ANÁLISIS SEMÁNTICO

OBJETIVO

Problemas para consolidar el diseño y construcción de analizadores semánticos.

PROBLEMA 1: LENGUAJE INTEXP

Un programa INTEXP es una secuencia de asignaciones. Cada asignación asocia una expresión entera a una variable. Las expresiones enteras pueden contener variables (p.e. `z`), variables pre-asignadas (p.e, `z()`), constantes (p.e. `7`), sumas (p.e. `+`), restas (p.e. `-`), productos (p.e. `*`) y divisiones (p.e. `/`) y sub-expresiones entre paréntesis (p.e. `(x+1)`). Toda variable pre-asignada se considerada inicializada a cero.

Ejemplo de programa INTEXP:

```
VARIABLES  x,y,z;
INSTRUCCIONES
x=1;
y=2*(x+7);
z=z+1;
x=z();
```

Suponga la siguiente gramática para INTEXP.

```
programa : declaracion_variables instrucciones EOF ;

declaracion_variables: VARIABLES variables PUNTOYCOMA
         ;
variables: IDENT COMA variables #Ids
         | IDENT                #Id
         ;
instrucciones: INSTRUCCIONES (asignacion)*   ;

asignacion : IDENT ASIG expresion PUNTOYCOMA ;

expresion : expresion (POR|DIV|MAS|MENOS) expresion       #OpBin
          | NUMERO                                        #Num
          | IDENT PARENTESISABIERTO PARENTESISCERRADO      #VarNoAmb
          | IDENT                                         #Var
          | PARENTESISABIERTO expresion PARENTESISCERRADO #Par
          ;
```

SE PIDE:

[SOLUCIÓN] Diseño de gramática atribuida para un analizador semántico de INTEXP capaz de detectar asignaciones ambiguas. Una asignación es ambigua cuando la expresión asignada contiene alguna variable ambigua. Una variable es ambigua si no tiene asignado ningún entero. En el programa de ejemplo, la asignación `z=z+1;` es ambigua porque la variable `z` es ambigua.

Por ejemplo, la ejecución del analizador semántico sobre el programa anterior emitirá los siguientes mensajes por pantalla:

```
Asignación z = z + 1 es ambigua (Instrucción 3).
```

[IMPLEMENTACIÓN] Implemente en ANTLR4 la solución propuesta.

PROBLEMA 2: LA RUPTURA DE CONTROL INALCANZABLE

Supongamos un lenguaje de programación secuencial con instrucción de ruptura de control. El objetivo del problema es detectar en los programas rupturas de control inalcanzables (y por tanto innecesarias). Los siguientes programas muestran rupturas inalcanzables.

Programa 1:

```
i=1;
while (i<=10){
   if (v==x){
      break;
      i=i+3;
      break;  //Inalcanzable
   }
   break;

   while (i>9)
      break;   //Inalcanzable
}
```

Programa 2:

```
i=1;
while (i<=10){
   if (v==x){
      break;
      i=i+3;
      break; //Inalcanzable
   }
   if (v==2) break;
   if (v==x) break;
   while (i>9)
      break;
}
```

Programa 3:

```
i=1;
while (i<=10){
   if (v==x){
      break;
      i=i+3;
      break; //Inalcanzable
   }
   break;
   break; //Inalcanzable
   while (i>9)
      break; //Inalcanzable
}
```

Suponga la siguiente gramática para el lenguaje:

```
programa : instrucciones EOF ;

instrucciones : instruccion instrucciones
              | instruccion
              ;

instruccion : asignacion
            | iteracion
            | seleccion
            | ruptura
            ;

asignacion : IDENT ASIG expr PyC ;

iteracion : WHILE PA expr PC  bloque   ;

seleccion : IF PA expr PC  bloque      ;

ruptura : BREAK PyC ;

bloque  : instruccion
        | LLA instrucciones LLC
        ;

expr : expr_suma
   ((MENOR|MAYOR|MENORIGUAL|MAYORIGUAL|IGUAL|DISTINTO) expr_suma)?
     ;

expr_suma : expr_mult ((MAS|MENOS) expr_mult)*
          ;

expr_mult : expr_base ((POR|DIV) expr_base)*
          ;

expr_base : NUMERO
          | IDENT
          | PA expr PC
          ;
```

SE PIDE:

[SOLUCIÓN] Diseño de gramática atribuida para un analizador semántico capaz de detectar rupturas de control inalcanzables en un programa.

[IMPLEMENTACIÓN] Implementar en ANTLR4 la solución propuesta.

PROBLEMA 3: 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 fuente, (b) datos fuente, (c) esquema destino y un conjunto de restricciones especificando la transferencia de datos entre la fuente y el destino. Todos los datos son de tipo cadena de caracteres. Las restricciones son implicaciones lógicas con antecedente formado por una tupla definida sobre una relación del esquema fuente y un consecuente formado por una tupla definida 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.

```
   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)
```

Cada dato `Xi` representa la instanciación de una variable cuantificada existencialmente en una restricción.

Suponga la siguiente gramática para LEXCHANGE:

```
entrada :   esquema_fuente datos_fuente esquema_destino restricciones EOF;

esquema_fuente : ESQUEMA FUENTE (signatura)+ ;

signatura: IDENT PA atributos PC ;

atributos: IDENT COMA atributos
         | IDENT
         ;

datos_fuente:  DATOS FUENTE (tupla)+ ;

esquema_destino: ESQUEMA DESTINO (signatura)+ ;

restricciones: RESTRICCIONES (restriccion)+ ;

restriccion: variables implicacion ;

variables: VAR vars PyC ;

vars: IDENT COMA vars
    | IDENT
    ;

implicacion: tupla IMPLICA tupla  ;

tupla: IDENT PA terminos PC ;

terminos: termino COMA terminos
        | termino
        ;

termino: IDENT
       | NUMERO
       ;
```

SE PIDE:

[SOLUCIÓN] Diseño de gramática atribuida para un analizador semántico de LEXCHANGE capaz de decidir si un programa LEXCHANGE es semánticamente correcto. Un programa es semánticamente correcto si cada tupla declarada en datos fuente pertenece a alguna relación definida en esquema fuente (mismo nombre y coincidencia en el número de argumentos).

