Práctica 6 - Tablas de Símbolos

Procesadores de Lenguajes - 4º curso - Grado en Ingeniería Informática

1 Objetivos

- Construir estructuras de datos válidas para gestionar tablas de símbolos en lenguajes estructurados en bloques utilizando una pila de tablas Hash.
- Elaborar una gramática adecuada para expresiones numéricas con operandos de tipos diferentes.
- Utilizar CUP y JFLEX para generar un analizador sintáctico ascendente LALR(1) para dicha gramática.
- Introducir acciones semánticas para gestionar una tabla de símbolos.

2 Tablas de Símbolos: pila de tablas Hash

Entre las diferentes posibilidades de implementar tablas de símbolos se propone la construcción de una estructura de datos consistente en una pila de tablas Hash basada en el empleo de tres clases de objetos:

- **Simbolo** Compuesto por tres atributos:
  - Nombre (java.lang.String)
  - Tipo de dato (java.lang.String)
  - Valor numérico asociado (java.lang.Number)
- **PilaTablasSimbolos**
  - Extiende a la clase java.util.LinkedList<TablaSimbolos> (lista doblemente enlazada que implementa los interfaces List y Deque) para disponer de la funcionalidad push y pop propia de las pilas LIFO.
  - Tanto su constructor como los métodos push y pop utilizan los métodos de la clase LinkedList para crear una pila, enviar una tabla de símbolos a la parte superior de la pila y extraer una tabla de símbolos de la parte superior de la pila, respectivamente.

- **TablaSimbolos**
  - Extiende a la clase HashMap<String, Simbolo> para almacenar los símbolos en mapas hash.
  - Contiene al menos un atributo,pila, de tipo PilaTablasSimbolos, necesario para conocer la pila en la que están almacenadas las demás tablas de símbolos (correspondientes a otros bloques).
  - Su constructor utiliza el constructor de la clase HashMap para crear una tabla de símbolos. Por otra parte, dicho constructor recibe como argumento un objeto de tipo PilaTablasSimbolos que se utiliza para asignar valor al atributo pila.
  - El método put(String clave, Simbolo valor) utiliza una llamada al método homónimo de esa superclase para almacenar la pareja de valores <clave,valor> en la tabla de símbolos. Se utiliza como clave el nombre de un símbolo y como valor un objeto de la clase Símbolo. Este método también sirve para modificar el valor asociado a una clave en una tabla de símbolos, pues si ya existe un objeto almacenado con esa clave, entonces se sobrescribe su valor en vez de crearse una entrada nueva.
  - El método Simbolo get(String clave) busca un símbolo en la tabla de símbolos. Si no lo encuentra, ha de buscarlo en las tablas de símbolos de bloques en los que esté anidado el bloque actual (almacenadas en la pila LIFO). El método devuelve el símbolo cuando lo encuentra en la jerarquía de tablas de símbolos, y devuelve **null** cuando no lo encuentra.
    Métodos útiles de la clase HashMap:
    * containsKey(clave) – Devuelve **true** si en la tabla de símbolos existe una entrada con la clave indicada
    * get(clave) – Devuelve el valor asociado a clave en la tabla de símbolos.
      Ejemplo de función get() utilizando un iterador para recorrer las tablas de símbolos almacenadas en la pila:

```
public Simbolo get (String clave) {
   Simbolo simb_encontrado=null;
   if (this.containsKey(clave))
      simb_encontrado=super.get(clave);
   else{
      TablaSimbolos ts=null;
      Iterator <TablaSimbolos> it=pila.iterator();
      while((it.hasNext()) && (simb_encontrado==null)) {
        ts=it.next();
        if (ts.containsKey(clave))
            simb_encontrado=ts.get(clave);
        }
   }
```

```
return simb_encontrado;
}
```

Todas las clases han de implementar el método **public** String toString() que permita utilizar la instrucción System.out.println(objeto) para visualizar el contenido de un símbolo, de una tabla de símbolos y de una pila de tablas de símbolos.

