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 Septiembre de 2005
3er Curso de I.T. Informática de Sistemas.

Apellidos, Nombre: _____

Calificación: _____

TEORÍA

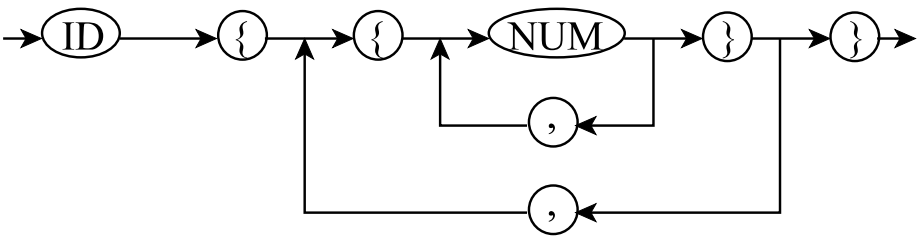
1.- Construir una gramática recursiva por la izquierda que:
a) Permita declarar matrices de dos dimensiones (no tienen porqué ser cuadradas).
b) Permita asignar un número a una posición de una matriz.
c) Permita asignar una secuencia de números a una fila de una matriz.
d) Permita asignar una matriz a otra.
Nota: Por número se entiende una secuencia de dígitos, o bien el valor de una posición de una matriz. Además, una fila de una matriz también es una secuencia de números.

Ejemplos de sentencias válidas son:

```
DECLARE m[3,2], n[2, 2];
DECLARE  h[3,2];
m[1,1] = 27;
m[1,2] = 45;
m[2] = m[1];
m[3] = {12, 33};
n[1] = m[3];
n[2] = {23, m[2, 2]};
```

2.- Dado el siguiente diagrama de sintaxis, codificar en C la función que la reconoce:

valorMatriz



3.- Sabemos que el objetivo de la generación de código intermedio (de tercetos), es generar una secuencia de tercetos semánticamente equivalente al programa de entrada. ¿Bajo qué circunstancias, varias expresiones aritméticas léxica o sintácticamente diferentes generarán exactamente la misma secuencia de tercetos? Justificarlo, además, con un ejemplo.

4.- Con respecto a la gestión de tipos complejos mediante pilas de caracteres vista en clase, responder a las siguientes preguntas:
a) Cuando el tamaño de la pila es 1, ¿qué puede asegurarse sobre el tipo almacenado en la pila?
b) ¿En qué posiciones pueden guardarse los caracteres que representan a los constructores de tipo?
c) Si a una variable **a** le corresponde un tipo expresado por la pila **p1**, y a la variable **b** le corresponde la pila **p2**, y si **p1** es igual a **p2**, ¿qué tipo de equivalencia de tipos podemos asegurar que existe entre **a** y **b**?

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 Septiembre de 2005
3er Curso de I.T. Informática de Sistemas.

Apellidos, Nombre: _____

Calificación: _____

PRÁCTICA

El lenguaje BASICA (BASIC Avanzado) es idéntico al visto en clase en el tema dedicado a la generación de código de tercetos, excepto porque se han ampliado las funcionalidades de tres de sus instrucciones: IF, REPEAT y CASE. A continuación se describen estas modificaciones:

1) Instrucción REPEAT. La instrucción REPEAT vista en clase tenía la estructura:
REPEAT sent ; UNTIL cond
mientras que ahora tiene la forma:
REPEAT sent ; UNTIL cond NUMERO TIMES
donde NUMERO TIMES es opcional, de forma que si no aparece, la instrucción REPAT se comporta como ya se ha visto en clase. La cláusula NUMERO TIMES indica cuantas veces debe ser cierta la condición para que el bucle finalice. Por ejemplo, en el código:

```
c := 0;
REPEAT
     c := c + 1;
UNTIL (c > 5) 3 TIMES;
```

el bucle finaliza con el valor 8 almacenado en la variable **c**.

2) Instrucción CASE. Cada caso de esta instrucción vista en clase tenía la estructura:
CASO expr : sent ;
mientras que ahora, **además**, puede tener la forma:
CASO $expr_1$ TO $expr_2$: sent ;
de tal manera que la sentencia se ejecuta cuando la expresión a comparar tiene un valor que cae en el intervalo cerrado [$expr_1$, $expr_2$]. Si la $expr_2$ es mayor que la $expr_1$, entonces el intervalo se supone vacío y no se entrará por este CASO. Por ejemplo, en el código:

```
CASE cont OF
     CASO 1: sal := 0;
     CASO 2 TO 8 : sal := 1;
     CASO 9 TO 15 : sal := 2;
     OTHERWISE sal := 3;
FIN CASE;
```

la variable **sal** valdrá **1** si **cont** vale entre 2 y 8 inclusives, valdrá **2** si **cont** vale entre 9 y 15 inclusive, valdrá **0** si **cont** vale 1; en caso contrario, **sal** valdrá **3**.

