Familiarização com a ferramenta JavaCC

Compiladores

Universidade do Porto/FEUP

Departamento de Engenharia Informática

ver. 1.0, Janeiro de 2011

Conteúdo

Objectivo: Familiarização com a ferramenta de construção de analisadores sintácticos (*parsers*) JavaCC.

Estratégia: Construção de um interpretador de expressões aritméticas tendo em conta várias opções.

---

1. Implementação de Analisadores Gramaticais Utilizando o JavaCC (1ª parte)

Este documento tem como objectivo iniciar a aprendizagem do gerador de analisadores sintácticos JavaCC [1][2]. Os conhecimentos sobre analisadores sintácticos descendentes permitirão mais tarde uma melhor percepção sobre o funcionamento dos geradores de *parsers*, dos quais o JavaCC é um exemplo.

O JavaCC utiliza um ficheiro com extensão **.jj** onde se encontram descritos os lexemas e a gramática para o *parser*, e gera um conjunto de classes Java de suporte[1], tendo uma delas o nome do *parser*. A Figura 1 ilustra genericamente a entrada do JavaCC e as classes geradas.

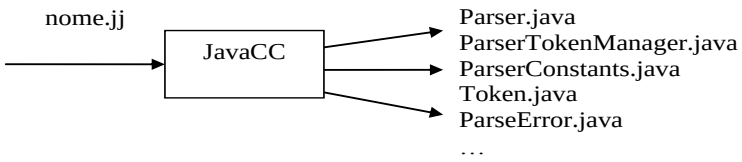

Figura 1. Entradas e saídas do JavaCC supondo que foi especificado "Parser" como nome do *parser* no ficheiro *nome.jj*.

Os ficheiros **.jj** são constituídos por:

1. Zona de opções (opcional): é onde pode ser definido o nível global de *lookahead*, por exemplo.
2. Unidade de compilação Java (PARSER_BEGIN(nome) ... PARSER_END(nome)) que define a classe principal do parser e onde pode ser incluido o método main do *parser*.
3. Lista de produções da gramática (as produções aceitam os símbolos +, *, e ? com o mesmo significado, aquando da utilização em expressões regulares)

Exemplo

Vamos supor que desejamos implementar um analisador sintáctico que reconheça expressões aritméticas com apenas um número inteiro positivo ou com uma adição ou uma subtracção de dois números inteiros positivos (por exemplo: 2+3, 1-4, 5). De seguida apresenta-se uma gramática com a especificação do símbolo terminal utilizando uma expressão regular:

```
INTEGER = [0-9]+       // símbolo terminal
```

[1] À medida que forem percebendo o JavaCC vão também tomando conhecimento do significado das classes de suporte.

Aritm → INTEGER [("+" | "-") INTEGER] // regra gramatical

Para desenvolver com o JavaCC um *parser* que implemente esta gramática temos de criar um ficheiro onde vamos especificar o *parser*. Vamos chamar-lhe *Exemplo.jj*.

Ficheiro Exemplo.jj:

```
PARSER_BEGIN(Exemplo)

// código Java que invoca o parser
    public class Exemplo {
        public static void main(String args[]) throws ParseException {
            // criação do objecto utilizando o constructor com argumento para
            // ler do standard input (teclado)
            Exemplo parser = new Exemplo(System.in);
            parser.Aritm();
        }
    }
PARSER_END(Exemplo)

// símbolos que não devem ser considerados na análise
SKIP :
{
    " " | "\t" | "\r"
}

// definição dos tokens (símbolos terminais)
TOKEN :
{
    < INTEGER : (["0" – "9"])+ >
    | < LF : "\n" >
}

// definição da produção
void Aritm() : {}
{
    <INTEGER> ( ("+" | "-") <INTEGER> )? <LF>       // "(...)?" é equivalente a "[...]"
}
```