3 Especificaciones del lenguaje

- Tipos de datos básicos: números enteros y números reales (se admite notación científica).
- Tipos de agrupaciones de datos: arrays (de una o más dimensiones) de datos de tipos básicos.
- Operaciones soportadas: suma, resta, producto, división, potenciación y raíz cuadrada. Se admiten los operadores unarios $+$ y $-$.
- El programa consta de bloques delimitados por llaves.
- Un bloque consta de una secuencia de sentencias y/o bloques anidados ({bloque}).
- Una sentencia consiste en la declaración de una variable, en la asignación del valor de una expresión a una variable o en la llamada a la función print(variable).
- Dentro de una expresión se pueden utilizar variables.
- Una expresión que combina valores enteros y reales da lugar a un resultado real.
- Se puede asignar un valor entero a una variable real (se hace un *cast* automático).
- No se contempla la asignación de valor a agrupaciones de datos ni su utilización dentro de una expresión.
- Las expresiones utilizan notación infija, respetando la precedencia de las operaciones.
- Se acepta la utilización de paréntesis en múltiples niveles.
- Una expresión finaliza con el carácter punto y coma.

  Ejemplo de archivo de entrada:

```
{
int var1;
float var2;
int[3][2] array1;
var1=2;
var2=-1.5;
```

```
print(var1);
        {
        float var2;
        var2=var1+1;
        print(var1);
        }
        {
        float[4][5][6] array2;
        print(array2);
        }
print (var1);
}
```

3.1 Especificaciones léxicas

El análisis léxico se encargará de procesar archivos con los siguientes tokens válidos:

TIPO_INT Cadena de caracteres `"int"` (declaración de variable entera)

TIPO_FLOAT Cadena de caracteres `"float"` (declaración de variable real)

INTEGER Número entero

FLOAT Número real

ID Identificador de variables (Nota: se puede utilizar el patrón [:jletter:][:jletterdigit:]*)

PLUS Signo $'+'$

MINUS Signo $'-'$

TIMES Signo $'*'$

DIV Signo $'/'$

POW Signo $'\hat{}'$ (potenciación)

SQRT Operación raíz cuadrada (*sqrt*)

PRINT Función imprimir el valor de una variable (almacenado en la tabla de símbolos)

LEFT_PARENTHESIS Paréntesis izquierdo $'('$

RIGHT_PARENTHESIS Paréntesis derecho $')'$

LEFT_SQBRACKET Corchete izquierdo $'['$

RIGHT_SQBRACKET Corchete derecho $']'$

LEFT_BRACE Llave izquierda ('{')

RIGHT_BRACE Llave derecha ('}')

ASSIGN Signo '='

SEMICOLON Terminador de expresión (';')

El analizador léxico permitirá el uso de mayúsculas y minúsculas, indistintamente, y deberá encargarse de eliminar todos los espacios en blanco del archivo de entrada así como de detectar los errores de tipo léxico.

3.2 Especificación sintáctica

A continuación se propone una gramática para el lenguaje descrito:

```
programa → bloque
bloque → { decls instrs }

decls → decls decl | ϵ
decl → tipo id;

tipo → t_basico t_componente
t_basico → tipo_int | tipo_float
t_componente → [num_int] t_componente | ϵ

instrs → instrs instr | ϵ
instr → bloque | factor = expr ; | print(factor);

factor → id
expr →  num_int | num_float| factor
      | expr + expr | expr − expr | expr ∗ expr |expr / expr
      | expr ^ expr | sqrt(expr) | +expr | −expr | (expr)
```

4 Especificaciones semánticas

Se puede gestionar la tabla de símbolos incrustando acciones semánticas en los cuerpos de las producciones gramaticales.

- El analizador sintáctico debe tener entre sus atributos un objeto de cada una de las clases vinculadas a las tablas de símbolos (Simbolo simb; TablaSimbolos ts; PilaTablasSimbolos pila). Nota: se pueden declarar en el bloque parser code del archivo de especificaciones sintácticas Cup.
- Antes de procesar la primera producción, se crea una pila de tablas de símbolos

```
programa →          {: pila=new PilaTablasSimbolos(); :}
            bloque
```

- Tras comenzar un bloque, se almacena la tabla de símbolos del bloque de nivel superior y se crea una nueva para gestionar los símbolos del ámbito del bloque que comienza. Después del terminal que indica el fin del bloque, se descarta la tabla de símbolos y se recupera la tabla de símbolos del bloque de nivel superior, que se encuentra en la parte superior de la pila.

```
bloque →'{'      {: if (ts!=null)
                       pila.push(ts);
                    ts=new TablaSimbolos(pila); :}
          decls instrs '}'   {: if (pila.size()>0)
                                   ts=pila.pop();
                               else
                                   ts=null; :}
```

