificadores sacan de la pila. Es decir, a medida que me voy encontrando modificadores de tipos vamos destruyendo, y conforme me encuentro constructores de tipos vamos construyendo la pila.

Si me encuentro con un modificador que no es el que se esperaba, entonces emitimos un error y en la pila pondremos *u* (undefined). (No podemos usar la *i* de “indefinido” porque se confunde con la *i* de “integer”)

Gramática a reconocer. Sistemas de tipos

```
%{
/*  El caracter 0 indica el número de posiciones ocupadas en
        el resto de la cadena */
typedef char cadena_tipo[21];
%}
%union
                {
                cadena_tipo tipo;
                char nombre[20];
                }
%token INTEGER REAL CHAR BOOLEAN POINTER TO
%token ARRAY OF PROCEDURE
%token NUM CARACTER NUMREAL CTELOGICA
%token <nombre> ID
%type <tipo> tipo expr decl
%start prog

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

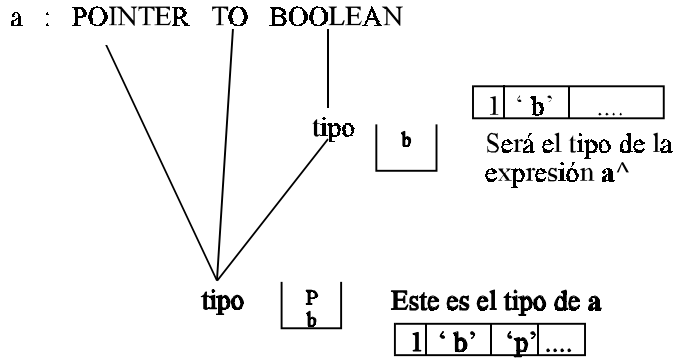
Sistemas de Tipos

En el lenguaje que estamos definiendo, existe zona de declaraciones y zona de utilizaciones de variables, por lo tanto es el analizador sintáctico (YACC) quien inserta en la tabla de símbolos.

- En la regla *prog* tenemos una regla ε que nos permite entrar alguna vez por *prog*, como paso base de la recursión de las reglas de *prog* sobre sí misma.

- En la regla *decl*, hacemos la gramática recursiva a la derecha
  a,b,c : POINTER TO INTEGER
  Los constructores de tipos son recursivos a la derecha, para crear la pila de derecha a izquierda.

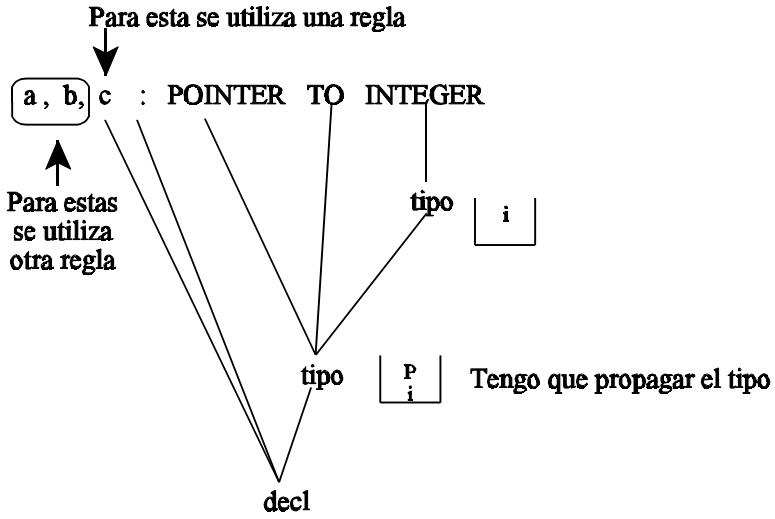
- *tipo* : es recursivo a la derecha, para, cuando se encuentra un tipo primitivo. Cuando hacemos el tipo en la declaración, una vez metido en la tabla de símbolos el tipo ya no se modifica, los que se modifican son las copias de los tipos.



Una vez construida la pila de tipos, tenemos que asociársela a los diferentes identificadores en la declaración.

decl	: ID, decl	→	esta *decl* también tiene un tipo, con objeto de ir propagándolo a todos los ID's.
	| ID ':' tipo	→	Si está en la tabla de símbolos se está intentando redeclarar ⇨ Error

¿Qué ocurre si me encuentro una redeclaración?



- expr : puede ser de 4 tipos : ‘i’, ‘r’, ‘c’, ‘b’
  Si es un identificador entonces lo busco en la tabla de símbolos, si no está quiere decir que estoy utilizando una variable que no he definido. Por lo tanto meto en la pila ‘u’ (indefinido)

  Una expresión también puede ser una expr (MT) con el modificador de tipo correspondiente.

  expr ‘ ^ ‘ Si la cima de la pila es ‘p’, copia el tipo de la pila menos la ‘p’. Si la cima no es la ‘p’ entonces emite un error, y en la pila se mete el tipo ‘u’

Fichero : TabSimb4

