JavaCC: Java compiler's compiler

Site Web: https://javacc.dev.java.net

(go to : Documentation > ... the JavaCC grammar file ...)

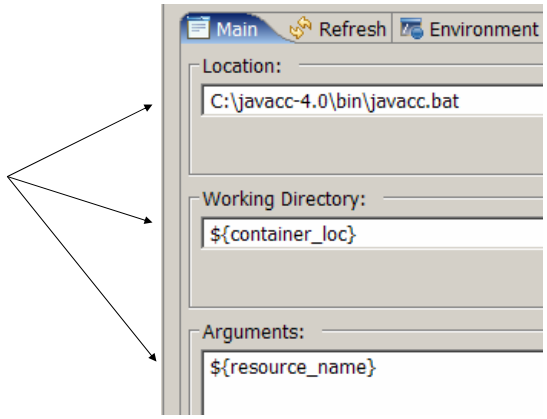
Download the version 4.0

(See http://www.epflpress.org/Book_Pages/petitpierre.html)

JavaCC: compilation of a .jj file

(the sources are contained in .jj files)

In external tools:



JavaCC: parts

1. Options
2. Program header
3. Tokens
4. Productions

JavaCC: (1) options

The first block of the source contains the options. Here are the two main ones.

```
options {
  STATIC = true;
  DEBUG_PARSER = true;
}
```

JavaCC: (2) the program header

```
PARSER_BEGIN(XMLParserSKIP)
package parser;
import java.io.FileReader;

public class XMLParserSKIP {
      public static void main(String args[]) throws Exception {
            FileReader in = new FileReader(args[0]);
            XMLParserSKIP parser = new XMLParserSKIP(in);
            parser.rootProduction();
      }
      // possibility to define attributes and methods
}
PARSER_END(XMLParserSKIP)
```

JavaCC: (3) ignored tokens

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

Specifies that the spaces, carriage returns, tabulations, new lines are ignored (skipped).

These tokens separate the other ones, of course, but they are not transmitted to the analyzer.

JavaCC: simple tokens

```
TOKEN :
{ < LT: "<" >
| < GT: ">" >
| <UNDERLINE: "_">
| <COLON: ":">
| <DOT: ".">
| <MINUS: "-">
| <ENDTAG: "</">
}
```

This source defines a few tokens

JavaCC: composed tokens

```
TOKEN :
{ <ID: ( <LETTER> | "_" | ":" )
        ( <DIGIT> | <LETTER> | "_" | ":" | "." | "-" )* >
| <#LETTER: ["a"-"z", "A"-"Z"] >
| <#DIGIT: ["0" - "9"] >
}
```

#LETTER defines a token known only locally

JavaCC: attention!

```
TOKEN :
{   < COLON:    ":" >
|   < ID:          ":"   (["a"-"z])* >
}
```

Input text:

:234 → <COLON> 2 3 4

:aaa → <ID>

The token is the longest possible path, but if two tokens have the same length, the first one is returned

JavaCC: definition of spaces

```
TOKEN :  // not correct (a single space is SPACE != S )
{
  <SPACE: " ">
| <CR: "\r">
| <TAB: "\t">
| <NL: "\n">
| <S: (<SPACE> | <CR> | <TAB> | <NL>)+  >
}
```

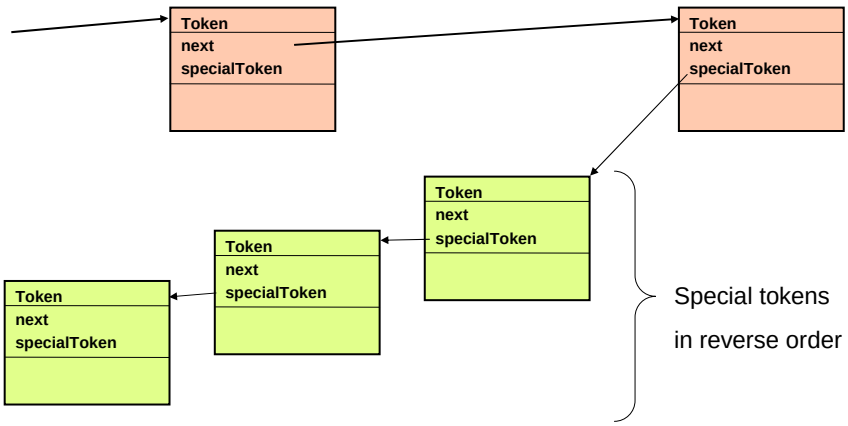
JavaCC: correct definition

```
TOKEN :          // correct
{
  <S: (<SPACE> | <CR> |<TAB> | <NL>)+  >
| <#SPACE: " ">    // local definition
| <#CR: "\r">
| <#TAB: "\t">
| <#NL: "\n">
}
```

