Resultados de los ejercicios del Examen de Procesadores de Lenguajes

(Convocatoria Ordinaria) celebrado el 30 de Junio de 2006.

3. (1 punto) Sea la gramática con las producciones siguientes:

Programa	$\longrightarrow$	**block** Sentencias **endblock**
Sentencias	$\longrightarrow$	Sentencias **;** Sentencia
	$\mid$	Sentencia
Sentencia	$\longrightarrow$	ident = Exp
	$\mid$	**do** Sentencia **until** Exp
	$\mid$	Programa
Exp	$\longrightarrow$	Exp + Exp
	$\mid$	Exp **-** Exp
	$\mid$	Exp ***** Exp
	$\mid$	Exp / Exp
	$\mid$	***** Exp
	$\mid$	**&** Exp
	$\mid$	**-** Exp
	$\mid$	const
	$\mid$	ident
const	$\longrightarrow$	const num
	$\mid$	num
num	$\longrightarrow$	$0\|1\|2\|\cdots\|9$
ident	$\longrightarrow$	ident letra
	$\mid$	letra
letra	$\longrightarrow$	$a\|b\|c\|\cdots\|z$

Obtener la tabla de tokens con el máximo nivel de abstracción suponiendo que:

(a) *(0.6 puntos) Se va a realizar traducción:* Las últimas producciones definen directamente secuencias de caracteres que se pueden definir mediante expresiones regulares. Alcanzando su máxima abstracción de acuerdo a su significado, la lista de palabras (secuencias de símbolos con significado propio) son las siguientes:

block, endblock, ;, ident, =, do, until, +, -, *, /, &, const

Teniendo en cuenta la misión sintáctica, sería posible agrupar los siguientes lexemas: **-,*** (son operadores binarios y unarios) y **+,/** (son operadores exclusivamente binarios). La tabla de tokens con el máximo nivel de abstracción queda como sigue:

Token	Patrón	Atributos
BLOCK	"block"	
ENDBLOCK	"endblock"	
PYC	";"	
ID	[a-z]+	
ASIG	"="	
DO	"do"	
UNTIL	"until"	
CONST	[0-9]+	
MENPOR	"−"|"∗"	0: "−" 1: "∗"
MASDIV	"+"|"/"	0: "+" 1: "/"
DIR	"&"	

(b) *(0.4 puntos) Sólo se va a realizar análisis sintáctico:* La gramática de partida obedece a una gramática libre de contexto causado por la iteración de sentencias basadas en la producción `Programa`. Sería posible abstraer hasta la sentencia de asignación (sería posible agrupar lexemas del apartado anterior que quedaban separados atendiendo a su significado). La tabla de tokens con el máximo nivel de abstracción para la realización única de análisis sintáctico sería la siguiente:

Token	Patrón	Atributos
BLOCK	"block"	
ENDBLOCK	"endblock"	
PYC	";"	
ASIGNACION	id"="{Exp}	
DO	"do"	
UNTIL	"until"	

4. (0.8 puntos) Construir la gramática abstracta para la tabla de tokens obtenida en el caso (a) del ejercicio anterior, eliminar la recursividad y factorizarla, si fuese necesario, y construir la tabla de análisis LL(1) para la gramática transformada.

La gramática abstracta es la siguiente:

$$
\begin{array}{rcl}
\text{Programa} & \longrightarrow & \text{BLOCK Sentencias ENDBLOCK} \\
\text{Sentencias} & \longrightarrow & \text{Sentencias PYC Sentencia} \\
 & | & \text{Sentencia} \\
\text{Sentencia} & \longrightarrow & \text{ID ASIG Exp} \\
 & | & \text{DO Sentencia UNTIL Exp} \\
 & | & \text{Programa} \\
\text{Exp} & \longrightarrow & \text{Exp MASDIV Exp} \\
 & | & \text{Exp MENPOR Exp} \\
 & | & \text{MENPOR Exp} \\
 & | & \text{DIR Exp} \\
 & | & \text{CONST} \\
 & | & \text{ID}
\end{array}
$$

Primero habría que eliminar la recursividad a la izquierda en las producciones que definen `Sentencias` y `Exp`.

Para el caso de `Sentencias` las producciones quedarían así:

$$
\begin{array}{rcl}
\text{Sentencias} & \longrightarrow & \text{Sentencia Sentencias'} \\
\text{Sentencias'} & \longrightarrow & \text{PYC Sentencia Sentencias'} \\
 & | & \epsilon
\end{array}
$$

Para el caso de `Exp` las producciones quedarían así:

$$
\begin{array}{rcl}
\text{Exp} & \longrightarrow & \text{MENPOR Exp Exp'} \\
 & | & \text{DIR Exp Exp'} \\
 & | & \text{CONST Exp'} \\
 & | & \text{ID Exp'} \\
\text{Exp'} & \longrightarrow & \text{MASDIV Exp Exp'} \\
 & | & \text{MENPOR Exp Exp'} \\
 & | & \epsilon
\end{array}
$$

No es necesario factorizar, por lo que la gramática resultante es la siguiente:

1)	Programa	$\longrightarrow$	BLOCK Sentencias ENDBLOCK
2)	Sentencias	$\longrightarrow$	Sentencia Sentencias'
3)	Sentencias'	$\longrightarrow$	PYC Sentencia Sentencias'
4)		$\mid$	ϵ
5)	Sentencia	$\longrightarrow$	ID ASIG Exp
6)		$\mid$	DO Sentencia UNTIL Exp
7)		$\mid$	Programa
8)	Exp	$\longrightarrow$	MENPOR Exp Exp'
9)		$\mid$	DIR Exp Exp'
10)		$\mid$	CONST Exp'
11)		$\mid$	ID Exp'
12)	Exp'	$\longrightarrow$	MASDIV Exp Exp'
13)		$\mid$	MENPOR Exp Exp'
14)		$\mid$	ϵ

La tabla de análisis LL(1) es la siguiente (obtenida con sefalas):

	BLOCK	ENDBLOCK	PYC	ID	ASIG	DO	UNTIL	CONST	MENPOR	MASDIV	DIR	$
Programa	1											
Sentencias	2			2		2						
Sentencia	7			5		6						
Exp				11				10	8		9	
Sentencias'		4	3									
Exp'		14	14				14		13/14	12/14		

Tal y como se muestra en la tabla, aparecen dos celdas con conflictos, por lo tanto, **no es LL(1)**.

5. (0.5 puntos) Dada la gramática con las producciones siguientes:

A	$\longrightarrow$	A b B Z
	$\mid$	ϵ
B	$\longrightarrow$	c d
	$\mid$	B d b
	$\mid$	ϵ
Z	$\longrightarrow$	g Z
	$\mid$	ϵ

Realizar los siguientes cálculos:

(a) (0.2 puntos) Si hacemos análisis LR(1), sea $I_p = \{[A \longrightarrow Ab \cdot BZ, b], \ldots\}$, construir el estado $goto(I_p, B)$ y obtener las acciones que le corresponde en la tabla de análisis.

Solución:

$$goto(I_p, B) = \{[A \longrightarrow AbB \cdot Z, b], [Z \longrightarrow \cdot gZ, b], [Z \longrightarrow \cdot, b]\}$$

Las acciones en dicho estado sería reducir la producción $Z \longrightarrow \epsilon$ ante el símbolo b y desplazar a otros estados ante los símbolos Z y g.

(b) (0.2 puntos) Suponiendo que hacemos análisis SLR y apareciera en un estado I_j el item $[B \longrightarrow cd\cdot]$, obtener las acciones que se deberían introducir en la tabla de análisis.

Solución: Reducir la producción $B \longrightarrow cd$ en $Seguidores(B) = \{g, d, b, \$\}$.

(c) (0.1 puntos) Si hacemos análisis SLR y aparece en el estado I_1 el item $[A' \longrightarrow A\cdot]$, obtener las acciones que se deberáin introducir en la tabla de análisis (siendo A' el símbolo inicial de la gramática aumentada).

Solución: Aceptar la gramática ante el símbolo $.

10. (1.1 puntos) A partir del archivo "host.ugr.es-access_log" de todas las peticiones a través del servidor web, se pretende controlar todos los accesos a cada dirección IP. El formato de dicho archivo se muestra en el examen.

(a) (0.7 puntos) Definir la gramática con atributos para poder controlar qué dirección IP ha solicitado el mayor número de accesos.

Solución: La gramática que define la sintaxis del archivo log es la siguiente:

$$\begin{array}{rcl} \text{Archivo\_log} & \longrightarrow & \text{Archivo\_log linea} \\ & | & \text{linea} \\ \text{linea} & \longrightarrow & \text{IP DATE HOUR NUM REQUEST NAV} \end{array}$$

Dado que se nos piden consideraciones sobre IP y NAV, el resto de los campos podrían considerarse de manera agrupada por simplicidad, sin embargo, se prefiere dejarlo tal y como se define con su significado.

