INTERPRETACIÓN Y COMPILACIÓN

OBJETIVO

Problemas para consolidar el diseño e implementación de intérpretes y compiladores. En todos los problemas se suponen que las entradas al procesador no contienen errores de ningún tipo (ni léxico, ni sintácticos, ni semánticos)

PROBLEMA 1: EL LENGUAJE COND1

COND1 es un lenguaje de programación. El programa COND1 consta de declaraciones de variables enteras y secuencia de instrucciones. Las instrucciones en COND1 son de cuatro tipos: asignación, instrucción condicional, impresión por pantalla de variables y ruptura de control. A continuación se muestran dos ejemplos de programas COND1 junto a la impresión por pantalla generada por sus respectivas interpretaciones:

Ejemplo 1:

```
PROGRAMA

VARIABLES x,y;

INSTRUCCIONES
  x = 1;
  y = 2;
  mostrar(x,y);
  si (x>y) entonces
     ruptura;
     x=y;
  sino
     x=y;
     mostrar(x,y);
     ruptura;
     si (x==y) entonces
       x=10; y=20;
     finsi
  finsi
  mostrar(x,y);
```

```
(mostrar)
x = 1
y = 2
(mostrar)
x = 2
y = 2
(mostrar)
x = 2
y = 2
```

Ejemplo 2:

```
PROGRAMA

VARIABLES x,y;

INSTRUCCIONES
```

```
x = 1;
y = 2;
mostrar(x,y);
si (x<y) entonces
   ruptura;
   x=y;
sino
   x=y;
   mostrar(x,y);
   ruptura;
   si (x==y) entonces
     x=10; y=20;
   finsi
finsi
mostrar(x,y);
```

```
(mostrar)
x = 1
y = 2
(mostrar)
x = 1
y = 2
```

La interpretación del programa COND1 se basa en las siguientes restricciones:

(1) La declaración de variables inicializa éstas al valor 0.
(2) Las instrucciones se ejecutan secuencialmente.
(3) La asignación de un valor a una variable se realiza con el operador =.
(4) Las expresiones aritméticas pueden utilizar operadores suma, resta y multiplicación. Las expresiones booleanas o condiciones pueden utilizar el operador de igualdad, mayor, menor, mayor o igual, menor o igual y distinto.
(5) La ejecución de una instrucción condicional implica evaluar la condición y ejecutar el bloque de instrucciones correspondiente.
(6) La instrucción de ruptura provoca la interrupción en la ejecución del bloque de instrucciones en el que ocurre la ruptura.
(7) Las impresiones por pantalla tienen como argumento una secuencia de variables. La impresión muestra el valor de las variables correspondientes. Por ejemplo, mostrar(x,y) mostrará un mensaje del tipo (mostrar) x=..., y=...

SE PIDE:

Intérprete de programas COND1. Suponga la gramática dada en el Anexo.

[SOLUCIÓN] Diseño del intérprete.

[IMPLEMENTACIÓN] Implementación ANTLR4 de la solución propuesta.

Anexo

```
programa : PROGRAMA variables instrucciones EOF ;

variables : VARIABLES (lista_vars)? PUNTOYCOMA;

lista_vars : VAR COMA lista_vars
           | VAR
           ;

instrucciones : INSTRUCCIONES (lista_instrs)? ;

lista_instrs : instruccion (lista_instrs)? ;
```

```
instruccion : asignacion
            | condicional
            | ruptura
            | impresion
            ;

asignacion : VAR ASIG expr PUNTOYCOMA;

condicional :  SI PARENTESISABIERTO condicion PARENTESISCERRADO ENTONCES
                  lista_instrs (alternativa)?
               FINSI;

alternativa : SINO lista_instrs ;

ruptura : RUPTURA PUNTOYCOMA ;

impresion : MOSTRAR PARENTESISABIERTO vars PARENTESISCERRADO PUNTOYCOMA ;

vars : VAR COMA vars
     | VAR
     ;

condicion : expr MAYOR expr         #Mayor
          | expr MENOR expr         #Menor
          | expr IGUAL expr         #Igual
          | expr DISTINTO expr      #Distinto
          | expr MAYORIGUAL expr    #MayorIgual
          | expr MENORIGUAL expr    #MenorIgual
          ;

expr : expr MAS expr    #Mas
     | expr MENOS expr  #Menos
     | expr POR expr    #Por
     | VAR              #Var
     | NUM              #Num
     | PARENTESISABIERTO expr PARENTESISCERRADO  #ParExp
     ;
```

