AULAS TEÓRICO-PRÁTICAS DE COMPILADORES

2º semestre de 2002/2003

FICHA Nº 9 (3 horas)

Esta ficha pretende que aprendam a utilizar o JJTree em conjunto com o JavaCC [1] para gerar *parsers* e árvores sintácticas.

Após a utilização do JavaCC nas duas últimas aulas, é agora tempo de aprenderem como se pode gerar automaticamente a árvore sintáctica. Para tal utilizaremos o JJTree (é uma ferramenta integrada no pacote de software JavaCC). O JJTree é uma ferramenta de pré-processamento que permite gerar um ficheiro para o JavaCC que, para além da descrição da gramática, integra código Java para a geração da árvore sintáctica. O ficheiro de entrada do JJTree é um ficheiro que especifica a gramática do mesmo modo que para o JavaCC e que adicionalmente inclui directivas para a geração dos nós da árvore (é utilizado jjt como extensão do ficheiro origem).

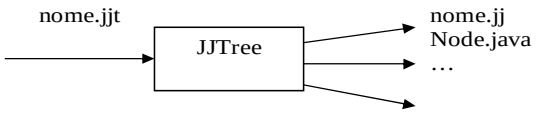

Figura 1. Entradas e saídas do JJTree.

1 Exemplo

Vamos considerar a gramática da FICHA 8 e vamos supor que pretendemos realizar um programa que calcule as mesmas expressões, desta vez gerando a árvore sintáctica e efectuando os cálculos sobre a árvore.

Para gerar a árvore sintáctica vamos alterar a extensão do ficheiro da gramática original para jjt e vamos adicionar as directivas que indicam ao JJTree para criar a árvore. O ficheiro apresentado de seguida contém as modificações introduzidas (é utilizado o método dump() para imprimir no ecrã a árvore gerada por cada expressão introduzida).

Ficheiro Calculator.jjt:

```
options
{
   LOOKAHEAD=2;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
   public static void main(String args[]) throws ParseException {
      Calculator parser = new Calculator(System.in);
      int i=0;
      while (true) {
       // the function will return the root node of the Syntax Tree
          SimpleNode rootNode = parser.parseOneLine();
```

```
        //print the Syntax Tree
        rootNode.dump("Syntax Tree "+(i++)+": ");
      }
    }
}

PARSER_END(Calculator)

SKIP :
{
   " "
   |  "\r"
   |  "\t"
}

TOKEN:
{
   < INTEGER: (["0"-"9"])+ >
   |  < EOL: "\n" >
}

SimpleNode parseOneLine() #Root : {}     // diz ao JJTree para criar o nó Root
{
  expr() <EOL>     {return jjtThis;}  // retorna o nó da árvore construído neste procedimento
  | <EOL>
  | <EOF>          { System.exit(-1); }
}

void expr():
{
}
{
  term()
   (
      "+" expr() #Add(2) // diz ao JJTree para criar um nó Add que tem dois filhos
    | "-" expr() #Sub(2) // diz ao JJTree para criar um nó Sub que tem dois filhos
   )*
   | "(" expr() ")"
}

void term(): {}
{
   unary()
   (
      "*" term() #Mult(2) // diz ao JJTree para criar um nó Mult que tem dois filhos
   |  "/" term() #Div(2)  // diz ao JJTree para criar um nó Div que tem dois filhos
   )*
   | "(" expr() ")"
}

void unary(): {}
{
   "-" <INTEGER>
   | <INTEGER>
}
```

Em seguida deve fazer-se:

```
jjtree Calculator.jjt
```

Gerar o código Java com o JavaCC:

```
javacc Calculator.jj
```

Compilar o código Java gerado:

```
Javac *.java
```

Executar o analisador sintáctico:

```
Java Calculator
```

Verifique de seguida as árvores geradas para um conjunto de expressões introduzidas.