Es necesario almacenar la lista de IPs con sus accesos producidos. Podemos suponer una estructura de datos al estilo de tabla de símbolos (array de registros formados por la pareja de valores IP, accesos). Para ello, se definen las siguientes funciones de uso de dicha estructura de datos:

- `int existeIP(char *IP)`: Busca la dirección IP dada como argumento, la busca en la estructura de datos y devuelve la posición del array, si ha sido encontrada y -1 en caso contrario.
- `void insertaIP(char *IP)`: Crea una nueva entrada para la IP dada como argumento y fija su número de accesos en 1.
- `void incAccesoIP(int indice)`: Accede a la posición del array indicado por el argumento e incrementa el número de accesos en dicha entrada.

El atributo que se evalúa en la dirección IP es de tipo cadena de caracteres. La acción semántica que habría que añadir en la gramática sería la siguiente (suponemos inicializada la estructura de datos totalmente vacía).

$$\begin{array}{rcl} \text{Archivo\_log} & \longrightarrow & \text{Archivo\_log linea} \\ & | & \text{linea} \\ \text{linea} & \longrightarrow & \text{IP DATE HOUR NUM REQUEST NAV} \end{array}$$

```
{
   if (int pos=existeIP($1)==-1)
      insertaIP($1) ;
   else
      incAccesoIP(pos);
}
```

(b) (0.4 puntos) Definir la gramática con atributos para poder controlar el número de accesos realizados por navegadores basados en **Mozilla** y el número de accesos realizados por navegadores basados en **iexplore**.

Solución: Existen varias formas para una correcta solución. La que se propone se basa en dos variables `numIexplore` y `numMozilla` ambos de tipo entero que alojarán el número de accesos registrados en ambos casos. El símbolo terminal NAV es de tipo cadena de caracteres.

Se considera la función `int contiene(char *cad, char *cont)` que devuelve verdadero si la cadena `cont` está contenida en la cadena `cad` y falso en caso contrario.

La gramática con atributos resultante queda como sigue:

$$
\begin{array}{rcl}
\text{Archivo\_log} & \longrightarrow & \text{Archivo\_log linea} \\
 & | & \text{linea} \\
\text{linea} & \longrightarrow & \text{IP DATE HOUR NUM REQUEST NAV}
\end{array}
$$

```
{
   if (contiene("iexplore", $6))
      numIexplore++ ;

   if (contiene("Mozilla", $6))
      numMozilla++ ;
}
```

Si se deseara imprimir el número de accesos, se podría hacer como una producción nueva de mayor abstracción cuya acción semántica fuera imprimir ambos valores.

También sería posible obtener una solución mediante atributos sintetizados, pero la manera más eficiente sería la que se ha mostrado.

6. [1.0 puntos] Dada la siguiente gramática que define un lenguaje ensamblador de un procesador RISC:

program	→	head block
head	→	**'CODE'** addr addr
block	→	instr block
	|	instr
instr	→	**'LOAD'** reg addr
	|	**'STORE'** addr reg
	|	**'STORE'** addr const
	|	**'MOVE'** reg reg
	|	**'ADD'** reg reg reg
	|	**'SUB'** reg reg reg
	|	**'CMP'** reg reg
	|	**'JMP'** const
	|	**'HALT'**
reg	→	**'R1'** | **'R2'** | **'R3'** | **'R4'**
addr	→	'#' const
const	→	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) [0.8 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	Atributos
1	CODE	"CODE"	
2	LOAD	"LOAD"	
3	STORE	"STORE"	
4	MOVECMP	"MOVE"|"CMP"	0:move, 1:cmp
5	ADDSUB	"ADD"|"SUB"	0:add, 1:sub
6	JMP	"JMP"	
7	HALT	"HALT"	
8	REG	"R1"|"R2"|"R3"|"R4"	0:R1, 1:R2, 2:R3, 3:R4
9	ADDR	'#'[0-9]+	
10	CONST	[0-9]+	

(b) [0.2 puntos] Análisis sintáctico.

Solución:

Cuando únicamente se va a realizar análisis sintáctico, no hay que distinguir palabras según su valor semántico elemental ni considerar secuencias de palabras atendiendo a determinados valores semánticos estructurados.

Se observa que la gramática original se rige bajo las condiciones de una **gramática regular**, por lo tanto, sería posible expresar la gramática mediante una única expresión regular. Se empleará la sintaxis de LEX para especificar el patrón:

```
%option noyywrap

sep     ([ \t\n])*
cons    [0-9]+
addr    "#"{cons}
reg     "R"[1234]
head    "HEAD"{sep}{addr}{sep}{addr}{sep}
load    "LOAD"{sep}{reg}{sep}{reg}
store   "STORE"{sep}{addr}{sep}({reg}|{cons})
movcmp  ("MOVE"|"CMP"){sep}{reg}{sep}{reg}
addsub  ("ADD"|"SUB"){sep}{reg}{sep}{reg}{sep}{reg}
jmp     "JMP"{sep}{cons}
halt    "HALT"
instr   (({load}|{store}|{movcmp}|{addsub}|{jmp}|{halt}){sep})*

%%
[ \t\n]+                        ;
{head}{sep}({instr})*           { ECHO; printf("\n *** Secuencia Correcta ***\n");  }
.                               { printf("[Linea %d] Error en %s\n", yylineno, yytext); }
%%
```

7. [0.25 puntos] Especificar la gramática abstracta del problema 6(a) usando la sintaxis de YACC y realizar las modificaciones adicionales para que considere recuperación de errores en la entrada ante las siguientes situaciones:

Solución:

En primer lugar vamos a especificar la gramática abstracta usando la sintaxis de YACC:

```
%token CODE LOAD STORE MOVECMP ADDSUB
%token JMP HALT REG ADDR CONST
%start program
%%
program : head block ;
head    : HEAD ADDR ADDR ;
block   : instr block
        | instr ;
instr   : LOAD REG ADDR
        | STORE ADDR REG
        | STORE ADDR CONST
        | MOVECMP REG REG
        | ADDSUB REG REG REG
        | JMP CONST
        | HALT ;
%%
```

(a) [0.10 puntos] Cuando acontezca un error en una instrucción (`instr`).

Solución:

Cuando aparezca un error por no poder construir sintácticamente las palabras que deben aparecer en una instrucción, debemos añadir la siguiente producción:

```
%token CODE LOAD STORE MOVECMP ADDSUB
%token JMP HALT REG ADDR CONST
%start program
%%
program : head block ;
head    : HEAD ADDR ADDR ;
block   : instr block
        | instr ;
instr   : LOAD REG ADDR
        | STORE ADDR REG
        | STORE ADDR CONST
        | MOVECMP REG REG
        | ADDSUB REG REG REG
        | JMP CONST
        | HALT
        | error ;     /* Producción de error */
%%
```

(b) [0.15 puntos] Cuando acontezca un error en la referencia a los registros que aparecen como parámetro de las instrucciones.

Solución:

Cuando aparezca un error debido a que no aparece la especificación de un registro (token `REG`) nos diría que ha aparecido cualquier otro token menos el del propio registro. Para esta situación, dado que `REG` es un token, debemos considerar un nuevo símbolo no terminal que nos lleve a la formación de `REG` o, si no aparece, se pueda recuperar el análisis indicando la producción de error ante ese nuevo no terminal que se crea. Por último, cualquier aparición del token `REG` deberá sustituirse por el nuevo no terminal `reg`:

```
%token CODE LOAD STORE MOVECMP ADDSUB
%token JMP HALT REG ADDR CONST
%start program
%%
program : head block ;
head    : HEAD ADDR ADDR ;
block   : instr block
        | instr ;
instr   : LOAD reg ADDR     /* Se sustituye REG por el no terminal ''reg'' */
        | STORE ADDR reg
        | STORE ADDR CONST
        | MOVECMP reg reg
        | ADDSUB reg reg reg
        | JMP CONST
        | HALT
        | error ;
reg     : REG         /* Un registro es REG */
        | error ;     /* en otro caso, error */
%%
```

8. [0.5 puntos] Dada la gramática abstracta del problema 6(a) usando la sintaxis de YACC, realizar los cambios que se estimen oportunos, construir la tabla de análisis LL(1) y verificar si dicha gramática cumple los requisitos para ser analizada empleando esta estrategia de análisis.

Solución:

Para realizar un análisis descendente mediante la estrategia de análisis LL(1), debemos considerar dos aspectos concretos en la gramática abstracta de partida:

- Recursividad a la izquierda.
- Factorización.

