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 2005
3er Curso de I.T. Informática de Sistemas.
Grupos A y B

Apellidos, Nombre: _____

Calificación: _____

PRÁCTICA

Algebraum es un lenguaje que permite expresar funciones recursivas finales (con recursión de cola). La estructura de un progrma en **Algebraum** es como sigue:
FUNC nombreFunc ($param_1$, $param_2$, ..., $param_n$) **DEV** retorno
$cond_1$ -> $expr_1$ de retorno;
$cond_2$ -> $expr_2$ de retorno;
...
$cond_u$ -> nombreFunc($expr_{u1}$, $expr_{u2}$, ... $expr_{un}$) ;
$cond_{v+1}$ -> nombreFunc($expr_{v1}$, $expr_{v2}$, ... $expr_{vn}$) ;
...
FIN FUNC

Básicamente, una función está formada por pares `condición -> resultado`, de tal manera que si se cumple la `condición` se devuelve el `resultado`. Como `resultado` de una función pueden ponerse dos cosas, o bien una `expresión` (a un par `condición->expresión` se le llama *caso base*), o bien una `invocación a la misma función` que se define (lo que constituyen los *casos no base*). Los parámetros formales sólo pueden ser expresiones simples. Por ejemplo, para calcular el factorial se haría así:

```
// Funcion que calcula el factorial
FUNC Factorial(N, A) DEV R
    N=0 -> A ;
    N>0 -> Factorial(N-1, A*N) ;
FIN FUNC
```

de manera que si se invoca a esta función de la forma Factorial(k, 1), el resultado que devuelve es k!. Para que el lector comprenda mejor lo que hace esta función, el código C equivalente sería:

```
int Factorial(int N, int A){
     if (N == 0) return A;
     else if (N > 0) return Factorial(N-1, A*N);
}
```

Como se deduce de esta equivalencia, en **Algebraum** no existen tipos, sino que se asume que todos los tipos son enteros.

Cualquier función recursiva expresada en **Algebraum** puede convertirse en un bloque de código iterativo siguiendo una serie de pasos. Para aclarar cuáles son, éstos se ilustran en código C:
1.- Se crea un bucle **`while`**`(true)`
2.- En su interior se pone una cascada de **`if`**, asociando a cada uno de ellos una de las `condiciones` originales.
3.- En el caso de las condiciones base, al **`if`** correspondiente se asocia un **`return`** `expresión`, donde `expresión` es la que acompaña a la condición base.
4.- Cada invocación a una función se sustituye por un bloque de código en el que la expresión asociada a cada parámetro formal se guarda en una variable temporal; por último, cada variable temporal se almacena en el parámetro formal correspondiente.

Siguiendo estos pasos, la función anterior se convierte en:

```
int Factorial(int N, int A){
     while(true) {
          if (N == 0) return A;
          else if (N > 0) {
               int tmp1, tmp2;
               tmp1 = N-1;
               tmp2 = A*N;
               N = tmp1;
               A = tmp2;
          }
     }
}
```

El propósito de este ejercicio es traducir una función cualquiera expresada en lenguaje **Algebraum** a código de tercetos sustituyendo la recursión final por una iteración. Antes de proceder a la traducción deben controlarse los siguientes errores semánticos:

- No puede haber varios parámetros con el mismo nombre. **0,25 pto.**
- El nombre de la función no puede coincidir con ningún otro identificador. **0,5 pto**.
- El nombre del resultado no puede coincidir con ningún otro identificador. **0,5 pto.**
- Los casos no base deben invocar a la función que se define. **0,25 pto.**
- En cada invocación el número de parámetros reales debe coincidir con el de formales. **1 pto.**
- Controlar que haya al menos un caso base que garantice que el algoritmo puede acabar. **0,25 pto.**

Para realizar la generación de tercetos, se parte de las reglas ya vistas en clase y que permiten generar el código de expresiones y condiciones. Así, el código generado asociado a la función factorial anterior sería:

```
 1    label etq1
 2            tmp1 = 0;
 3            if N = tmp1 goto etq3
 4            goto etq4
 5    label etq3
 6            R=A
 7            goto etq2
 8    label etq4
 9            tmp2 = 0;
10            if N > tmp2 goto etq5
11            goto etq6
12    label etq5
13            tmp3 = 1;
14            tmp4 = N - tmp3;
15            tmp5=tmp4
16            tmp6 = A * N;
17            tmp7=tmp6
18            N=tmp5
19            A=tmp7
20            goto etq1
21    label etq6
22            goto etq1
23    label etq2
```

Este código es equivalente al código anterior en C, con la diferencia de que se sustituye el `return` por una asignación a la variable de salida y un salto al final del código (líneas 6 y 7). Las líneas 15

y 17 ilustran la utilización de variables auxiliares para guardar los parámetros reales que luego se asignan de una sola tacada en las líneas 18 y 19 a los parámetros formales.

