# Chapter 1

# Introduction to JavaCC

## 1.1 JavaCC and Parser Generation

JavaCC is a parser generator and a lexical analyzer generator. Parsers and lexical analysers are software components for dealing with input of character sequences. Compilers and interpreters incorporate lexical analysers and parsers to decipher files containing programs, however lexical analysers and parsers can be used in a wide variety of other applications, as I hope the examples in this book will illustrate.

So what are lexical analysers and parsers? Lexical analysers can break a sequence of characters into a subsequences called *tokens* and it also classifies the tokens. Consider a short program in the C programming language.

```
int main() {
      return 0 ;
}
```

The lexical analyser of a C compiler would break this into the following sequence of tokens

“int”, “ ”, “main”, “(”, “)”,
“ ”, “{”, “\n”, “\t”, “return”
“ ”, “0”, “ ”, “;”, “\n”,
“}”, “\n”, “” .

The lexical analyser also identifies the *kind* of each token; in our example the sequence of

token kinds might be

KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF .

The token of kind EOF represents the end of the original file. The sequence of tokens is then passed on to the parser. In the case of C, the parser does not need all the tokens; in our example, those clasified as SPACE are not passed on to the parser. The parser then analyses the sequence of tokens to determine the structure of the program. Often in compilers, the parser outputs a tree representing the structure of the program. This tree then serves as an input to components of the compiler responsible for analysis and code generation. Consider a single statement in a program

fahrenheit = 32.0 + 9.0 * celcius / 5.0 ; .

The parser analyzes the statement according to the rules of the language and produces a tree

DIAGRAM TBD

The lexical analyser and parser also are responsible for generating error messages, if the input does not conform to the lexical or syntactic rules of the language.

JavaCC itself is not a parser or a lexical anaylzer but a *generator*. This means that it outputs lexical analyzers and parser according to a specification that it reads in from a file. JavaCC produces lexical analysers and parsers written in Java. See Figure TBD

DIAGRAM TBD

Parsers and lexical analysers tend to be long and complex components. A software engineer writing an efficient lexical analyser or parser directly in Java has to carefully consider the interactions between rules. For example in a lexical analyser for C, the code for dealing with integer constants and floating-point constants can not be sperated, since a floating constant starts off the same as a floating-point constant. Using a parser generator such as JavaCC, the rules for integer constants and floating-point constants are written separately and the commonality between them is extracted during the generation process. This increased modularity means that specification files are easier to write, read, and modify compared with a hand-written Java programs. By using a parser generator like JavaCC, the software engineer can save a lot of time and produce software components of better quality

## 1.2 A first example — adding integers

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

$$99 + 42 + 0 + 15 \quad .$$

We'll allow spaces and line breaks anywhere except within numbers. Otherwise the only characters in the input must be the 10 digits or the plus sign.

In the rest of this section the code examples 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.

### 1.2.1 Options and class declaration

The first part of the file is

```
/* adder.jj Adding up numbers */
options {
      STATIC = false ;
}
PARSER_BEGIN(Adder)
      class Adder {
           static void main( String[] args )
           throws ParseException, TokenMgrError {
                Adder parser = new Adder( System.in ) ;
                parser.Start() ; }
      }
PARSER_END(Adder)
```

After an initial comment is a section for options; all the standard values for JavaCC's options are fine for this example, except for the STATIC option, which defaults to true. More information about the options can be found in the JavaCC documentation, later in this book, and in the FAQ. Next comes a fragment of a Java class named Adder. What you see here is not the complete Adder class; JavaCC will add declarations to this class as part of the generation process. The main method is declared to potentially throw two classes of exceptions: ParseException and TokenMgrError; these classes will be generated by JavaCC.

