Project 1: vtscript

Assigned:	1/31
Beta Version Due:	2/20 at 8am
Final Version Due:	Friday 3/3 by 5pm **Extended** to 3/10 at 8am.

GitHub Invitation Link (https://classroom.github.com/assignment-invitations/89518404268f5695d3ea5d3cfb851403). Accept the GitHub invitation, wait until you get an email saying the import is complete, and then clone the git repository to your local machine.

The goal of this project is to gain practice designing and implementing multiple modules that are used together to solve a problem. You will also gain experience using the standard library as well as unit testing. You will write an interpreter (https://en.wikipedia.org/wiki/Interpreter_(computing)) for a simple programming language we will call *vtscript*. It is an expression-based language using prefix Lisp (https://en.wikipedia.org/wiki/Lisp_(programming_language)) notation [1].

Introduction

A C++ program is a collection of statements, each of which contains expressions, sequence of characters that when evaluated give a value. For example in the following code consisting of a single C++ statement.

```
int x = ((1+2)*3)/4;
```

the statement allocates a stack variable named `x` and assigns its value to the result of the expression `((1+2)*3)/4`. Some programming languages only have expressions. For example most Lisp-family languages, like scheme (http://www.schemers.org/), consist solely of expressions. This makes their syntax less complicated.

The syntax of a programming language is the rules that govern when a string of characters represents a valid sequence in the language. Semantics is the meaning of the sequence computationally, the result it produces. Some languages have a complicated syntax -- C++ notoriously so. Other are very simple.

The Lisp family of languages use prefix notation to represent an expression. Prefix notation puts the operator first. For example the prefix notation of the expression above is `(/ (* (+ 1 2) 3) 4)`. In general the syntax is `(PROC ARG1 ARG2 ... ARGN)`, where `PROC` is a *procedure* with *arguments* `ARG1`, `ARG2`, etc., where each argument can also be an expression. This is all the syntax there is in vtscript. A vtscript program then is just one, possibly very complex, expression. For example the following program is roughly equivalent to the C++ one above.

```
(define x (/  (* (+ 1 2) 3) 4))
```

It computes the result of the numerical expression and assigns the symbol `x` to have that value in the *environment*. The environment is a mapping from known symbols to either atoms or procedures. When the program starts there is a default environment that gets updated as the program runs adding symbol-atom

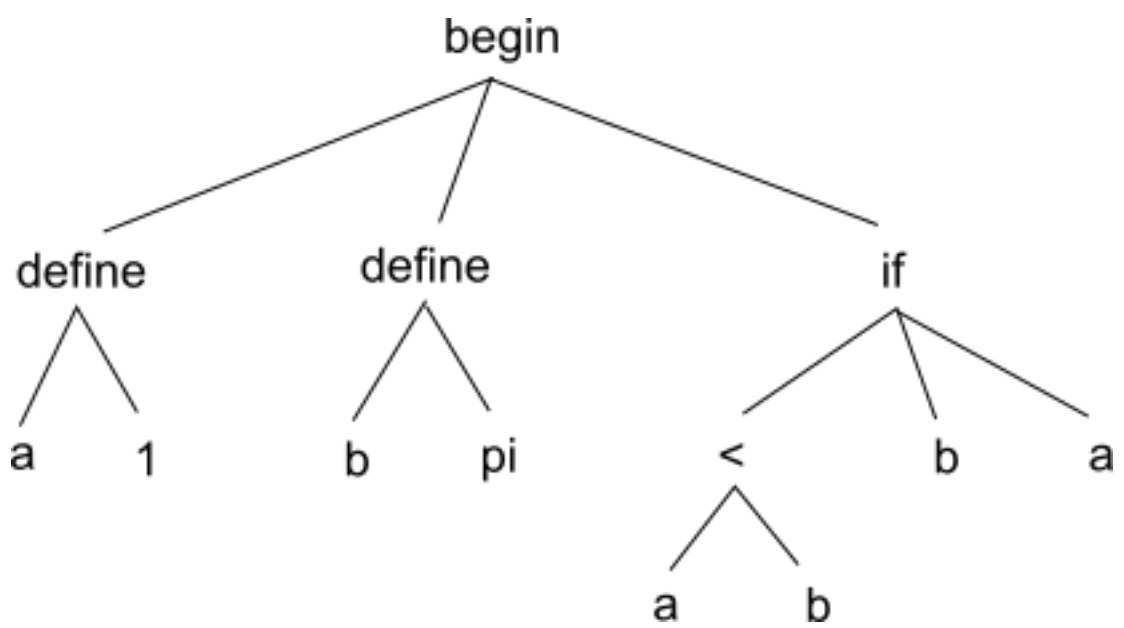
mappings.