PROBLEMA 2: EL LENGUAJE LC

LC es un lenguaje de programación secuencial. El programa LC consta de declaraciones de variables y secuencia de instrucciones. Las variables en LC son enteras. Las instrucciones son de tres tipos: asignaciones, instrucciones condicionales e instrucciones de lectura. Toda variable se considera inicializada a cero tras su declaración. Las expresiones de LC son de tipo entero y se definen recursivamente desde constantes naturales, variables, operaciones aritméticas y expresiones entre paréntesis. Las instrucciones de lectura tienen como argumento una variable y supone una interrupción del programa por un valor suministrado desde el exterior de la máquina (via teclado por ejemplo) A continuación se muestra un programa LC de ejemplo.

```
PROGRAMA
   VARIABLES
      x,y,z;
   INSTRUCCIONES
      si (z > 0) entonces
         x = 1;
         y = 2;
      fsi
      leer(x);
      si (x > y) entonces
        si (y < 7) entonces
           y=z+1;
           z = 2;
        si no
           y = x;
           z = 1;
        fsi
        z = 12;
      sino
        z=y+1;
        y = 2;
      fsi
```

SE PIDE:

Compilador de programas LC. El compilador debe generar un programa LC equivalente al programa fuente pero optimizado. La optimización consiste en eliminar las instrucciones condicionales innecesarias. La idea es que toda condición del programa fuente sea evaluada y decidir si el bloque de instrucciones correspondiente se debe o no incluir en el programa compilado. La lectura de una variable (p.e. leer(x)) no permite predecir su valor en tiempo de compilación. Por ejemplo, la compilación del programa LC de ejemplo producirá:

```
PROGRAMA
   VARIABLES
      x,y,z;
   INSTRUCCIONES
      leer(x);
      si (x > y) entonces
         y=z+1;
         z = 2;
         z = 12;
      sino
        z=y+1;
        y = 2;
      fsi
```

[COMPRENSIÓN DEL PROBLEMA] Determine los programas compilados para los ejemplos propuestos en Anexo I.

En los siguientes apartados suponga la gramática dada en el Anexo II para LC.

[SOLUCIÓN] Diseño del compilador.

[IMPLEMENTACIÓN] Implementación ANTLR4 de la solución propuesta.

ANEXO I

```
PROGRAMA

VARIABLES
   x,y,z;
INSTRUCCIONES
   leer(x);
   x = 7;
   si (x < 5) entonces
      si (x < 7) entonces
         y=z+1;
         z = 2;
      sino
         z = 3;
      fsi
      z = 12;
   sino
      leer(x);
      si (x < 10) entonces
         z=y+1;
         y = 2;
      sino
         y = z;
      fsi
   fsi
```

```
PROGRAMA

VARIABLES
   x,y,z;
INSTRUCCIONES
   x = 7;
   x = x + 1;
   si (x > 5) entonces
      y=1;
   sino
      z = 2;
   fsi
   z = 12;
   si (x < 10) entonces
      z=y+1;
      y = 2;
   fsi
```

ANEXO II:

Gramática LC

```
programa : PROGRAMA variables instrucciones EOF ;

variables : VARIABLES lista_vars PUNTOYCOMA;

lista_vars : IDENT COMA lista_vars #Vars
           | IDENT                 #Var
           ;

instrucciones : INSTRUCCIONES (lista_instrs)? ;

lista_instrs : instruccion (lista_instrs)? ;

instruccion : asignacion
            | condicional
            | leer
            ;

asignacion : IDENT ASIG expr PUNTOYCOMA;

condicional :  SI PARENTESISABIERTO condicion PARENTESISCERRADO ENTONCES
                  lista_instrs (alternativa)?
               FINSI;

alternativa : SINO lista_instrs ;

leer : LEER PARENTESISABIERTO IDENT PARENTESISCERRADO PUNTOYCOMA ;

condicion : expr MAYOR expr
          | expr MENOR expr
          | expr IGUAL expr
          | expr DISTINTO expr
          | expr MAYORIGUAL expr
          | expr MENORIGUAL expr;

expr : expr MAS expr
     | expr MENOS expr
     | expr POR expr
     | IDENT
     | NUMERO
     | PARENTESISABIERTO expr PARENTESISCERRADO;
```