### 1.2.2 Specifying a lexical analyser

We'll return to the main method later, but now let's look at the specification of the lexical analyser. In this simple example, the lexical analyzer can be specified in only four lines

```
SKIP : { " " }
SKIP : { "\n" | "\r" | "\r\n" }
TOKEN : { < PLUS : "+" > }
TOKEN : { < NUMBER : (["0"-"9"])+ > }
```

The first line says that space characters constitute tokens, but are to be skipped, that is, 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; in Unix and Linux, a newline character ("\n") is used, in DOS and Windows a carriage return ("\r") followed by a newline is used, and in older Macintoshes a carriage return alone is used. We tell JavaCC about all these possibilities, separating them with a vertical bar. The third line tells JavaCC that a plus sign alone is a token and gives a symbolic name to this kind of token: PLUS. Finally the fourth line tells JavaCC about the syntax to be used for numbers and gives a symbolic name, NUMBER, to this kind of token. If you are familiar with regular expressions in a language such as Perl or Java's regular expression package, then the specification of NUMBER tokens will probably be decipherable. We'll take a closer look at the regular expression (["0"-"9"])+. The ["0"-"9"] part is a regular expression that matches any digit, that is, any character whose unicode encoding is between that of 0 and that of 9. A regular expression of the form $(x)+$ matches any sequence of one or more strings, each of which is matched by regular expression $x$. So the regular expression (["0"-"9"])+ matches any sequence of one or more digits. Each of these four lines is called a *regular expression production*.

There is one more kind of token that the generated lexical analyser can produce, this has 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.

Consider an input file containing the following characters:

"123 + 456\n" .

The generated lexical analyser will find seven tokens: a NUMBER, a space, a PLUS, another space, another NUMBER, a newline, and an EOF. Of these, the tokens specified by regular expression productions marked SKIP are not passed on to the parser, so the parser sees only the sequence

NUMBER, PLUS, NUMBER, EOF .

Suppose that instead of a legitimate input file, we had one with unexpected characters, for example

"123 - 456\n" .

After finding the first space, the lexical analyser will confront a minus sign. Since no specified token can start with a minus sign, the lexical analyser will throw an exception of class TokenMgrError.

Now what if the input contains a character sequence

"123 ++ 456\n" .

This time the sequence lexical analyser can still deliver a sequence of tokens

NUMBER, PLUS, PLUS, NUMBER, EOF .

It is not the up to the lexical analyser to determine whether its sequence of tokens is sensible or not, that is usually left up to the parser. The parser that we are about to specify will detect the error after the lexical analyser has delivered the second PLUS token and will not request any more tokens from the lexical analyser after that. So the actual sequence of tokens delivered to the parser would be

NUMBER, PLUS, PLUS .

Skipping a character or character sequence is not the same as ignoring it. Consider an input sequence

"123 456\n"

The lexical analyser will recognize three tokens: two NUMBER tokens and a token in between coresponding to the space character; again the parser will detect an error.

### 1.2.3 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.

---

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

```
}
```

This BNF production specifies the legitimate sequences of token kinds in error free input. The production says that these sequences begin with a NUMBER token, end with an EOF token and in-between consist of zero or more subsequences each consisting of a PLUS token followed by a NUMBER token.

As is stands, the parser will only detect whether or not the input sequence is error free, it doesn't actually add up the numbers, yet. Will modify the parser soon to correct this, but first, let's generate the Java components and run them.

### 1.2.4 Generating a parser and lexical analyser

Having constructed the `adder.jj` file, we invoke JavaCC on it. Exactly how to do this depends a bit on the operating system. Below is how to do it on Windows NT, 2000, and XP. First using the "command prompt" program (CMD.EXE) we run JavaCC:

```
D:\home\JavaCC-Book\adder>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.
```

This generates seven Java classes, each in its own file:

- 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 (PLUS, NUMBER, 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.
- AdderTokenManager is the lexical analyser.
- Adder is the parser.

We can now compile these classes with a Java compiler:

```
D:\home\JavaCC-Book\adder>javac *.java
```

### 1.2.5 Running the example

Now let's take another look at the main method in class Adder.

```
  static void main( String[] args )
throws ParseException, TokenMgrError {
    Adder parser = new Adder( System.in ) ;
    parser.Start() ; }
```

First note that main might throw either of the two generated subclasses of Throwable. This is not very good style, as we really ought to catch these exceptions, however, it helps keep this first example short and uncluttered.

The first statement of the body creates a new parser object. The constructor used is automatically generated and takes an InputStream. There is also a constructor that takes a Reader. The constructor in turn constructs an instance of the generated SimpleCharacterStream class, and a lexical analyser object of class AdderTokenManager. Thus, the effect is that the parser will get its tokens from a lexical analyser that reads characters from the System.in object via a SimpleCharacterStream object.

