1 Introduction to JavaCC

JavaCC (Java Compiler Compiler) is a parser generator which takes as input a set of token definitions, a grammar, and a set of actions and produces a Java program which will parse input according to the grammar and execute the actions as appropriate.

This document is a quick introduction to this tool. The full documentation for JavaCC is indexed at

`https://javacc.dev.java.net/doc/docindex.html`

and may be viewed with any Web browser. All the examples mentioned below are subdirectories of `/soi/sw/courses/daveb/IN2009/` and are also available on WebCT. `README` files in each directory show how to compile and run the examples.

2 JavaCC

The format of the JavaCC grammar input file (file suffix '`.jj`') to be used is as follows (some of these definitions are subsets; see the full documentation for the complete story):

```
PARSER_BEGIN(Parser-name)

public class Parser-name {
}

PARSER_END(Parser-name)

/* Lexical items (ie token definitions). */

Token-definitions

/* Grammar rules. */

Syntax-definitions
```

Each element of this format is now described in more detail:

Parser-name

This name must be the same in all three places, and is used to produce a class called `Parser-name` which performs parsing. This name is also used as the prefix for the parsing-related JavaCC-generated files and classes.

Token-definitions

Token definitions are prefixed either with `SKIP` or `TOKEN`. Those prefixed `SKIP` define regular expressions for strings which, when matched, will not be passed to the parser. Typically these are 'white space' characters such as space and newline, or comments.

```
SKIP :   /* These items will be skipped by the lexical analyser. */
{
  "\n" | " " | "\t"
}
```

The `TOKEN` prefix is used to define named tokens so that they can be referred to in the syntax definitions later in the file. The generated lexical analyser matches tokens according to these `TOKEN` definitions and passes them to the parser generated from the syntax definitions. The general form of `TOKEN` definitions is:

```
TOKEN :
{
  regexpr_spec | regexpr_spec | ...
}
```

where the '|' represents alternatives.

A `regexpr_spec` has the form:

```
< NAME : regexp_choices>
```

and this definition then allows the expression `<NAME>` to be used elsewhere in token or syntax definitions to represent the regular expression `regexp_choices` defined. A '`#`' preceding the `NAME` restricts the use of `<NAME>` to other `TOKEN` definitions and is used to name expressions which are to be used *only* in defining lexical items. The `regexp_choices` have the form

```
regexp | regexp | ...
```

Once again the '|' separates alternatives.

A `regexp` (recursively) has any of the forms:

```
regexp regexp ...           matches first regexp then second and so on
( regexp )*                 matches zero or more regexps
( regexp )+                 matches one or more regexps
( regexp )?                 matches regexp or empty string (ie optional)
< NAME >                    matches expression defined as NAME
any string literal          matches the string (eg "bool")
[ char_list ]               matches any character in the char_list
```

A `char_list` is either quoted single characters (eg `";"`) or quoted character ranges (eg `"a"-"z"`) separated by commas. A '`~`' before the `[...]` matches any character not in the `char_list`. The expression `~[]` matches any single character.

Examples

Some expressions for matching identifiers and integer literals:

```
TOKEN : /* Matches signed and unsigned integers */
{
 <INTEGER_LITERAL: ("+"|"-")?(["0"-"9"])+ >
}

TOKEN :
{
 < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
}

TOKEN : /* Note LETTER and DIGIT restricted to other TOKEN definitions. */
{
 < #LETTER: [ "a"-"z", "A"-"Z" ] > | < #DIGIT: [ "0"-"9"] >
}
```

Finally, there is a special token `<EOF>` which matches the end of the input file to the parser and is usually deployed in one of the syntax-definitions (see below).

The `lextest` directory contains a simple token matcher (you'll need to read the rest to completely understand how it works, but a it does show token matching clearly).

Syntax-definitions

The syntax definitions are EBNF productions written in a stylised form suited to production of a parser in Java code. For a definition written in usual EBNF like this:

```
non-terminal-name   ->   right-hand-side
```

The general form written in JavaCC is:

```
java_return_type non-terminal-name ( java_parameter_list ) :
java_block
{ expansion_choices }
```

The first line gives the name of the non-terminal being defined (must be a Java-style identifier). The rest of the line dresses up this name as a Java-like method declaration; the form we will use most often is:

```
void non-terminal-name () :
```

The more general form can be used to allow values to be passed up and down the parse tree while the parse takes place (up via return values and down via parameters), since (as we will see) non-terminals in the productions are written like Java method calls.