Para esto se suministra un esqueleto de programa Yacc en el que sólo deben rellenarse las acciones semánticas. Desde el punto de vista de generación de código, se pide:

- Controlar la ubicación de las etiquetas de verdad y de falso en las condiciones. **2 ptos.**
- Generar las variables temporales necesarias en cada llamada no base y asignar los parámetros formales convenientemente (nótese la existencia del campo `tmpAux` en cada símbolo). **2 ptos.**
- Controlar convenientemente el resto del flujo del programa generado. **0,75 pto.**
- Asignar convenientemente la variable de salida. **0,75 pto.**
-Rellenar la función *imprimirAsignaciones* de la tabla de símbolos. **1,25 ptos.**
- Construir el programa Lex, ignorando los comentarios hasta el final de línea al estilo C (pueden comenzar en cualquier posición de la línea). **0,5 pto.**

Para finalizar el enunciado se muestran otros dos ejemplos:

Ejemplo	Tercetos equivalentes	

`// Funcion que calcula el factorial`
`FUNC Factorial(N, A) DEV R`
`  N=0 -> A ;`
`  N=1 -> A ;`
`  N=2 OR N=3 -> Factorial(N-2, A*N*(N-1)) ;`
`  N>3 -> Factorial(N-1, A*N) ;`
`FIN FUNC`
	label etq1
	tmp1 = 0;
	if N = tmp1 goto etq3
	goto etq4
	label etq3
	R=A
	goto etq2
	label etq4
	tmp2 = 1;
	if N = tmp2 goto etq5
	goto etq6
	label etq5
	R=A
	goto etq2
	label etq6
	tmp3 = 2;
	if N = tmp3 goto etq7
	goto etq8
	label etq8
	tmp4 = 3;
	if N = tmp4 goto etq9
	goto etq10
	label etq7
	goto etq9
	
	label etq12
	goto etq1
	label etq2
		label etq9
		tmp5 = 2;
		tmp6 = N - tmp5;
		tmp7=tmp6
		tmp8 = A * N;
		tmp9 = 1;
		tmp10 = N - tmp9;
		tmp11 = tmp8 * tmp10;
		tmp12=tmp11
		N=tmp7
		A=tmp12
		goto etq1
		label etq10
		tmp13 = 3;
		if N > tmp13 goto etq11
		goto etq12
		label etq11
		tmp14 = 1;
		tmp15 = N - tmp14;
		tmp16=tmp15
		tmp17 = A * N;
		tmp18=tmp17
		N=tmp16
		A=tmp18
		goto etq1

`// Funcion que calcula mal el factorial`
`FUNC Factorial(N, A, A, Factorial) DEV R`
`  N>0 -> Factoria(B-1, A*N) ;`
`FIN FUNC`
	Hay varios parámetros con el mismo nombre: A.
	El parámetro Factorial tiene el mismo nombre que la funcion.
	label etq1
	tmp1 = 0;
	if N > tmp1 goto etq3
	goto etq4
	label etq3
	La función no se llama Factoria.
	tmp2 = 1;
	tmp3 = B - tmp2;
	tmp4=tmp3
	tmp5 = A * N;
	tmp6=tmp5
	Se necesitan 4 parametros, y no 2.
	goto etq1
	label etq4
	goto etq1
	label etq2
	No hay ningun caso base.

Tabla de símbolos: TabSim05.c

```
#include <stdlib.h>
#include <stdio.h>
#define LONG_ID 21
typedef struct nulo {
      struct nulo * sig;
      char nombre[LONG_ID];
      char tmpAux[10];
} simbolo;

simbolo * crear(){
  return NULL;
};

// Se inserta por la cola, de forma que los parámetros formales
// están en el mismo orden en que se declaran
void insertar(simbolo * * p_t, simbolo * s){
  if ((*p_t) == NULL){
    (*p_t) = s;
    s->sig = NULL;
  }else {
    simbolo * aux = (*p_t);
    while(aux->sig != NULL) aux = aux->sig;
    s->sig = aux->sig;
    aux->sig = s;
  }
}

// Busca un símbolo por nombre
simbolo * buscarPorNombre(simbolo * t, char nombre[LONG_ID]){
  while( (t != NULL) && (strcmp(nombre, t->nombre)) )
      t = t->sig;
```

```
  return (t);
}

// Busca un símbolo por posición
simbolo * buscarPorPosicion(simbolo * t, int posicion){
  while( (t != NULL) && (posicion > 1) ){
      t = t->sig;
      posicion --;
  }
  return (t);
}

void imprimirAsignaciones(simbolo * t){




}
```