The second statement calls a generated method called Start. For each BNF production in the specification, JavaCC generates a corresponding method in the parser class. This method is responsible for attempting to find in its input stream a match for the description of the input. In the example, calling Start will cause the parser to attempt to find a sequence of tokens in the input that matches the description

<NUMBER> (<PLUS> <NUMBER>)* <EOF>

We can run the program by preparing a suitable input file and executing the command

```
D:\home\JavaCC-Book\adder>java Adder <input.txt
```

One of three things can happen when we run the main program with a given (finite) input file:

1. There is a lexical error found. In the example, lexical errors only happen when there is an unexpected character in the input. We can produce a lexical error by feeding in an input file consistsing of

   "123 - 456\n" .

   In this case the program will throw a TokenMgrError. The message field of the excpeption is: Exception in thread "main" TokenMgrError: Lexical error at line 1, column 5. Encountered: "-" (45), after : ""

2. There is a parsing error. This happens when the sequence of tokens does not match the specification of Start. For example

   "123 ++ 456\n"

   or

   "123 456\n"

   or

   "\n" .

   In this case the program will throw a ParseException. For the first of these the message of the exception is Exception in thread "main" ParseException: Encountered "+" at line 1, column 6.
   Was expecting:
   <NUMBER> ...

3. The input contains a sequence of tokens matching Start's specification. In this case, no exception is thrown and the program simply terminates.

Since this parser does nothing when the input is legitimate, it's use is limited to checking the legitimacy of its input. In the next sections we will make changes that will make the parser more useful.

### 1.2.6 The generated code

To get an idea of how a JavaCC generated parser works, it is worthwile looking at some of the generated code.

```
final public void Start() throws ParseException {
    jj_consume_token(NUMBER);
    label_1:
    while (true) {
        jj_consume_token(PLUS);
        jj_consume_token(NUMBER);
        switch ((jj_ntk == -1) ? jj_ntk() : jj_ntk) {
        case PLUS:
            ;
            break;
        default:
            jj_la1[0] = jj_gen;
            break label_1; } }
    jj_consume_token(0);
}
```

The method jj_consume_token takes a token kind as an argument and tries to obtain a token of that kind from the lexical analyser; if the next token has a different kind, then an exception is thrown. The expression

```
(jj_ntk == -1) ? jj_ntk() : jj_ntk
```

calculates the kind of the next unread token. The last line tries to obtain a token of type zero; JavaCC always uses zero to encode the kind of EOF tokens.

### 1.2.7 Augmenting the parser

The methods generated by JavaCC for BNF productions such as Start's, by default simply check the whether the input matches the specification. However we can augment BNF productions with Java code to be included in the generated methods. JavaCC provides the skeleton; it is up to us to flesh it out.

We'll make a few changes to the specification file to obtain `adder1.jj`. To the BNF production Start we add some declarations and some java code. The added or changed parts are shown in boldface

```
int Start() throws NumberFormatException :
{
    Token t ;
    int i ;
    int value ;
}
{
    t = <NUMBER>
    { i = Integer.parseInt( t.image ) ; }
    { value = i ; }
    (
        <PLUS>
        t = <NUMBER>
        { i = Integer.parseInt( t.image ) ; }
        { value += i ; }
    )*
    <EOF>
    { return value ; }
}
```

---

First, the return type of the BNF production, and hence the generated method, is changed from void to int. We've declared that NumberFormatException may be thrown from the generated method. We've declared three variables. Variable t is of type Token, which is a generated class that represents tokens; the image field of the Token class records the string of characters matched. When a token is matched in a BNF production, we can record the Token object by assigning a reference to it as in the lines

```
t = <NUMBER>
```

Within braces in a BNF production, we can add any Java statements that we want; these statements are copied essentially verbatim into the generated method.