```
#include <stdio.h>
typedef struct nulo
	{
		struct nulo * sig;
		char nombre[20];
		cadena_tipo tipo;
	} simbolo;


simbolo * crear()
{
	return NULL;
};

void insertar(p_t,s)
simbolo **p_t;
simbolo * s;
{
	s->sig = (*p_t);
	(*p_t) = s;
};

simbolo * buscar(t,nombre)
simbolo * t;
char nombre[20];
{
	while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
		t = t->sig;
	return (t);
};

void imprimir(t)
simbolo * t;
{
	while (t != NULL)
		{
			printf("%s\n", t->nombre);
			ver_tipo(t->tipo);
			t = t->sig;
		}
}
```

Código LEX : Ejem4l.lex

```
%%
[0-9]+  {return NUM;}
[0-9]+.[0-9]+  {return NUMREAL;}
"'".'"     {return CARACTER;}
"VERDAD"    |
"FALSO"       {return CTELOGICA;}
"INTEGER"   {return INTEGER;}
"REAL"        {return REAL;}
"CHAR"        {return CHAR;}
"BOOLEAN"  {return BOOLEAN;}
"POINTER"   {return POINTER;}
"TO"   {return TO;}
"ARRAY"       {return ARRAY;}
"OF"   {return OF;}
"PROCEDURE"        {return          PROCEDURE;}
[a-zA-Z_][a-zA-Z0-9_]+       {
                                    strcpy(yylval.nombre, yytext);
                                    return ID;
                                }
[ \t]+   {;}
.|\n       {return yytext[0];}
```

Código YACC : Ejem4y.yac

```
%{
#include <stdio.h>
#include <stdlib.h>
/*  El caracter 0 indica el número de posiciones ocupadas en
          el resto de la cadena */
typedef char cadena_tipo[21];

void ver_tipo(tipo)
cadena_tipo tipo;
{
unsigned char cont;
  printf("El tipo es ");
  for(cont = tipo[0]; cont>0; cont--)
        switch(tipo[cont])
        {
        case('i'):      {
                        printf("un entero.");
                        break;
                        }
        case('r'):      {
                        printf("un real.");
                        break;
                        }
        case('b'):      {
                        printf("un booleano.");
                        break;
                        }
        case('c'):      {
                        printf("un caracter.");
                        break;
                        }
        case('p'):      {
                        printf("un puntero a ");
                        break;
                        }
        case('a'):      {
                        printf("un array de ");
                        break;
                        }
        case('f'):      {
                        printf("una funcion que devuelve ");
                        break;
                        }
```

Código YACC : Ejem4y.yac

```
        case('u'):      {
                        printf("indefinido.");
                        break;
                        }
        };
        printf("\n");
}
#include "tabsimb4.c"
simbolo *t;

%}
%union
        {
        cadena_tipo tipo;
        char nombre[20];
        }

%token INTEGER REAL CHAR BOOLEAN POINTER TO
%token ARRAY OF PROCEDURE
%token NUM CARACTER NUMREAL CTELOGICA
%token <nombre> ID
%type <tipo> tipo expr decl
%start prog