El primero de ellos no acontece en la gramática pero el problema de la factorización sí que aparece en dos lugares. Una es cuando se especifica el no terminal `block`:

```
....
block   : instr block
        | instr ;
....
```

La solución sería la siguiente:

```
....
block   : instr block2
block2  : block
        |  ;
....
```

La otra factorización acontece en el no terminal `instr` en cuanto a sus opciones `STORE` y en el siguiente token `ADDR`.

```
....
instr   : ....
        | STORE ADDR REG
        | STORE ADDR CONST
        ....
....
```

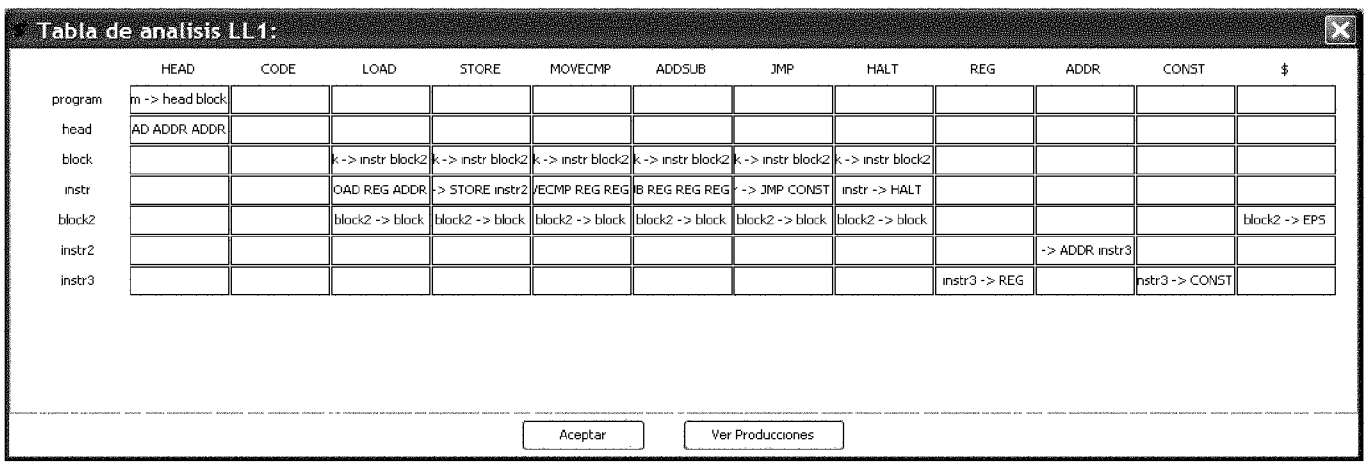
La solución sería la siguiente:

```
....
instr   : .....
        | STORE instr2
        | .....
instr2  : ADDR REG
        | ADDR CONST ;
....
```

Tendríamos que factorizar de nuevo en `instr2` dado que comienzan por el mismo terminal `ADDR`. La solución final sería la siguiente:

```
....
instr   : .....
        | STORE instr2
        | .....
instr2  : ADDR instr3 ;
instr3  : REG
        | CONST ;
....
```

Una vez calculados los iniciales y seguidores de aquellas producciones en las que aparezcan varias alternativas, la tabla de análisis LL(1) quedaría así (obtenida mediante Sefalas):



Por lo tanto, ante la ausencia de conflictos, esta gramática resultante es LL(1).

7. [1.2 puntos] Dada la siguiente gramática:

programa	→	funciones main
	|	main
funciones	→	funciones funcion
	|	funcion
funcion	→	"function" ident "()" bloque
main	→	"main" bloque
bloque	→	"{" sentencias "}"
sentencias	→	sentencias sentencia
	|	sentencia
sentencia	→	ident "=" expresion ";"
	|	bloque
expresion	→	expresion "<" ident ">" ident
	|	expresion "?" ident ":" ident
	|	expresion "+" expresion
	|	"*" expresion
	|	"&" expresion
	|	ident
	|	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	IDENT	[a-z]+
3	PARENTS	"()"
4	MAIN	"main"
5	LLAVEI	"{"
6	LLAVED	"}"
7	ASIGN	"="
8	PYC	";"
9	CONST	[0-9]"."[0-9]
10	OP_REL_MEN	"<"
11	OP_REL_MAY	">"
12	OP_INTERR	"?"
13	OP_DOSPUNT	":"
14	OP_MAS	"+"
15	OP_MUL_AMP	"*" | "&"

(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 aún más que la anterior pero no del todo dado que la gramática de partida es libre de contexto (las llaves de los bloques y la posibilidad de la recursividad de su contenido evitan toda posibilidad de gramática regular).

En este caso la abstracción es posible realizarla, desde abajo hasta arriba, hasta abstraer una sentencia de asignación. Dado que una expresión es posible mostrarla mediante un patrón, sería posible concatenarle antes un identificador junto al carácter de la asignación. Desde este punto hasta la primera de las producciones de la gramática lo único que podríamos abstraer es el

concepto de cabecera de la función. Desde ahí hasta el principio no es posible realizar mayor abstracción, por lo que la tabla de tokens quedaría así:

Nº	Token	Patrón
1	CABECERAFUNC	"function{ident}()"
2	MAIN	"main"
3	LLAVEI	"{"
4	LLAVED	"}"
5	ASIGNACION	"{ident}={expresion}"

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

1) A → B c d
2) | H d H
3) | F a
4) | ε
5) B → a
6) | ε
7) H → d
8) | ε
9) F → c
10) | ε

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: (H,d), (A,$) y (A,a) 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			$A \rightarrow Bcd$ $A \rightarrow Fa$	$A \rightarrow \varepsilon$
B				
H		$H \rightarrow d$ $H \rightarrow \varepsilon$		
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	*d*7/*r*6	*d*6/*r*8	*d*5/*r*10	*r*4	1	2	3	4

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      // "+", "-"
%left OP_MUL_DIV        // "*", "/"
%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 no admite coerciones.

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.
                                        }
//-------------------------------------------------------------------------------------------------------
expresion   : expresion OP_REL expresion
//-------------------------------------------------------------------------------------------------------
                                        { if ($1.tipo != $3.tipo)
                                          {
                                            printf ("Error semántico:
                                                     Tipos incompatibles en expresión relacional\n");
                                            $$.tipo= INCORRECTO ;
```