Contudo, a criação da árvore sintáctica não é suficiente para nós. Primeiro, a árvore gerada não armazena os valores dos números inteiros lidos. Segundo, é necessário programar o procedimento que atravessa a árvore e calcula o valor da expressão aritmética definida pela árvore.

Para resolvermos o primeiro obstáculo teremos que fazer algumas alterações. O JJTree vai gerar uma classe Java para o SimpleNode (nome da classe definido como retorno do procedimento: parseOneLine()). Esta classe, é utilizada para representar os nós da árvore sintáctica, e pode ser personalizada para implementar as funcionalidades necessárias. A classe inclui os seguintes métodos:

Alguns métodos da classe representativa dos nós da árvore sintáctica:	Descrição:
public Node jjtGetParent()	Retorna o nó pai do nó actual
public Node jjtGetChild(int i)	Retorna o nó filho nº i
public int jjtGetNumChildren()	Retorna o número de filhos do nó
Public void dump()	Escreve no ecrã a árvore sintáctica a partir do nó

Depois da classe SimpleNode ser gerada pela primeira vez pelo JJTree podemos adicionar-lhe métodos ou campos. Neste momento interessa-nos acrescentar à classe um campo que permita armazenar o valor do inteiro (no caso das folhas da árvore). Foram ainda adicionados dois métodos que permitem aceder ao campo (atribuir e retornar o seu valor). A classe é apresentada de seguida com as alterações efectuadas (a negrito).

```
/* Generated By:JJTree: Do not edit this line. SimpleNode.java */

public class SimpleNode implements Node {
  protected Node parent;
  protected Node[] children;
  protected int id;
  protected Calculator parser;

  int value;

  public void setValue(int a) {
    this.value = a;
```

```
}

public int getValue() {
  return this.value;
}

public SimpleNode(int i) {
  id = i;
}

public SimpleNode(Calculator p, int i) {
  this(i);
  parser = p;
}

public void jjtOpen() {
}

public void jjtClose() {
}

public void jjtSetParent(Node n) { parent = n; }
public Node jjtGetParent() { return parent; }

public void jjtAddChild(Node n, int i) {
  if (children == null) {
    children = new Node[i + 1];
  } else if (i >= children.length) {
    Node c[] = new Node[i + 1];
    System.arraycopy(children, 0, c, 0, children.length);
    children = c;
  }
  children[i] = n;
}

public Node jjtGetChild(int i) {
  return children[i];
}

public int jjtGetNumChildren() {
  return (children == null) ? 0 : children.length;
}

/* You can override these two methods in subclasses of SimpleNode to
   customize the way the node appears when the tree is dumped.  If
   your output uses more than one line you should override
   toString(String), otherwise overriding toString() is probably all
   you need to do. */

public String toString() { return CalculatorTreeConstants.jjtNodeName[id]; }
public String toString(String prefix) { return prefix + toString(); }

/* Override this method if you want to customize how the node dumps
   out its children. */

public void dump(String prefix) {
  System.out.print(toString(prefix));
```

```
        if (children != null) {
          for (int i = 0; i < children.length; ++i) {
         SimpleNode n = (SimpleNode)children[i];
         if (n != null) {
          System.out.println();
          n.dump(prefix + " ");
          if(n.id ==  CalculatorTreeConstants.JJTUNARY)
           System.out.println(" ["+n.getValue()+"]");
         }
        }
      }
    }
}
```

O novo ficheiro que descreve a gramática, que indica ao JJTree para gerar a árvore sintáctica e que calcula o valor das expressões aritméticas introduzidas é apresentado de seguida. Durante o atravessamento da árvore é importante que se possa verificar de que tipo é um determinado nó. Tal pode ser feito utilizando o campo id de cada nó da árvore (ver nos exemplos node.id). Para cada tido de nó da árvore o JJTree gera um ficheiro em que são especificados os identificadores atribuídos (ver ficheiro **CalculatorTreeConstants.java**). O tipo de nós corresponde aos procedimentos da gramática e/ou aos nós especificados nas directivas para o JJTree.

