13048 - LABORATORIO DE PROCESADORES DE LENGUAJE
Curso: 2012-2013
Práctica 2: Analizador léxico/sintáctico/semántico con Flex y Bison

Planteamiento del problema

En esta práctica se trata de realizar, mediante el generador de analizadores léxicos FLEX y sintácticos BISON, un analizador léxico, sintáctico y semántico que reconozca una simplificación del lenguaje de programación Modula-2. Además, la práctica deberá incluir la construcción de la Tabla de Símbolos (TS).

Especificación a nivel léxico

Consideraremos como comentario todo aquello que vaya precedido por un paréntesis de apertura y un asterisco y hasta que encontremos de nuevo otro asterisco y el paréntesis de cierre. Por ejemplo: (* Esto es un comentario. *)

Los identificadores (**ID**) constarán de una letra seguida opcionalmente de más letras, dígitos y símbolos de subrayado '_'.

Existen dos tipos de constantes numéricas; los números enteros (**TKN_NUMENT**) formada por uno o más dígitos y los números reales (**TKN_NUMREAL**) formado por uno ó más dígitos seguidos de un punto y uno o más dígitos.

Los símbolos especiales son: **( ) , . + - * / = <> > < <= >= := ; :**

Todas las palabras reservadas están en MAYUSCULAS. Estas palabras reservadas son: **MODULE FROM IMPORT VAR PROCEDURE CONST BEGIN END BOOLEAN INTEGER REAL WHILE FOR TO DO REPEAT UNTIL IF THEN ELSE RETURN OR AND**

Especificación a nivel sintáctico

Modula-2 es un lenguaje de alto nivel que permitir definir subprogramas dentro de otros programas. Para simplificar, en nuestro caso no vamos a permitir el anidamiento de programas. En nuestra versión simplificada del Modula-2 tendremos un único programa en el que se pueden definir variables, constantes y procedimientos o funciones (sin anidamiento).

Hay que señalar que la definición de las variables del programa principal puede hacerse antes o después de los procedimientos/funciones. Si la declaración de las variables del programa principal se realizan después de las declaraciones de los procedimientos o funciones, éstos pierden automáticamente el acceso a dichas variables. RECORDAR QUE SOLO SE PUEDE ACCEDER A ELEMENTOS YA DEFINIDOS.

Por otra parte, hay que observar que tanto en la llamada como en la definición de procedimientos, si no hay argumentos, los paréntesis son opcionales, mientras que en las funciones son siempre obligatorios.

Algunos ejemplos básicos serían:

```
MODULE primero;
(* Este programa escribe 111111 *)
FROM InOut IMPORT WriteInt;
BEGIN
   WriteInt ( 111111, 5 );
END primero.
```

```
MODULE suma_dos_numeros;
(* Este programa suma dos números enteros
  tanto positivos como negativos *)
FROM InOut IMPORT ReadInt, WriteLn,
WriteInt;
VAR
   numero_1, numero_2: INTEGER;

BEGIN
   ReadInt(numero_1);
   WriteLn;
   ReadInt(numero_2);
   WriteLn;
   WriteInt( numero_1+numero_2, 10 );
END suma_dos_numeros.
```

```
(* ----------------------------------------------------
  Este programa imprime la tabla de multiplicar de un número leido
como dato  ---------------------------------------------------- *)
 MODULE tabla_mult;

FROM InOut IMPORT ReadInt, WriteLn,WriteInt;

VAR
   multiplicando, multiplicador, resultado: INTEGER;

BEGIN
   ReadInt( multiplicando );
   WriteLn;
   multiplicador:=1;
   WHILE multiplicador < 11 DO
      resultado:=multiplicando * multiplicador;

      WriteInt( resultado, 5 );
      WriteLn;

      multiplicador:=multiplicador+1;
   END
END tabla_mult.
```

```
MODULE Ejemplo1;
IMPORT WrInteger;
PROCEDURE EscribeSumatorio ( N :
INTEGER );
VAR Suma, i : INTEGER;
BEGIN
      Suma := 0;
      FOR i := 1 TO N DO
            Suma := Suma + i;
      END;
      WrInteger (Suma, 0);
END EscribeSumatorio;
BEGIN
      EscribeSumatorio ( 6 );
END Ejemplo1.
```

```
MODULE M2;
(* Programa de ejemplo *)
IMPORT WriteLn, WriteInt ;

PROCEDURE Repetir( numero: INTEGER; veces: INTEGER );
(* Imprime el numero tantas veces como se indique, ocupando una
línea cada vez. Las líneas aparecen numeradas. *)
VAR k: INTEGER;
BEGIN
        FOR k := 1 TO veces DO
                WriteInt( k, 3 );
                WriteInt (numero , 5);
                WriteLn;
        END;
END Repetir;

BEGIN
        Repetir( 0 , 1 ); (* Escribe el 0 una vez *)
        Repetir( 10 , 10 ); (* Escribe el 10, 10 vecesl *)
END M2.
```