%%
prog    :       prog decl '\n'    {ver_tipo($2);}
        |       prog expr '\n' {ver_tipo($2);}
        |       prog error '\n' {yyerrok;}
        |
        ;
decl    :       ID ',' decl {
                        simbolo * aux;
                        strcpy($$,$3);
                        if(buscar(t,$1)!= NULL)
                                printf("%s ya existe.\n",$1);
                        else
                         {
                                aux = (simbolo *)malloc(sizeof(simbolo));
                                strcpy(aux->nombre,$1);
                                strcpy(aux->tipo,$3);
                                insertar(&t,aux);
                         };
                }
```

Código YACC : Ejem4y.yac

```
        |       ID ':' tipo {
                        simbolo * aux;
                        strcpy($$,$3);
                        if(buscar(t,$1)!= NULL)
                                printf("%s ya existe.\n",$1);
                        else
                         {
                                aux = (simbolo *)malloc(sizeof(simbolo));
                                strcpy(aux->nombre,$1);
                                strcpy(aux->tipo,$3);
                                insertar(&t,aux);
                         };
                        }
        ;
tipo    :       INTEGER     {
                                $$[0]=1;
                                $$[1]='i';
                            }
        |       REAL        {
                                $$[0]=1;
                                $$[1]='r';
                            }
        |       CHAR        {
                                $$[0]=1;
                                $$[1]='c';
                            }
        |       BOOLEAN     {
                                $$[0]=1;
                                $$[1]='b';
                            }

        |       POINTER TO tipo             {
                                                strcpy($$,$3);
                                                $$[0]++;
                                                $$[$$[0]]='p';
                                            }

        |       ARRAY '[' NUM ']' OF tipo   {
                                                strcpy($$,$6);
                                                $$[0]++;
                                                $$[$$[0]]='a';
                                            }
```

Código YACC : Ejem4y.yac

```
        |       PROCEDURE '(")":'tipo       {
                                                strcpy($$,$5);
                                                $$[0]++;
                                                $$[$$[0]]='f';
                                                }
        ;
expr    :       ID                  {  if(buscar(t,$1)==NULL)
                                        {
                                                $$[0] = 1;
                                                $$[1] ='u';
                                                printf("%s no encontrada.\n",$1);
                                        }
                                          else
                                                strcpy($$,buscar(t,$1)->tipo);
                                        }
        |       NUM                 {
                                                $$[0] = 1;
                                                $$[1] ='i';
                                        }

        |       NUMREAL             {
                                                $$[0] = 1;
                                                $$[1] ='r';
                                        }
        |       CARACTER            {
                                                $$[0] = 1;
                                                $$[1] ='c';
                                        }
        |       CTELOGICA           {
                                                $$[0] = 1;
                                                $$[1] ='b';
                                        }
        |       expr '^'            {
                                                if($1[$1[0]] != 'p')
                                                 {
                                                        $$[0] = 1;
                                                        $$[1] ='u';
                                                        printf("Esperaba un puntero.\n");
                                                 }
                                                else
                                                 {
                                                        strcpy($$,$1);
                                                        $$[0]--;
                                                 }
                                        }
```

Código YACC : Ejem4y.yac

```
        |       expr['NUM']'  {
                                        if($1[$1[0]] != 'a')
                                         {
                                                $$[0] = 1;
                                                $$[1] ='u';
                                                printf("Esperaba una tabla.\n");
                                         }
                                        else
                                         {
                                                strcpy($$,$1);
                                                $$[0]--;
                                         }
                                }
        |       expr'(' ')'             {
                                        if($1[$1[0]] != 'f')
                                         {
                                                $$[0] = 1;
                                                $$[1] ='u';
                                                printf("Esperaba una funcion.\n");
                                         }
                                        else
                                         {
                                                strcpy($$,$1);
                                                $$[0]--;
                                         }
                                }
        ;

%%
#include "ejem4l.c"

void main()
{
        t=crear();
        yyparse();
        imprimir(t);
}

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