Universidad de Málaga
Departamento de Lenguajes y
Ciencias de la Computación
Campus de Teatinos, 29071 MÁLAGA

Examen de Traductores, Intérpretes y Compiladores.
Convocatoria ordinaria de Junio de 2006
3er Curso de I.T. Informática de Sistemas.

Apellidos, Nombre: ________________________

Calificación: ____________

TEÓRICO

1.- Dado el programa Lex:

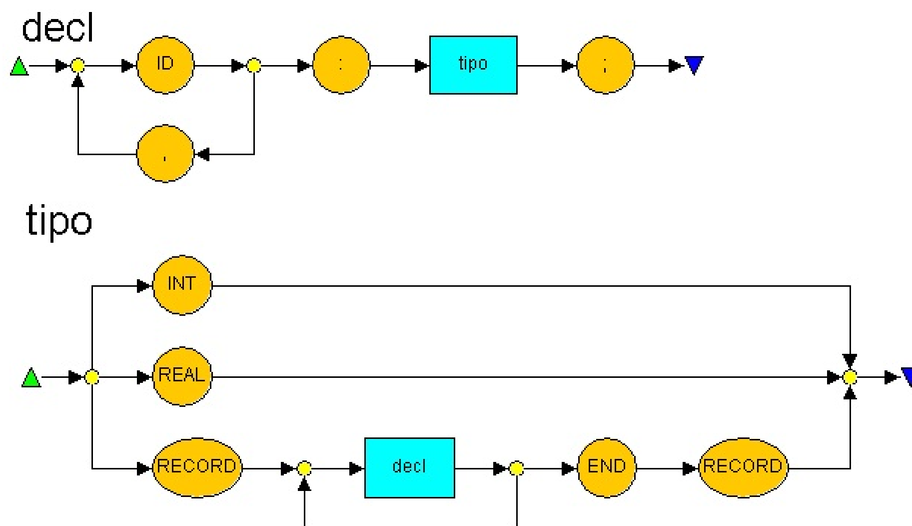
```
%start S
%%
^[ \t\n]*"*"        { BEGIN S; }
<S>(.|\n)+   { BEGIN 0; }
[a-z]+              { ; }
.            { ; }
\n           {; }
```

en cuántos *tokens* se divide la entrada:

```
\t\t\n\t* Hola \nejemplo@
```

¿cuáles son estos *tokens* y por qué?

2.- Dados los diagramas de sintaxis siguientes



se pide codificar en C las funciones recursivas que implementan su análisis sintáctico descendente.

3.- Dada la gramática:

```
❶ decl    :  ID ':' tipo ';'
❷         |  ID ',' decl
          ;
❸ tipo    :  INT
❹         |  REAL
❺         |  RECORD listaDecl END RECORD
          ;
❻ listaDecl     :  decl
❼               |  listaDecl decl
                ;
```

se pide suministrar el *parse* derecho que reconoce la sentencia:

```
ID , ID , ID : RECORD ID : INT ; ID , ID : REAL ; END RECORD ;
```

4.- Sea la gramática:

```
sent    :   FOR '(' listaExpr ';' cond ';' listaExpr ')'
        ;
listaExpr    :      expr listaExprP
           ;
listaExprP   :
             |      listaExpr ',' expr
             ;
expr    :   ID ASIG NUM
        |   NUM
        ;
cond    :   expr condP
        ;
condP   :   MaI expr
        |   MeI expr
        |   Igu expr
        ;
```

cuya tabla LL(1) es (está dividida en dos partes porque no cabe):

	ID	ASIG	NUM	MaI	MeI	Igu
expr	expr --> ID ASIG NUM		expr --> NUM			
cond	cond --> expr condP		cond --> expr condP			
condP				condP --> MaI expr	condP --> MeI expr	condP --> Igu expr
sent						
listaExpr	listaExpr --> expr listaExprP		listaExpr --> expr listaExprP			
listaExprP	listaExprP --> listaExpr , expr		listaExprP --> listaExpr , expr			

	FOR	(	;	)	,	$
expr						
cond						
condP						
sent	sent --> FOR (listaExpr ; cond ; listaExpr)					
listaExpr						
listaExprP			listaExprP --> EPS	listaExprP --> EPS	listaExprP --> EPS	

Se pide, mediante un análisis LL(1) con esta tabla, aceptar o rechazar la cadena:

FOR (ID ASIG NUM , NUM ; NUM MaI ID ; ID , ID) $

Universidad de Málaga
Departamento de Lenguajes y Ciencias de la Computación
Campus de Teatinos, 29071 MÁLAGA

Examen de Traductores, Intérpretes y Compiladores.
Convocatoria extraordinaria de Junio de 2006
3er Curso de I.T. Informática de Sistemas.

