

JavaCC

JavaCC is a Lexical Analyser Generator and a Parser Generator

- Input: Set of regular expressions (each of which describes a type of token in the language), and a grammar defined using these tokens as constants
- Output: A lexical analyser, which reads an input file and separates it into tokens, and a parser which reads an input file and performs a syntax analysis on it
- JavaCC files have the form `MyParser.jj`
- They are processed by JavaCC using a command of the form `javacc MyParser.jj`.
- The following files will be generated for `MyParser`:
 - MyParser.java** : The generated parser
 - MyParserTokenManager.java** : The generated token manager (lexical analyser)
 - MyParserConstants.java** : A bunch of useful constants

JavaCC File Structure

- The structure of a JavaCC file is as follows:

```
options{
    /* Code to set various options flags */
}

PARSER_BEGIN(MyParser)

public class MyParser {
    /* Java program is placed here */
}

PARSER_END(MyParser)

TOKEN_MGR_DECLS :
{
    /* Declarations used by lexical analyser */
}

/* Token Rules & Actions */
```

- The name following `PARSER_BEGIN` and `PARSER_END` must be the same.
- A Java program is placed between the `PARSER_BEGIN` and `PARSER_END`. This must contain a class declaration with the same name as the generated parser.

JavaCC Regular Expressions

- All characters and strings must be in quotation marks
 - "else"
 - "*"
 - ("a"|"b")
- All regular expressions involving * must be parenthesised
 - ("a")*, not "a"*

JavaCC Shorthand

<code>["a","b","c","d"]</code>	=	<code>("a" "b" "c" "d")</code>
<code>["d"-"g"]</code>	=	<code>["d","e","f","g"]</code>
	=	<code>("d" "e" "f" "g")</code>
<code>["d"-"f","M"-"O"]</code>	=	<code>["d","e","f","M","N","O"]</code>
	=	<code>("d" "e" "f" "M" "N" "O")</code>
<code>(α)?</code>	=	Optionally α (i.e., <code>(α \epsilon)</code>)
<code>(α)+</code>	=	$\alpha(\alpha)^*$
<code>(~["a","b"])</code>	=	Any character <i>except</i> "a" or "b". Can only be used with <code>[]</code> notation (<code>~(a(a b)*b)</code> is not legal)

Examples:

Regular Expression	Language
<code>"if"</code>	<code>{if}</code>
<code>["a"-"z"](["0"-"9","a"-"z"])*</code>	Set of legal identifiers
<code>["0"-"9"]+</code>	Set of integers (with leading zeroes)
<code>(["0"-"9"]+"."(["0"-"9"]*)) </code> <code>((["0"-"9"])*"."["0"-"9"]+)</code>	Set of real numbers

Token Rules in JavaCC

Tokens are described by rules with the following syntax:

```
TOKEN :  
{  
    <TOKEN_NAME: RegularExpression>  
}
```

- `TOKEN_NAME` is the name of the token being described
- `RegularExpression` is a regular expression that describes the token
- Examples:

```
TOKEN :  
{  
    <ELSE: "else">  
}
```

```
TOKEN :  
{  
    <INTEGER_LITERAL: (["0"-"9"])+>  
}
```

Token Rules in JavaCC

Several different tokens can be described in the same TOKEN block, with token descriptions separated by |.

For example:

```
TOKEN :  
{  
    <ELSE: "else">  
    | <INTEGER_LITERAL: ("0"-"9")+>  
    | <SEMICOLON: ";">  
}
```

getNextToken

- When we run `javacc` on the input file `MyParser.jj`, it creates the class `MyParserTokenManager`
- The class `MyParserTokenManager` contains the static method `getNextToken()`
- Every call to `getNextToken()` returns the next token in the input stream.
- When `getNextToken` is called, a regular expression is found that matches the next characters in the input stream.

getNextToken

What if more than one regular expression matches the input stream? For example:

```
TOKEN :
{
    <ELSE: "else">
  | <IDENTIFIER: ("a"-"z")+>
}
```

- Use the *longest match*
 - "elsed" should match to IDENTIFIER, not to ELSE followed by the identifier "d"
- If two matches have the same length, use the rule that appears first in the .jj file
 - "else" should match to ELSE, not IDENTIFIER

Tokens

- Each call to `getNextToken` returns a `Token` object
- `Token` class is automatically created by JavaCC.
- Variables of type `Token` contain the following public variables:
 - `public int beginLine, beginColumn, endLine, endColumn;` - the location of the token in the input file
 - `public String image;` - the text that was matched to create the token.
 - `public int kind;` - the type of token.
- When `javacc` is run on the file `MyParser.jj`, a file `MyParserConstants.java` is created, which contains the symbolic names for each constant:

```
public interface MyParserConstants {  
    int EOF = 0;  
    int SEMIC = 1;  
    int ASSIGN = 2;  
    int PRINT = 3;  
    ...  
}
```

Example

The following straight-line programming language from Appel's book (Chapter 1) will be used to illustrate some examples of using JavaCC.

