ANÁLISIS LÉXICO-SINTÁCTICO

OBJETIVO

Problemas para consolidar el diseño y construcción de analizadores léxico-sintácticos.

PROBLEMA 1: LENGUAJE EXPL

Supongamos un lenguaje de expresiones lógicas llamado EXPL con los siguientes recursos expresivos: variables, números enteros, funciones, predicados, negación, conjunción, disyunción, implicación y cuantificadores universal y existencial. Los predicados y funciones son símbolos prefijos con las excepciones de los predicados relacionales igual, mayor que, menor que, menor o igual que, mayor o igual que y distinto y de las funciones suma, resta, producto y división que son infijos.

Algunas de las expresiones de EXPL son:

```
1. PARATODO x,y:(p(f(x)+2-3*5) O r(x,y) => EXISTE z:(z+2 == 15))
2. PARATODO x,y:(p(x) Y NO r(x,y) => z == x)
3. (resultado == i Y i<=numeroElementos(v))
```

En la sentencia 1, `PARATODO` es el cuantificador universal y `x,y` sus variables cuantificadas, `EXISTE` es el cuantificador existencial y `z` su variable cuantificada `O` es la disyunción lógica, `=>` es la implicación lógica, `p` y `r` son predicados, `==` es el predicado de igualdad, `f(x)+2-3*5` y `z+2` son expresiones y `15` es un número entero.

En la sentencia 2, `NO` es la negación lógica.

En la sentencia 3, `Y` es la conjunción lógica y `<=` es el predicado binario menor o igual.

SE PIDE: Diseño de gramática independiente de contexto para EXPL. Usar este diseño como base de la implementación de un parser Antlr para EXPL. Implementación de un lexer Antlr para EXPL. Testar los analizadores sobre un conjunto de expresiones pertenecientes a EXPL y sobre un conjunto de expresiones no pertenecientes a EXPL.

PROBLEMA 2: LENGUAJE LEXCHANGE

Supongamos un lenguaje llamado LEXCHANGE para programar transferencias de datos entre dos bases de datos relacionales. Un programa típico en LEXCHANGE incluye: (a) esquema de la base de datos fuente, (b) datos fuente, (c) esquema de la base de datos destino y un conjunto de restricciones especificando la transferencia de datos entre la fuente y el destino. Las restricciones son implicaciones lógicas con antecedente formado por un átomo definido sobre una relación del esquema fuente y un consecuente formado por un átomo definido sobre una relación del esquema destino (ver programa de ejemplo). Hay dos clases de variables en una restricción, las que sólo aparecen en el consecuente y las demás. Las primeras se suponen cuantificadas existencialmente y las segundas universalmente.

Ejemplo. Programa LEXCHANGE.

```
ESQUEMA FUENTE
   estudiante(NOMBRE, NACIMIENTO, DNI)
   empleado(NOMBRE, DNI, TELEFONO)

DATOS FUENTE
   estudiante(Axel,1980,12122345)
   estudiante(Lorenzo,1982,10345321)
   estudiante(Antonio,1979,87654456)
   empleado(Axel,12122345,616234345)
   empleado(Manuel,50545318,617876654)

ESQUEMA DESTINO
   persona(NOMBRE, NACIMIENTO, DNI, TELEFONO)

RESTRICCIONES
   VAR x,y,z,u;
       estudiante(x,y,z) implica persona(x,y,z,u)
   VAR x,y,z,u;
       empleado(x,y,z) implica persona(x,u,y,z)
```

La ejecución del anterior programa transfiere los siguientes datos a la base de datos destino. Cada término `Xi` presente en los datos representa la instanciación de una variable cuantificada existencialmente en una restricción.

```
   persona(Axel,1980,12122345,X1)
   persona(Lorenzo,1982, 10345321,X2)
   persona(Antonio,1979, 87654456,X3)
   persona(Axel,X4,12122345, 616234345)
   persona(Manuel,X5,50545318,617876654)
```

SE PIDE: Diseño de gramática independiente de contexto para LEXCHANGE. Usar este diseño como base de la implementación de un parser Antlr para LEXCHANGE. Implementación de un lexer Antlr para LEXCHANGE. Testar los analizadores.

.PROBLEMA 3: CALCPROG

Supongamos un lenguaje llamado CALCPROG para programar secuencias de órdenes. La orden puede ser: expresión entera, asignación o declaración de función. Las declaraciones de funciones están restringidas a un parámetro. El siguiente ejemplo muestra un programa típico CALCPROG.

```
3 + (4 + 1);         // expresión
a = 1 + 1;           // asignación
f(a) = 10 * a;       // declaración de función
f(a);                // expresión
g(x) = 10*f(x) + a; // declaración de función
g(3);                // expresión
f(f(2));             // expresión
f(a+1);              // expresión
```

SE PIDE: Diseño de gramática independiente de contexto para CALCPROG. Usar este diseño como base de la implementación de un parser Antlr para CALCPROG. Implementación de un lexer Antlr para CALCPROG. Testar los analizadores.

