Revision history
181018: Initial version [CJG].

Lab assignment 6

Interpreter for the *M@* Language

This assignment is an exercise in an *interpreter* pattern, namely an *operator-precedence parser*. Your task is to implement a parser/interpreter for a small math-based language called M@. This language has three basic types of statements:

`AssgStmt` **Assignment** of an expression to a variable.

`PrintStmt` Issues **printing** of an expression to an output.

`ConfigStmt` Sets a **configuration**. Only one option is available: changing the number base. A base can be either decimal, hexadecimal or binary. All subsequent printing statements should use this base.

Additionally, there are **expressions**:

`MathExp` An expression consisting of either a *number* (integer), a *variable*, or an infix expression of the two with arithmetic operators.

Whereas other (some would say "real") programming languages, such as Java and C++, are compiled in a series of stages where the source code is scanned, analyzed, converted into intermediate code and so on, *M@* cuts this process very short: the source code is interpreted and executed on-the-fly, one statement (one line) at a time, inside a C++ program.

Example source and output

This is an example of valid *M@* source code:

Source code	Statement type
`config dec`	`ConfigStmt`
`print 1 + 1`	`PrintStmt`
`print 3 + 3 * 3`	`PrintStmt`
`x = 5 * ( 5 + 15 - 10 )`	`AssgStmt`
`y = x * -2`	`AssgStmt`
`print x`	`PrintStmt`
`print y`	`PrintStmt`
`x2 = x / 5 + 5`	`AssgStmt`
`print x2`	`PrintStmt`
`config hex`	`ConfigStmt`
`print x2`	`PrintStmt`
`config bin`	`ConfigStmt`
`print x2`	`PrintStmt`

Statements are separated by *line-break* and tokens are separated by *whitespace*. This makes parsing of M@ straightforward. The above code should generate the following output (or something close to it):

```
2
12
50
-100
15
0xf
00000000000000000000000000001111
```

Abstract grammar

The abstract grammar of M@ for given by:

```
Stmt:=           ConfigStmt | AssgStmt | PrintStmt

ConfigStmt :=    "config" [ "dec" | "hex" | "bin” ]
AssgStmt :=      Variable "=" MathExp
PrintStmt :=     "print" MathExp

MathExp :=       SumExp
SumExp :=        ProductExp [ "+" ProductExp | "-" ProductExp ]*
ProductExp :=    PrimaryExp [ "*" PrimaryExp | "/" PrimaryExp ]*
PrimaryExp :=    Int | Variable | "(" MathExp ")"

Variable :=      [a-zA-z][a-zA-z0-9]*
Int :=           -?[0-9]+
```

This grammar describes the language structure using abstract *blocks* with syntactic rules or regular expressions. A `ConfigStmt`, for instance, consists of the strings `config` followed by either `dec`, `hex` or `bin`.
Some blocks consists of other blocks: `SumExp`, for instance, consists of a left-hand-side `ProductExp` block, followed by *zero or more* right-hand-side `ProductExp`-blocks preceded by either plus or minus. Brackets with asterisks and plus signs indicate that content can be repeated: either *zero or more* times (asterisk), or *one or more* tims (plus sign).
The `Variable`-block is defined by a regular expression (regex) stating that variables may contain either letters or digits, but must start with a letter. The `Int`-block regex states that integers consist of one or more digits with an optional sign in the beginning.

A parser typically uses *one parsing function for each abstract block*. Parsing thus begins by calling the top-level block, say, `parse_Stmt`, which then delegates to other parsing functions depending on available tokens.

Instructions

The lecture which covers an interpreter for logical-comparison expressions is a good introduction to this lab.

Write a class `MathInterpreter` which is constructed with an out-stream,

```
    MathInterpreter(std::ostream& out_stream);
```

and has a function

```
    void evaluate( const std::vector<std::string>& tokens );
```

where `tokens` is one tokenized code line. If the source code contains multiple lines, `evaluate` is, in other words, called once per line. This function should parse and perform all actions stated in the code, such as storing variables, setting configurations, and making print-outs to the out-stream.

Before submitting code to `evaluate`, it has to be *tokenized* – i.e. broken down to a sequence of strings representing code elements (numbers, variables, operators etc). Start by splitting the code into lines, and then into tokens using whitespace as a separator.

Execute parsing according to the grammar by letting each *block* (`Stmt`, `AssgStmt`, `MathExp` etc) have its own parsing function. Each such function should identify and consume tokens, either by removing them from the token list or by advancing some stepping index. They then delegate parsing to other parsing functions depending on the available tokens and according to the grammar. Use a `peek`-function to identify statements and expressions from tokens before parsing them, and use a `consume`-function to remove or step past tokens as they are processed.

Variables (*symbols*) and variable values should be managed by a hash table within the interpreter class. This is called a *symbol table* in compiler jargong. Variables are created (or overwritten) in the symbol table by `AssgStmt`-statements, while `Variable`-expressions look up values for a particular variable name (or throw an error if the name does not exist).

As always it is a good idea to start small. Implement and test a subset of the grammar at first, e.g. just one type of statement, and make sure it works as expected before moving on. For instance, implement `parse_Stmt` first and have it distinguish between the other three main type of statements. Then move on to implement and call them, one by one.

A few helpful functions:

- The `<regex>`-header provides functionality for regular expression matching – see e.g. `std::regex` and `std::regex_match`.
- Regular expressions can also be matched by checking each char of the string manually. The `isdigit(char)` and `isalpha(char)` functions may be of help here.
- Feed `std::hex` to the `ostream` (using <<) to have it represent integers in hexadecimal base. Note that this only works for positive integers.
- One way to convert to binary is:
  `std::bitset<32>(int value).to_string()`

Requirements

Functionality

- The program reads all code from a separate source file.
- The program parses and executes code correctly according to the *M@* grammar, including but not limited to the provided example code. Add your own code (or code suggested by a teacher) and test more examples as well.
- Output is sent to an `ostream`, such as `cout` or `ofstream`.
- Blocks defined by regular expressions (`Int` and `Variable`) should be matched properly to the provided pattern. `Int`-tokens should be matched before they are cast to C++-integers.
- Invalid syntax (e.g. "`print 1 + - 2`") should cause an error with an error message stating something about what went wrong.

Design

- The interpreter/parser should be well structured and implemented with the grammar in mind.
- The interpreter/parser should be implemented in well formed C++.

Limitations

The program may terminate (throw a runtime error) immediately if invalid M@ syntax is encountered. The error message does not have provide precise information about the violation, but it should say something of help regarding what went wrong. If a previously undefined variable is used in an expression, for instance, the error message should say so.

Though C++ has support for regular expressions, it is sufficient to iterate the source string manually and match characters one by one according to the pattern. See the suggested built-in functions mentioned above.

Extensions (non-mandatory)

Try to think about how M@ can be extended with new functionality (and maybe even become useful?). What about unary operators such as negation "`-x`" and incrementation "`x++`"?

How would the current grammar tie into the grammar for logical-comparison expressions used in the lecture?

What about real programming stuff such as flow control (if-else-statements) and loops? Hint: Google *Abstract Syntax Tree*.