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 Febrero de 2004
3er Curso de I.T. Informática de Sistemas.

Apellidos, Nombre: _____ ***Calificación:*** ___

TEORÍA

1.- Construir una gramática capaz de reconocer llamadas a funciones. No es necesario construir las expresiones regulares que den lugar a los *tokens*. Como parámetros formales de una llamada a función sólo se admiten variable y otras llamadas a función. los parámetros deben ir separados por comas y se admite la existencia de funciones sin parámetros. Los parámetros deben seguir al nombre de la función e ir delimitados por paréntesis.

Nota: Sólo se pide la gramática. no son necesarias las acciones semánticas.

2.- Sea la siguiente gramática LL(1):

```
decl   → ID DOSPNT lTip FLECHA ID
lTip   → ε
       | lnTp
lnTp   → ID lnTp'
lnTp'  → ε
       |  POR ID lnTp'
```

Y sea su tabla LL(1):

T / N	ID	DOSPNT	POR	FLECHA	$
decl	decl → ID DOSPNT lTip FLECHA ID				
lTip	lTip → lnTp			lTip → ε	
lnTp	lnTp → ID lnTp'				
lnTp '			lnTp' → POR ID lnTp'	lnTp' → ε	

Se pide: aplicando el algoritmo LL(1) a la tabla anterior, aceptar o rechazar la cadena:

ID DOSPNT ID POR ID POR ID POR ID FLECHA ID

e indicar el árbol sintáctico que se genera (si se produce error, indicar el máximo árbol sintáctico que se llega a producir).

3.- En PCYacc pueden colocarse acciones semánticas entre los símbolos del consecuente de una reglas de producción, dando lugar a un esquema de traducción. Sin embargo, PCYacc realiza algunas transformaciones sobre este tipo de reglas. Indicar de qué transformación se trata y cómo afecta a la utilización de atributos.

4.- Responder concisamente a las siguientes cuestiones (unas tres líneas por pregunta):

a) ¿Qué tipo de gramáticas son inadmisibles por el análisis descendente recursivo visto en clase?
b) ¿Qué tipo de gramáticas son inadmisibles por el análisis ascendente recursivo visto en clase?
c) ¿Cuáles son las directrices fundamentales de PCLex en el reconocimiento de lexemas?
d) ¿Qué ventajas tiene la generación de código intermedio en lugar de código máquina?
e) ¿Qué es la reubicación de programas en memoria?

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 Febrero de 2004
3er Curso de I.T. Informática de Sistemas.

Apellidos, Nombre: _____ ***Calificación:*** ___

PRÁCTICA

Se desean reconocer especificaciones algebraicas reducidas, mediante el lenguaje LEAR y generar estructuras en memoria para posteriores procesamientos.

Un programa LEAR consta de:

a) Una primera parte de declaración de funciones.
b) Una segunda de declaración de variables.
c) Una tercera de especificación de ecuaciones.

Un ejemplo de programa en LEAR puede ser:

```
FUNCIONES
      suma: Nat * Nat -> Nat;
      cero: -> Nat;
      suc : Nat -> Nat;
      mayI: Nat * Nat -> Log;
      verd: -> Log;
      fals: -> Log;
VAR
      a, b, c : Nat;
      d : Log;
      e : Nat;
      f : Otro;
ECUACIONES
      suma(cero(), a) == a;
      suma(suc(b), a) == suc(suma(b, a));
      mayI(a, cero()) == verd();
      mayI(cero(), suc(a)) == fals();
      mayI(suc(a), suc(b)) == mayI(a, b);
```

Como puede observarse, cada parte del programa debe ir precedida de la correspondiente palabra reservada (**FUNCIONES**, **VAR** y **ECUACIONES**, respectivamente).

Cada declaración de función finaliza en punto y coma y comienza por un identificador seguido de dos puntos, que es el nombre de la función; a continuación una lista de parámetros (identificadores que suponemos que son tipos de datos) separados por asteriscos, los caracteres guión y mayor que (->) y otro identificador (que suponemos que es otro tipo de salida). Como puede observarse en el ejemplo, se admiten funciones sin parámetros.