```
                                        }
                                        else
                                           $$.tipo= $1.tipo ;
                                     }
//-------------------------------------------------------------------------------------------------------
          | expresion OP_MAS_MENOS expresion
//-------------------------------------------------------------------------------------------------------
                                     { if (($1.tipo==$3.tipo) && (($1.tipo==ENTERO) || ($1.tipo==REAL)))
                                           $$.tipo= $1.tipo ;
                                        else
                                        {
                                           printf ("Error semántico:
                                                    Tipos incompatibles en expresión aritmética\n");
                                           $$.tipo= INCORRECTO ;
                                        }
                                     }
//-------------------------------------------------------------------------------------------------------
          | expresion OP_MUL_DIV expresion
//-------------------------------------------------------------------------------------------------------
                                     { if (($1.tipo==$3.tipo) && (($1.tipo==ENTERO) || ($1.tipo==REAL)))
                                           $$.tipo= $1.tipo ;
                                        else
                                        {
                                           printf ("Error semántico:
                                                    Tipos incompatibles en expresión aritmética\n");
                                           $$.tipo= INCORRECTO ;
                                        }
                                     }
//-------------------------------------------------------------------------------------------------------
          | OP_MAS_MENOS expresion %prec OP_NOT
//-------------------------------------------------------------------------------------------------------
                                     { if (($1.tipo==ENTERO) || ($1.tipo==REAL))
                                           $$.tipo= $1.tipo ;
                                        else
                                        {
                                           printf ("Error semántico:
                                                    Tipo incompatible en expresión aritmética\n");
                                           $$.tipo= INCORRECTO ;
                                        }
                                     }
//-------------------------------------------------------------------------------------------------------
          | OP_NOT expresion
//-------------------------------------------------------------------------------------------------------
                                     { if ($1.tipo==BOOLEANO)
                                           $$.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 ; }
%%
........
```

PROCESADORES DE LENGUAJES
4º INGENIERÍA INFORMÁTICA

PARCIAL DE 2008 – PREGUNTAS DE TEORÍA

1. **¿Es susceptible la aplicación de un nuevo proceso de abstracción léxica a una gramática abstracta? En caso afirmativo razone bajo qué condiciones podría realizarse.**

2. **Si una gramática es susceptible de ser factorizada, no se factoriza y se construye la tabla de análisis LL(1), ¿cómo quedaría reflejada esta situación en la tabla? ¿Y en una LL(k) para un *k* adecuado?**

3. **Supóngase el análisis sintáctico ascendente de una cadena. Si en la pila del autómata aparece una secuencia de símbolos en los que se pueden distinguir más de 1 pivote, ¿surgiría algún tipo de conflicto? ¿Qué tipo de conflicto? ¿Admitiría un análisis predictivo? Razone la respuesta.**

4. **En una estrategia de análisis sintáctico ascendente, cuando se desplaza un símbolo de la pila, ¿se puede asegurar que la secuencia de entrada es siempre correcta hasta ese símbolo desplazado? Razone la respuesta.**

5. **Dada la tabla de análisis LR de una gramática en la que bajo los operadores + y * aparecen conflictos desplaza/reduce, elija las acciones que deben aparecer para resolver esos conflictos. Razone la respuesta en los casos siguientes:**

   1. * > +
   2. * = + **pero el + asociativo por la izqda y * por la derecha**

6. [1.0 puntos] Dada la siguiente gramática:

programa	→	funciones main
	|	main
funciones	→	funciones funcion
	|	funcion
funcion	→	'**function**' ident '()' bloque
main	→	'**main**' bloque
bloque	→	'{' sentencias '}'
sentencias	→	sentencias sentencia
	|	sentencia
sentencia	→	ident '=' expresion '**;**'
	|	bloque
expresion	→	expresion '[' ident ']' ident
	|	expresion '**?**' ident '**:**' ident
	|	expresion '**+**' const
	|	'***** expresion
	|	'**&**' expresion
	|	ident
	|	'[' 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) [0.7 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	CONST	`[0-9]+"."[0-9]+`
2	IDENT	`"[a-z]+"`
3	CORI	`"["`
4	CORD	`"]"`
5	AMPAST	`"&"|"*"`
6	MAS	`"+"`
7	INTER	`"?"`
8	DOSP	`":"`
9	ASIGN	`"="`
10	PYC	`";"`
11	LLAVEI	`"{"`
12	LLAVED	`"}"`
13	MAIN	`"main"`
14	FUNCTION	`"function"`
15	PARENT	`"()"`

(b) [0.3 puntos] Análisis sintáctico.

Solución:

Cuando únicamente se va a realizar análisis sintáctico, primero se observa que la gramática abstracta resultante del proceso anterior es libre de contexto debido a una producción en la forma de construcción de una `sentencia`:

sentencia → bloque

Por lo tanto, sería posible realizar una abstracción mayor hasta el símbolo no terminal `expresion`, cuyas formas de producción obedecen a una gramática regular. De esta forma, la tabla de tokens con máximo nivel de abstracción quedaría así:

Nº	Token	Patrón
1	SENTASIG	`[a-b]+"="{expresion}";"`
2	LLAVEI	`"{"`
3	LLAVED	`"}"`
4	MAIN	`"main"`
5	CABFUNCTION	`"function()"`

Donde `{expresion}` representa el patrón de construcción de las formas de producción de una `expresion`.

7. 0.5 puntos Dada la gramática siguiente:

$$
\begin{array}{lcl}
A & \rightarrow & A\ b\ B\ Z \\
 & | & \varepsilon \\
B & \rightarrow & c\ d \\
 & | & B\ d\ b \\
 & | & \varepsilon \\
Z & \rightarrow & g\ Z \\
 & | & \varepsilon
\end{array}
$$

Realizar los cambios que se estimen oportunos sobre la misma y construir la tabla de análisis LL(1).

Solución:

Se observan varias producciones con recursividad a la izquierda. Los símbolos no terminales que aparecen con recursividad a la izquierda son A y B. Tras eliminar la recursividad a la izquierda nos queda la siguiente gramática equivalente:

$$
\begin{array}{lcl}
A & \rightarrow & A' \\
A' & \rightarrow & b\ B\ Z\ A' \\
 & | & \varepsilon \\
B & \rightarrow & c\ d\ B' \\
 & | & B' \\
B' & \rightarrow & d\ b\ B' \\
 & | & \varepsilon \\
Z & \rightarrow & g\ Z \\
 & | & \varepsilon
\end{array}
$$

No es necesario factorizar la gramática anterior. Para la construcción de la tabla de análisis LL(1) calculamos los iniciales y seguidores.

Iniciales(A)	=	{b, ε}	Seguidores(A)	=	{$}
Iniciales(B)	=	{c, d, ε}	Seguidores(B)	=	{g, b, $}
Iniciales(Z)	=	{g, ε}	Seguidores(Z)	=	{b, $}
Iniciales(A')	=	{b, ε}	Seguidores(A')	=	{$}
Iniciales(B')	=	{d, ε}	Seguidores(B')	=	{g, b, $}

Por lo que la tabla de análisis LL(1) quedaría así:

	b	c	d	g	$
A	$A \rightarrow A'$				$A \rightarrow A'$
B	$B \rightarrow B'$	$B \rightarrow cdB'$	$B \rightarrow B'$	$B \rightarrow B'$	$B \rightarrow B'$
Z	$Z \rightarrow \varepsilon$			$Z \rightarrow gZ$	$Z \rightarrow \varepsilon$
A'	$A' \rightarrow bBZA'$				$A' \rightarrow \varepsilon$
B'	$B' \rightarrow \varepsilon$		$B' \rightarrow dbB'$	$B' \rightarrow \varepsilon$	$B' \rightarrow \varepsilon$

No presenta conflictos la tabla de análisis, por lo tanto, se trata de una gramática LL(1).

8. 0.2 puntos Dada la gramática del problema 7, completar las acciones que debe aparecer en el estado inicial I_0 de la tabla de análisis LR(1).

Estado	b	c	d	g	$	A	B	Z
I_0								

Solución:

Los items del estado I_0 son los siguientes:

$$I_0 = \{[A' \rightarrow \cdot A, \$], [A \rightarrow \cdot AbBZ, \$/b], [A \rightarrow \varepsilon\cdot, \$/b]\}$$

A partir de dicho estado la única transición constituiría el siguiente estado:

$$I_1 = goto(I_0, A) = \{[A' \rightarrow A\cdot, \$], [A \rightarrow A \cdot bBZ, \$/b]\}$$

Por lo tanto, el estado I_0 de la tabla de análisis LR(1) quedaría así:

Estado	b	c	d	g	$	A	B	Z
I_0	r_2				r_2	1		

PROCESADORES DE LENGUAJES
4º INGENIERÍA INFORMÁTICA

EXAMEN DE SEPTIEMBRE DE 2008 :
PREGUNTAS DE TEORÍA

1. **Relación entre los conceptos de token y lexema con la semántica de un lenguaje de programación (0.6 puntos)**

2. **Tenemos una gramática que es tanto de precedencia simple como LR(1). ¿Por qué sería mejor construir un analizador LR(1) que uno de precedencia simple? (0.6 puntos)**

3. **Supongamos que tenemos una ALU (unidad aritmético-lógica) capaz de evaluar expresiones aritméticas complejas. ¿Tendría sentido la fase de generación de código intermedio? (0.6 puntos)**

4. **En una gramática con atributos en la que el símbolo no terminal de la parte izquierda de la regla tiene un atributo heredado, ¿se podría evaluar ese atributo en análisis ascendente? ¿Y si el símbolo aparece en la parte derecha, en qué casos se podría evaluar? (0.6 puntos)**

5. **¿Qué diferencias existen entre un compilador y un intérprete recursivo? (0.6 puntos)**

6. **Construir el GDA asociado al siguiente código y escribir el código que genera: (0.6 puntos)**

```
temp1 = i+4;
temp2 = i+4;
i = 3 + temp2;
h = i + 1;
j = h + 1;
z = h;
```

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_MI_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 → F H B
2) | H d H
3) | a F
4) | ε
5) B → d
6) | ε
7) H → a
8) | ε
9) F → c
10) | ε

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			$B \rightarrow \varepsilon$
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	*d*5	*r*10/*r*8	*d*4/*r*10	*r*4/*r*10	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.
                                    }
//-------------------------------------------------------------------------------------------------------------
expresion  : expresion OP_REL expresion
//-------------------------------------------------------------------------------------------------------------
                                    { if ($1.tipo != $3.tipo)
                                      {
                                         printf ("Error semántico:
                                                  Tipos incompatibles en expresión relacional\n");
                                         $$.tipo= INCORRECTO ;
                                      }
                                      else
                                         $$.tipo= $1.tipo ;
                                    }
//-------------------------------------------------------------------------------------------------------------
           | expresion OP_MAS_MENOS expresion
//-------------------------------------------------------------------------------------------------------------
                                    { 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 ;
                                      }
                                    }
//-------------------------------------------------------------------------------------------------------------
           | expresion OP_MUL_DIV expresion
//-------------------------------------------------------------------------------------------------------------
                                    { 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 expresion %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 expresion
//-------------------------------------------------------------------------------------------------------------
                                    { 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 ; }
%%
.........
```

Procesadores de Lenguajes

Examen Parcial 2009

7 de mayo de 2009

El examen está calificado sobre 3.5 puntos, deben obtenerse al menos 0.875 puntos en cada parte.