Los intervalos pueden solaparse y se evaluarán en riguroso orden de aparición (del primero al último).

3) Instrucción IF. La instrucción IF vista en clase tenía la estructura:
IF cond THEN sent ; ELSE sent ; FIN IF
donde la parte ELSE es opcional; ahora **pueden** incluirse **además** tantas cláusulas ELSIF como se quiera, de la forma:
IF $cond_0$ THEN $sent_0$;
ELSIF $cond_1$ THEN $sent_1$
ELSIF $cond_2$ THEN sen_2
...

ELSIF $cond_n$ THEN $sent_n$
ELSE $sent_{n+1}$;
FIN IF

donde cada $sent_i$ se ejecuta siempre y cuando $cond_i$ sea cierta y $cond_0$ a $cond_{i-1}$ sean falsas. La $sent_{n+1}$ se ejecuta siempre que $cond_0$ a $cond_n$ sean falsas. Por ejemplo, el código:

```
IF cont = 1 THEN sal := 0;
ELSIF cont >= 2 AND cont <= 8 THEN sal := 1;
ELSIF cont >= 9 AND cont <= 15 THEN sal := 2;
ELSE sal := 3;
FIN IF;
```

se comporta de igual manera que la sentencia CASE del ejemplo anterior.

A continuación se muestra un ejemplo de entrada válida y de la salida que se debe producir.

Entrada:

```
CONT := 2;
DIAPASON := 7;
REPEAT {
      IF DIAPASON = 7 THEN
            CONT := CONT + 1;
      ELSIF DIAPASON = 6 OR DIAPASON = 4 THEN
            CONT := CONT - 1;
      ELSIF DIAPASON = 1 THEN
            CONT := 2;
      ELSE
            CONT := 0;
      FIN IF;
      CASE DIAPASON OF
            CASO 5 :    CONT := 3;
            CASO 2 TO 3 :     CONT := 5;
            CASO -1 TO 0 :    CONT := 0;
            OTHERWISE   CONT := -1;
      FIN CASE;
}; UNTIL (CONT <= 0) 3 TIMES;
```

Salida:

```
  tmp1 = 2;
  CONT = tmp1
  tmp2 = 7;
  DIAPASON = tmp2
  tmp3=0
label etq1
  tmp4 = 7;
  if DIAPASON = tmp4 goto etq2
  goto etq3
label etq2
  tmp5 = 1;
  tmp6 = CONT + tmp5;
  CONT = tmp6
  goto etq4
label etq3
  tmp7 = 6;
  if DIAPASON = tmp7 goto etq6
  goto etq7
label etq7
  tmp8 = 4;
  if DIAPASON = tmp8 goto etq8
  goto etq9
label etq6
  goto etq8
label etq8
  tmp9 = 1;
  tmp10 = CONT - tmp9;
  CONT = tmp10
  goto etq5
label etq9
  tmp11 = 1;
  if DIAPASON = tmp11 goto etq10
  goto etq11
label etq10
  tmp12 = 2;
  CONT = tmp12
  goto etq5
label etq11
  tmp13 = 0;
  CONT = tmp13
label etq5
label etq4
  tmp14 = 5;
  if DIAPASON != tmp14 goto etq13
  tmp15 = 3;
  CONT = tmp15
  goto etq12
label etq13
  tmp16 = 2;
  tmp17 = 3;
  if DIAPASON < tmp16 goto etq14
  if DIAPASON > tmp17 goto etq14
```

```
     tmp18 = 5;
     CONT = tmp18
     goto etq12
label etq14
     tmp19 = 1;
     tmp20 = - tmp19;
     tmp21 = 0;
     if DIAPASON < tmp20 goto etq15
     if DIAPASON > tmp21 goto etq15
     tmp22 = 0;
     CONT = tmp22
     goto etq12
label etq15
     tmp23 = 1;
     tmp24 = - tmp23;
     CONT = tmp24
label etq12
     tmp25 = 0;
     if CONT <= tmp25 goto etq16
     goto etq17
label etq17
     goto etq1
label etq16
     tmp3=tmp3+1
     if tmp3<3 goto etq1
```

Se pide:

- Construir el programa YACC que genere el código de tercetos equivalente a la semántica de las sentencias propuestas. Para ello se suministra un esqueleto en el ya se encuentra rellenada la parte estudiada en clase.