- Como resultado de la declaración de una variable, se crea un símbolo y se inserta en la tabla de símbolos actual.

```
decl → tipo id;      {: simb=new Simbolo(id0, t1, 0);
                        ts.put(simb.nombre,simb); :}
```

- Los no terminales tipo, t_basico y t_componente tienen como atributos una cadena de caracteres. En el caso de un array, la cadena tiene una estructura como la del siguiente ejemplo: **int** [2][3][4] x; → tipo: `"Array de (2x3x3, TIPO_INT)"`

```
tipo → tipo_basico:tb tipo_componente:tc
         {:if (tc==null)
            RESULT=tb;
          else
            RESULT="Array de ("+tipo_componente.valor+   \
                               ", "+tipo_basico.valor+")");:}


t_basico → tipo_int {:RESULT="TIPO_INT";:}
         | tipo_float {:RESULT="TIPO_FLOAT";:}

t_componente → [num_int] t_componente:tc
         {:if (tc!=null)
              RESULT=""+num_int.valor+ "x" + tc;
            else
              RESULT=""+num_int.valor;:}
```

- La acción semántica correspondiente a la producción de la instrucción **print**(factor) se encarga de comprobar que el factor está en la tabla de símbolos y, en su caso, mostrarlo por pantalla. Si el factor corresponde a una variable no declarada, se muestra un mensaje de error. Nota: En Cup, el nombre de una variable terminado en 'left' o en 'right' devuelve el valor del campo correspondiente del objeto Symbol devuelto por el analizador léxico).

```
instr → print(factor:f1);
  {:
  try{
    simb=ts.get(f1.nombre);
    if (simb!=null)
      System.out.println("Simbolo: "+simb);
    else
      System.out.println("Variable no declarada: "+ f1.nombre + \
               " [linea: "+ f1left  +", col: "+ f1right +"] \n");
  }catch(Exception exc){}
  :}
```

- La producción de una instrucción de asignación implica comprobar, antes de realizar la asignación, que el tipo del factor y el tipo de la expresión sean iguales. Si el factor es una variable real y la expresión es un valor entero, entonces se hace un *cast* y se completa la operación de asignación. En cualquier otro caso se muestra un mensaje de error.

```
instr → factor:f1=expr:e;
{:
try{
   simb=ts.get(f1.nombre);
   if (simb!=null){
     if (f1.tipo.equals(e.tipo)){
        Number nuevo_valor=e.valor;
        Simbolo simb_aux=new Simbolo(f1.nombre,f1.tipo, nuevo_valor);
        ts.put(f.nombre,simb_aux);
     }
     else if (f1.tipo.equals("TIPO_FLOAT") && e.tipo.equals("TIPO_INT")){
        Number nuevo_valor=e.valor.floatValue();
        Simbolo simb_aux=new Simbolo(f1.nombre, f1.tipo, nuevo_valor);
        ts.put(f1.nombre,simb_aux);
     }
     else{
        System.out.println("Incompatibilidad de tipos: variable " \
           +f1.nombre+" no es de tipo "+ e.tipo+. \
           " [linea: "+ f1left  +", col: "+ f1right +"] \n");
     }
```

```
  }
  else
     System.out.println("Variable no declarada: "+ \
       f1.nombre +" [linea: "+ f1left  +", col: "+ f1right +"] \n");
}
catch(Exception exc){}
:}
```

- La producción de un factor implica devolver el objeto Símbolo que está asociado al identificador en la tabla de símbolos. Si no se encuentra, se muestra un mensaje de error.

```
factor → id:id1
{: simb=ts.get(id1);
   if (simb!=null)
      RESULT=simb;
   else{
      System.out.println("Variable no declarada: "+ id1 +. \
          " [linea: "+ id1left  +", col: "+ id1right +"] \n");
      RESULT=null;
   }
:}

expr → factor:f1
{:
try{
simb=ts.get(f1.nombre);
if (simb!=null)
  RESULT=simb;
else{
  System.out.println("Variable no declarada: "+ \
     f1 +" [linea: "+ f1left  +", col: "+ f1right +"] \n");
  RESULT=null;
}
}
catch(Exception exc){RESULT=null;}
:}
```