Consider another example, a program that finds the max of two numbers:

```
(begin
 (define a 1)
 (define b pi)
 (if (< a b) b a)
 )
```

The outermost expression is a *special form* named `begin`, that simply evaluates each argument in turn. Its first argument is an expression that when evaluated adds a *symbol* `a` to the *environment* with a value of 1. The second argument of `begin` is another expression, this time a special form `define`, that adds a symbol `b` to the environment with a value of the built-in symbol `pi` (that is, the symbol pi is in the default environment). The third argument is an expression that evaluates the expression `(< a b)` evaluating the expression `b` if the former evaluates to true, else evaluating the expression `a`. In this case, since 1 < pi, the if expression evaluates to pi (3.14.159...). Notice white-space (e.g. tabs, spaces, and newlines) is unimportant in the syntax, but can be used to make the code more readable.

There are two equivalent views of the above syntax, as a list of lists or as a tree, called the *abstract syntax tree* or AST.



When viewed as an AST, the evaluation of the program (the outer-most expression), corresponds to a *post-order* traversal of the tree, where the children are considered in order left-to-right. Each leaf expression, which in our language must be an atom, is evaluated (to itself). Then the special-form or procedure is evaluated with its arguments. This continues in the post-order traversal until the root of the AST is evaluated, giving the final result of the program, in this case the expression consisting of a single atom, the numerical value of pi (the max of 1 and pi). If at any time during the traversal this process cannot be completed, the program is invalid and an error is emitted (this will be specified more concretely below). Such an invalid program might be syntactically correct, but not semantically correct. For example suppose the programmer forgot to define a value for `a`, as in

```
(begin
 (define b pi)
 (if (< a b) b a)
 )
```

This is not a semantically valid program in our language and so interpreting it should produce an error.

The process of converting from the stream of text characters that constitute the candidate program to an AST is called *parsing*. This is typically broken up into two steps, tokenization and AST construction. The tokenization step converts the stream of characters into a list of tokens, where each token is a syntactically meaningful string. In our language this means splitting the stream into tokens consisting of `(`, `)`, or strings containing no white-space. For example the stream `(+ a ( - 4))` would become the list

```
"(", "+", "a", "(", "-", "4", ")", ")"
```

This can be implemented as a finite state machine operating on the input stream to produce the token stream.

The process of AST construction then uses the token list to build the AST. Every time a `(` token is encountered a new node in the AST is created using the following token in the list. Its children are then constructed recursively in the same fashion. This is called a recursive descent parser since it builds the AST top-down in a recursive fashion.

Simple syntax makes languages much easier to learn, since there is less to remember, and easier to program in. This makes lisp/scheme syntax a good candidate for *scripting languages*, programs written to extend the run-time capabilities of larger programs. In fact we will be using vtscript in such a fashion in Project 2. Scripting languages are generally interpreted rather than compiled. An interpreter reads the source code and computes it's result and side effects, without converting (compiling) to machine code [2]. Interpreters then are programs that read programs and produce output. They can usually be invoked a few different ways, for example reading the program to be interpreted from a file or interactively with user input. The latter is called a Read-Eval-Print-Loop or REPL.