(*Ejemplo: El siguiente subprograma toma un argumento real y lo redondea:*)

```
MODULE Ejemplo2;
FROM IO IMPORT WrReal;
IMPORT TRUNC;
PROCEDURE Redondea (VAR Num:REAL);
BEGIN
  Num := (TRUNC (Num + 0.5));
END Redondea;
VAR x : REAL;
BEGIN
   x := 10.7;
   Redondea (x);
   WrReal(x, 5, 0);
END Ejemplo2.
```

```
MODULE Ejemplo3;
FROM IO IMPORT WrReal;
VAR x, y : REAL;
PROCEDURE EscribirDivCuadrados (x, y : REAL);
CONST
    PRECISION = 5;
    ANCHO = 0;
VAR Cociente : REAL;
BEGIN
    Cociente := (x * x) / (y * y);
    WrReal (Cociente, PRECISION, ANCHO);
END EscribirDivCuadrados;
BEGIN
    x := 1.0;
    y := 2.0;
    EscribirDivCuadrados (y, x);
END Ejemplo3.
```

La gramática correspondiente a este lenguaje es la siguiente:

programa → **TKN_MODULE TKN_ID TKN_PTOCOMA** import_list_opt block **TKN_ID TKN_PTO**

import_list_opt → import_list | ε
import_list → import import_list_opt
import → from_opt **TKN_IMPORT TKN_ID** ident_list **TKN_PTOCOMA**
from_opt → **TKN_FROM TKN_ID** | ε

ident_list → **TKN_COMA TKN_ID** ident_list | ε

block → declaration_list_opt begin_and_stmts_opt **TKN_END**

declaration_list_opt → declaration_list | ε
declaration_list → declaration declaration_list_opt
declaration → **TKN_CONST** constant_declaration_list_opt |
TKN_VAR variable_declaration_list_opt |
procedure_declaration **TKN_PTOCOMA**

constant_declaration_list_opt → constant_declaration_list | ε
constant_declaration_list → constant_declaration constant_declaration_list_opt
constant_declaration → **TKN_ID TKN_IGUAL** expression **TKN_PTOCOMA**

variable_declaration_list_opt → variable_declaration_list | ε
variable_declaration_list → variable_declaration variable_declaration_list_opt
variable_declaration → **TKN_ID** ident_list **TKN_DOSPTOS** type **TKN_PTOCOMA**

type → **TKN_INTEGER** | **TKN_REAL** | **TKN_BOOLEAN**

declaration_list_opt2 → declaration_list2 | ε
declaration_list2 → declaration2 declaration_list_opt2
declaration2 → **TKN_CONST** constant_declaration_list_opt |
TKN_VAR variable_declaration_list_opt

procedure_declaration → procedure_heading **TKN_PTOCOMA** declaration_list_opt2
begin_and_stmts_opt **TKN_END TKN_ID**

procedure_heading → **TKN_PROCEDURE TKN_ID** formal_parameters_opt function_opt

formal_parameters_opt → formal_parameters | ε
formal_parameters → **TKN_PARA** fp_section_list_opt **TKN_PARC**
fp_section_list_opt → fp_section fp_section_list | ε
fp_section_list → **TKN_PTOCOMA** fp_section fp_section_list | ε
fp_section → var_opt **TKN_ID** ident_list **TKN_DOSPTOS** type
var_opt → **TKN_VAR** | ε