JavaCC: special token definition

```
SPECIAL_TOKEN :
{
  <S: (<SPACE> | <CR> |<TAB> | <NL>)+  >
| <#SPACE: " ">    // local definition
| <#CR: "\r">
| <#TAB: "\t">
| <#NL: "\n">
}
```

JavaCC: queue of tokens and special tokens



JavaCC: special token and token class

```
package tree;
public class Token {
   public int kind;
   public int beginLine, beginColumn, endLine, endColumn;
   public String image;
   public Token next;
   public Token specialToken;
   public String toString()  {    return image;   }
   public static final Token newToken(int ofKind) {
       switch(ofKind)   {                  // to create specific tokens
       default : return new Token();
} } }
```

JavaCC: token in contexts

<![CDATA[x x x x x]]>

<DEFAULT> **MORE** : { "<![CDATA[" : **IN_CDATA** }

<IN_CDATA> **MORE** : { < ~[] > }

<IN_CDATA> **TOKEN** : { <CDATA : "]]>"> : DEFAULT }

In the default mode (start of the program), ***introduce*** *token* "<![CDATA[" *in the current token and* ***jump*** *to context* <IN_DATA>. In this context, **add** *any character to the same token,* ***add*** *the token* "]]>", *return to the default mode and return a single token made of all pieces appearing in the MORE lines. The final token is named CDATA.*

(not in the book!)

JavaCC: (4) production

```
void tag() :
{ }
{
   "<“   <ID>   ">"
}
```

```
String tag() :
{ }
{
   "<"   <ID>   ">"
   { return token.image; }
}
```

token exists by default

```
String tag() :
{
   String s;
   Token t;    // extra token
}
{
   "<"   t =<ID>   ">"
   { return t.image; }
}
```

JavaCC: repetitions

```
void product() :
{ }
{
   tag()
   (   <ID>  )*
   endTag()
}
void tag() :  { }
{ "<"   <ID>   ">" }

void endTag() :  { }
{ "</"  <ID>   ">" }
```

(x)*	0 – n times
(x)+	1 – n times
(x)?	optional
[x]	same as above

JavaCC: choices

```
void product() :
{ }
{
   "("
   (
       <ID>
   |
       tag()
   )
   endTag()
}
```

Either

"("	<ID>	endTag()

or

"("	tag()	endTag()

JavaCC: another production

```
void command() throws Exception :
{  String s, t;  }
{
   s = tag()
   (  (<ID>)+
   |  (command())+
   |  <CDATA>
   )
   t = endTag()
   { if (!s.image.equals(t.image))      // Java code
        throw new Exception("end tag != from start tag");
   }
}
```

*<a> ID ID *

*<a> <c>x</c> *

*<a> <![CDATA[x x x]]> *

JavaCC: token versus production

```
TOKEN:
{
  <TAG:  "<"  <ID>  ">"  >
}

String tag () :
{ String s; }
{
    "<" <ID>{s=token.image;} ">"
    { return s; }
}
```

JavaCC: lookahead

```
void product() :
{ }
{
   (   tag()
   |   endTag()
   (
}
void tag() :  { }
{ "<" <ID> ">" }

void endTag() :  { }
{ "<"   "/"   <ID>   ">" }
```

```
void product() :
{ }
{
   (   LOOKAHEAD (2)  tag()
   |                  endTag()
   (
}
void tag() :  { }
{ "<" <ID> ">" }

void endTag() :  { }
{ "<"   "/"   <ID>   ">" }
```

both continue with "<"

JavaCC: LOOKAHEAD

The LOOKAHEAD command is placed at the beginning of the alternative

Three possibilities:

LOOKAHEAD (2)

LOOKAHEAD ("<" id() ">")

LOOKAHEAD ({ methodReturningABoolean() })

. . .

```
JAVACODE
boolean methodReturningBoolean()  {
    if (…) return true; else return false;
}
```