Since the generated Start method now returns a value, we must alter the main method, which calls it:

---

```
static void main( String[] args )
throws ParseException, TokenMgrError, NumberFormatException {
    Adder parser = new Adder( System.in ) ;
    int val = parser.Start() ;
    System.out.println(val); }
```

There is one more minor improvement to make before leaving this example. The two lines

```
t = <NUMBER>
{ i = Integer.parseInt( t.image ) ; }
```

appear twice. Although it doesn't make a big difference in this case, as only two lines are involved, this sort of repetition can lead to maintenence problems. So we will factor out these two lines into another BNF production, this one named Primary. Again the most recent changes are shown in bold face.

```
int Start() throws NumberFormatException :
{
    int i ;
    int value ;
}
{
    value = Primary()
    (
        <PLUS>
        i = Primary()
        { value += i ; }
    )*
    <EOF>
    { return value ; }
}

int Primary() throws NumberFormatException :
{
    Token t ;
}
{
    t=<NUMBER>
    { return Integer.parseInt( t.image ) ; }
}
```

Looking at the generated methods shows how JavaCC integrates the Java declarations and statements into the skeleton of the generated methods. The

```
final public int Start() throws ParseException, NumberFormatException {
      int i ;
      int value ;
      value = Primary();
      label_1:
      while (true) {
           switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
           case PLUS:
                ;
                break;
           default:
                jj_la1[0] = jj_gen;
                break label_1; }
           jj_consume_token(PLUS);
           i = Primary();
           value += i ; }
      jj_consume_token(0);
      {if (true) return value ;}
      throw new Error("Missing return statement in function");
}

final public int Primary() throws ParseException, NumberFormatException {
      Token t ;
      t = jj_consume_token(NUMBER);
      {if (true) return Integer.parseInt( t.image ) ;}
      throw new Error("Missing return statement in function");
}
```

We will see later that it is possible to pass parameters into BNF productions.

## 1.3 A second example: A calculator

We will transform our adder into a simple four function interactive calculator.

As a first step, we'll make the calculator more interactive, printing out a value of each line. At first, we'll just add up numbers and worry about other operations, subtraction, multiplication, and division later.

### 1.3.1 Options and class declaration.

The first part of our file `calculator0.jj` is much as before:

```
/* calculator0.jj An interactive calculator. */

options {
    STATIC = false ;
}

PARSER_BEGIN(Calculator)
    import java.io.PrintStream ;

    class Calculator {
        static void main( String[] args )
        throws ParseException, TokenMgrError, NumberFormatException {
            Calculator parser = new Calculator( System.in ) ;
            parser.Start( System.out ) ;
        }

        double previousValue = 0.0 ;
    }
PARSER_END(Calculator)
```

The previousValue field of the Calculator class is used to store the result of evaluating the previous line, we'll use it in a future version where a dollar sign can be used to represent it. The import statement illustrates that import declarations are possible between the PARSER_BEGIN and PARSER_END brackets; these are copied into the generated parser and token manager classes. Package declarations can also be used and will be copied into all generated classes.

### 1.3.2 Lexical specification

The specification of the lexical analyser changes a little. First the end of line is declared as a TOKEN and given a symbolic name so that it is passed on to the parser.

```
SKIP : { " " }
TOKEN : { < EOL : "\n" | "\r" | "\r\n" > }
TOKEN : { < PLUS : "+" > }
```

Second we'll allow decimal points in numbers. We change the NUMBER kind of token to allow a decimal point in the number. There are four choices which we separate by a vertical bar. The four choices are respectively: no decimal point, decimal point in the middle, decimal point at the end, and decimal point at the start. A perfectly good specification is:

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

As sometimes happens, the same regular expression appears many times. For readability, it is nice to give such regular expresssions a symbolic name. We can invent a name for a regular expression that is purely local to the lexical analyser; such name does not represent a token kind. These named regular expressions are marked by a # in their defintion. An equivalent to the previous snippet is

```
TOKEN : { < NUMBER : <DIGITS> | <DIGITS> "." <DIGITS> | <DIGITS> "." | "."
<DIGITS> > }
TOKEN : { < #DIGITS : (["0"-"9"])+ > }
```

### 1.3.3 Parser specification

The input to the parser consists of a sequence of zero or more lines, each containing an expression. Using BNF notation, which will be further explained in the next chapter, we can write this as

$$Start \longrightarrow (Expression\ EOL) *\ EOF$$

This gives us the skeleton of the Start BNF production:

```
void Start() :
{}
{
    (
        Expression()
        <EOL>
    )*
    <EOF>
}
```