- La producción de una expresión a partir de un número (num_float o num_int) se encarga de devolver un objeto Símbolo con ese valor.

```
expr →  num_float:n1
{:RESULT=new Simbolo("","TIPO_FLOAT",n1.floatValue());:}
```

- Las demás formas de producir expresiones devuelven un objeto de la clase Simbolo con información del valor y tipo de su resultado.

```
expr → −expr:e1
{:
try{
  if (e1.tipo.equals("TIPO_FLOAT")){
    float valor= e1.valor.floatValue();
    RESULT= new Simbolo("","TIPO_FLOAT",valor);
  }
  else{
    int valor= −e1.valor.intValue();
    RESULT= new Simbolo("","TIPO_INT",valor);
  }
}catch (Exception exc){RESULT=null;}
:}

expr →  expr:e1 + expr:e2
{:
try{
  if ((e1.tipo.equals("TIPO_FLOAT"))|| \
        (e2.tipo.equals("TIPO_FLOAT"))){
    float res= e1.valor.floatValue() + e2.valor.floatValue();
    RESULT= new Simbolo("","TIPO_FLOAT",res);
  }
  else{
    int res= e1.valor.intValue() + e2.valor.intValue();
    RESULT= new Simbolo("","TIPO_INT",res);
  }
}catch (Exception exc)
    RESULT=null;
:}

expr → sqrt(expr:e1)
{:
try{
  float valor= Float.valueOf((float)Math.sqrt(e1.valor.floatValue()));
  RESULT= new Simbolo("","TIPO_FLOAT",valor);
}catch (Exception exc){RESULT=null;}
:}
```

5 Instrucciones

1. Construir una aplicación basada en JFlex y CUP utilizando como referencia la gramática propuesta y adaptándola, si fuese necesario, para que cumpla con las especificaciones del lenguaje.

2. Cuando se detecte un error léxico se mostrará un mensaje con el siguiente formato:

   ```
   [Lex] Error léxico en línea ##, columna ##
   ```

3. Los lexemas que den lugar a errores léxicos no se enviarán al analizador sintáctico, por lo que no tendrán efecto en el resultado final.

4. Al comienzo y final de cada bloque, la acción semántica se encargará de mostrar por pantalla los mensajes <INICIO DE BLOQUE> y <FIN DE BLOQUE>, respectivamente.

5. El analizador sintáctico se encargará de calcular los valores numéricos resultado de las operaciones en las que no existan errores de sintaxis. Además, ha de gestionar adecuadamente la estructura de tablas de símbolos, insertando símbolos y actualizando sus contenidos a medida que cambian los valores de los símbolos.

6. Recuperación ante errores de sintaxis. Cuando el analizador sintáctico encuentre un token que no se adecúe a la gramática, mostrará un mensaje con el formato que se indica a continuación e intentará recuperarse tras el siguiente carácter punto y coma.

   ```
   [Parser] Error de sintaxis: 'tipo_token'  en línea ##, columna ##
            Tipos de tokens esperados: [lista_tokens_esperados]
   ```

7. Ejemplo de funcionamiento:
   Contenido del archivo de entrada:

   ```
   {
   int x;
   float y;
   x=8;
   print(x);
           {
           float[3][5] array;
           int y;
           print(array);
           y=x+2;
           print(x);
           print(y);
           }
   y=y+x+2.1;
   print(y);
   }
   ```

   Salida por pantalla:

```
<INICIO DE BLOQUE>
[x; TIPO_INT; 8]

<INICIO DE BLOQUE>
[array; Array de (3 x 5, TIPO_FLOAT); 0]
[x; TIPO_INT; 8]
[y; TIPO_INT; 10]
<FIN DE BLOQUE>

[y; TIPO_FLOAT; 10.1]
<FIN DE BLOQUE>

<FIN DE BLOQUE>
```

Se entregarán, al menos, los siguientes archivos:

1. Especificación léxica para JFlex (.jflex)
2. Especificación sintáctica para Cup (.cup)
3. Archivos generados por JFlex y Cup (.java)
4. Archivos con el resto de clases elaboradas (.java)
5. Un archivo de entrada