Para gerar o *parser*:

javacc Exemplo.jj

Compilar o código Java gerado:

javac *.java

Executar o analisador sintáctico:

java Exemplo

Poderemos associar às funções referentes aos símbolos não-terminais pedaços de código Java. Por exemplo, as modificações apresentadas de seguida permitem escrever no ecrã mensagens a indicar os números que são lidos pelo *parser*:

```
void Aritm() : {Token t1, t2;}
{
   t1=<INTEGER> {
      System.out.println("Integer = "+t1.image);
```

```
    }
    ( ("+" | "-")  t2=<INTEGER> {
        System.out.println("Integer = "+t2.image);
    }
    )? (<LF>)
}
```

Por cada símbolo terminal INTEGER, foi inserida uma linha de código Java que imprime no ecrã o valor do *token* lido (o atributo *image* da classe *Token* retorna uma *String* representativa do valor do *token*[2])

Insira estas modificações no ficheiro *Exemplo.jj* e volte a repetir o processo até à execução do *parser*. Verifique o que acontece.

Como poderíamos escrever no ecrã o sinal (+ ou -) lido?

2. Implementação de Analisadores Gramaticais Utilizando o JavaCC (2ª parte)

Depois de uma primeira utilização do JavaCC vamos agora ver como se pode gerar automaticamente a árvore sintáctica. Para tal utilizaremos o JJTree [5]. O JJTree é uma ferramenta de pré-processamento integrada no pacote de software JavaCC que gera classes Java e 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 de entrada do JJTree). A Figura 2 ilustra as entradas e as saídas do JJTree.

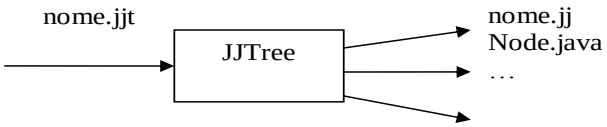

Figura 2. Entradas e saídas do JJTree.

Exemplo

Para testarmos o JJTree vamos construir uma calculadora de expressões aritméticas simples sobre números inteiros. Para tal, vamos usar a árvore sintáctica gerada pelo *parser* e efectuar os cálculos sobre essa árvore.

[2] Outros métodos serão apresentados posteriormente.

Vamos considerar os seguintes *tokens* para a gramática:

LF=”\n”

INTEGER=[0-9]+

E a seguinte gramática para as expressões aritméticas:

Expression → Expr1 LF

Expr1 → Expr2 [(“+” | “-“) Expr2]

Expr2 → Expr3 [(“*” | “/“) Expr3]

Expr3 → INTEGER

Expr3 → “-“ Expr3

Expr3 → “(“ Expr1 “)”

Para gerar a árvore sintáctica vamos especificar a gramática original num ficheiro com extensão **jjt** e vamos adicionar as directivas que indicam ao JJTree para gerar o código necessário para criar a árvore.

O ficheiro apresentado de seguida contém as regras gramaticais anteriores e as modificações introduzidas, identificadas a negrito, para gerar a árvore. O método *dump(String)* foi também modificado para imprimir no ecrã a árvore gerada para cada expressão introduzida.

Ficheiro Calculator.jjt:

```
options
{
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
   public static void main(String args[]) throws ParseException {
       Calculator myCalc = new Calculator(System.in);
       SimpleNode root = myCalc.Expression(); // devolve referência para o nó raiz da árvore

       root.dump(""); // imprime no ecrã a árvore
   }
}

PARSER_END(Calculator)

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

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

```
SimpleNode Expression(): {}
{
  Expr1() <LF>  {return jjtThis;} // código Java entre chavetas
}

void Expr1(): {}
{
  Expr2() [("+" | "-") Expr2()]
}

void Expr2(): {}
{
  Expr3() [("*" | "/") Expr3()]
}

void Expr3(): {}
{
   <INTEGER>
   | "-" Expr3()
   | "(" Expr1() ")"
}
```

Executar o JJTree:

```
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 as expressões que introduzir.

A criação da árvore sintáctica não é suficiente para desenvolvermos a calculadora de expressões aritméticas. 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 representada pela árvore. Vamos ver de seguida o passo necessário para incluirmos anotações na árvore.

Anotação da Árvore

Para resolvermos o primeiro problema teremos que fazer algumas alterações. O JJTree apenas gera a classe *SimpleNode*[3] se esta não existir. Esta classe é utilizada para representar os nós[4] da árvore sintáctica, e pode ser personalizada para implementar as funcionalidades necessárias. A classe inclui os seguintes métodos:

[3] nome da classe definido como instância a devolver pela função *myCalc()*.
[4] Existe a possibilidade de geração de uma classe para cada nó (ver opção MULTI).