We augment this skeleton with Java actions to record and print the result of each line.

```
void Start(PrintStream printStream) throws NumberFormatException :
{}
{
    (
        previousValue = Expression()
        <EOL>
        { printStream.println( previousValue ) ; }
    )*
    <EOF>
}
```

Each expression consists of one or more numbers separated (for now) by plus signs. In BNF notation we have

$$Expression \longrightarrow Primary\ (PLUS\ Primary)*$$

where *Primary*, for the moment, simply represents numbers. This is translated to JavaCC notation as follows (with augmentation shown in bold face).

```
double Expression() throws NumberFormatException :
{
    double i ;
    double value ;
}
{
    value = Primary()
    (
        <PLUS>
        i = Primary()
        { value += i ; }
    )*
    { return value ; }
}
```

This is essentially the same as the Start BNF production of the `adder1.jj` example, except we've changed the type of the numbers involved from int to double.

Primary is much as in the `adder1.jj` example. In BNF notation it is simply

$$Primary \longrightarrow NUMBER$$

The JavaCC is just the same except that it now computes a double precision number.

```
double Primary() throws NumberFormatException :
{
    Token t ;
}
{
    t = <NUMBER>
    { return Double.parseDouble( t.image ) ; }
}
```

Sumarizing the parser in BNF notation, we have

$$Start \longrightarrow (Expression\ EOL) *\ EOF$$
$$Expression \longrightarrow Primary\ (PLUS\ Primary)*$$
$$Primary \longrightarrow NUMBER$$

At this point we are done the `calculator.jj` file and can try running a few examples.

### 1.3.4 Adding subtraction

To obtain a more functional calculator we need some more operators such as subtraction, multiplication, and division. We'll start with subtraction.

In the specification of the lexical analyser, we add a new production

```
TOKEN : { < MINUS : "-" > }
```

In the regular expression productions for EOL and NUMBER, we've used the vertical bar character to separate choices; we can do the same in the BNF productions that define the parser. In this case we need a choice between a PLUS token and a MINUS token. In terms of BNF notation, we could change the production for Expression to

$$Expression \longrightarrow Primary\ ((PLUS \mid MINUS)\ Primary) *\qquad ,$$

but instead we use the equivalent

$$Expression \longrightarrow Primary\ (PLUS\,Primary \mid MINUS\,Primary)*$$

because it makes the Java actions a bit simpler. In JavaCC notation, with the additions in boldface, the production is

```
double Expression() throws NumberFormatException :
{
      double i ;
      double value ;
}
{
      value = Primary()
      (
            <PLUS>
            i = Primary()
            { value += i ; }
      |
            <MINUS>
            i = Primary()
            { value -= i ; }
      )*
      { return value ; }
}
```

### 1.3.5 Adding multiplication and division

To add multiplication and division the changes to the lexical specification are easy. We just add two productions

```
TOKEN : { < TIMES : "*" > }
TOKEN : { < DIVIDE : "/" > }
```

We could change the Expression production in a way similar to the way we did for subtraction, that is to change it to

$$Expression \longrightarrow Primary\ (PLUS\ Primary \mid MINUS\ Primary \mid TIMES\ Primary \mid DIVIDE\ Primary)*$$

From a purely syntactic point of view, there is nothing wrong with this approach, however it does not mesh well with our method of evaluation because it does not recognize that multiplication and division should have higher precedence than addition and multiplication. For example, if we evaluate the line

```
2 * 3 + 4 * 5
```

we would get a result of $((2 \times 3) + 4) \times 5$, which is 50, rather than $(2 \times 3) + (4 \times 5)$. Instead we use two productions

$$\textit{Expression} \longrightarrow \textit{Term}\,(\textit{PLUS}\,\textit{Term} \mid \textit{MINUS}\,\textit{Term})*$$
$$\textit{Term} \longrightarrow \textit{Primary}\,(\textit{TIMES}\,\textit{Primary} \mid \textit{DIVIDE}\,\textit{Primary})*$$

This divides each expression into a sequence of one or more terms which are added to, or subtracted from, each other. In our example, the terms are shown in boxes:

`2 * 3` + `4 * 5`

The change to Expression is only to change the references to Primary to references to Term:

```
double Expression() throws NumberFormatException :
{
    double i ;
    double value ;
}
{
    value = Term()
    (
        <PLUS>
        i = Term()
        { value += i ; }
    |
        <MINUS>
        i = Term()
        { value -= i ; }
    )*
    { return value ; }
}
```