function_opt → **TKN_DOSPTOS** type | ε

begin_and_stmts_opt → begin_and_stmts | ε
begin_and_stmts → **TKN_BEGIN** statement **TKN_PTOCOMA** statement_list

statement_list → statement **TKN_PTOCOMA** statement_list | ε
statement → **TKN_ID** sentencia | if_statement | while_statement |
repeat_statement | for_statement | **TKN_RETURN** expression_opt

expression_opt → expression | ε

sentencia → assignment | actual_parameters_opt

assignment → **TKN_ASIGNACION** expression

actual_parameters_opt → actual_parameters | ε

```
actual_parameters → TKN_PARA exp_list_opt TKN_PARC
exp_list_opt → expression exp_list  |  ε
exp_list → TKN_COMA expression exp_list | ε

if_statement → TKN_IF expression TKN_THEN statement_list else_opt TKN_END
else_opt → TKN_ELSE statement_list |  ε

while_statement → TKN_WHILE expression TKN_DO statement_list TKN_END

repeat_statement → TKN_REPEAT statement_list TKN_UNTIL expression

for_statement → TKN_FOR TKN_ID TKN_ASIGNACION expression TKN_TO expression
                TKN_DO statement_list TKN_END

expression → simple_expression  simple_expression_opt
simple_expression_opt → relation  expression | ε
sign_opt → TKN_MAS | TKN_MENOS | ε

relation → TKN_IGUAL | TKN_DISTINTO | TKN_MENOR |
           TKN_MENORIGUAL | TKN_MAYOR | TKN_MAYORIGUAL

add_operator → TKN_MAS | TKN_MENOS | TKN_OR
mul_operator → TKN_MULTI | TKN_DIV | TKN_AND

simple_expression → sign_opt term | simple_expression add_operator term

term → factor | term mul_operator factor
factor → TKN_NUMENT | TKN_NUMREAL | TKN_ID actual_parameters_opt |
         TKN_PARA expression TKN_PARC
```

Importante: Habrá que tener en cuenta la precedencia entre operadores y los signos positivo y negativo en las expresiones que trabajan con dichos operadores. Por otra parte, deberemos indicar la fila y columna de los posibles errores, tanto léxicos como sintácticos y semánticos. Por último, se deberá poder visualizar la tabla de símbolos una vez esta haya sido completamente generada.

Nota: Se podrán incluir modificaciones a la gramática siempre que se genere el mismo lenguaje.

Generación de la tabla de símbolos

En este apartado se trata de generar la tabla de símbolos. Esta estructura, junto con el árbol sintáctico, se usará también en la siguiente práctica para generar código intermedio.

A la hora de generar la **tabla de símbolos** se debe tener en cuenta que es necesario almacenar tanto las variables como las funciones, para ello tendremos que diferenciarlas mediante un campo que nos indique de que tipo de símbolo se trata. Además, como Modula-2 permite importar funciones de otras librerías, también deberemos tenerlas en cuenta aunque para este caso no sabemos el número de argumentos ni su tipo y no haremos la comprobación.

Para el caso de las variables, deberemos contemplar la posibilidad de que estén definidas en el cuerpo del programa (que serán globales) o dentro de alguna función (que serán locales). Necesitaremos un campo que nos almacene el ámbito al que pertenece la variable. La primera función definida tiene ámbito uno, la segunda tiene ámbito dos, así sucesivamente. También habrá que tener en cuenta el número de argumentos que se le pasa a las funciones y cuales son estas variables.

