JavaCC

(Java Compiler Compiler)

JavaCC and Parser generation

- JavaCC (Java Compiler Compiler) is an open source parser generator for the Java programming language.
- JavaCC is similar to Yacc
- It generates a parser (and a lexical analyser) for a formal grammar provided in EBNF (Extended BNF) notation
- The output is a Java source code.
- Unlike Yacc, however, JavaCC generates top-down parsers, which limits it to the **LL(k)** class of grammars (in particular, left recursion cannot be used).

A first example: adding integers

- As a first example we'll add lists of numbers such as

  99 + 42 + 0 + 15

- We’ll allow spaces and line breaks anywhere except within numbers
- The code example of this section will be parts of one file called `Adder.jj`
- This file contains the JavaCC specification for the parser and the lexical analyser and will be used as input to JavaCC the program

Part 1: Options and class declaration

```
options{
    LOOKAHEAD = 1;
}
PARSER_BEGIN(Adder)
    class Adder{
    public static void main(String[] args)
    throws ParseException, TokenMgrError{
        Adder parser = new Adder(System.in);
        parser.Start();
    }
}
PARSER_END(Adder)
```

Part 1: Options and class declaration

- After an initial comment there is a section for options
- Next comes a fragment of a Java class named `Adder` (this is enclosed between PARSER_BEGIN and PARSER_END)
- This fragment is not the complete Adder class - JavaCC will add declarations to this class as a part of the generation process
- The main method is declared to potentially throw two classes of exceptions:
  - ParseException, and
  - TokenMgrError;

  Both these classes will be generated by JavaCC

Part 2: specifying the lexical analyser

For this simple example, the lexical analyser can be specified by the following four lines

```
SKIP: {" "}

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

TOKEN: {<PLUS: "+">}

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

Part 2: specifying the lexical analyser

- The first line says that **space characters** constitute tokens, but they have to be skipped, i.e. they are not to be passed on to the parser.
- The second line says the same thing about **line breaks**. Different operating systems represent line breaks with different character sequences. We tell JavaCC about all possibilities, separating them with a vertical bar.
- The third line tells that a **plus sign** alone is a token, and gives a *symbolic name* (`PLUS`) to this kind of token
- Similarly, the fourth line tells JavaCC about the syntax to be used for **numbers** and gives a symbolic name (`NUM`)
- There is one more kind of token that the generated lexical analyser can produce, this as the symbolic name `EOF` and represents the **end of the input** sequence. There is no need to have a regular expression production for EOF; JavaCC deals with the end of the file automatically.

Part 3: specifying the parser

```
void Start():
{}
{
    <NUM>
    (
        <PLUS> <NUM>
    )*
    <EOF>
}
```

Part 3: specifying the parser

- The specification of the parser consists of what is called **a BNF production**. It looks a little like a Java method definition.
- This BNF production specifies the legitimate sequences of token kinds in error free input.
- The production says that these sequences begin with a NUM token, end with an EOF token and in-between consist of zero or more sub-sequences each consisting of a PLUS token followed by a NUM token.
- As it stands, the parser will only detect whether or not the input sequence is error free, it doesn’t actually add up the numbers, yet.

Generating the parser and lexical analyser

- Having constructed the Adder.jj file, we invoke JavaCC on it (`javacc Adder.jj`)

```
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file Adder.jj . . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
```

Generating the parser and lexical analyser

This process generates 7 Java classes, each in its own file:

- TokenMgrError
- ParseException
- Token
- SimpleCharStream
- AdderConstants
- AdderTokenManager – the lexical analyser
- Adder – the parser

Generating the parser and lexical analyser

- TokenMgrError is a simple error class; it is used for errors detected by the lexical analyser and is a subclass of Throwable.
- ParseException is another error class; it is used for errors detected by the parser and is a subclass of Exception and hence of Throwable.
- Token is a class representing tokens. Each Token object has an *integer* field kind that represents the *kind* of the token (in our example PLUS, NUM, or EOF) and a *String* field *image*, which represents the sequence of characters from the input file that the token represents.
- SimpleCharStream is an adapter class that delivers characters to the lexical analyser.
- AdderConstants is an interface that defines a number of classes used in both the lexical analyser and the parser

Compiling the parser

- We can now compile these classes with a Java compiler:

```
javac *.java
```

- We run the parser by typing

```
java  Adder < input.txt
```

Running the parser

- There is a lexical error found. Lexical errors only happen when there is an unexpected sequence of characters in the input.

  EX: “123 - 456\n”

  In this case the program will throw a `TokenMgrError`
- There is a parsing error. This happens when the sequence of tokens does not match

  the specification of Start. EX: “123 ++ 456\n”, or “123 456\n”, or “\n”

  In this case the program will throw a `ParseException`
- The input contains a sequence of tokens matching `Start`’s specification. In this case, no exception is thrown and the program simply terminates