The second line (`java_block`) introduces some Java code which is usually used to declare variables which will be used later in the production (see below).

The third line represents the EBNF `right-hand-side`, once again written in a stylised form as a series of expansion choices. As in EBNF, the `expansion_choices` have the form:

```
expansion | expansion | ...
```

where the '`|`' separates alternatives.

Each expansion (recursively) may be of the form:

```
expansion expansion ... matches first expansion then second and so on
( expansion_choices )*  matches zero or more expansion_choices
( expansion_choices )+  matches one or more expansion_choices
( expansion_choices )?  matches expansion_choices or empty string (ie optional)
[ expansion_choices ]   matches expansion_choices or empty string (ie optional)
regexp                  matches the token matched by the regexp
java_id = regexp        matches the token matched by the regexp and assigns
                        it to java_id
non-terminal-name (optional_params)
                        matches the non-terminal, passing in any parameters
java_id = non-terminal-name (optional_params)
                        matches the non-terminal and assigns any returned
                        value to java_id
```

The `java_id` will usually be declared in the `java_block`.

Any of these expansions may be followed by some Java code written in `{...}` and this code (often called an action) will be executed when the generated parser matches the expansion.

Example: bracket matching

A EBNF for simple brace bracket matching is:

```
Input          ->  MatchedBraces NEWLINE EOF
MatchedBraces  ->  { [ MatchedBraces ] }
```

This can be written in JavaCC form as:

```
void Input() :
{}
{
  MatchedBraces() ("\n"|"\r")* <EOF>
}

void MatchedBraces() :
{}
{
  "{" [ MatchedBraces() ] "}"
}
```

The first production, for non-terminal `Input`, matches `MatchedBraces` followed by a newline or return character (notice this is a `regexp`) followed by the end-of-file. The second production matches a left brace followed by either another `MatchedBraces` followed by a right brace, or a right brace alone.

We might wish to count the levels of nesting of braces. To do this we introduce some action code and use the facility for returning values from non-terminal matching:

```
void Input() :
{ int count; }
{
  count=MatchedBraces() ("\n"|"\r")* <EOF>
  { System.out.println("The levels of nesting is " + count); }
}

int MatchedBraces() :
{ int nested_count=0; }
{
  "{" [ nested_count=MatchedBraces() ] "}"
  { nested_count = nested_count + 1; return nested_count; }
}
```

Accessing Token values

A syntax-definition `java_id` (here called '`t`') may be declared as type `Token` and then assigned with a token reference in the syntax-definition. Token objects carry fields which can be used usefully in the action code, including the field '`image`', which is the string matched for the token, and the field '`kind`', which can be used to index the array '`tokenImage`' to obtain a printable representation of the kind of the token. Given some token definitions, the following is a simple syntax-definition to match and print them, and demonstrates the use of a `Token` object (here called '`t`'):

```
void TokenList() :
{Token t;}
{
   (
     (t = <KEYWHILE> | t = <INTEGER_LITERAL> | t = <KEYTRUE> | t = <KEYFALSE> |
         t = <IDENTIFIER>)
           { System.out.println ("token found: "+ tokenImage[t.kind]+
                                 " ('"+t.image+"')"); }
   )* <EOF>
}
```

This can be found in the `lextest` directory.

A complete example

This is a complete JavaCC grammar, showing `SKIP` and `TOKEN` definitions, and showing the braces defined as named tokens `LBRACE` and `RBRACE`:

```
PARSER_BEGIN(Simple)

public class Simple {
}

PARSER_END(Simple)

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

TOKEN :
{
  <LBRACE: "{"> | <RBRACE: "}">
}

void Input() :
{ int count; }
{
  count=MatchedBraces() <EOF>
  { System.out.println("The levels of nesting is " + count); }
}

int MatchedBraces() :
{ int nested_count=0; }
{
  <LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
  { nested_count = nested_count + 1; return nested_count; }
}
```

To use this grammar to create a parser, we must run the specification through JavaCC, and provide a Java class and `main()` which creates a parser object and starts the parse. Such a class might be written simply:

```
class DoSimple {
  public static void main(String args[]) throws ParseException {
    Simple parser = new Simple(System.in);
    parser.Input();
  }
}
```

The parser class constructor (here `Simple`) takes the source of input as its argument, so this parser, when called, reads from the standard input `System.in` (your keyboard unless redirected). The parser is called by invoking the non-terminal name you want recognised (here `Input()`).

This example can be found in the `braces` directory.