JavaCC: SimpleExamples

This directory contains five examples to get you started using JavaCC. Each example is contained in a single grammar file and is listed below: (1) Simple1.jj, (2) Simple2.jj, (3) Simple3.jj, (4) Simple4.jj, (5) NL_Xlator.jj.

Summary instructions

If you are a parser and lexical analyzer expert and can understand the examples by just reading them, the following instructions show you how to get started with JavaCC. The instructions below are with respect to Simple1.jj, but you can build any parser using the same set of commands.

1. Run javacc on the grammar input file to generate all Java files that implement the parser and lexical analyzer (or token manager):

```
javacc Simple1.jj
```

2. Now compile the resulting Java programs:

```
javac *.java
```

3. The parser is now ready to use. To run the parser, type:

```
java Simple1
```

DETAILED DESCRIPTION OF Simple1.jj

This is a simple JavaCC grammar that recognizes a set of left braces followed by the same number of right braces, by zero or more line terminators and finally an end of file.

```
PARSER_BEGIN(Simple1)

public class Simple1 {

  public static void main(String args[]) throws ParseException {

    Simple1 parser = new Simple1(System.in);

    parser.Input();

  }

}

PARSER_END(Simple1)
```

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

The `Simple1` parser is designed to take input from *standard input*. `Simple1` recognizes matching braces followed by zero or more line terminators and then an end of file.

Examples of legal strings in this grammar are: `"{}", "{{{{{}}}}}".`

Examples of illegal strings are: `"{{{{", "{}{}", "{}}", "{{}{}}", "{ }", "{x}".`

The Java compilation unit is enclosed between `PARSER_BEGIN(parser-name)` and `PARSER_END(parser-name)`. This compilation unit can be of arbitrary complexity. The only constraint is that *it must define a class called* `parser-name,` where `parser-name` is the argument of `PARSER_BEGIN` and `PARSER_END`. This is the name that is used as the prefix for the Java files generated by the parser generator. The parser code that is generated is inserted immediately before the closing brace of the class called "parser-name". In the above example, the class in which the parser is generated contains a main program. *This main program creates an instance of the parser object* (an object of type `Simple1`) *by using a constructor that takes one argument of type* `java.io.InputStream` (`System.in` in this case).

The main program then makes a call to the non-terminal in the grammar that it would like to parse, `Input` in this case. All non-terminals have equal status in a JavaCC generated parser, and hence one may parse with respect to any grammar non-terminal. In this example, there are two productions that define the non-terminals, `Input` and `MatchedBraces`, respectively. *In JavaCC, non-terminals are written and implemented as Java methods*. When the non-terminal is used on the left-hand side of a production, it is considered to be declared and its syntax follows the Java syntax. On the right-hand side, its use is similar to a Java method call.

Each production defines its left-hand side non-terminal followed by a colon. This is followed by a bunch of declarations and statements within braces (in both cases in the above example, there are no declarations and hence this appears as "`{}`") which are generated as common declarations and statements into the generated method. This is then followed by a set of expansions also enclosed within braces.

Lexical tokens (regular expressions) in a JavaCC input grammar are either simple strings ("`{`", "`}`", "`\n`", and "`\r`" in the above example), or a more complex regular expression. In our example above, there is one such regular expression `<EOF>` *which is matched by the end of file*.

All complex regular expressions are enclosed within angular brackets (<, >).

The first production above says that the non-terminal `Input` expands to the non-terminal `MethodBraces` followed by zero or more line terminators (`"\n"` or `"\r"`) and then the end of file.

The second production above says that the non-terminal `MatchedBraces` expands to the token `{` followed by an optional nested expansion of `MatchedBraces` followed by the token `}`. Square brackets `[...]` in a JavaCC input file indicate that the `...` is optional. This may also be written as `(...)?`. These two forms are equivalent. Other structures that may appear in expansions are:

```
e1 | e2 | e3 | ...   : A choice of e1, e2, e3, etc.

( e )+               : One or more occurrences of e

( e )*               : Zero or more occurrences of e
```

Note that these may be nested within each other, so we can have something like:

```
(( e1 | e2 )* [ e3 ] ) | e4
```

To build this parser, simply run JavaCC on this file and compile the resulting Java files:

```
javacc Simple1.jj

javac *.java
```

Now you should be able to run the generated parser. Make sure that the current directory is in your `CLASSPATH` and type:

```
java Simple1
```

Now type a sequence of matching braces followed by a return and an end of file (`CTRL-D` on UNIX machines). If this is a problem on your machine, you can create a file and pipe it as input to the generated parser in this manner (piping also does not work on all machines – if this is a problem, just replace "`System.in`" in the grammar file with

new FileInputStream(testfile)

and place your input inside this file):

```
java Simple1 < myfile
```

Also try entering illegal sequences such as mismatched braces, spaces, and carriage returns between braces as well as other characters and take a look at the error messages produced by the parser.

