Calculator example

We wish to write a program that reads a sequence of simple arithmetic expressions from standard input terminating at end of file (or from a BreezyGUI text field), evaluates each expression in turn, and prints their values.

Sample input

100
123 + 4.5
123 + 2*4 + 100/2

Sample output

100.0
127.5
300.0

(For simplicity, we apply operators from left to right, with no relative precedence, and no parentheses allowed. Here, as usual, each line is evaluated as soon as it is read.)

Calculator example (cont.)

Initial design

```
For each line, exp, of input
    Let val = evaluate(exp);
    Print val
```

Note that this design does not specify whether the expressions to evaluate are read from standard input or from a text field, nor whether the output is written to standard output or to a text area. So the design applies to both console and graphical applications.

The interesting part of this program is method evaluate. (Including its use of a string tokenizer. And the conversion of tokens to numbers. And...)

Calculator example (cont.)

Initial design of method `evaluate()`

```
Let val be the first token in exp;
while there are more tokens in exp
    Let op be the next token;
    Let num be the following token;
    Let val be the result of applying op to val and num;
return val
```

Now, with input 3 + 4*5, a trace of the method would be:

```
    Val       op      Num     Remaining tokens
                              3 + 4 * 5
    3         +       4       * 5
    7         *       5
    35
```

(Note that we can – and should – test our ideas at this design stage.)

Calculator example (cont.)

Outline implementation of this design

```
public static double evaluate(String exp) {
    StringTokenizer st =
        new StringTokenizer(exp, "+-*/", true);
    double val = number corresponding to first token in st;
    while (st.hasMoreTokens()) {
        char op =
            operator corresponding to next token in st;
        double num =
            number corresponding to next token in st;
        val = result of applying op to val and num;
    }
    return val;
}
```

The italicized constructs can be completed using methods described above.

Calculator example (cont.)

More detailed implementation

```
public static double evaluate(String exp) {
    StringTokenizer st =
        new StringTokenizer(exp, "+-*/", true);
    double val = Double.parseDouble(st.nextToken().trim());
    while (st.hasMoreTokens()) {
        char op = st.nextToken().charAt(0);
        double num =
            Double.parseDouble(st.nextToken().trim());
        switch (op) {
            case '+' : val += num; break;
            case '-' : val -= num; break;
            // ...
        }
    }
    return val;
}
```

Calculator example (cont.)

Why the following construction of the string tokenizer?

```
StringTokenizer st = new StringTokenizer(exp, "+-*/", true);
```

Suppose we had used simply:

```
StringTokenizer st = new StringTokenizer(exp);
```

This default uses spaces, tabs, etc., as token separators, so this could work if every token was separated by white space, "100 + 200", *but would fail to separate the expression into tokens otherwise:* "100+200".

So let's make the operator characters token separators:

```
StringTokenizer st = new StringTokenizer(exp, "+-*/");
```

This string tokenizer now separates "100 + 200" into the two tokens "100 " and " 200". *The operator* (like the spaces previously) *is omitted*.

The final version (at top), uses operator characters as separators *and returns separators as tokens* : "100 + 200" => "100 ", "+", " 200".

Calculator example (cont.)

Hence, to convert a number token token to a number num, we must write something like:

```
double num = Double.parseDouble(token.trim());
```

and to convert an operator token token to a character op for use in the switch statement, we must write something like:

```
char op = token.charAt(0);
```

Exercise 1 (important) Complete this calculator program.

Exercise 2 (important) Rewrite the calculator program as a BreezySwing application.

Exercise 3 Extend the calculator program so that it recognises operator precedence and parentheses as in Java, i.e., so that the input expression "1 + (3+3*4) / 5" evaluates to 4. (See over.)

Recursive descent parsing (solution to Ex. 3 above, not examinable)

Here is a grammar for the class of expressions to be evaluated:

expression ⇒ *term* { ('+' | '–') *term* }
term ⇒ *factor* { ('*' | '/') *factor* }
factor ⇒ *number* | '(' *expression* ')'

These grammar rules simply that say an *expression* is a sum (or difference) of *terms*, a *term* is a product (or quotient) of *factors*, and a *factor* is a number or a parenthesised *expression*.

Such a grammar generates sentences from the starting symbol *expression* as follows:

expression ⇒ *term*
⇒ *factor* * *factor*
⇒ 3 * (*expression*)
⇒ 3 * (*term* + *term*)
⇒ 3 * (*factor* + *factor*)
⇒ 3 * (4 + 5)

Recursive descent parsing (cont.)

To recognise (and evaluate) sentences generated from the nonterminal symbol expression, we can do the following:

- Declare a global variable `token`
- Write a method of no arguments for each nonterminal symbol *N* to recognise (and evaluate) sentences *s* generated from *N*.
- Each such method must have the following properties:
  - On entry, the first token of *s* must already be assigned to the variable `token`.
  - On exit, the variable `token` must contain the first token following *s*.

The implementation of these rules is actually very mechanical! (See over.)

Recursive descent parsing (cont.)

For example, given the rule

expression ⇒ *term* { ('+' | '–') *term* }

we could define a method to read and evaluate an expression as follows:

```
// On entry, token is the first token of the expression
// On exit, token is the first token after the expression
// and the value of the expression is returned
double expression () {
    double val = term();
    // On exit, token is the first token after the term
    while (token.equals("+") || token.equals("-")) {
        char op = character corresponding to token;
        token = next token;
        double num = term();
        val = result of applying op to val and num;
    }
    return val;
 }
```

Recursive descent parsing (cont.)

We can define a method to read and evaluate a factor using the rule

term ⇒ *factor* { ('*' | '/') *factor* }

as follows:

```
// On entry, token is the first token of the term
// On exit, token is the first token after the term
// and the value of the term is returned
double term () {
    double val = factor();
    // On exit, token is the first token after the factor
    while (token.equals("*") || token.equals("/")) {
        char op = character corresponding to token;
        token = next token; // first token of next factor
        double num = factor();
        val = result of applying op to val and num;
    }
    return val;
 }
```

Recursive descent parsing (cont.)

We can define a method to read and evaluate a factor using the rule

factor ⇒ *number* | '(' *expression* ')'

as follows:

```
// On entry, token is the first token of the factor
// On exit, token is the first token after the factor
// and the value of the factor is returned
double factor () {
    if (token is a number) {
        val = number corresponding to the token;
    } else if (token.equals("(")) {
        token = next token;       // first token of expression
        val = expression();
        if (token.equals(")")
            token = next token; // token following expression
        else
            Report an error;     // expression not followed by ")"
    } else {
        Report an error; // factor doesn't start with number or ")"
    }
    return val;
 }
```

Exercises (also not examinable)

1. Here is a grammar for a part of the Java language. Write a recursive descent parser to determine whether the input stream of tokens is a valid *statement* or not.

statement ⇒ *variable* = *expression* ';'
| 'if' '(' *condition* ')' *statements* ['else' *statements*]
| 'while' '(' *condition* ')' *statements*

statements ⇒ *statement* | '{' *statement* { *statement* } '}'

condition ⇒ *expression* ('==' | '!=') *expression*

2. Write an alternative program to evaluate expressions using operator precedence and parentheses *without* using recursive descent parsing!

Hint Assign an integer precedence to each operator, so that multiplication (and division) have higher precedence than addition (and subtraction). Push each number onto a stack (a FIFO list) as it is read. Compare the precedence of an incoming operator with the precedence of the top operator on the stack to decide whether to apply the operator on the stack or to push the new operator onto the stack.