Teoría

1. ¿Qué características debe tener un lenguaje para que sea posible aplicar un proceso de abstención léxica? Razone la respuesta. (0.35 puntos)

2. Dada una gramática sin producciones a la cadena vacía, ¿sería necesario realizar el cálculo de los conjuntos SEGUIDORES para una estrategia de análisis sintáctico SLR? Razone la respuesta. (0.35 puntos)

3. Dada una tabla de análisis de precedencia simple, ¿qué condiciones debería tener la gramática para que pudiéramos eliminar de dicha tabla de análisis las columnas referentes a los símbolos no terminales? Razone la respuesta. (0.35 puntos)

4. ¿En qué circunstancias se puede producir un conflicto reduce/desplaza en una gramática SLR? ¿Y en un conflicto de tipo desplaza/desplaza? Razone la respuesta. (0.35 puntos)

5. ¿Qué aporta la estrategia de análisis sintáctico ascendente LR con respecto a la de precedencia simple y sobre qué concepto se sustenta? Razone la respuesta. (0.35 puntos)

Problemas

6. Dada la siguiente gramática que representa la declaración de tipos de registro según la sintaxis de Pascal. (1 punto)

   registro → registros registro | ∈

   registro → **type** nombre = **record** lista_campos **end** pyc

   lista_campos → lista_campos campo | campo
   campo → nombre: tipo pyc

   tipo → **integer** | **real** | **character** |
   boolean | nombre

   nombre → nombre caracter_digito | caracter

   caracter_digito → caracter | digito

   caracter → a | b | c | ... | z

   digito → 0 | 1 | 2 | ... | 9

   pyc → ;

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

   a) Traducción de un texto en sintaxis de Pascal a otro con sintaxis de C. (0.7 puntos)

   b) Sólo análisis sintáctico. (0.3 puntos)

7. Escriba la gramática abstracta del supuesto a) del ejercicio anterior, realice los cambios que se estimen oportunos sobre la misma y construya la tabla de análisis LL(1). (0.5 puntos)

8. Escriba la gramática abstracta del ejercicio 6, construya el estado I_0 para el caso de análisis LR(1) y los estados que son alcanzados por una transición desde el estado I_0 . ¿Habría conflictos en la fila del estado 0 de la tabla de análisis?

PROBLEMAS

6. [1.0 puntos] Dada la siguiente gramática que representa la declaración de tipos registro según la sintaxis de Pascal

registros	→	registros registro
	|	registro
registro	→	**type** nombre = **record** lista_campos **end** pyc
lista_campos	→	lista_campos campo
	|	campo
campo	→	nombre **:** tipo pyc
tipo	→	**integer** | **real** | **character** | **boolean** | nombre
nombre	→	nombre (caracter|digito)
	|	caracter
caracter	→	a|b|c|···|z
digito	→	0|1|2|···|9
pyc	→	**;**

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

(a) [0.7 puntos] Traducción de un texto en sintaxis de Pascal a otro con sintaxis de C.

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	TYPE	"type"
2	ASIGNRECORD	"="{sep}*"record"
3	END	"end"
4	PYC	";"
5	DPTOS	":"
6	TIPO	"integer"|"real"|"character"|"boolean"
7	NOMBRE	[a-z]+([a-z]|[0-9])*

donde {sep} es cualquier secuencia de espacios en blanco, tabuladores o retornos de carro.

En cuanto a los atributos, tan sólo el token TIPO es susceptible de llevarlos. En concreto, uno para cada tipo básico. Para contemplar que el tipo de uno de los campos de un registro se basa en un tipo definido por el usuario, no es posible situar el patrón {nombre} como parte del token TIPO porque en otro apartado del lenguaje aparece con significado propio. En este caso, ya que los tipos de un campo pueden ser básicos o definidos, habrá una producción para los básicos y otra para el caso de un nombre definido por el usuario.

(b) [0.3 puntos] Sólo análisis sintáctico.

Solución:

Cuando únicamente se va a realizar análisis sintáctico, primero se observa que la gramática abstracta resultante del proceso anterior es libre de contexto debido a una producción en la forma de construcción de un `registro`:

Nº	Token	Patrón
1	REGISTROS	{registro}*

donde {registro} se define de la siguiente forma usando notación Lex:

```
sep       [ \n\t]*
letra     [a-z]
digito    [0-9]
nombre    {letra}({letra}|{digito})*
tipo      "integer"|"real"|"character"|"boolean"|{nombre}
campo     {nombre}{sep}":"{sep}{tipo}{sep}";"
registro  {sep}"type"{sep}{nombre}{sep}"="    \
          {sep}"record"{sep}{campo}+{sep}"end"{sep}";"
%%
{sep}              ;
{registro}*       ;
.                  printf (''[Linea %d]: Error leyendo caracter %s\n'', yylineno, yytex
%%
```

7. [0.5 puntos] Escriba la gramática abstracta del supuesto (a) del ejercicio anterior, realice los cambios que se estimen oportunos sobre la misma y construya la tabla de análisis LL(1):

Solución:

La gramática abstracta para el supuesto (a) del ejercicio anterior, usando notación YACC es la siguiente:

```
%token TYPE ASIGNRECORD END PYC DPTOS TIPO NOMBRE
%start registros
%%
registros    : registros registro
             | ;
registro     : TYPE NOMBRE ASIGNRECORD lista_campos END PYC ;
lista_campos : lista_campos campo
             | campo ;
campo        : NOMBRE DPTOS tipo_nombre PYC ;
tipo_nombre  : TIPO
             | NOMBRE ;
%%
```

Inicialmente se aprecian varias producciones en las que hay recursividad a la izquierda. En concreto serían las siguientes:

```
registros    : registros registro
             | ;
```

y también en:

```
lista_campos : lista_campos campo
             | campo ;
```

Para la primera es posible realizar un cambio tan simple como intercambiar los símbolos no terminales que intervienen en la correspondiente producción `registros` y `registro`. de esa forma se resuelve el problema de la recursividad a la izquierda. El resultado sería:

```
registros    : registro registros
             | ;
```

Para la segunda es posible realizar lo mismo que en el caso anterior, salvo que ahora se produce el problema de la factorización. Por lo tanto, aplicando la regla de intercambios entre los símbolos no terminales `campo` y `lista_campos` y la regla de sustitución para el problema de la factorización nos quedaría así:

```
lista_campos  : campo lista_campos2 ;
lista_campos2 : lista_campos
              | ;
```

La gramática abstracta resultante final para la realización de la tabla de análisis LL(1) quedaría así:

```
%token TYPE ASIGNRECORD END PYC DPTOS TIPO NOMBRE
%start registros
%%
(1) registros     : registro registros
(2)               | ;
(3) registro      : TYPE NOMBRE ASIGNRECORD lista_campos END PYC ;
(4) lista_campos  : campo lista_campos2 ;
(5) lista_campos2 : lista_campos
(6)               | ;
(7) campo         : NOMBRE DPTOS tipo_nombre PYC ;
(8) tipo_nombre   : TIPO
(9)               | NOMBRE ;
%%
```

La tabla de análisis para análisis sintáctico descendente LL(1) quedaría así:

	TYPE	NOMBRE	ASIGNRECORD	END	PYC	DPTOS	TIPO	$
registros	(1)							(2)
registro	(3)							
lista_campos		(4)						
lista_campos2		(5)		(6)				
campo		(7)						
tipo_nombre		(9)					(8)	

No presenta conflictos la tabla de análisis, por lo tanto, se trata de una gramática LL(1).

8. 0.25 puntos Escriba la gramática abstracta del ejercicio 6(a), construya el estado I_0 para el caso de análisis LR(1) y los estados que son alcanzados por una transición desde ese estado I_0. ¿Habría conflictos en la fila del estado 0 de la tabla de análisis?