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

Depois da classe *SimpleNode* ser gerada pela primeira vez pelo JJTree podemos adicionar-lhe métodos e/ou atributos. Neste momento interessa-nos acrescentar à classe dois atributos que permitam armazenar o valor do inteiro (no caso das folhas da árvore) [5] e a operação a realizar[6].

Para este exemplo vamos definir uma classe com as operações que precisamos:

Ficheiro MyConstants.java:

```
public class MyConstants {

        public static final int ADD =1;
        public static final int SUB =2;
        public static final int MUL =3;
        public static final int DIV =4;

        public static final char[] ops = {'+','-','*','/'};
}
```

A classe *SimpleNode* é apresentada de seguida com as alterações efectuadas (a negrito). Note-se que em vez de anotarmos os nós com a operação a realizar poderíamos orientar o JJTree para associar nós da árvore à respectiva operação aritmética (usando directivas do tipo "#Add"). Mais à frente veremos como usar essas directivas.

Ficheiro SimpleNode.java:

```
/* 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 Object value;
  protected Calculator parser;

  // added
  public int val;
```

[5] Poderiam ter sido adicionados dois métodos que permitem aceder ao campo (atribuir e retornar o seu valor).

[6] Poderíamos usar o ID do nó (ver campo "id" da classe "SimpleNode" e os tipos de nós para uma dada gramática na classe "CalculatorTreeConstants").

```
public int Op=0;

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;
}

public void jjtSetValue(Object value) { this.value = value; }
public Object jjtGetValue() { return value; }

/* 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.println(toString(prefix));

  if(this.Op != 0)
        System.out.println("\t[ "+MyConstants.ops[this.Op-1]+" ]");
  if(children == null)
       System.out.println("\t[ "+this.val+" ]");

  if (children != null) {
    for (int i = 0; i < children.length; ++i) {
      SimpleNode n = (SimpleNode)children[i];
```

```
      if (n != null) {
        n.dump(prefix + " ");
      }
    }
  }
 }
}
```

É apresentado de seguida o ficheiro que permite obter o analisador sintáctico que inclui a geração da árvore sintáctica anotada.

Para utilizarmos a árvore sintáctica é importante que possamos verificar de que tipo é um determinado nó. Uma das possibilidades é anotar cada nó com a informação relacionada com o respectivo tipo. Neste exemplo usamos as constantes definidas na classe *MyConstants*.

Novo ficheiro Calculator.jjt:

```
options
{
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
   public static void main(String args[]) throws ParseException {
       Calculator myCalc = new Calculator(System.in);
       SimpleNode root = myCalc.Expression(); // devolve referência para o nó raíz da árvore

       root.dump(""); // imprime no ecrã a árvore
   }
}

PARSER_END(Calculator)

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

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

SimpleNode Expression(): {}
{
  Expr1() <LF>  {return jjtThis;}
}

void Expr1(): {}
{
  Expr2(1)
  [
    ("+" {jjtThis.Op = MyConstants.ADD;}
    | "-" {jjtThis.Op = MyConstants.SUB;}
```

```
    )
    Expr2(1)
    ]
}

void Expr2(int sign): {}    // 1: positive; -1: negative, por causa do operador unitário '-'
{
  Expr3(sign)
  [
   ("*" {jjtThis.Op = MyConstants.MUL;}
   | "/" {jjtThis.Op = MyConstants.DIV;}
   )
   Expr3(1)
  ]
}

void Expr3(int sign):  {Token t;}
{
   t=<INTEGER>
   {
    jjtThis.val = sign *Integer.parseInt(t.image);
   }
   | "-" Expr3(-1)
   | "(" Expr1() ")"
}
```