PROBLEMA 4: UMLTEXT

Supongamos un lenguaje llamado UMLTEXT diseñado para expresar textualmente diagramas de clase y diagramas de objetos UML. Los diagramas de clase tienen la siguiente expresividad: declaración de clase, declaración de asociación y declaración de restricciones de multiplicidad sobre los extremos de la asociación. Un ejemplo de sentencia UMLTEXT es la siguiente:

```
BEGIN_CLASS_DIAGRAM                 //Diagrama de clase.
(Class A)                           //Clase A
(Class B)                           //Clase B
(Association R between A and B)     //Asociación entre A y B
(Multiplicity R on A is = 1)        //Restricción multiplicidad sobre R
                                    //en A
(Multiplicity R on B is >= 1)       //Restricción multiplicidad sobre R
                                    //en B
END_CLASS_DIAGRAM


BEGIN_OBJECT_DIAGRAM               //Diagrama de objetos.
(Object o1 Class A)                //Objeto clase A
(Object o2 Class B)                //Objeto clase B
(Object o3 Class A)                //Objeto clase A
(Object o4 Class B)                //Objeto clase B

(Link l1 Association R LinkedObjects o1 o2) //Enlace de R
(Link l2 Association R LinkedObjects o3 o2) //Enlace de R
END_OBJECT_DIAGRAM
```

SE PIDE: Diseño de gramática independiente de contexto para UMLTEXT. Usar este diseño como base de la implementación de un parser Antlr para UMLTEXT. Implementación de un lexer Antlr para UMLTEXT. Testar los analizadores.

PROBLEMA 5: LPROC

Supongamos un lenguaje llamado LPROC diseñado para expresar procedimientos como el mostrado en el siguiente ejemplo.

```
buscar(entero e, vector(entero)[1..n]  v) devuelve (booleano, entero)

variables locales:
   booleano resultado;
   entero elemento, i;

instrucciones:
   (resultado, i)=(falso, 1);
    mientras (resultado == falso y i<=numeroElementos(v)) hacer
       elemento = v[i];
       si (elemento == e) entonces
          resultado = cierto;
       sino
          i=i+1;
       finsi
    finmientras
    devuelve (resultado,i);

fin
```

El perfil de un procedimiento consta de nombre, parámetros y resultados. Internamente el procedimiento se estructura con un conjunto de variables locales y una secuencia de instrucciones. Los tipos elementales en LPROC son el tipo lógico y el tipo entero. El único tipo no elemental es el tipo vector (de elementos enteros o lógicos). La declaración de un vector siempre incluirá el tipo de sus elementos y el rango de sus índices. LPROC tiene 5 tipos de instrucciones: asignaciones simples, asignaciones múltiples, iteraciones, condicionales y devolución de resultados.

SE PIDE: Diseño de gramática independiente de contexto para LPROC. Usar este diseño como base de la implementación de un parser Antlr para LPROC. Implementación de un lexer Antlr para LPROC. Testar los analizadores.

PROBLEMA 6: ASSERTION

ASSERTION es un lenguaje de programación. Sus programas pueden incluir asertos. Un aserto es una condición lógica construida con las siguientes reglas:
(1) Si f y g son expresiones enteras entonces f == g, f != g, f > g y f < g son asertos.

(2) Las constantes T y F son asertos.

(3) Si f y g son asertos también lo son su conjunción (ej. f & g), disyunción (ej. f | g) y negación (ej. –f, -g).

(4) Si f es un aserto también lo es f entre paréntesis (ej. (f)).

Sintácticamente, un programa ASSERTION está compuesto por una declaración de variables y una secuencia de instrucciones. Las instrucciones pueden ser (a) asignaciones, (b) instrucciones condicionales e (c) iteraciones. Un programa ASSERTION puede incluir un aserto entre llaves en cualquier punto de su secuencia de instrucciones.

Ejemplo de programa ASSERTION:

```
PROGRAMA
VARIABLES x, y, z;
LEER(x);
{ x > 0 }
z = 1;
{ z < x/2 | z == x/2 }
y = 1;
MIENTRAS (y < x/2 | y == x/2) HACER
SI ((x/y)*y == x) ENTONCES
z = y;
FINSI
y = y + 1;
{ (x/z) * z == x & (z == 1 | z > 1) & (z < x/2 | z == x/2) }
FINMIENTRAS
{ (x/z) * z == x & (z == 1 | z > 1) & (z < x/2 | z == x/2) }
```

SE PIDE: Diseño de gramática independiente de contexto para ASSERTION. Usar este diseño como base de la implementación de un parser Antlr para ASSERTION. Implementación de un lexer Antlr para ASSERTION. Testar los analizadores.