Solución:

La gramática abstracta para el ejercicio 6(a), usando notación YACC, es la siguiente:

```
%token TYPE ASIGNRECORD END PYC DPTOS TIPO NOMBRE
%start registros
%%
(1) registros    : registros registro
(2)              | ;
(3) registro     : TYPE NOMBRE ASIGNRECORD lista_campos END PYC ;
(4) lista_campos : lista_campos campo
(5)              | campo ;
(5) campo        : NOMBRE DPTOS tipo_nombre PYC ;
(6) tipo_nombre  : TIPO
(7)              | NOMBRE ;
%%
```

Considerando $[registros' \rightarrow registros]$ como la producción de la gramática aumentada, calculamos el estado I_0 del AFD que modela los prefijos viables para análisis LR(1) de la siguiente forma:

$$\begin{aligned} I_0 = \{ \quad & [registros' \rightarrow \cdot registros,\ \$], \\ & [registros \rightarrow \cdot registros\ registro,\ \$/\texttt{TYPE}], \\ & [registros \rightarrow \cdot,\ \$/\texttt{TYPE}] \ \} \end{aligned}$$

Las transiciones desde el estado I_0 sería tras realizar la reducción del símbolo no terminal `registros`:

$$\begin{aligned} I_1 = goto(I_0, registros) = \{ \quad & [registros' \rightarrow registros\cdot,\ \$], \\ & [registros \rightarrow registros \cdot registro,\ \$/\texttt{TYPE}], \\ & [registro \rightarrow \cdot\texttt{TYPE NOMBRE ASIGNRECORD lista\_campos END PYC},\ \$/\texttt{TYPE}] \ \} \end{aligned}$$

La primera fila de la tabla de análisis LR(1) quedaría así:

Est.	TYPE	ASIGNRECORD	END	PYC	DPTOS	TIPO	NOMBRE	$	registros	registro	lista_campos	campo	tipo_nombre
0	r2							r2	1				

Como puede apreciarse, no existen conflictos en dicha fila, según se solicita en el ejercicio.

Resultados del Examen de Prácticas de Procesadores de Lenguajes

(Convocatoria Ordinaria) celebrado el 30 de Junio de 2006.

1. Dado el lenguaje asignado para la realización de las prácticas, se va a añadir funcionalidad en dos aspectos:

(a) ***División entera*: Consiste en la división de dos expresiones de tipo entero por medio del operador binario // devolviendo un valor entero (ejemplo: $a - b//c + d$, donde todas las variables son de tipo entero). La prioridad del operador división entera es igual que la división.**

Solución:

Análisis Léxico

Dado que posee la misma misión sintáctica y precedencia que la división, se agrupa con dicho operador. Sea `OP_MULDIV` el token que identifica aquellos lexemas que son operadores binarios exclusivamente y que poseen la misma precedencia (normalmente se agrupa con el de multiplicación y otros que tengan asignada la misma precedencia).

Según lo dicho, sería necesario añadir las siguientes líneas al fuente LEX

```
...
%%
.....

 "/"    { yylval.atrib= ATR_DIV_FLO ; return OP_MULDIV ; }
 "//"   { yylval.atrib= ATR_DIV_ENT ; return OP_MULDIV ; }

.....
%%
...
```

donde `ATR_DIV_FLO` y `ATR_DIV_ENT` son dos constantes enteras de distinto valor que sirven para dar valor al atributo del token `OP_MULDIV` e `yylval` es la variable de LEX/YACC de tipo `YYSTYPE`, definida con la misma estructura que la práctica de análisis semántico. En dicha variable existe un campo denominado "`atrib`"que es de tipo entero que almacena valores referentes al atributo del token (de cara a distinguir los lexemas en el análisis semántico).

Análisis Sintáctico

Para contemplar expresiones de división entera ya existe la regla que así lo expresa en sintaxis Yacc:

```
...
%%
...
Expresion : ...
          | Expresion OP_MULDIV Expresion
          | ...
;
...
%%
...
```

Por lo tanto, no habría que añadir ningún tipo de regla sintáctica para contemplar este tipo de expresiones.

Análisis Semántico

Dado que ya existe la regla sintáctica que puede representar expresiones binarias, incluyendo las de división entera, habría que añadir la comprobación semántica de que ambas expresiones de los operandos son de tipo entero. En sintaxis de Yacc quedaría así:

```
...
%%
...
Expresion : ...
          | Expresion OP_MULDIV Expresion
              {
                  ....
                  if ($2.atrib == ATR_DIV_ENT)     /* Lexema "//" */
                  {
                     if ($1.tipo==TIPO_ENTERO && $3.tipo==TIPO_ENTERO)
                        $$.tipo=TIPO_ENTERO ;
                     else
                     {
                        printf ("Error semántico:
                                 Tipos incompatibles en división entera\n");
                        $$.tipo= incorrecto ;
                     }
                  }
                  ....
              }
          | ...
;
...
%%
...
```

aquí, `TIPO_ENTERO` es una constante entera que codifica el tipo entero.

(b) ***Operador condicional ternario*: Consiste en la evaluación de una expresión de tipo lógico y dependiendo del resultado devolverá una expresión u otra (ejemplo: $a < b?c+10 : d-10$, donde las expresiones $c+10$ y $d-10$ deben ser del mismo tipo básico y $a < b$ es la expresión de tipo lógico). La prioridad del operador ternario condicional es la mínima del resto de operadores.**

Solución:

Análisis Léxico

Son necesarios dos nuevos tokens `OPINTERR` (?) y `DOSPUNTOS` (:) (los alumnos que tengan como lenguaje base Pascal ya poseen este último token). El papel sintáctico de estos tokens es diferente del papel de los tokens correspondientes a cualquier otro operador. Por tanto, los nuevos tokens no se pueden agrupar con los ya existentes para otros operadores.

La inclusión de estos nuevos tokens puede mostrarse en la siguiente sintaxis Lex:

```
...
%%
 ....
 "?"   { return OPINTERR ; }
 ":"   { return DOSPUNTOS ; }
 ....
%%
...
```

Análisis Sintáctico

Habría que añadir una nueva regla para el nuevo tipo de expresiones. La declaración de los token debe hacerse de forma que nos aseguremos que se asigna a este operador menor precedencia que a los demás, así que deberíamos de escribirlos antes que la declaración de los tokens correspondientes a otros operadores de expresiones. En YACC, junto a la nueva regla sintáctica que surge, se escribiría así:

```
%left OPINTERR
%left DOSPUNTOS
// resto de tokens de operadores de expresiones
........
%%
...
Expresion : ....
          | Expresion INTERR Expresion DOSPUNTOS Expresion
          | ....
;
...
%%
...
```

Análisis Semántico

A partir de la nueva regla gramatical introducida, debemos añadir aquellas comprobaciones semánticas necesarias, es decir, que la primera expresión sea de tipo lógico y que las otras dos sean del mismo tipo. Si todo es correcto, se sintetiza el tipo de cualquiera de las dos últimas expresiones.

En sintaxis de Yacc quedaría así:

```
%%
...
Expresion : ...
          | Expresion INTERR Expresion DOSPUNTOS Expresion
             {
                 if ($1.tipo==TIPO_LOGICO)
                 {
                    if ($3.tipo==$5.tipo && EsTipoBasico($3.tipo) )
                       $$.tipo= $3.tipo ;
                    else
                    {
                       $$.tipo= incorrecto ;
                       printf ("Error semántico en el operador condicional:
                                tipos diferentes o no básicos tras '?'\n");
                    }
                 }
                 else
                 {
                    $$.tipo= incorrecto ;
                    printf ("Error semántico en el operador condicional:
                             se esperaba una expresión lógica antes
                             de '?'\n");
                 }
             }
          | ...

...
%%
...
```

Donde:

- `TIPO_LOGICO` es una constante entera que codifica el tipo lógico.
- `EsTipoBasico(tipo)` es una función lógica que devuelve `true` si el tipo (que es un entero que se pasa como parámetro) es un entero, flotante o carácter, y `false` en caso contrario (arrays, listas, etc.) (no es necesario implementar esta función).

PROCESADORES DE LENGUAJES
4º Ingeniería Informática
Convocatoria Extraordinaria de Septiembre
1-Septiembre-2006

DNI:		Apellidos y Nombre:		Grupo:	
Profesor de Prácticas:					

Observaciones: Este examen es obligatorio para aquellos que ya han presentado y aprobado las prácticas de laboratorio en este curso 2005/2006 (se consideran superadas las prácticas de laboratorio con una calificación igual o superior a 1.5 puntos). La calificación total del apartado de prácticas se considera un factor entre 0 y 1 a multiplicar por la calificación total obtenida en las prácticas de laboratorio.

PRÁCTICAS

1. Dado el lenguaje asignado para la realización de las prácticas, se va a añadir como tipo básico el **número complejo** con las operaciones de **suma, resta, multiplicación** y **división** entre complejos (con la misma precedencia que los operadores empleados para expresiones de tipo entero y real), los operadores unarios **mas** y **menos** (con la misma precedencia que los unarios ya existentes) y operadores unarios para la extracción de la **parte real** y la **parte imaginaria** (ambos con la máxima prioridad que el resto de los operadores que existan).