Language Specification for vtscript

For this project our language will be kept relatively simple (we will extend it in Project 2). It can be specified as follows:

An *Atom* has a type and a value. The type may be one of None, Boolean, Number, or Symbol. The type `None` indicates the expression has no value. The possible values of a Boolean are `True` or `False`. The possible values of a Number are any IEEE double floating point value strictly parsed. The possible values of a Symbol is any string, not containing white-space, not possible to parse as a Number, not beginning with a numerical digit, and not one of the *special forms* defined below.

Examples of Numbers are: `1`, `6.02`, `-12`, `1e-4`

Examples of Symbols are: `a`, `length`, `start`

An *Expression* is an Atom or a special form, followed by a (possibly empty) list of Expressions surrounded by parenthesis and separated by spaces. When an expression consists only of an atom the parenthesis may be omitted.

- `<atom>`
- `(<atom>)`
- `(<atom> <expression> <expression> ...)`

There are three special-form expressions that begin with `define`, `begin`, and `if`. All other expressions are of the form `(<symbol> <expression> <expression> ...)` where the symbol names a *procedure*.

Procedures take the one or more arguments and return an expression according to their name. For example the expression `(+ a b c)` where `a`, `b`, and `c` are atoms representing numbers or expressions evaluating to such an atom and the result is an expression consisting of a single number atom. This expression is *m-ary* meaning it can take m arguments. Some expressions are binary, meaning they take only two arguments, i.e. `(- a b)` subtracts b from a. Other are unary, e.g. `(not True)`. Thus all procedures have an *arity* of 1,2,...m.

The *Environment* is a mapping from symbols to either an Expression or a Procedure. The process of *evaluating* an Expression may modify the Environment (a side-effect) and results in an expression, which consists of a single Atom.

Our language has the following special-forms:

- `(define <symbol> <expression>)` adds a mapping from the symbol to the result of the expression in the environment. It is an error to redefine a symbol. This evaluates to the expression the symbol is defined as (maps to in the environment).
- `(begin <expression> <expression> ...)` evaluates each expression in order, evaluating to the last.
- `(if <expression1> <expression2> <expression3>)` evaluates to the result of `<expression2>` if `<expression1>` evaluates to `True`, else it evaluates to the result of `<expression3>`. It is an error if `<expression1>` does not result in a Boolean Atom. The conditional expression is always evaluated. The others are conditionally evaluated. Thus side effects are conditional.

Our language has the following procedures:

- `not`, unary expression of Booleans, return the logical negation of the argument.
- `and`, m-ary expression of Booleans, return the logical conjunction of the arguments. There is no short circuit. All expressions are evaluated.
- `or`, m-ary expression of Booleans, return the logical disjunction of the arguments. There is no short circuit. All expressions are evaluated.
- `<`, binary expression of Numbers, returns `True` if the first argument is numerically less than the second, else `False`
- `<=`, binary expression of Numbers, returns `True` if the first argument is numerically less than or equal to the second, else `False`
- `>`, binary expression of Numbers, returns `True` if the first argument is numerically greater than the second, else `False`
- `>=`, binary expression of Numbers, returns `True` if the first argument is numerically greater than or equal to the second, else `False`
- `=`, binary expression of Numbers, returns `True` if the first argument is numerically equal to the second,, else `False`
- `+`, m-ary expression of Numbers, returns the sum of the arguments
- `-`, unary expression of Numbers, returns the negative of the argument
- `-`, binary expression of Numbers, return the first argument minus the second
- `*`, m-ary expression of Number arguments, returns the product of the arguments
- `/`, binary expression of Numbers, return the first argument divided by the second

It is an error to evaluate a procedure with an incorrect arity or incorrect argument type.

Our language has the following built-in symbol:

- `pi` , a Number, evaluates to the numerical value of pi, given by atan2(0, -1)

Our language also supports comments using the traditional lisp notation. Any content after and including the character `;` up to a newline is considered a comment and ignored by the parser.

See the directory `tests` in the repository for several example vtscript programs demonstrating the above syntax.

Module Specifications

Your C++ code implementing the vtscript interpreter must be divided into *at least* the following modules, consisting of a header and implementation pair (.hpp and .cpp). You are free to define additional units of code as desired.