<i>Stm</i>	→	<i>Stm ; Stm</i>	(CompoundStm)
<i>Stm</i>	→	<i>id := Exp</i>	(AssignStm)
<i>Stm</i>	→	<i>print (ExpList)</i>	(PrintStm)
<i>Exp</i>	→	<i>id</i>	(IdExp)
<i>Exp</i>	→	<i>num</i>	(NumExp)
<i>Exp</i>	→	<i>Exp Binop Exp</i>	(OpExp)
<i>Exp</i>	→	<i>(Stm , Exp)</i>	(EseqExp)
<i>ExpList</i>	→	<i>Exp , ExpList</i>	(PairExpList)
<i>ExpList</i>	→	<i>Exp</i>	(LastExpList)
<i>Binop</i>	→	<i>+</i>	(Plus)
<i>Binop</i>	→	<i>-</i>	(Minus)
<i>Binop</i>	→	<i>×</i>	(Times)
<i>Binop</i>	→	<i>/</i>	(Div)

Specifying Tokens For the Straight Line Programming Language

```
TOKEN : /* Keywords and punctuation */
{
  < SEMIC : ";" >
| < ASSIGN : ":@" >
| < PRINT : "print" >
| < LBR : "(" >
| < RBR : ")" >
| < COMMA : "," >
| < PLUS_SIGN : "+" >
| < MINUS_SIGN : "-" >
| < MULT_SIGN : "*" >
| < DIV_SIGN : "/" >
}
```

```
TOKEN : /* Numbers and identifiers */
{
  < NUM : (<DIGIT>)+ >
| < #DIGIT : ["0" - "9"] >
| < ID : (<LETTER>)+ >
| < #LETTER : ["a" - "z", "A" - "Z"] >
}
```

```
TOKEN : /* Anything not recognised so far */
{
  < OTHER : ~[] >
}
```

SKIP Rules

- Tell JavaCC what to ignore (typically whitespace) using SKIP rules
- SKIP rule is just like a TOKEN rule, except that no TOKEN is returned.

```
SKIP:
{
    < regularexpression1 >
  | < regularexpression2 >
  | ...
  | < regularexpressionn >
}
```

- For example:

```
SKIP : /* Ignoring spaces/tabs/newlines */
{
    " "
  | "\t"
  | "\n"
  | "\r"
  | "\f"
}
```

JavaCC States

- Comments can be dealt with using SKIP rules
- How could we skip over 1-line C++ Style comments?

```
// This is a comment
```

- Using a SKIP rule:

```
SKIP :  
{  
    < "//" (~["\n"])* "\n" >  
}
```

- Writing a regular expression to match multi-line comments (using /* and */) is much more difficult
- Writing a regular expression to match nested comments is impossible
- Therefore need to use *JavaCC States*

JavaCC States

- We can label each TOKEN and SKIP rule with a "state"
- Unlabeled TOKEN and SKIP rules are assumed to be in the *default state* (named DEFAULT)
- Can switch to a new state after matching a TOKEN or SKIP rule using the : NEWSTATE notation
- For example:

```
SKIP :  
{  
    < "/*" > : IN_COMMENT  
}
```

```
<IN_COMMENT> SKIP :  
{  
    < "*/" > : DEFAULT  
    | < ~[] >  
}
```

Actions in TOKEN and SKIP

- How do we deal with nested comments?
- Through the use of *actions*
- We can add Java code to any SKIP or TOKEN rule
- That code will be executed when the SKIP or TOKEN rule is matched.
- Any methods/variables defined in the TOKEN_MGR_DECLS section can be used by these actions.
- For example:

```
TOKEN_MGR_DECLS :
{
    static int commentNesting = 0;
}

SKIP : /* COMMENTS */
{
    "/"* { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/"* { commentNesting++; }
    | "*"/* { commentNesting--;
        if (commentNesting == 0)
            SwitchTo(DEFAULT);
    }
    | <~ []>
}
```

The TokenManager Code

```
/******  
**** SECTION 1 - OPTIONS ****  
*****/  
  
options { JAVA_UNICODE_ESCAPE = true; }  
  
/******  
**** SECTION 2 - USER CODE ****  
*****/  
  
PARSER_BEGIN(SLPTokeniser)  
  
public class SLPTokeniser {  
  
    public static void main(String args[]) {  
  
        SLPTokeniser tokeniser;  
        if (args.length == 0) {  
            System.out.println("Reading from standard input . . .");  
            tokeniser = new SLPTokeniser(System.in);  
        } else if (args.length == 1) {  
            try {  
                tokeniser = new SLPTokeniser(new java.io.FileInputStream(args[0]));  
            } catch (java.io.FileNotFoundException e) {  
                System.err.println("File " + args[0] + " not found.");  
                return;  
            }  
        } else {  
            System.out.println("SLP Tokeniser: Usage is one of:");  
            System.out.println("        java SLPTokeniser < inputfile");  
            System.out.println("OR");  
            System.out.println("        java SLPTokeniser inputfile");  
            return;  
        }  
    }  
}
```

The TokenManager Code

```
/*
 * We've now initialised the tokeniser to read from the appropriate place,
 * so just keep reading tokens and printing them until we hit EOF
 */

for (Token t = getNextToken(); t.kind!=EOF; t = getNextToken()) {

    // Print out the actual text for the constants, identifiers etc.
    if (t.kind==NUM)
    {
        System.out.print("Number");
        System.out.print("(" + t.image + " ");
    }
    else if (t.kind==ID)
    {
        System.out.print("Identifier");
        System.out.print("(" + t.image + " ");
    }
    else
        System.out.print(t.image + " ");
}

}

PARSER_END(SLPTokeniser)
```