DETAILED DESCRIPTION OF Simple2.jj

`Simple2.jj` is a minor modification to `Simple1.jj` to allow white space characters to be interspersed among the braces. So then input such as: `{{   }\n}\n\n` will now be legal.

```
PARSER_BEGIN(Simple2)

public class Simple2 {

  public static void main(String args[]) throws ParseException {

    Simple2 parser = new Simple2(System.in);

    parser.Input();

  }

}

PARSER_END(Simple2)

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

void Input() :

{}

{

  MatchedBraces() <EOF>

}

void MatchedBraces() :

{}

{

  "{" [ MatchedBraces() ] "}"

}
```

The difference between this file and `Simple1.jj` is that *this file contains a lexical specification* - the portion that starts with `SKIP`. Within this region are 4 regular expressions - space, tab, newline, and return. *This says that matches of these regular expressions are to be ignored (and not considered for parsing).* Hence whenever any of these 4 characters are encountered, they are just thrown away. In addition to `SKIP`, JavaCC has three other lexical specification regions. These are:

- `TOKEN`: used to specify lexical tokens (see next example)
- `SPECIAL_TOKEN`: used to specify lexical tokens that are to be ignored during parsing.

In this sense, SPECIAL_TOKEN is the same as SKIP. However, these tokens can be recovered within parser actions to be handled appropriately.

- MORE: specifies a partial token. A complete token is made up of a sequence of MORE's followed by a TOKEN or SPECIAL_TOKEN.

Please take a look at some of the more complex grammars such as the Java grammars for examples of usage of these lexical specification regions. You may build Simple2 and invoke the generated parser with input from the keyboard as standard input. *You can also try generating the parser with the various debug options turned on and see what the output looks like*. To do this type:

```
javacc -debug_parser Simple2.jj

javac Simple2*.java

java Simple2
```

Then type:

```
javacc -debug_token_manager Simple2.jj

javac Simple2*.java

java Simple2
```

Note that token manager debugging produces a lot of diagnostic information and it is typically used to look at debug traces a single token at a time.

DETAILED DESCRIPTION OF Simple3.jj

This is the final version of our matching brace detector.

```
PARSER_END(Simple3)

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>

  { return ++nested_count; }

}
```

This example illustrates the use of the `TOKEN` region for specifying lexical tokens. In this case, "{" and "}" are defined as tokens and given names `LBRACE` and `RBRACE` respectively. T*hese labels can then be used within angular brackets (as in the example) to refer to this token.* Typically such token specifications are used for complex tokens such as identifiers and literals. Tokens that are simple strings are left as is (in the previous examples).

This example also *illustrates the use of actions in the grammar productions*. The actions inserted in this example count the number of matching braces. Note the use of the declaration region to declare variables count and nested_count. Also note how the non-terminal `MatchedBraces` returns its value as a function return value.