- Expression Module (`expression.hpp` , `expression.cpp`): This module must define a struct or class named `Expression` with at least the following public member functions

```
// Default construct an Expression of type None
Expression();

// Construct an Expression with a single Boolean atom with value
Expression(bool value);

// Construct an Expression with a single Number atom with value
Expression(double value);

// Construct an Expression with a single Symbol atom with value
Expression(const std::string & value);

// Equality operator for two Expressions, two expressions are equal if the have
the same
// type, atom value, and number of arguments
bool operator==(const Expression & exp) const noexcept;
```

- Tokenize Module (`tokenize.hpp` , `tokenize.cpp`): This module should define the C++ types and code required to parse a vtscript program into an AST.
- Environment Module (`environment.hpp` , `environment.cpp`): This module should define the C++ types and code required to implement the vtscript environment mapping.
- Interpreter Module (`interpreter.hpp` , `interpreter.cpp`): This module must define a class named "Interpreter`` with at least the following public member functions

```
// Default construct an Interpreter with the default environment and an empty AS
T
Interpreter();

// Given a vtscript program as a std::istream, attempt to parse into an internal
AST
// return true on success, false on failure
bool parse(std::istream & expression) noexcept;

// Evaluate the current AST and return the resulting Expression
// throws InterpreterSemanticError if a semantic error is encountered
// the exception message string should document the nature of the semantic error
Expression eval();
```

The exception class `InterpreterSemanticError` is provided in the file `interpreter_semantic_error.hpp`.

Interpreter Program Specifications

The interpreter module needs some user interface code. This should be a command-line application that compiles to an executable named `vtscript.exe` on Windows and just `vtscript` on mac/linux. The executable should be usable in one of three ways:

- For executing short simple programs, pass a flag `-e` followed by a quoted string with the program. For example (> is the prompt):

  ```
  > vtscript -e "(+ 1 (- 3) 12)"
  ```

  This should evaluate the program in the string and print the result in the format below or produce an appropriate error message, beginning with "Error", if the program cannot be parsed or encounters a semantic error. If an error occurs vtscript should return `EXIT_FAILURE` from main, otherwise it should return `EXIT_SUCCESS`.
- For executing programs stored in external files, provide the file-name containing the vtscript program as a command-line argument. For example, assuming a file named `mycode.vts` is in the current working directory with the executable:

  ```
  > vtscript mycode.vts
  ```

  This should evaluate the program in the file and print the result in the format below or produce an appropriate error message, beginning with "Error", if the program cannot be parsed or encounters a semantic error. If an error occurs vtscript should return `EXIT_FAILURE` from main, otherwise it should return `EXIT_SUCCESS`.
- For interactive execution of programs using a REPL, just type the executable name:

  ```
  > vtscript
  ```

This should print a prompt `vtscript>` to standard output and wait for the user to type an expression on standard input. It should then evaluate the provided expression and print the result in the format below, or print an error message, beginning with "Error", if the line cannot be parsed or encounters a semantic error during evaluation. If a semantic error is encountered the environment should be reset to the default state. After printing the result the REPL should prompt again. This should continue until the user types the EOF character (Control-k on Windows and Control-d on unix). Changes to the environment should be persistent during the use of the REPL (unless an evaluation error occurred). If the user provides an empty line at the REPL (just types Enter) is should just ignore the input and prompt again.

Output Format: Expressions returned from the interpreter evaluation should be printed as `(<atom>)`. Errors should be printed on a single line as the string "Error: " followed by an error message describing the error.