   Determinar todos los cambios necesarios en el traductor a nivel de:

   (a) **Análisis Léxico:** Determinar qué lexemas y/o tokens nuevos surgen, en su caso, y especificarlos en sintaxis de LEX.

   (b) **Análisis Sintáctico:** Añadir reglas gramaticales para permitir esta nueva funcionalidad, en su caso, y especificarlas usando la sintaxis de YACC.

   (c) **Análisis Semántico:** Añadir las acciones semánticas correspondientes a estas reglas y especificarlas usando la sintaxis de YACC.

   Solución:

   Análisis Léxico

   En primer lugar aparece un nuevo tipo de dato dentro de los cuatro básicos que denominaremos `complejo` para aquellos que tengan palabras reservadas en castellano y `complex` para los que no. No se crea un nuevo token sino que se añade un nuevo lexema al token `TIPO`. El valor del atributo lo define la constante `ATR_TIPO_COMPL`, que se inicializará al siguiente valor de los atributos posibles de un tipo.

   Por otro lado, los operadores que permiten definir el álgebra de tratamiento de este tipo de dato serán sobrecargados a excepción de los unarios que extraen la parte real y la parte imaginaria. Los operadores sobrecargados serán los de multiplicación, división, suma y resta entre dos datos de tipo complejo y la suma y resta unaria. Aparecerán dos nuevos operadores que definimos así:

   - ($) Operador unario que devuelve la parte real de un número complejo.
   - (#) Operador unario que devuelve la parte imaginaria de un número complejo.

   Al tener estos operadores unario otorgada la mayor de las precedencias, no sería posible agruparlos dentro del token de operadores unarios ya que éstos tendrían menos precedencia que los nuevos. Por lo tanto, al tener la misma misión sintáctica, asociatividad y precedencia, creamos el nuevo token denominado `OP_COMPL_UNA`

   La descripción en sintaxis de LEX quedaría así:

```
.....
%%
.....
"complex"     { yylval.atrib= ATR_TIPO_COMPL ; return TIPO ; }
.....
"$"           { yylval.atrib= ATR_REAL_COMPL ; return OP_COMPL_UNA ; }
"#"           { yylval.atrib= ATR_IMAG_COMPL ; return OP_COMPL_UNA ; }
%%
.....
```

   donde `ATR_REAL_COMPL` y `ATR_IMAG_COMPL` son dos constantes enteras de distinto valor que sirven para dar valor al atributo del token `OP_COMPL_UNA` e `yylval` es la variable de LEX/YACC de tipo `YYSTYPE`, definida con la misma estructura que la práctica de análisis semántico. En dicha variable existe un campo denominado "`atrib`"que es de tipo entero que almacena valores referentes al atributo del token (de cara a distinguir los lexemas en el análisis semántico).

   Obviamente, es posible definir estos nuevos operadores con otros caracteres, en el caso de que éstos hubiesen sido definidos para otro propósito.

Análisis Sintáctico

Se debe definir la asociatividad y precedencia del nuevo token `OP_COMPL_UNA` que debe ser a la derecha y situado al final de todos para otorgarle la mayor precedencia.

En cuanto a producciones, la única regla gramatical que se debe añadir sería la que permite la extracción de la parte real y la parte imaginaria de una expresión formada por un tipo complejo. En sintaxis de YACC sería así:

```
.....
%left  ....
.....
%right OP_UNA
%right OP_COMPL_UNA

%%
.....
Expresion : ...
          | OP_COMPL_UNA Expresion
          | ...
;
.....
%%
.....
```

Análisis Semántico

Dado que ya existen las reglas sintácticas que operan con expresiones unarias y binarias para los operadores aritméticos binarios y unarios, a nivel semántico habría que considerar el tratamiento cuando las expresiones sean de tipo complejo. En cuanto a la regla sintáctica que aparece nueva, también tendríamos que realizar la pertinente consulta para determinar si el operador correspondiente tiene asociada una expresión de tipo complejo.

En sintaxis de YACC sería así:

```
.....
%%
.....
expresion : .....
          | expresion OP_MUL_DIV_BIN expresion
              {
                 .....
                 // Tipo Complejo en expresión
                 if ($1.tipo==ATR_TIPO_COMPL && $1.tipo==$3.tipo`
                    $$.tipo= $1.tipo ;
                 else
                 {
                    $$.tipo= incorrecto ;
                    printf ("Error semántico:
                             tipos incompatibles en dato complejo\n");
                 }
                 .....
              }
          | expresion OP_SUM_RES_UBI expresion
              {
                 .....
                 // Tipo complejo en expresión
                 if ($1.tipo==ATR_TIPO_COMPL && $1.tipo==$3.tipo)
                    $$.tipo= $1.tipo ;
                 else
                 {
                    $$.tipo= incorrecto ;
                    printf ("Error semántico:
                             tipos incompatibles en dato complejo\n");
```

```
            }
            .....
         }
      | OP_SUM_RES_UBI expresion %prec OP_UNA
         {
            .....
            // Tipo complejo en expresión
            if ($2.tipo==ATR_TIPO_COMPL)
               $$.tipo= complejo ;     // Se sintetiza un dato complejo.
            .....
         }
      | OP_COMPL_UNA expresion
         {
            if ($2.tipo=ATR_TIPO_COMPL)
               $$.tipo= real ;        // Se sintetiza un dato de tipo float.
            else
            {
               $$.tipo= incorrecto ;
               printf ("Error semántico:
                        tipo incorrecto en dato complejo\n");
            }

      | .....
%%
.....
```

7. [3.4 puntos] Dada una empresa que dispone de una serie de teléfonos móviles asignados a sus trabajadores, se pretende realizar un control tanto del consumo mensual como del número de llamadas que se hacen por cada línea (con independencia de la duración de la llamada). Con ese propósito, la compañía de teléfono facilita cada mes a la empresa un documento en la que se detallan todas y cada una de las llamadas realizadas de cada móvil. Un ejemplo del detalle para un teléfono móvil es el siguiente:

```
Total Factura, 185.10, 639121190, Plan tarde

2007-04-04,13:59,651345045,Promoción,Alphatel,4:53,Normal,,0.1200
2007-04-04,17:04,678686082,Nacional,Móvil,2:43,Super R.,,0.4760
2007-04-04,18:42,686456441,Promoción,Alphatel,1:00,Normal,,0.1200
2007-04-05,18:05,678626034,Nacional,Móvil,4:29,Super R.,,0.6880
2007-04-05,19:44,622441670,Promoción,Alphatel,1:17,Normal,,0.1200
2007-04-05,21:03,633686089,Nacional,Móvil,0:37,Super R.,,0.2240
2007-04-06,11:40,666377047,Mensaje Corto,Alphatel,0:00,Normal,,0.1500
2007-04-07,11:43,699758805,Promoción,Alphatel,1:00,Normal,,0.1200
2007-04-30,21:21,958280654,Provincial,Granada,1:19,Super R.,,0.2290
2007-04-02,14:59,617222225,Núm. Alphatel,Alphatel,2:14,Normal,,0.7977
.....................................................
```

Como se puede apreciar, aparece una cabecera y luego el detalle de las llamadas realizadas. Los campos de la cabecera la componen un texto indicativo, el consumo total y posteriormente aparece el número de teléfono móvil y el plan asignado

El detalle de las llamadas aparece justo debajo con los siguientes campos separados por comas:

- **Fecha:** formato numérico `AAAA-MM-DD`.
- **Hora:** formato numérico `HH:MM`.
- **Nº Receptor:** 9 cifras del número de teléfono al que se llama.
- **Tipo:** uno de los siguientes: `Provincial`, `Nacional`, `Num. Alphatel`, `Mensaje corto` y `Promocion`.
- **Destino:** uno de los siguientes: `Alphatel`, `Móvil` y `Provincia` (nombre de la provincia destinataria).
- **Tarifa:** uno de los siguientes: `Normal` y `Super R.`

Se desean realizar las siguientes comprobaciones:

- Determinar si el consumo total/mes de un teléfono supera la cantidad de **150 euros**.
- En base al horario en el que realiza las llamadas, determinar el plan de ahorro más apropiado en cada móvil. Si existe un balance favorable de llamadas en horario de mañana, de tarde o no es significante la diferencia. Según lo anterior, existen tres planes de ahorro disponibles en la empresa de móviles:
  - **Plan mañana:** precio reducido en llamadas realizadas en la franja que va desde las 08:00h hasta las 13:00h.
  - **Plan tarde:** precio reducido en llamadas realizadas en la franja que va desde las 17:00h hasta las 22:00h.
  - **Plan constante:** precio único en todas las llamadas con independencia del horario.