Apellidos, Nombre: ________________________

Calificación: ____________

PRÁCTICO

Las primeras versiones del lenguaje BASIC (Beginners' All-purpose Set Instruction Code) estaban muy restringidas en cuanto al uso de funciones y procedimientos. Las únicas funciones que existían no se parecían en nada a las que conocemos hoy día, sino que eran forzosamente funciones numéricas que producían un único valor también numérico como resultado de su evaluación. Por tanto, se parecían más al concepto de función matemática que al de función informática que conocemos hoy día.

Una función de éstas se define de la forma:
DEF FN nombreFuncion(parametros) = expresión;
donde:

- **nombreFuncion** es un identificador.
- **parametros** es una lista de identificadores separados por comas. Al menos debe haber un identificador.
- **expresion** es una expresión aritmética en la que sólo pueden intervenir números y los identificadores declarados como parámetros en la cabecera de la función.

Ejemplos de funciones válidas son:
DEF FN suma(x, y) = x+y;
DEF FN polinomio(x) = x*x*5+x*2+7;
DEF FN derivadaPolinomio(te) = 2*te*5+2;
DEF FN variosParam(x, y, z, u) = x+y*(z-u);

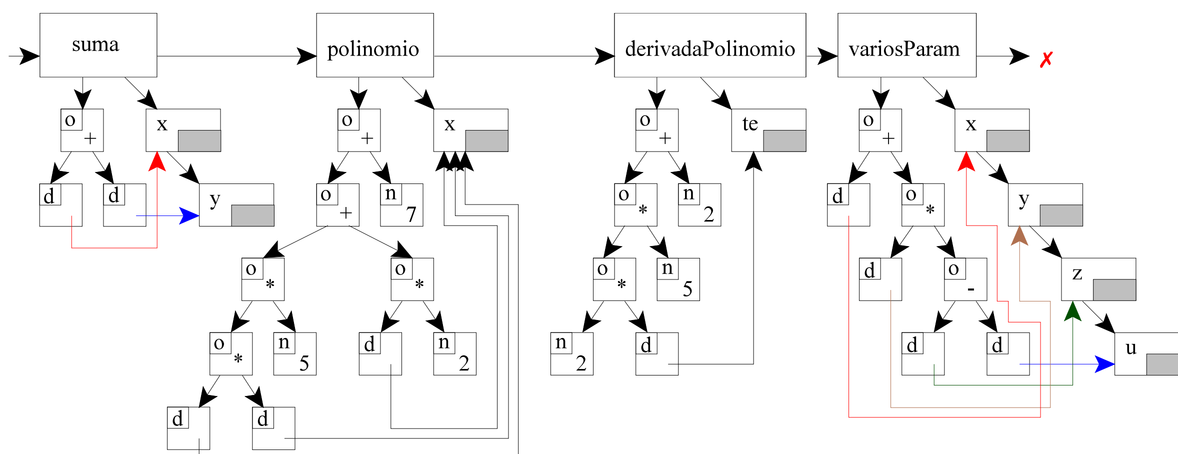
Por supuesto, la idea es poder utilizar con posterioridad estas funciones en cualquier lugar del código donde se espere una expresión. Por ejemplo:

a := 7+suma(3, 8);	asigna a la variable **a** el valor 18
b := polinomio(2);	asigna a la variable **b** el valor 31
c := derivadaPolinomio(a);	asigna a la variable **c** el valor 182
d := variosParam(2+a, 3, c+2*9, 100) - a*10;	asigna a la variable **d** el valor 140

NUESTRO OBJETIVO ES GENERAR CÓDIGO PARA ESTAS ÚLTIMAS EXPRESIONES Y ASIGNACIONES.