Novo ficheiro Calculator.jjt:

```
options
{
   LOOKAHEAD=2;
}

PARSER_BEGIN(Calculator)

public class Calculator {
   public static void main(String args[]) throws ParseException {
      Calculator parser = new Calculator(System.in);
       int i=0;
      while (true) {
           // the function will return the root node of the Syntax Tree
           SimpleNode rootNode = parser.parseOneLine();

           //print the Syntax Tree
           rootNode.dump("Syntax Tree "+(i++)+": ");

           // call the evaluation method with the ROOT node
           System.out.println("Result: "+parser.eval(rootNode));
      }
   }

   /*
       Método recursivo que realiza o cálculo da expressão percorrendo a árvore sintáctica.
   */
   int eval(SimpleNode node) {
      // each node contains an id field identifying its type.
      int id = node.id;

      //System.out.println("VALUE "+node.getValue()+" "+id);
```

```
        if(id == CalculatorTreeConstants.JJTUNARY) // node with integer value
            return node.getValue();
        else if(node.jjtGetNumChildren() == 1) // only one child
            return this.eval((SimpleNode) node.jjtGetChild(0));

        // nodes with two childs
        SimpleNode lhs = (SimpleNode) node.jjtGetChild(0); //left child
        SimpleNode rhs = (SimpleNode) node.jjtGetChild(1); // right child

        switch(id) {
        case CalculatorTreeConstants.JJTADD : return eval( lhs ) + eval( rhs );
        case CalculatorTreeConstants.JJTSUB : return eval( lhs ) - eval( rhs );
        case CalculatorTreeConstants.JJTMULT : return eval( lhs ) * eval( rhs );
        case CalculatorTreeConstants.JJTDIV : return eval( lhs ) / eval( rhs );
        default : // abort
            System.out.println("Operador ilegal!");
            System.exit(1);
        }
        return 0;
    }
}

PARSER_END(Calculator)

SKIP :
{
    " "
    |  "\r"
    |  "\t"
}

TOKEN:
{
    < INTEGER: (["0"-"9"])+ >
    |  < EOL: "\n" >
}

SimpleNode parseOneLine() #Root : {} // diz ao JJTree para criar o nó Root
{
  expr() <EOL>     {return jjtThis;} // retorna o nó da árvore construído neste procedimento
  | <EOL>
  | <EOF>          { System.exit(-1); }
}

void expr(): {}
{
  term()
   (
      "+" expr() #Add(2)       // diz ao JJTree para criar um nó Add que tem dois filhos
    | "-" expr() #Sub(2)       // diz ao JJTree para criar um nó Sub que tem dois filhos
   )*
   | "(" expr() ")"
}

void term(): {}
{
   unary()
```

```
    (
       "*" term() #Mult(2)      // diz ao JJTree para criar um nó Mult que tem dois filhos
    |  "/" term() #Div(2)       // diz ao JJTree para criar um nó Div que tem dois filhos
    )*
    | "(" expr() ")"
}

void unary(): {Token t;}
{
    "-" (t=<INTEGER> {
        // diz ao JJTree para colocar o valor do inteiro num dos campos deste nó
        // (para tal é necessário adicionar à classe Java SimpleNode o método setValue() e
        // o campo que armazena o valor inteiro
        jjtThis.setValue(-Integer.parseInt(t.image)[1]);  })
    | (t=<INTEGER> {
        // diz ao JJTree para colocar o valor do inteiro num dos campos deste nó
        // (para tal é necessário adicionar à classe Java SimpleNode o método setValue() e
        // o campo que armazena o valor inteiro
        jjtThis.setValue(Integer.parseInt(t.image));
    })
}
```

Acha que a árvore sintáctica gerada pelos ficheiros jjt apresentados é uma árvore sintáctica concreta ou uma AST (árvore sintáctica abstracta)?