Avaliação das Expressões Usando a Árvore Anotada

Com a anotação dos inteiros e das operações na árvore sintáctica já podemos programar um método capaz de determinar o valor da expressão aritmética introduzida pelo utilizador. Para tal, podemos incluir uma função recursiva como a que é apresentada de seguida:

```
int eval(SimpleNode node) {

        if(node.jjtGetNumChildren() == 0) // leaf node with integer value
                return node.val;
        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(node.Op) {
                 case MyConstants.ADD : return eval( lhs ) + eval( rhs );
                 case MyConstants.SUB : return eval( lhs ) - eval( rhs );
                 case MyConstants.MUL : return eval( lhs ) * eval( rhs );
                 case MyConstants.DIV : return eval( lhs ) / eval( rhs );
                 default : // abort
                         System.out.println("Operador ilegal!");
                         System.exit(1);
         }
         return 0;
 }
```

O método anterior pode ser colocado entre

"PARSER_BEGIN(Calculator)" e "PARSER_END(Calculator)".

Podemos depois invocá-lo no "*main*", como é ilustrado pela instrução:

System.out.println("Valor da expressão: "+myCalc.eval(root));

Simplificação da Árvore Gerada e Utilização do Tipo de Nó para Representar os Operadores

As árvores sintácticas geradas nos exemplos apresentados são designadas por árvores sintácticas concretas (representam fidedignamente as produções da gramática). A simplificação destas árvores permite obter as árvores sintácticas abstractas (ASTs[7]).

Para que não seja gerado um nó por cada produção na gramática (procedimento na especificação da produção no ficheiro **jjt**) utiliza-se a directiva **#void**. Estas directivas devem ser colocadas a seguir aos nomes dos procedimentos para os quais não queremos um nó na árvore. Este método permite gerar as ASTs automaticamente sem por isso ser necessário transformar a árvore concreta numa árvore abstracta (AST). O ficheiro seguinte apresenta uma versão da calculadora em que são geradas ASTs. Vamos utilizar este exemplo para ilustrar também como podemos usar os tipos de nós atríbuidos automaticamente. O identificador do tipo de nó é armazenado no atributo *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 as constantes identificadoras dos tipos de nós que podem constar nas árvores sintácticas geradas a partir das regras gramaticais (ver ficheiro **CalculatorTreeConstants.java**). O tipo de nó corresponde aos procedimentos relacionados com as regras da gramática que não usem a directiva *"#void"* e/ou aos nós especificados nas directivas para o JJTree. Esta versão ilustra a associação de identificadores diferentes para regras gramaticais no mesmo procedimento (ver #Add(2), #Mul(2), por exemplo).

Novo ficheiro Calculator.jjt:

```
options {
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculator)
public class Calculator
{
   public static void main(String args[]) throws ParseException {
      Calculator myCalc = new Calculator(System.in);
      SimpleNode root = myCalc.Expression();
      root.dump("");

      System.out.println("Valor da expressão: "+myCalc.eval(root));
   }

int eval(SimpleNode node) {

        if(node.jjtGetNumChildren() == 0) // leaf node with integer value
```

[7] Do inglês: abstract syntax tree.

```
                return node.val;
        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(node.id) {
                case CalculatorTreeConstants.JJTADD : return eval( lhs ) + eval( rhs );
                case CalculatorTreeConstants.JJTSUB : return eval( lhs ) - eval( rhs );
                case CalculatorTreeConstants.JJTMUL : 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"])+ >
        |   < LF: "\n" >
}

SimpleNode Expression(): {}
{
  Expr1() <LF>  {return jjtThis;}
}

void Expr1() #void: {}
{
  Expr2(1)
  [
        ("+" Expr2(1) #Add(2)
        | "-" Expr2(1) #Sub(2))
  ]
}

void Expr2(int sign) #void: {}    // 1: positive; -1: negative
{
  Expr3(sign)
  ("*" Expr3(1) #Mul(2)
        | "/" Expr3(1) #Div(2)
        )?                          // (…)? é equivalente a […]
}

void Expr3(int sign) #void:  {Token t;}
{
   t=<INTEGER>
        {
         jjtThis.val = sign * Integer.parseInt(t.image);
        } #Term
   | "-" Expr3(-1)
   | "(" Expr1() ")"
```

```
}
```

