Interpreter for the C@ language

In this assignment you will apply the *interpreter pattern* to implement an *operator-precedence parser* [1] in C++ for a small improvised language called C@ (😺). This language has three basic types of statements:

`AssgStmt`	**Assign** the result of an *expression* to a *variable*.
`PrintStmt`	Issue **printing** of (the result of) an *expression* to an output.
`ConfigStmt`	Set a **configuration**. Only one option is available: changing the number base. A base can be either decimal, hexadecimal or binary. Affects subsequent printing statements.

Additionally, there are **expressions**:

`MathExp`	A mathematical expression. Valid operands are *numbers* (integers) and *variables*. Valid operators are the basic arithmetic operators, `+-*/`. Additionally, parentheses are allowed to define scopes for nested expressions.

Other well known (semi-)interpreted programming languages, such as C# and Java, are compiled and then executed in a series of stages where the source code is scanned, analyzed, converted into intermediate code and so on. In C@, we cut 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

Below is an example of valid C@ source code. The statement types are defined in the next section.

C@ source code	Statement type
config dec	ConfigStmt
print 1 + 1	PrintStmt
print 3 + 3 * 3	PrintStmt
print (3 + 3) * 3	PrintStmt
x = 2 - -2	AssgStmt
y = x	AssgStmt
z = y * (16 / (y - 2))	AssgStmt
print x	PrintStmt
print y	PrintStmt
print z	PrintStmt
config hex	ConfigStmt
print z	PrintStmt
config bin	ConfigStmt
print z	PrintStmt

Statements are separated by *line-break* and tokens by *whitespace*. Parsing into statements and tokens can be aided by for example `std::getline` and `std::stringstream`.
The above code should generate the following output (or something close to it):

```
2
12
18
4
4
32
0x20
0000000000100000
```

Abstract grammar

The abstract grammar for C@ is 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 *expression blocks* with syntactic rules or regular expressions. A `ConfigStmt`, for instance, is defined `:=` as the string `config`, followed by either `dec`, `hex` or `bin` (all strings).

Some blocks consist 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 `[]*`, or *one or more* times `[]+`.

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.

One way to design a parser for this grammar is to use **one parsing function per abstract block**, as described in lecture 11 (for logical-comparison expressions). Parsing thus begins by calling the top-level block, say, `parse_Stmt`, which then delegates to other parsing functions depending on available tokens.

Instructions

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

```
Interpreter(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 for every 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. Start by parsing the code into statements and tokens. Then implement and test a subset of the grammar, e.g. just one type of statement, and make sure it works as expected before moving on. For example, implement `parse_Stmt` first and have it distinguish between the other three main types of statements. Then move on to implement and call them, one by one.

Potentially helpful functions & other stuff:

- 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 print integers in hexadecimal base. Note that this only works for positive integers.
- Convert an `int` to binary (state the number of bits) and then to string: `std::bitset<32>(int).to_string()`

Requirements

Functionality

- The program reads all code from a separate source file.
- The program parses and executes code correctly according to the C@ 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 `std::ostream`, such as `cout` (standard output) or an `ofstream` (file stream), See the interpreter constructor above.
- Blocks defined by regular expressions (`Int` and `Variable`) should be matched properly to the provided pattern. `Int`-tokens should be matched before being 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 C@ syntax is encountered. The error message does not have to 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.

Future Work

Do not allow language-specific keywords such as `print` and `config` to be used as variable names. We then no longer have to deal with ambiguous statements such as `print = 1` (is it an `AssgStmt` or an (ill-formed) `PrintStmt`?).

What about unary operators such as negation `-x` (currently works for `int` literals but not for variables) and incrementation `x++`? We might need extra parsing functions for this, or rethink how tokens are separated.

How would the current grammar tie into the grammar for logical-comparison expressions (<, `&&`, `!=` etc) used in lecture 11?

What about real programming stuff such as flow control (`if-else`-statements), loops, scopes and functions? Hint: we need further abstraction for this – Google *Abstract Syntax Tree* (AST) for more information. You can also check out [2] for a fairly complete and also very accessible interpreter tutorial. For an even deeper look into parsing, AST's and compiler design in general, [3] is a good read.

---

[1] See lecture 11.
Also see e.g. https://en.wikipedia.org/wiki/Operator-precedence_parser (200930).

[2] Nystrom, https://craftinginterpreters.com (200930).

[3] Appel, *Modern Compiler Implementation in Java*, 2nd Edition.
There is a version of this book written for C as well (which I haven't read...)