Interpreter for the C@ language

In this assignment you will apply an *interpreter pattern* to implement a *recursive-descent parser* [1] in C++ for a small improvised language called C@ (🐱).

C@ 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.

More comprehensive (semi-)interpreted languages, such as C# or Java, compiles the source code in a series of stages before it can be executed. These stages include scanning, syntax analysis, conversion 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
`var1 = 2 - -2`	AssgStmt
`var2 = var1`	AssgStmt
`var3 = var2 * ( 16 / ( var2 - 2 ) )`	AssgStmt
`print var1`	PrintStmt
`print var2`	PrintStmt
`print var3`	PrintStmt
`config hex`	ConfigStmt
`print var3`	PrintStmt
`config bin`	ConfigStmt
`print var3`	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 grammar (in EBNF form [4]) 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 using *terminals* (operators, strings etc) and *nonterminals* (`Stmt`, `MathExp` etc). Nonterminals are defined by *production rules* that prescribe a sequence of terminals or other nonterminals. The `ConfigStmt` nonterminal, for example, is defined := as the terminal `config`, followed by either `dec`, `hex` or `bin` (all terminals).

Nonterminals may consist of other nonterminals. `SumExp`, for example, consists of a `ProductExp`, followed by *zero or more* additional `ProductExp`, each preceded by either plus + or minus -. Brackets with an asterisk `[]*` indicate content that may repeat *zero or more times*, while brackets with a plus sign `[]+` indicate content that may repeat *one or more times*.

`Variable` is a terminal defined by a *regular expression* (regex), stating that names of variables may contain either letters or digits but must start with a letter. The `Int` 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 interpret each nonterminal as a *parsing function* that implements the given production rule. As other nonterminals are encountered, their respective parsing functions are called recursively. This is called a *recursive-descent parser* and is described a bit further in lecture 11 (in the context of logical-comparison expressions).

Instructions

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

```
// Constructor
Interpreter(std::ostream& out_stream) : ... { ... }
```

and has a function

```
// Evaluate & interpret one tokenized statement
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.

Raw code line	`"value = 10 + y"`
Tokenized code line	`"value", "=", "10", "+", "y"`

Execute parsing according to the grammar by letting each nonterminal (`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.

Variable names and values should be managed as key-value pairs in a hash table. This is called a *symbol table* in compiler jargong. These key-value pairs are created (or overwritten) in the symbol table by `AssgStmt`-statements. Values are fetched from the hash table whenever expressions with a `Variable` are encountered (using an undefined variable is a grammatical error).

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.
- Stream `std::hex` to an `ostream` (using <<) to have it print integers in hexadecimal base. Note that this only works for positive integers.
  Stream `std::showbase` to include a base prefix, e.g. `0x`.
- Convert an `int` to a binary representation 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.
- Terminals 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 throw or display 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, but should be as helpful as possible. If, for example, a previously undefined variable was used in an expression – the error message should say so.

Though C++ has support for regular expressions, it is sufficient to iterate strings letter by letter and match them to given patterns. See the suggested built-in functions mentioned above.

Future Work (non-compulsory)

Have a list of language-specific keywords, such as `print` and `config`, and throw an error if they are used as variable names. We then no longer have to deal with unintuitive statements such as `print = 1` (should be invalid for both `PrintStmt` and `AssgStmt`).

What about unary operators, such as negation `-x`, or incrementation `x++`? We need to extend the grammar for this. For negation, we could start with:

```
PrimaryExp :=  Int
               | Variable
               | "(" MathExp ")"
               | -PrimaryExp
```

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/Recursive_descent_parser (211014).

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

[3] Appel, *Modern Compiler Implementation in Java*, 2nd Edition.
A version of this book written for C exists as well.

[4] Extended Backus–Naur form
https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form
(211014)