JavaCC: JAVACODE command

```
TOKEN:
{
   <UNDEFINED:  ~[]>
}
void command() { }
{
     tag()   <CDATA_START>   {getCDATA();}   endTag()
}
JAVACODE
void getCDATA() {
     while ( getNextToken().kind != CDATA_END)
          System.out.print("-"+getToken(0).image+"-");
     if (getToken(1).kind==S)
          getNextToken();
}                                          // getToken(1)  == lookahead
```

JavaCC: an example

(CVS project Patterns, package tree, compile *Parser.jjt* with *javacc* for now)

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

TOKEN :
{
  < LPAR: "(" >
| < RPAR: ")" >
| < PLUS: "+" >
| < MINUS: "-" >
| < INTEGER: (["0" - "9"])+ >
}
```

compiler for expressions like this one

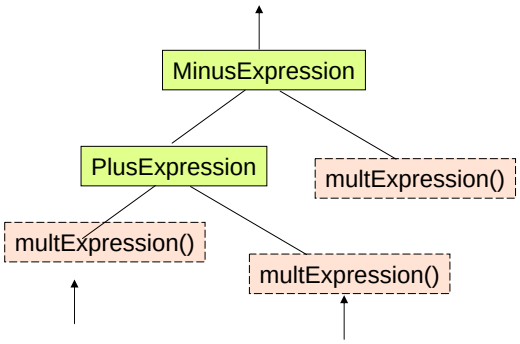
4*(8+7-9)+(6*(7-2))

JavaCC: continuation of the example

```
void expression() throws Exception :
{ }
{   multExpression()
     (  "+“  multExpression()
     |  "-"   multExpression()
     )*
}
void multExpression() throws Exception :
{ }
{   primaryExpression()
    (  "*"   primaryExpression()   )*
}
void primaryExpression() throws Exception :
{ }
{
    (   <INTEGER>
    |    "(" expression() ")"
    )
}
```

4*(8+7-9)+(6*(7-2))

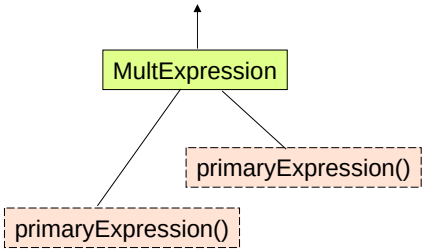
JavaCC: creation of an interpreter tree




```
AbstractExpression expression()
                    throws Exception :
{
   AbstractExpression ae, m;
}
{
   m = multExpression()
   (
     "+" ae = multExpression() {m=new PlusExpression(m,ae);}
   |
     "-" ae = multExpression() {m=new MinusExpression(m,ae);}
   )*
   {return m;}
}
```

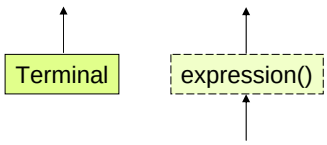
JavaCC: creation of an interpreter tree

```
AbstractExpression multExpression()
                        throws Exception :
{
   AbstractExpression ae, m;
}
{
   m = primaryExpression()
   (
      "*" ae = primaryExpression()  {m=new MultExpression(m, ae);}
   )*
   { return m; }
}
```



JavaCC: creation of an interpreter tree

```
AbstractExpression primaryExpression()
                        throws Exception :
{
   AbstractExpression m;
}
{
   (
      <INTEGER>  {m = new Terminal(Integer.parseInt(token.image));}
   |
      "(" m=expression() ")"
   )
   {return m;}
}
```



JavaCC: continuation with Interpreter Pattern

```
public class Terminal extends AbstractExpression{
    public Terminal( AbstractExpression left,
                     AbstractExpression right) {
        super(left, right);
    }
    int value;
    public Terminal(int value) {
        super(null,null);
        this.value = value;
    }
    public void interpret(Stack stack) {
        stack.push(value);
}   }
```

JavaCC: continuation with Interpreter Pattern

```
public class PlusExpression extends AbstractExpression {

public PlusExpression(AbstractExpression left,
        AbstractExpression right) {
        super(left, right);
    }
    public void interpret(Stack stack) {
        left.interpret(stack);
        right.interpret(stack);
        stack.push(stack.pop()+stack.pop());
}   }
```

// Available in project Patterns, package tree, ParserDump.jjt