[IMPLEMENTACIÓN] Implementación ANTLR4 de la solución propuesta..

PROBLEMA 4: LENGUAJE BLQ

BLQ es un lenguaje de programación basado en el concepto de bloque. Un bloque BLQ es una construcción formada por una declaración de variables y una secuencia de instrucciones. Las instrucciones en BLQ son de dos tipos: (a) asignaciones y (b) bloques. Las declaraciones de variables en un bloque se propagan a los bloques internos a éste pero no al revés.

Ejemplo de programa BLQ:

```
BLOQUE x,z;
   x = 1;
   m = 2*(x+7);
   BLOQUE x,m;
      x = m+z;
      n = 3;
   FBLOQUE
   z = m+1;
FBLOQUE
```

Suponga la siguiente gramática para BLQ.

```
programa: bloque EOF ;
bloque: BLOQUE variables instrucciones FBLOQUE ;
variables: vars PyC ;
vars: VAR COMA vars
    | VAR
    ;
instrucciones: (asignacion | bloque)*   ;
asignacion: VAR ASIG expr PyC ;
expr: expr1 (MAS|MENOS) expr
    | expr1
    ;
expr1: expr2 (POR|DIV) expr1
    | expr2
    ;
expr2: NUMERO      #NumExpr
     | VAR         #VarExpr
     | PA expr PC  #ParExpr
     ;
```

SE PIDE:

[SOLUCIÓN] Diseño de gramática atribuida para un analizador semántico de BLQ capaz de detectar el uso de variables no declaradas. Una variable se dice no declarada si no tiene declaración en el punto en el que ésta ocurre. Por ejemplo, la ejecución del analizador semántico sobre el programa anterior emitirá los siguientes mensajes por pantalla:

```
Variable m no declarada: instrucción m = 2*(x+7);
Variable n no declarada: instrucción n = 3;
Variable m no declarada: instrucción z = m+1;
```

[IMPLEMENTACIÓN] Implementación ANTLR4 de la solución propuesta.

PROBLEMA 5: LENGUAJE LEC

El siguiente ejemplo muestra un programa escrito en el lenguaje de expresiones contextualizadas (LEC):

```
CONTEXTO 1
   x = 5;
   y = 7;
   z = 1;
   x = 9;

CONTEXTO 3
   x = 9;
   z = 3;

DEFECTO  2

EXPRESIONES
   (x{10} - y{2}) * z;
   (x{1} + y{1}) * z{3} + y{3};
   (x - y) * m;
```

Un programa LEC es un conjunto de expresiones aritméticas contextualizadas. Los contextos son secciones del programa en las que se inicializan las variables con constantes numéricas (ej. `z = 1`). Un programa LEC puede tener uno o más contextos. Todo programa LEC tiene un contexto por defecto (ej. `DEFECTO  2`). Las expresiones LEC se construyen con operadores aritméticos, constantes numéricas y variables. Los operadores aritméticos son la suma (+), la resta (-) y el producto (*). Las constantes numéricas son enteros positivos o cero. La variable LEC puede referenciar a un contexto (ej. `z{3}` es una variable del contexto 3) o al contexto por defecto (ej. `x`).

El lenguaje LEC define un conjunto de restricciones semánticas que deben ser satisfechas para poder interpretar el programa LEC:

(1) Ninguna variable puede inicializarse más de una vez en un mismo contexto (ej. `x = 9` en nuestro programa de ejemplo).
(2) No se permite seleccionar un contexto por defecto inexistente (ej. `DEFECTO  2` en nuestro programa de ejemplo).
(3) No se permiten el uso de variables con contexto inexistentes (ej. `x{10}` o `y{2}` en nuestro programa de ejemplo).
(4) No se permite el uso de variables sin inicializar (ej. `y{3}` en nuestro programa de ejemplo).

Suponga la siguiente gramática para LEC:

```
programa : contextos defecto expresiones EOF ;

contextos : (contexto)+ ;
```

```
contexto : CONTEXTO NUMERO asignaciones ;

asignaciones : (asignacion)+ ;

asignacion : VARIABLE ASIG NUMERO PUNTOYCOMA ;

defecto : DEFECTO NUMERO ;

expresiones : EXPRESIONES (expresion PUNTOYCOMA)*  ;

expresion : expresion1 ((MAS|MENOS) expresion)? ;

expresion1 : expresion2 (POR expresion1)? ;

expresion2 : VARIABLE LLAVEABIERTA NUMERO LLAVECERRADA      #VarCtxt
           | VARIABLE                                       #Var
           | NUMERO                                         #Num
           | PARENTESISABIERTO expresion PARENTESISCERRADO  #ParExp
           ;
```

SE PIDE:

[SOLUCIÓN] Diseño de gramática atribuida para un analizador semántico de LEC.

[IMPLEMENTACIÓN] Implementación ANTLR4 de la solución propuesta.

Aclaraciones:

El analizador semántico dará mensajes por pantalla al detectar un error. El análisis semántico de nuestro programa de ejemplo generará los siguientes mensajes:

```
Error: variable x reinicializada en contexto 1
Error: contexto por defecto 2 inexistente
Error: contexto 10 inexistente para variable x en línea 14
Error: contexto 2 inexistente para variable y en línea 14
Error: variable y{3} sin inicializar en línea 15
```