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. A continuación se muestra un programa de ejemplo CALCPROG y su interpretación.

```
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
```

```
Expresión:  3 + (4 + 1) vale 8
Asignación: a vale 2
Función: f(a) vale  10 * a
Expresión:  f(a) vale 20
Función: g(x) vale  10*f(x) + a
Expresión:  g(3) vale 302
Expresión:  f(f(2)) vale 200
Expresión:  f(a+1) vale 30
```

Gramática CALCPROG:

```
programa : (orden PUNTOYCOMA)* EOF  ;
orden: expresion  #OrdenExpr
     | asignacion #OrdenAsig
     | funcion    #OrdenFunc
     ;
asignacion :  IDENT ASIG expresion ;
funcion : IDENT PARENTESISABIERTO IDENT PARENTESISCERRADO ASIG expresion ;
expresion : PARENTESISABIERTO expresion PARENTESISCERRADO        #ParExpr
          | MENOS expresion
#UnarioMenosExpr
          | expresion POR expresion                              #PorExpr
          | expresion MAS expresion                              #MasExpr
          | expresion MENOS expresion                            #MenosExpr
          | NUMERO                                               #NumExpr
          | IDENT                                                #IdentExpr
          | IDENT PARENTESISABIERTO expresion PARENTESISCERRADO  #LlamadaExpr
          ;
```

SE PIDE:

[SOLUCIÓN] Diseño del intérprete de programas CALCPROG.

[IMPLEMENTACIÓN] Implementación ANTLR4 de la solución propuesta.

PROBLEMA 4: DIAGRAMAS DE ESTADOS

El lenguaje de los diagramas de estados es una notación que permite especificar software mediante secuencias de estados. Los diagramas de estados contienen un único estado inicial y al menos un estado final. Cada estado especifica una secuencia de asignaciones (se permite la secuencia vacía). La asignación permite cambiar el valor de una variable (entera) con el valor de una expresión aritmética (entera). El cambio de estado se expresa mediante el uso de transiciones. La transición está anotada con una condición booleana. Se requiere el cumplimiento de la condición para hacer efectivo el cambio de estado. Un estado se considera final si no es origen de ninguna transición. Los estados no finales son origen de transiciones con condiciones excluyentes y complementarias. El diagrama de estado no contiene ciclos.

A continuación, se muestra un diagrama de estado que describe el comportamiento de un software capaz de ordenar el valor de tres variables enteras de la manera siguiente: en a queda el valor menor, en b el siguiente menor y en c el componente mayor de los tres.

```
DIAGRAMA diagrama3

VARIABLES a b c d

ESTADOS
 estado1 a=6; b=1; c=0; d=0;
 estado2
 estado3
 estado4
 estado5
 estado6 d=c; c=a; a=d;
 estado7
 estado8 d=c; c=a; a=d; d=c; c=b; b=d;
 estado9 d=c; c=b; b=d;
 estado10 d=c; c=b; b=d; d=c; c=a; a=d;
 estado11 d=b; b=a; a=d;

INICIAL estado1

TRANSICIONES
 estado1 estado2 b>a;
 estado1 estado3 a>=b;
 estado2 estado4 b>=c;
 estado2 estado5 c>b;
 estado3 estado6 b>=c;
 estado3 estado7 c>b;
 estado4 estado8 a>c;
 estado4 estado9 c>=a;
 estado7 estado10 a>c;
 estado7 estado11 c>=a;
```

El diagrama del ejemplo contiene 11 estados. El estado inicial es estado1 y los estados finales son estado5, estado6, estado8, estado9, estado10 y estado11 (no son origen de ninguna transición). Los estados estado2, estado3, estado4, estado5 y estado7 no contienen ninguna asignación.. El diagrama de ejemplo define 6 secuencias de estados: estado1-estado3-estado7-estado10, estado1-estado3-estado6, estado1-estado2-estado5, estado1-estado2-estado4-

estado9, estado1-estado3-estado7-estado11 y estado1-estado2-estado4-estado8. Para realizar la traducción a Java es importante remarcar que todas las secuencias comienzan en un mismo estado (el estado inicial) y son excluyentes entre sí (sólo se ejecutará una de ellas) debido a la restricciones impuestas sobre las condiciones en las transiciones.

SE PIDE: Compilador de diagrama de estados a Java. Suponga gramática dada en Anexo.

