

7. 1.2 puntos Dada la siguiente gramática:

```

programa → subprogramas main
          | main
subprogramas → subprogramas subprograma
              | subprograma
subprograma → "function" ident bloque
main       → "program" bloque
bloque     → "begin" sentencias "end"
sentencias → expresiones expresion ";"
          | expresion ";"
expresion  → ident "=" expresion
          | expresion ">=" ident
          | expresion "[" ident "]"
          | expresion "+" ident
          | "*" expresion
          | "&" expresion
          | ident
          | const
ident      → ident letra
          | letra
letra      → "a" | "b" | ... | "z"
const      → num "." num
num        → "0" | "1" | ... | "9"

```

Obtener la tabla de tokens con el máximo nivel de abstracción suponiendo que se va a realizar:

- (a) 0.6 puntos Traducción.

Solución:

La tabla de tokens con el máximo nivel de abstracción cuando se realiza el proceso completo de traducción sería la siguiente:

Nº	Token	Patrón
1	FUNCTION	"function"
2	PROGRAM	"program"
3	BEGIN	"begin"
4	END	"end"
5	PCOMA	","
6	ASIG	"="
7	OP_ML_MAS	">="
8	OP_AST_AMP	"*"
9	CORIZQ	"["
10	CORDER	"]"
11	IDENT	"[a-z]+"
12	CONST	"[0-9]+"

- (b) 0.6 puntos Análisis sintáctico.

Solución:

La tabla de tokens con el máximo nivel de abstracción cuando únicamente se va a realizar análisis sintáctico puede abstraerse hasta un único token dado que la gramática obtenida es regular. De esta forma, empleando un analizador de léxico es posible realizar la fase de análisis sintáctico.

8. 1.0 puntos Dada la gramática con las producciones siguientes:

- 1) $A \rightarrow F H B$
- 2) $\quad \quad \quad | \quad H d H$
- 3) $\quad \quad \quad | \quad a F$
- 4) $\quad \quad \quad | \quad \epsilon$
- 5) $B \rightarrow d$
- 6) $\quad \quad \quad | \quad \epsilon$
- 7) $H \rightarrow a$
- 8) $\quad \quad \quad | \quad \epsilon$
- 9) $F \rightarrow c$
- 10) $\quad \quad \quad | \quad \epsilon$

Realizar los siguientes cálculos:

- (a) 0.5 puntos Rellenar el número o los números de las producciones que deben figurar en las celdas formadas por las parejas de símbolo no terminal y terminal siguientes: (B,c), (B,\$) y (H,c) de la tabla de análisis LL(1).

	c	d	a	\$
A				
B				
H				
F				

Solución:

La tabla de análisis con las celdas completadas con sus correspondientes producciones quedaría así:

	c	d	a	\$
A				
B	vacío			$B \rightarrow \epsilon$
H	vacío			
F				

- (b) 0.5 puntos Determinar las acciones del estado 0 de la tabla de análisis LR(1).

Estado	c	d	a	\$	A	B	H	F
0								

Solución:

Estado	c	d	a	\$	A	B	H	F
0	d5	r10/r8	d4/r10	r4/r10	1		3	2

9. 1.2 puntos Dada la siguiente especificación YACC:

```
%token PROGRAM          // "program"
%token PI PD             // "(", ")"
%token BEGIN END         // "begin", "end"
%token IDENT ASIGN       // [a-z]+, "="
%token PYC               // ";"
%token FOR TO DO         // "for", "to", "do"
%token WHILE COMA        // "while", ",", " "
%token TIPO              // "int", "char", "float", "boolean"
%token CONST             // [0-9]+, '[a-z0-9]', [0-9]+.[0-9]+, "true"|"false"

%left OP_MAS_MENOS       // "+", "-", Suma y resta de valores REALES.
                        // "++", "--", Suma y resta de valores ENTEROS.
%left OP_MUL_DIV         // "*", "/", Multiplicación y División de valores REALES.
                        // "**", "/", Multiplicación y División de valores ENTEROS.
%nonassoc OP_REL         // "<", "<=", "==", "!=", ">=", ">"
%right OP_NOT            // "!"

%%
programa : PROGRAM bloque ;
bloque : BEGIN decl_var sentencias END ;
decl_var : TIPO lista_var PYC
        | ;
lista_var : lista_var COMA IDENT
        | IDENT ;
sentencias : sentencias sentencia
        | sentencia ;
sentencia : IDENT ASIGN expresion PYC
        | FOR IDENT ASIGN expresion TO expresion DO sentencia
        | WHILE expresion DO sentencia ;
expresion : expresion OP_REL expresion
        | expresion OP_MAS_MENOS expresion
        | expresion OP_MUL_DIV expresion
        | OP_MAS_MENOS expresion %prec OP_NOT
        | OP_NOT expresion
        | IDENT
        | CONST ;

%%
```

Obtener la gramática con atributos para realizar las comprobaciones semánticas sobre tipos. Se debe indicar qué información sería necesaria tener previamente en la tabla de símbolos, estructura de ésta y aquellas funciones que se necesiten para su gestión (no es necesario implementar dichas funciones). Hay que tener en cuenta que el lenguaje admite la adaptación de tipos (coerción) de entero a real y de carácter a entero.

Solución:

Al insertar las acciones semánticas que se deben considerar en el problema, tendremos la necesidad de usar una **tabla de símbolos** para alojar las variables que se declaren. Las consideraciones semánticas necesarias son:

- Evitar duplicidad en declaración de variables.
- En sentencia de asignación, el identificador debe estar declarado y ser del mismo tipo que la expresión.