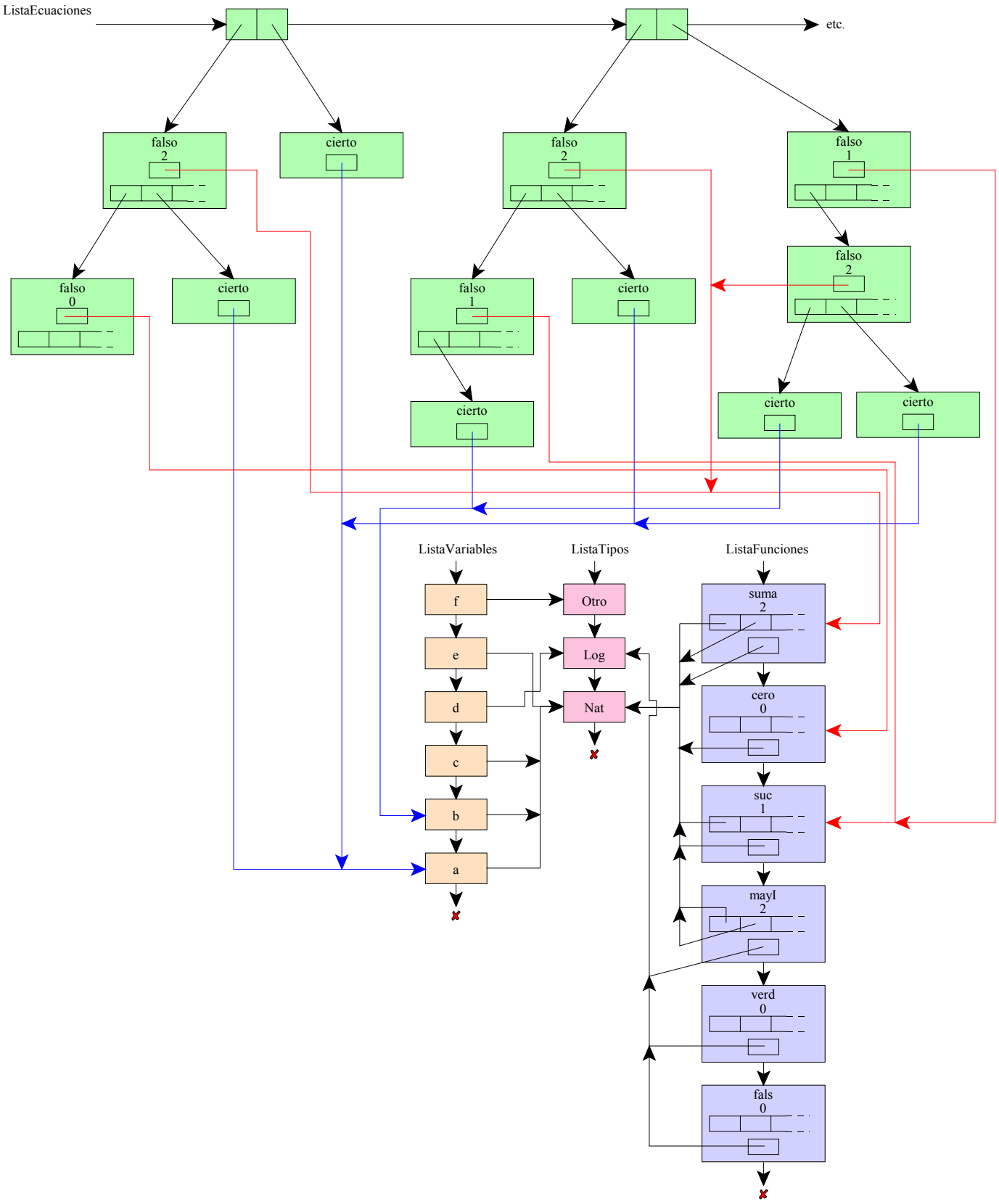
Una declaración de variable consiste en una lista no vacía de identificadores separados por comas, seguida de dos puntos y un nuevo identificador, que suponemos que es el tipo de dichos identificadores o variables. Cada declaración finaliza en punto y coma.

Cada ecuación está formada por dos términos separados por dos símbolos de guión y finaliza en punto y coma. Un término es cualquier expresión valida, entendiéndose por expresión válida:

a) Una variable.
b) Una invocación a una función con el mismo número de parámetros que en su declaración. Cada parámetro debe ser, a su vez, un término.

Por último, los tipos no necesitan declaración, y pueden utilizarse libremente con el nombre que se desee.