[COMPRENSIÓN DEL PROBLEMA] Determine el programa Java compilado para el diagrama de estado del ejemplo.

[SOLUCIÓN] Diseño del compilador.

[IMPLEMENTACIÓN] Implementación Antlr4 de la solución propuesta.

Anexo

```
parser grammar Anasint;
options{
   tokenVocab=Analex;
}

diagrama: DIAGRAMA IDENT variables estados inicial transiciones EOF;

variables: VARIABLES vars ;

vars: IDENT vars
    |
    ;
estados: ESTADOS (estado)* ;

estado: IDENT asignaciones ;

asignaciones: asignacion asignaciones
            |
            ;
asignacion: IDENT ASIG term PUNTOYCOMA ;

inicial: INICIAL IDENT ;

transiciones: TRANSICIONES (transicion)* ;

transicion: IDENT IDENT condicion PUNTOYCOMA ;

condicion: condicion Y condicion                              #condY
         | condicion O condicion                              #condO
         | NO condicion                                       #condNo
         | PARENTESISABIERTO condicion PARENTESISCERRADO      #condPar
         | term MAYOR term                                    #condMayor
         | term MENOR term                                    #condMenor
         | term IGUAL term                                    #condIgual
         | term DISTINTO term                                 #condDistinto
         | term MAYORIGUAL term                               #condMayorIgual
         | term MENORIGUAL term                               #condMenorIgual
         ;

term: NUMERO | IDENT ;
```

PROBLEMA 5: 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 datos fuente, (b) datos fuente, (c) esquema de datos 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.

SE PIDE: Intérprete de programas LEXCHANGE que genere tuplas de la base de datos destino en formato fichero de texto. Suponga gramática dada en Anexo.

[SOLUCIÓN] Diseño del intérprete

[IMPLEMENTACIÓN] Implementación Antlr4 de la solución propuesta.

Anexo

```
parser grammar Anasint ;
options{
   tokenVocab = Analex;
}

entrada : esquema_fuente datos_fuente esquema_destino restricciones EOF ;

esquema_fuente : ESQUEMA FUENTE (esquema)+ ;

esquema: IDENT PA atributos PC ;

atributos : IDENT COMA atributos
          | IDENT
          ;

datos_fuente:  DATOS FUENTE (tupla)+ ;

esquema_destino: ESQUEMA DESTINO (esquema)+ ;

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
       ;
```

PROBLEMA 6: EL LENGUAJE OWL

OWL es un lenguaje diseñado para expresar ontologías en la web. Una ontología es un conjunto de declaraciones que describen un dominio de interés. En este ejercicio se consideran 4 tipos de declaraciones:

Asertos de clase: formalizan la pertenencia de un individuo a una clase de individuos.

Ejemplo:
```
ClassAssertion(Person  Mary)  // Mary es una persona
ClassAssertion(Woman  Mary)   // Mary es una mujer
```

Asertos de propiedad: formalizan una determinada relación entre dos individuos.

Ejemplo: `ObjectPropertyAssertion(hasWife John Mary)  // Mary es la esposa de John`

Jerarquía de clase: formalizan una relación de inclusión entre dos clases de individuos.

Ejemplo: `SubClassOf(Mother Woman)  //Toda madre es una mujer`

Definición de clase: formaliza la definición de una clase primitiva.

Ejemplo: `EquivalentClasses(Person Human) //persona (clase primitiva) equivale a ser humano`

Aparte de las clases primitivas tales como `Woman` o `Person`, OWL dispone de operadores adicionales para formalizar el concepto clase de individuo. Estos operadores son **intersección, unión, complemento** y **restricciones universales o existenciales sobre propiedad**:

Ejemplos:
```
// individuos que son mujeres y madres
      ObjectIntersectionOf(Woman Mother)
// individuos que son padres, madres o abuelos
      ObjectUnionOf(Father Mother GrandFather)
// individuos que no son padres
      ObjectComplementOf(Father)
// individuos con algún hijo feliz
      ObjectSomeValuesFrom(hasChild Happy)
// individuos con todos sus hijos felices
      ObjectAllValuesFrom(hasChild Happy)
```