Para que não seja gerado um nó por cada procedimento na gramática utiliza-se a directiva **#void** a seguir ao nome do procedimento para o qual não se quer nó na árvore. Este método permite gerar as ASTs automaticamente sem por isso ser necessário transformar a árvore concreta numa AST. O ficheiro seguinte apresenta uma versão da calculadora em que são geradas ASTs. Verifique a colocação da directiva a indicar os símbolos não-terminais para os quais não será representado qualquer nó na árvore.

Novo ficheiro Calculator.jjt:

```
options
{
   LOOKAHEAD=2;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
   public static void main(String args[]) throws ParseException {
      Calculator parser = new Calculator(System.in);
    int i=0;
      while (true) {
       // the function will return the root node of the Syntax Tree
           SimpleNode rootNode = parser.parseOneLine();

       //print the Syntax Tree
       rootNode.dump("Syntax Tree "+(i++)+": ");
```

[1] Como o valor de um Token é guardado como String, é necessário converter a representação em String para inteiro. Tal pode ser conseguido utilizando métodos existentes nos API do Java. Por exemplo, `Integer.parseInt(t.image)` retorna o inteiro representado pela String `t.image`.

```
        // call the evaluation method with the ROOT node
        System.out.println("Result: "+parser.eval(rootNode));
      }
    }

    int eval(SimpleNode node) {
        // each node contains an id field identifying its type.
        // we switch on these.
        // enum values such as JJTUNARY come from the interface file
        // SimpleParserTreeConstants, which SimpleParser implements.
        // This interface file is one of several auxilliary Java sources
        // generated by JJTree. JavaCC contributes several others.
        int id = node.id;

        //System.out.println("VALUE "+node.getValue()+" "+id);

        if(id == CalculatorTreeConstants.JJTUNARY) // node with integer value
            return node.getValue();
        else if(node.jjtGetNumChildren() == 1) // only one child
            return this.eval((SimpleNode) node.jjtGetChild(0));

        SimpleNode lhs = (SimpleNode) node.jjtGetChild(0); //left child
        SimpleNode rhs = (SimpleNode) node.jjtGetChild(1); // right child

        switch(id) {
        case CalculatorTreeConstants.JJTADD : return eval( lhs ) + eval( rhs );
        case CalculatorTreeConstants.JJTSUB : return eval( lhs ) - eval( rhs );
        case CalculatorTreeConstants.JJTMULT : return eval( lhs ) * eval( rhs );
        case CalculatorTreeConstants.JJTDIV : return eval( lhs ) / eval( rhs );
        default : // abort
            System.out.println("Operador ilegal!");
            System.exit(1);
        }
        return 0;
    }
}

PARSER_END(Calculator)

SKIP :
{
    " "
    |  "\r"
    |  "\t"
}

TOKEN:
{
    < INTEGER: (["0"-"9"])+ >
    |  < EOL: "\n" >
}

SimpleNode parseOneLine() #Root : {}
{
  expr() <EOL>      {return jjtThis;}
  | <EOL>
```

```
  | <EOF>          { System.exit(-1); }
}

void expr() #void : {}
{
  term()
   (
      "+" expr() #Add(2)
    | "-" expr() #Sub(2)
   )*
   | "(" expr() ")"
}

void term() #void : {}
{
   unary()
   (
      "*" term() #Mult(2)
   |   "/" term() #Div(2)
   )*
   | "(" expr() ")"
}

void unary() : {Token t;}
{
    "-" (t=<INTEGER> {
        jjtThis.setValue(-Integer.parseInt(t.image));
    })
    | (t=<INTEGER> {
        jjtThis.setValue(Integer.parseInt(t.image));
    })
}
```

2 Referências de Apoio

1. JavaCC: http://www.experimentalstuff.com/Technologies/JavaCC/index.html

2. Documento de introdução ao JJTree incluído na distribuição da ferramenta: http://w3.ualg.pt/~jmcardo/ensino/PS2003/jjtree.intro