Comprobaciones semánticas

Se deberán realizar todas las siguientes comprobaciones semánticas básicas con la tabla de símbolos:

- ✓ No insertar un identificador de una variable o constante dos veces con el mismo nombre y con el mismo ámbito o un identificador de función dos veces.
- ✓ No podrá haber funciones y variables (o constantes) con el mismo nombre independientemente del ámbito en el que esté declarada la variable.
- ✓ En las llamadas a funciones, deberemos comprobar que esa función ha sido definida y que el número de argumentos que se le pasa es el correcto y, además, que los argumentos son del mismo tipo.
- ✓ Para el caso particular de usar las funciones importadas solo comprobaremos que fueron importadas, pero no el número de argumentos ni el tipo (dado que no se conoce).
- ✓ Cuando se usen las variables o constantes dentro de las sentencias de la gramática o de las funciones, deberemos comprobar que dichas variables o constantes hayan sido previamente declaradas. Igualmente, no se puede llamar a un procedimiento si éste no existe.
- ✓ Se comprobará que en la parte izquierda de una asignación no hay llamadas a funciones.
- ✓ Se comprobará que el identificador que aparece junto con el END se corresponde con el del módulo o procedimiento correspondiente.
- ✓ Se comprobará en las expresiones aritméticas que ambos operandos son del mismo tipo. De lo contrario, se declarará un error. Sólo están definidas las operaciones aritméticas para los tipos de datos enteros y reales.

Nota: Opcionalmente se podrán incluir otro tipo de comprobaciones.

Gestión de errores

Siempre que se produzca un error, tanto léxico como sintáctico como semántico, habrá que indicar el tipo de error de que se trata así como la fila y la columna donde se ha producido dicho error.

Más Ejemplos del lenguaje definido por la gramática:

```
MODULE Ejemplo;
FROM IO IMPORT WrReal,  WrLn;
FROM MATHLIB IMPORT Sqrt;
VAR x, y : REAL;
PROCEDURE EscribirModulo (Real, Imag : REAL);
VAR Modulo : REAL;
BEGIN
      WrReal (Real, 5, 0);
      WrReal (Imag, 5, 0);
      Real := Real * Real;
      Imag := Imag * Imag;
      Modulo := Sqrt (Real + Imag);
      WrReal (Modulo, 5, 0);
      WrLn;
END EscribirModulo;
BEGIN
      x := 5.0;
      y := 6.0;
      EscribirModulo (x, y);
      WrReal (x, 5, 0); WrLn;
      WrReal (y, 5, 0); WrLn;
```

```
END Ejemplo.
```

(*Ejemplo: El siguiente subprograma intercambia los valores almacenados en dos variables de tipo INTEGER, utilizando para ello una tercera variable temporal: *)

```
MODULE Ejem;
FROM IO IMPORT WrInt;
VAR x, y : INTEGER;
PROCEDURE Intercambiar (VAR Var1, Var2 : INTEGER);
VAR Temp : INTEGER;
BEGIN
      Temp := Var1;
      Var1 := Var2;
      Var2 := Temp;
END Intercambiar;
BEGIN
      x := 10;
      y := 20;
      Intercambiar(x,y);
      WrInt(x,0); WrLn;
      WrInt(y,0);
END Ejem.
```

```
MODULE Matematicas;
FROM IO IMPORT WrReal;
PROCEDURE VolumenCubo (l : REAL) : REAL;
BEGIN
      RETURN (l*l*l);
END VolumenCubo;
PROCEDURE Potencia (x : REAL; n : INTEGER) : REAL;
VAR i : INTEGER;
Pot : REAL;
BEGIN
      Pot := 1.0;
      FOR i := 1 TO n DO
            Pot := Pot * x;
      END;
      RETURN Pot;
END Potencia;
PROCEDURE Maximo (a, b : REAL) : REAL;
VAR Max : REAL;
      BEGIN
            IF (a>b) THEN
                  Max := a;
            ELSE
                  Max := b;
      END;
      RETURN Max;
END Maximo;
VAR x, y , z: REAL;
BEGIN
      x := VolumenCubo (3.25);
      y := Potencia( x , 2 );
      z := Potencia( x , y );
      WrReal (x, 5, 0); WrLn;
      WrReal (y, 5, 0); WrLn;
      WrReal (z, 5, 0); WrLn;
END Matematicas.
```

Fecha de entrega:

Todos los grupos deberán entregar la práctica con fecha máxima el 12 de Marzo, mediante Aula Virtual.

DURACIÓN: 2 sesiones