Para ello, seguiremos los siguientes pasos:

- Cualquier expresión en la que no intervenga una función será transformada en código de tercetos y tal y como se ha visto en los ejercicios de clase.
- Cuando encontremos la definición de una función hemos de introducirla en la tabla de símbolos, junto con la lista de sus nombres de parámetros y la estructura de la expresión a que equivale. En nuestro caso, la tabla de símbolos quedaría como se indica en la figura.



- La estructura de la expresión está formada por nodos de diferentes tipos según su contenido:
  - Nodos n: contiene números constantes enteros.
  - Nodos o: contienen operadores aritméticos y dos punteros a sus operadores.
  - Nodos d: contienen identificadores. Realmente como los únicos identificadores que pueden usarse en una función son sus parámetros, lo que contienen es un puntero al parámetro referido.
- No puede haber dos funciones con el mismo nombre, ni puede redefinirse una función ya creada.
- En la definición de una función nopuede haber identificadores que no sena parámetros.
- No es necesario comprobar que todos los parámetros de una función aparezcan en la expresión que define a ésta.
- Al llamar a una función hay que comprobar que el número de parámetros formales y reales sea igual.

Una vez creada la estructura de una función, puede pasarse a utilizarla. Cuando se utiliza una función como expresión o como parte de ella, debe generarse todo el código de la función, pero sustituyendo cada parámetro formal por uno real. Así, la generación de código de la utilización de una función se divide en dos partes:

1ª: Se genera el código correspondiente a las expresiones que hacen de parámetros reales.

2ª: Se genera todo el código correspondiente a la función recorriendo su árbol de expresión y sustituyendo cada parámetro formal por el real.

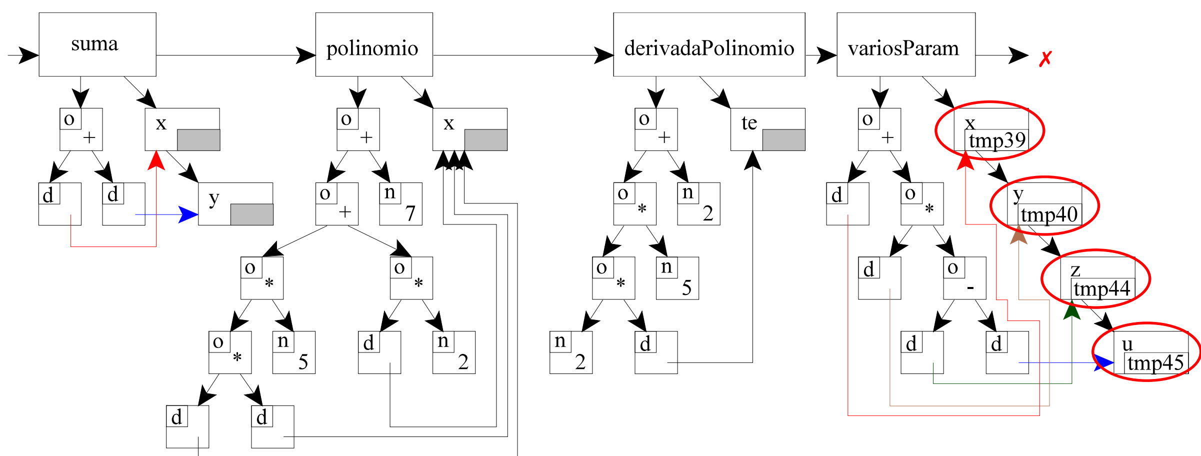
Por ejemplo, cuando nos encontramos la asignación

d := variosParam(2+a, 3, c+2*9, 100) - a*10;

lo primero que se genera es el código de los parámetros reales, y el representante de cada parámetro real se carga en el nodo del parámetro formal. Así se obtiene el siguiente código de tercetos

```
tmp38 = 2;
tmp39 = tmp38 + a;
tmp40 = 3;
tmp41 = 2;
tmp42 = 9;
tmp43 = tmp41 * tmp42;
tmp44 = c + tmp43;
tmp45 = 100;
```

y la siguiente modificación en la tabla de símbolos:



De esta forma, el segundo paso consiste en recorrer el árbol de la función **variosParam** y generar el código siguiente siguiendo los punteros:

```
tmp48=tmp44 - tmp45
tmp47=tmp40 * tmp48
tmp46=tmp39 + tmp47
tmp49 = 10;
tmp50 = a * tmp49;
tmp51 = tmp46 - tmp50;
d = tmp51
```

Por supuesto, cada vez que se utilice una función, se machacarán los valores que hubiese junto a los parámetros formales con los de la nueva invocación real, pudiendo reutilizarse el árbol de una función cuantas veces sea necesario.

Para todo esto se dispone de una tabla de símbolos que es la siguiente:

Fichero tsjun06.c

```
#include <stdlib.h>
#include <stdio.h>

typedef struct _parametro {
   char nombre[21];
   struct _parametro * sig;
   char aux[21];
} parametro;
typedef struct _nodo {
   char tipo; // 'n'umero, i'd'entificador, 'o'perador
   union {
      int numero;
      struct _parametro * identificador;
      struct {
         char tipo;
         struct _nodo * izq;
         struct _nodo * der;
      } operador;
   } contenido;
} nodo;
typedef struct _funcion {
   char nombre[21];
   struct _funcion * sig;
   struct _parametro * listaParametros;
   struct _nodo * expresion;
} funcion;

funcion * tabla = NULL;

funcion * buscarFuncion(char * nombre){
  funcion * t = tabla;
  while( (t != NULL) && (strcmp(nombre, t->nombre)) )
      t = t->sig;
  return (t);
}

void insertarFuncion(char * nombre){
   if (buscarFuncion(nombre) != NULL)
      printf("La funcion %s ya esta declarada.\n", nombre);
   else {
      funcion * aux = (funcion *)malloc(sizeof(funcion));
      strcpy(aux->nombre, nombre);
      aux->sig = tabla;
      tabla = aux;
      aux->listaParametros = aux->expresion = NULL;
   }
}

void insertarParametro(char * nombreF, char * nombreP){
   funcion * aux = buscarFuncion(nombreF);
   if ( aux == NULL)
      printf("La funcion %s no esta declarada.\n", nombreF);
   else {
      // Mete al final de la lista.
      parametro * nuevo = (parametro *) malloc(sizeof(parametro));
      parametro * * ptr_ptr_param = &(aux->listaParametros);
      while((* ptr_ptr_param) != NULL)
         ptr_ptr_param = &((* ptr_ptr_param)->sig);
      (* ptr_ptr_param) = nuevo;
      strcpy(nuevo->nombre, nombreP);
      nuevo->sig = NULL;
   }
}

parametro * buscarParametro(char * nombreF, char * nombreP){
   funcion * aux = buscarFuncion(nombreF);
   if ( aux == NULL)
      printf("La funcion %s no esta declarada.\n", nombreF);
   else {
      parametro * p = aux->listaParametros;
```

```
      while( (p != NULL) && (strcmp(nombreP, p->nombre)) )
         p = p->sig;
      return (p);
   }
   return NULL;
}

void asociarParametro(char * nombreF, int numP, char * paramReal){
   funcion * aux = buscarFuncion(nombreF);
   if ( aux == NULL)
      printf("La funcion %s no esta declarada.\n", nombreF);
   else {
      parametro * p = aux->listaParametros;
      while( (p != NULL) && (numP-- > 1) )
         p = p->sig;
      if (p != NULL)
         strcpy(p->aux, paramReal);

   }
}
int numeroParam(char * nombreF){
   int i = 0;
   funcion * aux = buscarFuncion(nombreF);
   if ( aux == NULL)
      printf("La funcion %s no esta declarada.\n", nombreF);
   else {
      parametro * p = aux->listaParametros;
      while( (p != NULL) ){
         i++;
         p=p->sig;
      }
   }
   return i;
}
```

Se pide:

- Construir el programa YACC que conceda la funcionalidad pedida. Para ello se da un esqueleto en el que también hay que indicar los atributos de los símbolos que los necesiten.
- Incluir la función `char * emitirTercetos(nodo * ptrNodo)` que recorre el árbol de nodos ya relleno y genera los tercetos correspondientes.