DETAILED DESCRIPTION OF NL_Xlator.jj

This example goes into the details of writing regular expressions in JavaCC grammar files. It also illustrates a slightly more complex set of actions that translate the expressions described by the grammar into English.

```
PARSER_BEGIN(NL_Xlator)

public class NL_Xlator {

  public static void main(String args[]) throws ParseException {

    NL_Xlator parser = new NL_Xlator(System.in);

    parser.ExpressionList();

  }

}

PARSER_END(NL_Xlator)

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

TOKEN :

{

  < ID: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )* >

|

  < NUM: ( ["0"-"9"] )+ >

}



void ExpressionList() :

{

     String s;

}

{

  {

     System.out.println("Type an expr. followed by a \";\" or ^D to quit:\n");

  }

  ( s=Expression() ";"
     {
       System.out.println(s);
       System.out.println("Type an expr. followed by a \";\" or ^D to quit:");
       System.out.println("");
     }
  )*    <EOF>
}
```

```
String Expression() :
{
          java.util.Vector termimage = new java.util.Vector();
          String s;
}
{
          s=Term()
          { termimage.addElement(s); }
          (
                    "+" s=Term()
                    { termimage.addElement(s); }
          )*
          {
            if (termimage.size() == 1) {
              return (String)termimage.elementAt(0);
            }
            else {
             s = "the sum of " + (String)termimage.elementAt(0);
              for (int i = 1; i < termimage.size()-1; i++) {
                    s += ", " + (String)termimage.elementAt(i);
              }
              if (termimage.size() > 2) {  s += ","; }
              s += " and " + (String)termimage.elementAt(termimage.size()-1);
             return s;
            }
          }
}
```

```
String Term() :

{

          java.util.Vector factorimage = new java.util.Vector();

          String s;

}

{

  s=Factor()

          { factorimage.addElement(s); }

  ( "*" s=Factor()

          { factorimage.addElement(s); }

  )*

  {

    if (factorimage.size() == 1) {

      return (String)factorimage.elementAt(0);

    }

    else {

      s = "the product of " + (String)factorimage.elementAt(0);

      for (int i = 1; i < factorimage.size()-1; i++) {

         s += ", " + (String)factorimage.elementAt(i);

      }

    if (factorimage.size() > 2) { s += ",";}

    s+=" and " + (String)factorimage.elementAt(factorimage.size()- 1);

    return s;

    }

  }

}
```

```
String Factor() :
{
         Token t;
         String s;
}
{
  t=<ID> { return t.image;}
|
  t=<NUM> {return t.image; }
|
  "(" s=Expression() ")" { return s; }
}
```

The new concept in the above example is the use of more complex regular expressions. The regular expression:

```
 <ID: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )*>
```

creates a new regular expression whose name is `ID`. *This can be referred anywhere else in the grammar simply as* `<ID>`. What follows in square brackets are a set of allowable characters - in this case it is any of the lower or upper case letters or the underscore. This is followed by 0 or more occurrences of any of the lower or upper case letters, digits, or the underscore. Other constructs that may appear in regular expressions are:

```
(r1)+             One or more occurrences of r1
(r1)?              An optional occurrence of r2
(r1 | r2 | ... )   Any one of r1, r2, ...
```

A construct of the form `[...]` is a pattern that is matched by the characters specified in `...` These characters can be individual characters or character ranges. A ~ before this construct is a pattern that *matches any character not specified* in ... . Therefore:

- `["a"-"z"] matches all lower case letters`
- `~[] matches any character`
- `~["\n","\r"] matches any character except the new line characters`

When a regular expression is used in an expansion, it takes a value of type `Token`. This is generated into the generated parser directory as `Token.java`. In the above example, we have defined a variable of type `Token` and assigned the value of the regular expression to it.

DETAILED DESCRIPTION OF IdList.jj

This example illustrates an important *attribute* of the SKIP specification. The main point to note is that the regular expressions in the SKIP specification *are only ignored between tokens* and *not within tokens*.

```
PARSER_BEGIN(IdList)

public class IdList {

  public static void main(String args[]) throws ParseException {

    IdList parser = new IdList(System.in);

    parser.Input();

  }

}

PARSER_END(IdList)


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


TOKEN :

{

  < ID: ["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z","0"-"9"] )* >

}

void Input() :

{}

{

  ( <ID> )+ <EOF>

}
```

This grammar *accepts any sequence of identifiers with white space in between*. A legal input for this grammar is: `abc  xyz123  A  B  C  \t\n  aaa`. This is because any number of the `SKIP` regular expressions are allowed in between consecutive `<ID>`'s. However, the following is **not** a legal input: `xyz  123`. This is because the space character after "xyz" is in the `SKIP` category and therefore causes one token to end and another to begin. This requires "123" to be a separate token and hence does not match the grammar. If spaces were OK within `<ID>`'s, then all one has to do is to replace the definition of `ID`to:

```
TOKEN :

{

  < ID: ["a"-"z","A"-"Z"] ( (" ") * ["a"-"z","A"-"Z","0"-"9"] )* >

}
```

Note that having a space character within a `TOKEN` specification does not mean that the space character cannot be used in the `SKIP` specification. All this means is *that any space character that appears in the context where it can be placed within an identifier will participate in the match for* `<ID>`, whereas all other space characters will be ignored. The details of the matching algorithm are described in the JavaCC documentation in the web pages.