Fichero ExJun05L.lex

Programa ExJun05Y.yac

```
%{
#include "tabsim05.c"
typedef struct {
      char  etqVerdad[10],
            etqFalso[10];
} dobleCond;
// Vars. globales
simbolo * parametros;
char nombreFunc[LONG_ID];
char etqIniFunc[10],
     etqFinFunc[10];
char idOut[LONG_ID];
int numParamFormales=0;
int numParamReales;
int casoBase=0;
%}

%union
{
      char numero[10];
      char variable[LONG_ID];
      dobleCond bloqueCond;
      char texto[5];
}

%token <numero> NUMERO
%token <variable> ID
%token MAI MEI DIF FUNC DEV FIN FLECHA

%type <variable> expr
%type <bloqueCond> cond
%type <texto> opRel

%left OR
%left AND
%left NOT
%left '+' '-'
%left '*' '/'
%left MENOS_UNARIO

%%
func  :     FUNC ID                 {                           }
            '(' listaParam ')' DEV idSalida {




                                    }
            cuerpo FIN FUNC         {




                                    }
      ;
listaParam :      idEntrada
      |     listaParam ',' idEntrada
      ;
idEntrada : ID    {
```

```
                    }
        ;
idSalida :  ID      {
                    }
        ;
cuerpo      :       expr        {
                            }
        |       listaCasos
        ;
listaCasos  :       caso ';'
        |       listaCasos caso ';'
        ;
caso    :       cond FLECHA             {                                   }
                  exprOLlamadaAFunc     {                                 }
        ;
exprOLlamadaAFunc :     expr    {
                            }
                |       ID          {
                                    }
        ;
```

```
listaExpr    :      exprReal
             |      listaExpr ',' exprReal
             ;
exprReal :   expr          {
                                    }
        ;
expr     :          NUMERO              {
                                      nuevaVar($$);
                                      printf("\t%s = %s;\n", $$, $1);
                                    }
         |          ID                  {
                                      strcpy($$, $1);
                                    }
         |          expr '+' expr    { nuevaVar($$); printf("\t%s = %s + %s;\n", $$, $1, $3); }
         |          expr '-' expr    { nuevaVar($$); printf("\t%s = %s - %s;\n", $$, $1, $3); }
         |          expr '*' expr    { nuevaVar($$); printf("\t%s = %s * %s;\n", $$, $1, $3); }
         |          expr '/' expr    { nuevaVar($$); printf("\t%s = %s / %s;\n", $$, $1, $3); }
         |          '-' expr %prec MENOS_UNARIO
                                    {
                                      nuevaVar($$);
                                      printf("\t%s = - %s;\n", $$, $2);
                                    }
         |          '(' expr ')'    {
                                      strcpy($$, $2);
                                    }
        ;
cond     :          expr opRel expr   {
                                      nuevaEtq($$.etqVerdad);
                                      nuevaEtq($$.etqFalso);
                                      printf("\tif %s %s %s goto %s\n", $1, $2, $3, $$.etqVerdad);
                                      printf("\tgoto %s\n", $$.etqFalso);
                                    }
         |          NOT cond  {
                                      strcpy($$.etqVerdad, $2.etqFalso);
                                      strcpy($$.etqFalso, $2.etqVerdad);
                                    }
         |          cond AND
                              {
                                printf("label %s\n", $1.etqVerdad);
                              }
                    cond          {
                                      printf("label %s\n", $1.etqFalso);
                                      printf("\tgoto %s\n", $4.etqFalso);
                                      strcpy($$.etqVerdad, $4.etqVerdad);
                                      strcpy($$.etqFalso, $4.etqFalso);
                                    }
         |          cond OR
                              {
                                printf("label %s\n", $1.etqFalso);
                              }
                    cond           {
                                      printf("label %s\n", $1.etqVerdad);
                                      printf("\tgoto %s\n", $4.etqVerdad);
                                      strcpy($$.etqVerdad, $4.etqVerdad);
                                      strcpy($$.etqFalso, $4.etqFalso);
                                    }
         |          '(' cond ')'    {
                                      strcpy($$.etqVerdad, $2.etqVerdad);
                                      strcpy($$.etqFalso, $2.etqFalso);
                                    }
        ;
opRel    :          '>'        { strcpy($$, ">"); }
         |          '<'        { strcpy($$, "<"); }
         |          MAI        { strcpy($$, ">="); }
         |          MEI        { strcpy($$, "<="); }
         |          '='        { strcpy($$, "="); }
         |          DIF        { strcpy($$, "!="); }
         ;
%%
#include "ExJun05l.c"

void main() {
          parametros = crear();
          yyparse();
}

void yyerror(char * s)
{
    fprintf(stderr, "Error de sintaxis.\n");
}

void nuevaVar(char * s)
{
  static actual=0;
    strcpy(s, &"tmp");
    itoa(++actual, &(s[3]), 10);
}

void nuevaEtq(char * s)
{
  static actual=0;
    strcpy(s, &"etq");
    itoa(++actual, &(s[3]), 10);
}
```