Novo ficheiro SimpleNode.java:

```
public class SimpleNode implements Node {

  protected Node parent;
  protected Node[] children;
  protected int id;
  protected Object value;
  protected Calculator parser;


  //  added
  public int val;

  …
  public void dump(String prefix) {
        System.out.println(toString(prefix));
        switch(this.id) {
              case CalculatorTreeConstants.JJTADD:
                    System.out.println("\t[ + ]");break;
              case CalculatorTreeConstants.JJTSUB:
                    System.out.println("\t[ - ]");break;
              case CalculatorTreeConstants.JJTMUL:
                    System.out.println("\t[ * ]");break;
              case CalculatorTreeConstants.JJTDIV:
                    System.out.println("\t[ / ]");break;
         }
        if(children == null)
                        System.out.println("\t[ "+this.val+" ]");
        if (children != null) {
             for (int i = 0; i < children.length; ++i) {
               SimpleNode n = (SimpleNode)children[i];
               if (n != null) {
                 n.dump(prefix + " ");
               }
             }
        }
   }
}
```

Ficheiro gerado CalculatorTreeConstants.java:

```
public interface CalculatorTreeConstants {

  public int JJTEXPRESSION = 0;
  public int JJTVOID = 1;
  public int JJTADD = 2;
  public int JJTSUB = 3;
  public int JJTMUL = 4;
  public int JJTDIV = 5;
  public int JJTTERM = 6;

  public String[] jjtNodeName = { "Expression", "void",   "Add",   "Sub",  "Mul",  "Div",  "Term" };
}
```

3. Exercícios Propostos

Exercício 1

Modifique a gramática anterior para que aceite atribuições a símbolos e a utilização de símbolos nas expressões aritméticas, como é ilustrado no exemplo seguinte:

```
A=2;
B=3;
A*B; // neste caso deve apresentar no ecrã o valor 6.
```

Considere que os símbolos são definidos pela expressão regular [A-Za-z][0-9A-Za-z]*.

De seguida, acrescente o código necessário para calcular as expressões aritméticas.

Exercício 2

A calculadora implementada não calcula correctamente expressões do tipo -(2), -(2+3), etc. O sinal '-' imediatamente anterior ao símbolo de abrir parêntesis não é considerado nos cálculos. Que modificações teremos de fazer para que expressões como as anteriores possam ser avaliadas correctamente?

Exercício 3

A gramática apresentada não aceita expressões aritméticas que contenham sequências de operações do mesmo tipo (como 2+3+4 ou 2*3*4, por exemplo). A única forma de introduzir expressões deste tipo é utilizando parêntesis para agrupar as operações como: 2+(3+4) ou 2*(3*4), por exemplo. Ilustra-se em baixo a gramática alterada para aceitar sequências de operações do mesmo tipo ("{...}" indica 0 ou mais).

Expression → Expr1 LF
Expr1 → Expr2 **{**("+" | "-") Expr2**} // alteração**
Expr2 → Expr3 **{**("*" | "/") Expr3**} // alteração**
Expr3 → INTEGER
Expr3 → "-" Expr3
Expr3 → "(" Expr1 ")"

Identifique os problemas que esta gramática introduzirá no cálculo das expressões aritméticas utilizando a árvore sintáctica.

4. Referências

[1] JavaCC, https://javacc.dev.java.net/
[2] Tom Copeland, *Generating Parsers with JavaCC*, Second Edition, Centennial Books, Alexandria, VA, 2009. ISBN 0-9762214-3-8, http://generatingparserswithjavacc.com/
[3] JavaCC [tm]: Documentation Index, https://javacc.dev.java.net/doc/docindex.html

[4] Oliver Enseling, “Build your own languages with JavaCC”, Copyright © 2003 JavaWorld.com, an IDG company, http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools_p.html

[5] JavaCC [tm]: JJTree Reference Documentation, https://javacc.dev.java.net/doc/JJTree.html

5. Ferramentas para o JavaCC

[6] JTB: Java Tree Builder: http://compilers.cs.ucla.edu/jtb/

6. Outras Ferramentas para Gerar Parsers

[7] SableCC: http://sablecc.org/

[8] ANTLRWorks: The ANTLR GUI Development Environment, http://www.antlr.org/