- En sentencia `for`, el identificador debe estar declarado, ser del mismo tipo que la expresión que se le asigna (la primera de ellas) y la segunda expresión también ha de ser del mismo tipo.
- En sentencia `while` la expresión debe ser de tipo lógico (dado que no se menciona la necesidad de evaluar expresiones lógicas de manera forzada en este tipo de sentencias, también se considera correcto evaluar expresiones de cualquier tipo considerando que toda expresión que no sea 0 es verdadera).
- En expresiones, dado que se indica claramente en el enunciado, no admite la posibilidad de adaptación de tipos, por lo tanto se evalúan expresiones que sean concordantes al operador y, donde haya dos expresiones, ambas deben ser del mismo tipo.

La estructura de la tabla de símbolos estará formada por la misma que la propuesta en las prácticas de la asignatura y los métodos de manejo de la misma serán:

- `int TS_InsertaId (char *nombre, tipodato tipo)` Busca en la TS una entrada que sea igual al nombre dado como argumento, en caso de encontrarla devuelve 0 y en otro caso lo inserta como variable asignándole el tipo que también se le aporta como argumento.
- `int TS_ExisteId (char *nombre)` Busca en la TS una entrada que sea igual al nombre dado como argumento, en caso de encontrarla devuelve el valor del campo `tipo` y `-1` en otro caso.

De esta forma, la gramática con atributos que resuelve el problema quedaría así:

```
.....
%%
programa      : PROGRAM bloque ;
bloque       : BEGIN decl_var sentencias END ;

// Aquí se puede solucionar insertando una acción en mitad de la regla.

decl_var      : TIPO { tmp= $1.atrib ; } lista_var PYC
               | ;
//-----
lista_var     : lista_var COMA IDENT
//-----
               { if (!TS_InsertaId ($3.nombre, tmp))
                 printf ("Error semántico:
                           Identificador duplicado\n");
               }
//-----
               | IDENT
//-----
               { if (!TS_InsertaId ($1.nombre, tmp))
                 printf ("Error semántico:
                           Identificador duplicado\n");
               }
sentencias    : sentencias sentencia
               | sentencia ;
//-----
sentencia     : IDENT ASIGN expresion PYC
//-----
               { if ((tipo=TS_ExisteId ($1.nombre))!=-1)
                 printf ("Error semántico:
                           Identificador no declarado en asignación\n");
                 else if (tipo!=$3.tipo)
                 printf ("Error semántico:
                           Tipos incompatibles en asignación\n");
               }
//-----
               | FOR IDENT ASIGN expresion TO expresion DO sentencia
//-----
               { if ((tipo=TS_ExisteId ($1.nombre))!=-1)
                 printf ("Error semántico:
                           Identificador no declarado en sentencia for\n");
                 else if (tipo != $4.tipo || tipo != $6.tipo)
                 printf ("Error semántico:
                           Tipos incompatibles en expresiones
                           de sentencia for\n");
               }
//-----
               | WHILE expresion DO sentencia
```

```

//-----
{ // Si admitimos únicamente expresiones lógicas:
  if ($2.tipo!=BOOLEANA)
    printf ("Error semántico:
              Tipo no booleano en sentencia while\n");
  // Si admitimos cualquier tipo de expresión, no habría que
  // añadir regla semántica.
}
//-----
expression : expression OP_REL expression
//-----
{ if ($1.tipo != $3.tipo)
  {
    printf ("Error semántico:
              Tipos incompatibles en expresión relacional\n");
    $$.tipo= INCORRECTO ;
  }
  else
    $$.tipo= $1.tipo ;
}
//-----
| expression OP_MAS_MENOS expression
//-----
{ if (($1.tipo==$3.tipo) && (($1.tipo==ENTERO) || ($1.tipo==REAL)))
  $$.tipo= $1.tipo ;
  else
  {
    printf ("Error semántico:
              Tipos incompatibles en expresión aritmética\n");
    $$.tipo= INCORRECTO ;
  }
}
//-----
| expression OP_MUL_DIV expression
//-----
{ if (($1.tipo==$3.tipo) && (($1.tipo==ENTERO) || ($1.tipo==REAL)))
  $$.tipo= $1.tipo ;
  else
  {
    printf ("Error semántico:
              Tipos incompatibles en expresión aritmética\n");
    $$.tipo= INCORRECTO ;
  }
}
//-----
| OP_MAS_MENOS expression %prec OP_NOT
//-----
{ if (($1.tipo==ENTERO) || ($1.tipo==REAL))
  $$.tipo= $1.tipo ;
  else
  {
    printf ("Error semántico:
              Tipo incompatible en expresión aritmética\n");
    $$.tipo= INCORRECTO ;
  }
}
//-----
| OP_NOT expression
//-----
{ if ($1.tipo==BOOLEANO)
  $$.tipo= $1.tipo ;
  else
  {
    printf ("Error semántico:
              Tipo incompatible en expresión lógica\n");
    $$.tipo= INCORRECTO ;
  }
}
//-----
| IDENT
//-----
{ if (!(tipo=TS_ExisteId ($1.nombre)))
  {
    printf ("Error semántico:

```

```

                                Identificador no declarado en asignación\n");
                                $$tipo= INCORRECTO ;
                                }
                                else
                                    $$tipo= tipo ;
                                }
//-----
| CONST
//-----
{ $$tipo= $1.atrib ; }
%%
.....
```