Ejemplo de ontología OWL:

```
   ClassAssertion(Person  Mary)  // Mary es una persona
   SubClassOf(Woman Person)  //Toda mujer es una persona
   SubClassOf(Mother Woman)   //Toda madre es una mujer
   ClassAssertion(Person John) // John es una persona
// Madre feliz es una madre con todos sus hijos felices
   SubClass(HappyMother
     ObjectIntersectionOf(Mother ObjectAllValuesFrom(hasChild Happy)))
```

Gramática ANTLR4 para OWL,

```
parser grammar Anasint ;
options{
   tokenVocab = Analex;
}

ontologia :  ONTOLOGY  (declaracion)* EOF    ;

declaracion : aserto_clase
            | aserto_propiedad
            | subclase
            | definicion
            ;

aserto_clase : CLASS_ASSERTION PARENTESISABIERTO individuo clase
PARENTESISCERRADO  ;

aserto_propiedad : OBJECT_PROPERTY_ASSERTION PARENTESISABIERTO propiedad
individuo individuo  PARENTESISCERRADO   ;

subclase : SUB_CLASS_OF PARENTESISABIERTO clase clase  PARENTESISCERRADO   ;

definicion : EQUIVALENT_CLASSES PARENTESISABIERTO clase_primitiva clase
PARENTESISCERRADO    ;

individuo : IDENT     ;

propiedad : IDENT     ;

clase : clase_primitiva
      | restriccion_existencial
      | restriccion_universal
      | union
      | interseccion
      | complementario
      ;
clase_primitiva : IDENT  ;

restriccion_existencial :  OBJECT_SOME_VALUES_FROM PARENTESISABIERTO
propiedad clase PARENTESISCERRADO ;

restriccion_universal :  OBJECT_ALL_VALUES_FROM PARENTESISABIERTO propiedad
clase PARENTESISCERRADO  ;

union :  OBJECT_UNION_OF PARENTESISABIERTO clase (clase)+ PARENTESISCERRADO
;

interseccion :  OBJECT_INTERSECTION_OF PARENTESISABIERTO clase (clase)+
PARENTESISCERRADO  ;

complementario :  OBJECT_COMPLEMENT_OF PARENTESISABIERTO clase
PARENTESISCERRADO      ;
```

SE PIDE:

Implementación ANTLR4 de compilador de ontología OWL a teoría lógica de primer orden. La compilación de ontología OWL se define como la traducción de cada declaración en formula lógica de primer orden equivalente. Esta traducción se define formalmente de la siguiente manera.

(a) `ClassAssertion(C I)` equivale a la formula F(`C,I`)
(b) `ObjectPropertyAssertion(P I1 I2)` equivale al predicado *`P(I1,I2)`*
(c) `SubClassOf(C1 C2)` equivale a la formula lógica *forall* `x` *(*F(`C1,x`) *implies* F(`C2,x`)*)*.
(d) `EquivalentClasses(C1 C2)` equivale a la formula lógica *forall* `x` *(*F(`C1,x`) *equiv* F(`C2,x`)*)*.

Siendo

F(`C,x`) = *`C(x)`* si `C` una clase primitiva.
F(`ObjectSomeValuesFrom(P C), x`)= *exists* `y` *(`P(x,y)` and* F(`C,y`)*)* siendo `y` una variable nueva.
F(`ObjectAllValuesFrom(P C), x`)= *forall* `y` *(`P(x,y)` implies* F(`C,y`)*)* siendo `y` una variable nueva.
F(`ObjectUnionOf(C1,…,Cn), x`)= (F(`C1, x`) *or … or* F(`Cn, x`)).
F(`ObjectIntersectionOf(C1,…,Cn), x`)= (F(`C1, x`) *and … and* F(`Cn, x`)).
F(`ObjectComplementOf(C), x`)= *not* F(`C, x`).

Ejemplo de traducción:

(ontología)

```
Ontology

       ClassAssertion(Person  Mary)
        SubClassOf(Woman Person)
        SubClassOf(Mother Woman)
        ClassAssertion(Person John)
        EquivalentClasses(HappyMother
            ObjectIntersectionOf(Mother ObjectAllValuesFrom(hasChild Happy)))
```

(teoría primer orden)

```
Theory

      Person(Mary)
      forall x0 (Woman(x0) implies Person(x0))
      forall x0 (Mother(x0) implies  Woman(x0))
      Person(John)
      forall x0 (HappyMother(x0) equiv  Mother(x0)  and  forall x1 (hasChild(x0,x1) implies
                                                          Happy(x1)))
```