The production for Term is similar:

```
double Term() throws NumberFormatException :
{
```

```
    double i ;
    double value ;
}
{
    value = Primary()
    (
        <TIMES>
        i = Primary()
        { value *= i ; }
    |
        <DIVIDE>
        i = Primary()
        { value /= i ; }
    )*
    { return value ; }
}
```

---

### 1.3.6 Adding parentheses, a unary operator, and history

We just need a few more features before we have a useful four function calculator. We'll allow parenthesized expressions, negation, and the previously calculated value to be accessed by using a dollar sign.

The changes to the lexical specification are straight-forward, we just add productions:

---

```
TOKEN : { < OPEN_PAR : "(" > }
TOKEN : { < CLOSE_PAR : ")" > }
TOKEN : { < PREVIOUS : "$" > }
```

---

There is no need add a regular production for negation, since the hyphen character is already recognized as a token of kind MINUS.

The changes to the parser specification are all in the production for Primary. There are now four possibilities: A number (as before), the dollar sign, a parenthesized expression, or a minus sign followed by any of these possibilities. In BNF notation we have:

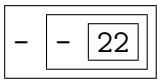
$$
\begin{array}{lll}
\textit{Primary} & \longrightarrow & \textit{NUMBER} \\
 & | & \textit{PREVIOUS} \\
 & | & \textit{OPEN\_PAR Expression CLOSE\_PAR} \\
 & | & \textit{MINUS Primary}
\end{array}
$$

This BNF production is recursive in two ways. The last alternative is directly recursive. The second last alternative is indirectly recursive, since *Expression* ultimately depends on *Primary*. There is nothing wrong with using recursion in BNF productions, although there are some limitations that we will discuss later. Consider an expression.

```
- - 22
```

.

The *Primary*s are shown in boxes here:

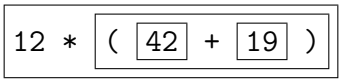



.

In executing the generated parser on this input, there will be one call to the Primary method for each of these boxes. Similarly, given an input of

```
12 * ( 42 + 19 )
```

we can again box the *Primary*s:




Again the nested boxes show recursive calls to the generated Primary method.

Here is the production in JavaCC notation, with the changes high-lighted in bold.

---

```
double Primary() throws NumberFormatException :
{
    Token t ;
    double d ;
}
{
    t=<NUMBER>
    { return Double.parseDouble( t.image ) ; }
|
    <PREVIOUS>
    { return previousValue ; }
|
    <OPEN_PAR> d=Expression() <CLOSE_PAR>
    { return d ; }
|
```

```
    <MINUS> d=Primary()
    { return -d ; }
}
```

---

This concludes the calculator example. The full calculator specification is in file `calculator1.jj`. There are many improvements we could still make, most of which would be quite straight-forward, for example new operations, and the reader is invited to do so.

The method of calculating the result of an expression that we have illustrated in this chapter is "direct interpretation", that is the parser itself calculates the numerical result of each expression. This method works fine for simple expressions, but not so well when some form of looping is involved. Consider for example an expression

```
sum i : 1..10 of i*i
```

corresponding to the mathematical expression

$$\sum_{i=1}^{100} i^2 \quad .$$

In this case direct interpretation does not work so well since there is no one number that corresponds to the subexpression

```
i*i
```

.

For this sort of thing it may be best for the parser to compute some other representation of the input expression —perhaps a tree or an abstract machine code— that can be evaluated after parsing is complete.

## 1.4 Text processing

The last example of the chapter has a slightly different character. Whereas the adder and calculator example show the processing of artificial languages —and will later be expanded to a full programming language—, the examples in this section show that JavaCC is useful also for text processing tasks with input that is largely unstructured.

### 1.4.1 A Bowdlerizer

The task here is to replace certain patterns in the input with some other text. The pattern that we'll look for are four-letter words. We'll take this specification literally and replace any word with four letters regardless of whether it is in good taste or not.