Nota: Suponemos que nunca se va a dar el caso de invocaciones recursivas a funciones, de la forma: a := suma(3, suma(2, suma(5, 2))). Pero, si se dieran, ¿funcionaría nuestro programa?¿por qué?¿qué modificaciones habría que hacer en caso de que no funcionase?

Fichero exjun06l.lex

```
%START COMENT

%%
^[ \t]*"*" { BEGIN COMENT; }
<COMENT>.+   { ; }
<COMENT>\n   { BEGIN 0; yylineno ++; }
":="    { return ASIG; }
">="    { return MAI; }
"<="    { return MEI; }
"!="    { return DIF; }
DEF     { return DEF; }
FN      { return FN; }
AND     { return AND; }
OR      { return OR; }
NOT     { return NOT; }

[0-9]+ {
         yylval.numero = atoi(yytext);
         return NUMERO;
       }
[A-Za-z_][A-Za-z0-9_]* {
                          strcpy(yylval.nombre, yytext);
                          return ID;
                        }
```

```
[ \t]+ { ; }
\n     { yylineno++; }
.      { return yytext[0]; }
```

Fichero exjun06y.yac

```
%{
#include "tsjun06.c"

void nueva_var(char * s){
  static actual=0;
    sprintf(s, "tmp%d", ++actual);
}

void nueva_etq(char * s){
  static actual=0;
    sprintf(s, "etq%d", ++actual);
}

char * emitirTercetos(nodo * ptrNodo){
   // No hace falta liberar la memoria.
   char * retorno=(char *)malloc(21);;















   return retorno;
}

char funcionActual[21];

%}

%union {
   int numero;
   char nombre[21];
   nodo * ptr_nodo;
   struct {
      char  nombre[21];
      int   numParamActual;
   } datos_fun;
}










%left '+' '-'
%left '*' '/'

%%
prog  :  prog sent ';'
   |  prog error ';' { yyerrok; }
   |
   ;
sent  :  ID ASIG expr   {
               printf("\t%s = %s;\n", $1, $3);
            }
   |  DEF FN ID
         {
```

```
          }
       '(' lista_param ')'
       '=' expr_restringida
          {

          }
   ;
lista_param :  ID
          {

          }
       |  lista_param ',' ID
          {

          }
       ;
expr   :  NUMERO      {
              nueva_var($$);
              printf("\t%s = %d;\n", $$, $1);
            }
   |  ID    {
              strcpy($$, $1);
            }
   |  llamadaAFuncion ')'
            {




            }
   |  expr '+' expr   {
              nueva_var($$);
              printf("\t%s = %s + %s;\n", $$, $1, $3);
            }
   |  expr '-' expr   {
              nueva_var($$);
              printf("\t%s = %s - %s;\n", $$, $1, $3);
            }
   |  expr '*' expr   {
              nueva_var($$);
              printf("\t%s = %s * %s;\n", $$, $1, $3);
            }
   |  expr '/' expr   {
              nueva_var($$);
              printf("\t%s = %s / %s;\n", $$, $1, $3);
            }
   |  '(' expr ')'    {
              strcpy($$, $2);
            }
   ;
expr_restringida  :  NUMERO   {
              $$ = (nodo *) malloc(sizeof(nodo));
              $$->tipo = 'n';
              $$->contenido.numero = $1;
            }
   |  ID    {









            }
   |  expr_restringida '+' expr_restringida   {
```

```
            }
    |  expr_restringida '-' expr_restringida   {




            }
    |  expr_restringida '*' expr_restringida   {




            }
    |  expr_restringida '/' expr_restringida   {




            }
    |  '(' expr_restringida ')'    {

            }
    ;
llamadaAFuncion   :  ID '(' expr
            {



            }
      |  llamadaAFuncion ',' expr
            {



            }
      ;
%%
#include "errorlib.c"
#include "exjun06l.c"

void main() {
    yylineno = 1;
    yyparse();
}
```