En base a la descripción anterior y los requisitos expuestos:

(a) [1.4 puntos] Definir el lenguaje del documento anterior mediante una gramática con las producciones en forma gramatical de tal forma que si existe algún error en la lectura de los campos que no son determinantes, es decir, en todos excepto en `Total Factura` correspondiente a la cabecera y en `Hora` del detalle de las llamadas, se pueda recuperar y continuar analizando.

Solución:

En primer lugar definiremos el lenguaje sin las producciones de error. De esta forma, vamos a considerar el lenguaje sin ningún tipo de abstracción, si bien, es posible definirlo en base a la abstracción solicitada en el siguiente apartado.

El lenguaje quedaría así:

moviles	→	moviles movil
	|	movil
movil	→	cabecera llamadas
cabecera	→	totalfactura "," numero "," plan
totalfactura	→	**"Total Factura,"** precio
numero	→	numero num
	|	num
num	→	**0 | 1 | ⋯ | 9**
plan	→	**"Plan mañana" | "Plan tarde" | "Plan constante"**
precio	→	numero "." numero
llamadas	→	llamadas llamada
	|	ε
llamada	→	fecha "," hora "," numero "," tipo "," destino "," hora "," horario "„" precio
fecha	→	numero **"-"** numero **"-"** numero
hora	→	numero ":" numero
tipo	→	**"Promocion" | "Nacional" | "Mensaje Corto" | "Provincial" | "Num. Alphatel"**
destino	→	**"Alphatel" | "Móvil" | "Álava"** | ⋯ | **"Zaragoza"**
horario	→	**"Normal" | "Super R."**

A continuación, añadimos las producciones de error en aquellos lugares en los que es necesario recuperarse. Según el problema, en cualquiera de los campos anteriormente especificados excepto en `precio` de la producción que define al símbolo no terminal `totalfactura` y en la primera aparición de `hora` de la producción `llamada`.

Con esas premisas, podemos permitir un error sintáctico en el campo de texto del total de la factura pero no en el precio final, es decir, quedaría la siguiente producción:

totalfactura	→	texto precio
texto	→	**"Total Factura,"**
	|	**error**

También podemos permitir errores sintácticos en los detalles de una llamada a excepción de la hora en la que se efectúa la llamada. De esta forma, la descripción gramatical quedaría así:

llamada	→	fecha hora restollamada
fecha	→	numero **"-"** numero **"-"** numero
	|	**error**
restollamada	→	"," numero "," tipo "," destino "," hora "," horario "„" precio
	|	**error**

(b) 1.0 puntos **Definir los tokens con el máximo nivel de abstracción para contemplar las comprobaciones anteriormente mencionadas y obtener la gramática abstracta correspondiente.**

Solución:

El problema nos dice que la auditoría en las llamadas presenta dos requisitos fundamentales. El primero es observar el precio total gastado en el mes y en segunda instancia la determinación del plan de llamadas que sea más apropiado. Para lo primero debemos observar el campo PRECIO y para lo segundo debemos prestar atención, dentro del detalle de la llamada, a la hora a la cual se realiza la misma. El resto de la información no es necesaria para las consideraciones semánticas, en cuyo caso, todo lo que se pueda unir formando gramática regular se podrá realizar sin ningún tipo de problema.

En estas circunstancias, la tabla de tokens quedaría así:

Nº	Token	Patrón
1	TOTALFACT	`"Total Factura,"`
2	PLAN	`"Plan mañana" | "Plan tarde" | "Plan constante"`
		`atrib=0, atrib=1, atrib=2`
3	PRECIO	`{dig}+"."{dig}+`
4	HORA	`{dig}+":"{dig}+`
5	FECHA	`{dig}+"-"{dig}+"-"{dig}+`
6	RESTOLLAM	`{restollam}`
7	NUMMOVIL	`{dig}+`

Para una mayor comprensión, se muestra la descripción de esta tabla de tokens según la sintaxis de LEX/FLEX:

```
%{
char *numMovil;   // variable que almacena el número del móvil.
int   consumoMes; // variable que almacena el consumo.

extern yylval;
%}

%option noyywrap

dig       [0-9]+
nhora     [0-9]{2,2}
nyear     [0-9]{4,4}
nummovil  [0-9]{9,9}
precio    {dig}"."{dig}
hora      {nhora}":"{nhora}
fecha     {nyear}"-"{nhora}"-"{nhora}
tipo      ("Nacional"|"Promocion"|"Mensaje Corto"|"Provincial"|"Num. Alphatel")
destino   ("Movil"|"Alphatel"|"Granada")
horario   ("Normal"|"Super R.")
restollam {nummovil}","{tipo}","{destino}","{hora}","{horario}",,"{precio}

%%
[ \t\n]+                        ;
"Total Factura,"                { return TOTALFACT ; }
"Plan mañana"                   { yylval= 0; return PLAN ; }
"Plan tarde"                    { yylval= 1; return PLAN ; }
"Plan constante"                { yylval= 2; return PLAN ; }
{precio}                        { consumoMes= atoi(yytext); return PRECIO ; }
{hora}                          { return HORA ; }
{fecha}                         { return FECHA ; }
{restollam}                     { return RESTOLLAM ; }
{nummovil}                      { numMovil= strdup (yytext); return NUMMOVIL ; }
.                               { printf("[Linea %d] Error en %s\n", yylineno, yytext); }
%%
```

La gramática abstracta resultante, según la sintaxis de YACC y sin considerar las producciones de error quedaría así:

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

%token TOTALFACT PLAN PRECIO HORA FECHA RESTOLLAM NUMMOVIL
%%

moviles       : moviles movil
              | movil ;
movil         : cabecera llamadas ;
cabecera      : totalfactura PLAN ;
totalfactura  : texto PRECIO ;
texto         : TOTALFACT NUMMOVIL
llamadas      : llamadas llamada
              | ;
llamada       : FECHA HORA RESTOLLAM ;

%%
#include "lex.yy.c"

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

(c) [1.0 puntos] Definir la gramática con atributos en sintaxis de YACC a partir de la gramática anterior para que contemple las comprobaciones que se exigen en el problema.

Solución:

Aunque se puede plantear una solución utilizando una tabla de almacenamiento temporal (TS), también es posible una solución que no considere dicha estructura dada la forma en la que se representan las llamadas para cada teléfono móvil. En este caso, la solución sería más simple dado que con el uso de variables que representen el total de llamadas en cada franja horaria se podría llevar dicha contabilidad. Dichas variables se tendrían que reiniciar al aparecer el detalle de facturación de

un nuevo número de teléfono.
Vamos a suponer que existe una función que analice la hora en la que se ha realizado la llamada y nos determine si pertenece al plan de mañana o al plan de tarde:

- `int PlanDeLlamada (char *hora)`: Devuelve `0` si se ha realizado en horario del plan de mañana y `1` si fue en horario de plan de tarde.

Con toda esta información, tomando la gramática anterior sin las producciones de error, la gramática con atributos quedaría así:

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

#define MAXCONSUMO 150 // Máximo consumo que se tolera.
#define PLANMAN    0   // Plan de mañana.
#define PLANTAR    1   // Plan de tarde.

int llamDia ;          // Número de llamadas en horario de mañana.
int llamTarde ;        // Número de llamadas en horario de tarde.
int planInicial ;      // Plan de llamadas que tiene asignado.
%}
%token TOTALFACT PLAN PRECIO HORA FECHA RESTOLLAM NUMMOVIL
%%
moviles       : moviles movil
              | movil ;
movil         : cabecera { llamDia= 0; llamTarde= 0; }
                llamadas
                      {
                            // INFORME DEL NÚMERO DE LLAMADAS REALIZADAS
                            if (planInicial== PLANMAN)
                            {
                               if (llamDia < llamTarde)
                                  printf ("\nNº Movil: %s, se aconseja cambiar
                                            al Plan de Tarde\n", numMovil);
                            }
                            else // plan es de tarde
                            {
                               if (llamDia > llamTarde)
                                  printf ("\nNº Movil: %s, se aconseja cambiar
                                            al Plan de Mañana\n", numMovil);
                            }

                            if (llamDia == llamTarde)
                               printf ("\nNº Movil: %s, se aconseja cambiar
                                        al Plan Constante\n", numMovil);
                      }

cabecera      : totalfactura PLAN          { planInicial= $2; }
totalfactura  : texto PRECIO
texto         : TOTALFACT NUMMOVIL
llamadas      : llamadas llamada
              | ;
llamada       : FECHA HORA RESTOLLAM       { int plan= PlanDeLlamada ($2);
                                             if (plan==PLANMAN) // Horario de mañana.
                                                llamDia++;
                                             else if (plan=PLANTAR)
                                                llamTarde++;
                                             }
%%
```