Example transcripts of use:

- Executing a simple example at the command line:

```
> vtscript –e "(* 2 3)"
(6)
```

- Execute the program in a file (showing it first using cat):

```
> cat program.vts
; print the larger of two numbers
(begin
  (define a 1)
  (define b 2)
  (if (< a b) b a)
)
> vtscript program.vts
(2)
```

- Execute some expressions in the REPL:

```
> vtscript
vtscript> (define a 12)
(12)
vtscript> (define b 10)
(10)
vtscript> (– a b c)
Error: unknown symbol
vtscript> (– a b)
Error: unknown symbol
vtscript> (– 12 10)
(2)
```

Unit Testing

For each module of code above you should have a set of unit tests covering its functionality using Catch. Basic tests for the interpreter module are included in the file `test_interpreter.cpp`. These tests should be built as part of your overall project using CMake as described below. At a minimum you should:

- add tests to cover your tokenize, environment and expression modules,
- add additional tests to `test_interpreter.cpp` to cover it more fully,
- and add tests to cover any additional modules or helper functions you might write.

Using CMake to build and test your software

The repository contains a `CMakeLists.txt` file that sets up the tests I am providing and builds the vtscript executable. It also creates a configuration header `test_config.hpp` from the file `test_config.hpp.in` when cmake is run so the tests can find the directory containing example vtscript programs (the tests subdirectory).

You will need to modify this file. See the sections at the top marked `EDIT`. In the appropriate section add any source files you create not in the specification for implementing the interpreter, unit-tests, or the vtscript application. This should allow all parts of the software and tests to be built and run. In the virtual machine this translates to the following:

```
> cd ~
> cmake /vagrant
> make
> make test
```

This treats the source directory as the shared host directory (`/vagrant`) and places the build in the home directory of the virtual machine user (`/home/ubuntu`). Using CMake on your host system will vary slightly by platform and compiler/IDE.

Integration Testing

A basic script is included that tests operation of the vtscript user interface. It should be run inside the virtual machine created using the included Vagrantfile. Note it tests for partial correctness only and does not cover all cases. It is intended primarily to ensure our full grading scripts can interact with your vtscript application. In the virtual machine this would typically look like:

```
> cd ~
> cmake /vagrant
> make
> make test
> python3 /vagrant/integration_test.py
```

UPDATE: So that you can see for yourself what is the intended output given a script, I am providing the binary of my solution. You can install this in the VM using the following:

```
wget https://filebox.ece.vt.edu/~ECE3574/projects/01-vtscript/VTSCRIPT_1.0.2.deb
sudo dpkg -i VTSCRIPT_1.0.2.deb
```

This will add a program vtscript-clw to your path (installed in `/usr/local/bin`) which you can run to get the

expected output of various expressions using any of the command line modes.

Submission

Your submission must consist of a git repository containing **only** the source code and any supporting test and configuration files (Vangrantfile, CMakeLists.txt, tests directory).

To submit your assignment, either the beta or final versions:

1. Tag the git commit that you wish to be considered for grading as "beta" or "final".

   ```
   git tag <tagname>
   ```

   If you want to have multiple versions of the tags, name them sequentially, i.e. beta, beta2, beta3, final, final2, final3, etc. We will consider the last one present in the repository when it is pulled for grading.
2. Push this change to GitHub

   ```
   git push origin <tagname>
   ```

Be sure you have committed all the changes you intend to by the respective due dates. **Failure to complete these steps by the due date will result in no credit being assigned.**

Grading

There are 100 points allocated to this assignment distributed among the beta and final versions.

	Beta	Final
Correctness	15	35
Testing	10	30
Code Quality	-	10

- The correctness score is determined by the fraction of correctness tests your code passes and whether or not your code has memory errors (leaks, invalid reads/writes, etc).
- The testing score is determined by the quality of your unit tests, that is the amount of coverage they test.
- The code quality score is determined by the code compiling cleanly with no warnings (at the same warning level as P0) as well as a other criteria that will be provided as part of the beta grading.

Note: Some partial credit may be assigned based on evidence of work and good incremental development practice, as exhibited by your git commit history. Be sure to record your development progress via commits.

Notes

---

1. This project is loosely based on the lispy interpreter (http://norvig.com/lispy.html) by Peter Norvig.↩
2. This distinction is not always clear, many interpreters do compile to machine code or to a virtual machine. These are called just-in-time or JITing interpreters.↩