### Options and class declaration

The initial part of the specification file declares a static method that maps a String to a String.

```
/* four-letter-words.jj A simple report writer. */
options {
    STATIC = false ;
}

PARSER_BEGIN(FLW)
    import java.io.Reader ;
    import java.io.StringReader ;

    class FLW {
        static String substitute( String inString ) {
            Reader reader = new StringReader( inString ) ;
            FLW parser = new FLW( reader ) ;
            StringBuffer buffer = new StringBuffer() ;
            try {
                parser.Start( buffer ) ; }
            catch( TokenMgrError e ) {
                throw new IllegalStateException() ; }
            catch( ParseException e ) {
                throw new IllegalStateException() ; }
            return buffer.toString() ; }
    }
PARSER_END(FLW)
```

The point of the try statement here is to transform ParseExceptions into exceptions that don't have to be declared. The reason is that this parser should never throw ParseException (or TokenMgrError's either); any character sequence at all should be legal input. (If the assert statement is supported by our Java compiler, we could replace the throw statements with assert false; statements.)

### The lexical analyzer

The specification of the lexical analyzer is the crucial part. We will divide the file into tokens of three categories: four letter words, words of more than four letters, and any other

character, including letters that are in one, two and three letter words. We'll go through the specification one line at a time.

The four letter words are easily specified using an abbreviation $(x)\{n\}$ which specifies exactly $n$ repetitions of regular expression $x$.

```
TOKEN : { < FOUR_LETTER_WORD : (<LETTER>){4} > }
```

We've already seen that $(x)+$ stands for one or more repetitions of regular expression $x$. Similarly $(x)*$ means zero or more repetitions of regular expression $x$.Thus five letter words can be specified as:

```
TOKEN : { < FIVE_OR_MORE_LETTER_WORD : (<LETTER>){5} (<LETTER>)* > }
```

We specified digits with ["0"-"9"]. We can write a regular expression to match a single letter in a number of ways; one is ["a"-"z","A"-"Z"], which gives a list two ranges[1].

```
TOKEN : { < #LETTER : ["a"-"z","A"-"Z"] > }
```

We could have specified the digits by listing all the digits individually; that is the regular expression ["0"-"9"] as

```
["0","1","2","3","4","5","6","7","8","9"]
```

.

In general we can give a list of individual characters or character ranges. For example

```
["0"-"9","a"-"z","A"-"Z","'","-"]
```

matches any single character that is a digit, letter, apostrophe, or hyphen. We can also write a regular expression for the complement of a set of letters. For example

```
~["0"-"9","a"-"z","A"-"Z","'","-"]
```

matches any single character that is not a digit. An extreme case is where the set is empty. The regular expression [] matches any single character that is in the empty set; that is, it does not match anything; the set of character sequences that it matches is the empty set. The regular expression [] is not very useful, but the complement ~[] matches any single character that is not in the empty set; that is it matches any single character. This is just what we need to match characters that are not part of four letter or longer words.

[1]For simplicity we restrict ourselves to the 52 upper and lower case letters of the roman alphabet. JavaCC is perfectly capable of dealing with any Unicode character, which means that it could easily deal with accented letters and letters from other alphabets.

```
TOKEN : { < OTHER : ~[] > }
```

### Maximal munch

Consider the input sequence "sinister". We can break it up a number of ways so that each part matches one of our three nonlocal regular expression productions. For example, we could consider it to consist of eight individual characters, in which case it breaks into eight tokens each of kind OTHER, or we could consider it to consist of two tokens of kind OTHER, one of kind FOUR_LETTER_WORD, and then two more tokens of kind OTHER, or we could consider it to consist of one token of kind FIVE_OR_MORE_LETTER_WORD followed by zero, one, two, or three tokens of kind OTHER, and so on. (There are 17 possbilities in all.)

What we want of course is for it to be matched as a single token of kind FIVE_OR_MORE_LETTER_WORD this is exactly what happens, but it is important to understand why. The lexical analyser always tries to cram as many of the remaining input characters as possible into the next token that it produces. This is called the "maximal munch" rule. Suppose the input is "sinister cats". All three productions match some beginning part of the input: OTHER matches the one character sequence "s"; FOUR_LETTER_WORD matches the first four characters "sini"; and FIVE_OR_MORE_LETTER_WORD matches any of the sequences "sinis", "sinist", "siniste", and "sinister". The longest possible match is the first eight characters. This leaves " cats". As the next character is not a letter, the only production that matches is OTHER. The remaining sequence is then "cats". Both the OTHER and the FOUR_LETTER_WORD productions match, but by the maximal munch rule, the FOUR_LETTER_WORD production wins out. The remaining input is then the empty sequence, which leads to an EOF token being produced.