El programa Lex **no** es necesario especificarlo, pues casi igual al ya visto en clase.

Las reglas asociadas a las expresiones y a las condiciones, ni se han adjuntado al esqueleto (por falta de espacio), ni deben ser incluídas por el alumno, ya que son idénticas a las vistas en clase.

No es necesario realizar ningún control de tipos, ni existe declaración de variables.

YACC:

```
%{
typedef struct  {
        char    etqVerdad[21],
                etqFalso[21];
} dobleCond;
typedef struct  {
        char    etqFinal[21];
        char    variableExpr[21];
} datosCase;
typedef struct {
        char variableAux[21];
        char etiquetaAux[21];
} datosRepeat;
%}

%union {
        int numero;
        char variableAux[21];
        char etiquetaAux[21];
        char etiquetaSiguiente[21];
        dobleCond bloqueCond;
        datosCase bloqueCase;
        datosRepeat bloqueRepeat;
}

%token <numero> NUMERO
%token <variableAux> ID
%token <etiquetaAux> IF
%token <bloqueRepeat> REPEAT
```

```
%token <etiquetaSiguiente> CASO
%token ASIG THEN ELSE FIN DO UNTIL CASE OF OTHERWISE TO TIMES ELSIF
%token MAI MEI DIF

%type <numero> opcionalRepeat
%type <variableAux> expr
%type <bloqueCond> cond
%type <bloqueCase> inicio_case
%type <etiquetaAux> opcionalElsIfs

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

%%
prog    :       prog sent ';'
        |       prog error ';' { yyerrok; }
        |
        ;
sent    :       ID ASIG expr  {
                                printf("\t%s = %s\n", $1, $3);
                              }
        |       IF cond
                        {
                         printf("label %s\n", $2.etqVerdad);
                        }
                THEN sent ';'
                        {
                         nueva_etq($1);
                         printf("\tgoto %s\n", $1);
                         printf("label %s\n", $2.etqFalso);
                        }
                opcionalElsIfs
                opcional
                FIN IF
                        {
                        }
        |       '{' lista_sent '}' { ; }
        |       REPEAT
                        {
                         nueva_etq($1.etiquetaAux);
                         nueva_var($1.variableAux);
                         printf("\t%s=0\n", $1.variableAux);
                         printf("label %s\n", $1.etiquetaAux);
                        }
                sent ';'
                UNTIL cond
                        {
```

```
                        printf("label %s\n", $6.etqFalso);
                        printf("\tgoto %s\n", $1.etiquetaAux);
                        printf("label %s\n", $6.etqVerdad);
                    }
                opcionalRepeat
                    {
                    }
        |       sent_case
        ;
opcionalRepeat      :       /* Epsilon */           {
                                                    }
                |       NUMERO TIMES                {
                                                    }
                ;
opcionalElsIfs      :       /* Epsilon */   {
                                            }
                |       opcionalElsIfs ELSIF cond
                            {
                            }
                        THEN sent ';'
                            {
                            }
                ;
opcional        :       ELSE sent ';'
                |
                ;
lista_sent      :       /* Epsilon */
                |       lista_sent sent ';'
                |       lista_sent error ';' { yyerrok; }
                ;
sent_case       :       inicio_case
                        OTHERWISE sent ';'
                        FIN CASE
                            {
                              printf("label %s\n", $1.etqFinal);
                            }
                |       inicio_case
                        FIN CASE
                            {
```

```
                        printf("label %s\n", $1.etqFinal);
                    }
        ;
inicio_case :   CASE expr OF
                    {
                     strcpy($$.variableExpr, $2);
                     nueva_etq($$.etqFinal);
                    }
        |       inicio_case
                CASO expr ':'
                    {
                     nueva_etq($2);
                     printf("\tif %s != %s goto %s\n",
                            $1.variableExpr, $3, $2);
                    }
                sent ';'
                    {
                     printf("\tgoto %s\n", $$.etqFinal);
                     printf("label %s\n", $2);
                     strcpy($$.variableExpr, $1.variableExpr);
                     strcpy($$.etqFinal, $1.etqFinal);
                    }
        |       inicio_case
                CASO expr TO expr ':'
                    {
                    }
                sent ';'
                    {
                     printf("\tgoto %s\n", $$.etqFinal);
                     printf("label %s\n", $2);
                     strcpy($$.variableExpr, $1.variableExpr);
                     strcpy($$.etqFinal, $1.etqFinal);
                    }
        ;
...
/* El resto es igual a lo visto en clase */
```