El objetivo del ejercicio es generar una estructura en memoria que represente un programa en LEAR. Para ello se tienen cuatro tablas de símbolos: tipos, funciones, variables y ecuaciones. Las definiciones de todas estas tablas de símbolos se suministran al final de este ejercicio. En base a estas estructuras, las tablas de símbolos obtenidas en memoria tras el reconocimiento del programa anterior es el siguiente (se han omitido las tres últimas ecuaciones):



Para reconocer este lenguaje se suministra una gramática cuyo esqueleto también se suministra al final de este enunciado.

Se pide:

a) Indicar el tipo de atributo de cada terminal y no terminal de la gramática.

b) Enumerar los tipos de error semántico que podrían detectarse en este lenguaje. Supondremos que un mismo identificador puede ser a la vez tipo, nombre de función y nombre de variable.

c) Indicar para error semántico enunciado en el apartado anterior, dónde se debe controlar según el esqueleto dado al final del enunciado (no es necesario indicar código, sólo el punto concreto donde habría que controlarlo. Utilizar para ello los números entre círculos que se suministran).

d) Rellenar las acciones semánticas del esqueleto, **suponiendo que no se van a producir errores semánticos de ningún tipo**.

Pistas: `funcionActual` se utiliza en la regla `listaFunciones`.

`principioTermino` propaga los atributos de manera parecida a la regla `inicioCase` vista en clase.

Esqueleto - Código Lex:

```
%%
FUNCIONES   { return FUNCIONES; }
VAR         { return VAR; }
ECUACIONES  { return ECUACIONES; }
[a-zA-Z][a-zA-Z0-9]*    { strcpy(yylval.nombre, yytext); return ID; }
"->"  { return FLECHA; }
"=="        { return EQUIVALE; }
"*"         { return POR; }
[ \t\n]     { ; }
.           { return yytext[0]; }
```

Esqueleto - Código Yacc:

```
%{
#include "TabSimb.c"
simboloFuncion * listaFunciones = NULL;
simboloTipo * listaTipos = NULL;
simboloVariable * listaVariables = NULL;
simboloEcuacion * listaEcuaciones;
simboloFuncion * funcionActual = NULL;
%}
%union{




}




%%
prog  :     FUNCIONES listaFunciones
            VAR listaDeclaraciones
            ECUACIONES listaEcuaciones
      ;
listaFunciones    :     /* Epsilon */
            |     listaFunciones ID ':'   {
①
                                         }
                        listaTipos FLECHA tipo ';'
                                    {
②
                                    }
            ;
listaTipos  :     /* Epsilon */
            |     listaTiposNovacia
            ;
listaTiposNovacia :     tipo  {
```

```
③
                                }
                |     listaTiposNovacia POR tipo    {
④
                                                    }
                ;
tipo  : ID  {
⑤
            }
      ;
listaDeclaraciones      :     /* Epsilon */
                |     listaDeclaraciones declaracion
                ;
declaracion :     ID ':' tipo ';'   {
⑥
                                                }
            |     ID ',' declaracion      {
⑦
                                                }
            ;
listaEcuaciones   :     /* Epsilon */
            |     listaEcuaciones termino EQUIVALE termino ';'
                        {
⑧
                        }
            ;
termino     :     ID    {
⑨
                        }
      |     ID '(' ')'  {
⑩
                        }
      |     principioTermino ')'    {
①①
                                    }
      ;
principioTermino  :     ID '(' termino    {
①②
                                                }
                |     principioTermino ',' termino  {
①③
                                                    }
                ;
%%
#include "exFeb04l.c"

void main(){
```

```
        yylineno = 1;
        yyparse();
        verTipos(listaTipos);
        verFunciones(listaFunciones);
        verVariables(listaVariables);
        verEcuaciones(listaEcuaciones);
}
```

A rellenar por el alumno:
Tipos de errores semánticos y acción semántica donde controlarlos:

Código de las tablas de símbolos:

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

// -------------------------- TIPOS DE DATOS  --------------------------

typedef struct _simboloTipo {
      struct _simboloTipo * sig;
      char nombre[20];
} simboloTipo;

simboloTipo * insertarTipo(simboloTipo * * p_t, char nombre[20]){
       simboloTipo * s;
       s = (simboloTipo *)malloc(sizeof(simboloTipo));
       strcpy(s->nombre, nombre);
       s->sig = (*p_t);
       (*p_t) = s;
       return s;
}