You might wonder what happens if the maximal munch rule does not determine the production because two productions can both match the longest possible match. This doesn't happen in the current example since the three (nonlocal) productions match inputs of lengths one, four, and five or more respectively. But consider a lexical analyser for the Java programming language. We might have the following production rules.

```
TOKEN : { < KWINT : "int" > }
TOKEN : { < IDENTIFIER : (<LETTER> | "_") (<LETTER> | <DIGIT> | "_")* > }
```

When the remaining input is "int0 = 0 ; ... " then by the maximal munch rule "int0" is matched as an INDENTIFIER. However, when the remaining input is "int i ; ...". The the both rules match the maximum number of characters that any rule matches — three. In this case the rule that appears first in the specification file has priority. So in our example

"int" is matched as a KWINT.

The presence of the OTHER rule in our specification ensures that some token can always be produced by the lexical analyser. If the input is not empty, then an OTHER token can be produced (although it may be that some other production is actually prefered) and if the remaining input is the empty sequence then an EOF token will be produced. Thus the lexical analyser will never throw a TokenMgrError.[2]

### Parsing for the Bowdlerizer

The specification of the parser is straight forward. The three token kinds can occur in any number and in any order. For FOUR_LETTER_WORD tokens we copy four asterisks to the output and for either of the other kinds we simply echo the image of the token to the output.

```
void Start( StringBuffer buffer ) :
{
    Token t ;
}
{
    (
        <FOUR_LETTER_WORD>
        { buffer.append("****"); }
    |
        ( t=<FIVE_OR_MORE_LETTER_WORD> | t=<OTHER> )
        { buffer.append( t.image ) ; }
    )*
    <EOF>
}
```

Since the parser accepts any sequence of input tokens that the lexical analyser might produce, the parser can not throw a ParseException. Both our lexical analyser and parser are "total": they accept any input string at all.

[2] In a later chapter, we will look at the use of the MORE keyword. When the MORE keyword is used in a lexical analyser specification, then the presence of a catch-all production such as our OTHER production may not be enough to ensure that TokenMgrErrors can not be thrown. See the JavaCC FAQ for more information on this point.

## 1.5 Summary and prospectus

We have seen that JavaCC allows concise specification of lexical analysers and parsers using the notation of regular expressions and BNF productions.

The input to the lexical analyser is a sequence of character — represented by a Java InputStream object or a Java Reader object. The output of the lexical analysers is fixed by JavaCC: it is a sequence of Token objects. The input to the parser is again fixed, it is a sequence of Token objects. This relationship between the lexical analyser and the parser is shown in figure [TBD].

The output of the parser is, however, not prescribed by JavaCC at all; it is whatever the programmer wants it to be, as long as it can be expressed in Java. Typically it is some abstracted respresentation of the input. In the adder and calculator examples of this Chapter, the output is a number, either a Java int or a Java double. These are abstractions in that the same number might be produced by different inputs. In compilers the output of the parser might take the form of machine or assembly code. More commonly the parser of a compiler produces an intermediate representation of the input program, which is then further translated by other parts of the compiler. For other applications the output will take other forms. For example, the output might be a string, perhaps a modified version of the input string, as it is in our Bowdlerizer example; the output might be a Java object representing configuration settings, if the input is a configuration file; and so on.

A particularly common case is where the output of the parser is a tree that closely conforms to the tree of method calls made by the generated parser. In this case, there are additional tools that can be used in conjuction with JavaCC to automate the augmentation of the grammar. These tools are JJTree and JTB and are the subject of Chapter [TBD].

It is important to note that the lexical analyser works quite independently of the parser; its choice of how to divide the input stream into tokens is not influenced by which token kind the parser might expect at any given moment, but only by the rules that constitute its specification. In fact, as we will see later, the lexical analyser be working several tokens ahead of the parser and so is in no position to be influenced by the expectations of the parser.

...TBD...