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

void verTipo(simboloTipo * s){
       printf("%s", s->nombre);
}

void verTipos(simboloTipo * t){
  while( (t != NULL) ){
      verTipo(t); printf("\n");
      t = t->sig;
  }
}

// -------------------------- FUNCIONES --------------------------

typedef struct _simboloFuncion {
      struct _simboloFuncion * sig;
      char nombre[20];
      simboloTipo * salida;
      short int numeroParametros;
      simboloTipo * parametros[20];
} simboloFuncion;

simboloFuncion * insertarFuncion(simboloFuncion * * p_t, char nombre[20]){
       simboloFuncion * s;
       s = (simboloFuncion *)malloc(sizeof(simboloFuncion));
       strcpy(s->nombre, nombre);
       s->salida = NULL;
       s->numeroParametros = 0;
       s->sig = (*p_t);
       (*p_t) = s;
       return s;
}

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

void verFuncion(simboloFuncion * s){
       int i;
       printf("%s:", s->nombre);
       for(i=0;i<s->numeroParametros;i++) printf("%s * ", s->parametros[i]->nombre);
       printf("-> %s", s->salida->nombre);
}

void verFunciones(simboloFuncion * t){
  while( (t != NULL) ){
      verFuncion(t); printf("\n");
      t = t->sig;
```

```
  }
}

// --------------------------- VARIABLES ---------------------------

typedef struct _simboloVariable {
      struct _simboloVariable * sig;
      char nombre[20];
      simboloTipo * tipo;
} simboloVariable;

simboloVariable * insertarVariable(simboloVariable * * p_t, char nombre[20], simboloTipo * tipo){
       simboloVariable * s;
       s = (simboloVariable *)malloc(sizeof(simboloVariable));
       strcpy(s->nombre, nombre);
       s->tipo = tipo;
       s->sig = (*p_t);
       (*p_t) = s;
       return s;
}

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

void verVariables(simboloVariable * t){
  while( (t != NULL) ){
      printf("%s : %s\n", t->nombre, t->tipo->nombre);
      t = t->sig;
  }
}

// --------------------------- TÉRMINOS ---------------------------

typedef struct _simboloTermino {
      short int esVariable;
      simboloVariable * variable;       // Se usa si este término es una variables
                                        // En caso contrario se usan los siguientes campos
      simboloFuncion * funcion;
      short int numeroParametros;
      struct _simboloTermino * parametros[20];
} simboloTermino;

simboloTermino * crearTermino(){
       simboloTermino * s;
       s = (simboloTermino *)malloc(sizeof(simboloTermino));
       s->esVariable = 1;
       s->variable = s->funcion = NULL;
       s->numeroParametros = 0;
       return s;
};

simboloTipo * tipoTermino(simboloTermino * s){
       if (s->esVariable)
           return s->variable->tipo;
       else
           return s->funcion->salida;
}

void liberarTermino( simboloTermino * * p_s){
       if (((*p_s) != NULL) && (!((*p_s)->esVariable))){
           simboloTermino * s = (*p_s);
           int i;
           for(i=0;i<s->numeroParametros;i++)
               liberarTermino(&(s->parametros[i]));
       }
       free(*p_s);
}

void verTermino(simboloTermino * s){
       int i;
       if (s->esVariable)
           printf("%s", s->variable->nombre);
       else{
           printf("%s(", s->funcion->nombre);
```

```
            for(i=0;i<s->numeroParametros;i++){
                verTermino(s->parametros[i]); printf(", ");
            }
            printf(")");
        }
   }

   // --------------------------- ECUACIONES ---------------------------

   typedef struct _simboloEcuacion{
          simboloTermino * izq, * dch;
          struct _simboloEcuacion * sig;
   } simboloEcuacion;

simboloEcuacion * insertarEcuacion(simboloEcuacion * * p_t, simboloTermino * izq, simboloTermino * dch){
       simboloEcuacion * s;
       s = (simboloEcuacion *) malloc(sizeof(simboloEcuacion));
       s->izq = izq;
       s->dch = dch;
       s->sig = NULL;
       s->sig = (*p_t);
       (*p_t) = s;
       return s;
}


void verEcuaciones(simboloEcuacion * t){
  while( (t != NULL) ){
      verTermino(t->izq);
      printf(" == ");
      verTermino(t->dch);
      printf("\n");
      t = t->sig;
  }
}
```