CS 6371	**Advanced Programming Languages**	Assignment 2:
Spring 2020	Due: Jan 30, 2020 by 1:05pm	SIMPL Interpreter

1 Introduction

In this assignment we will implement an interpreter for the language SIMPL, whose syntax and operational semantics are defined in Section 6 of this document. An interpreter executes an arbitrary program step-by-step according to the programming language's operational semantics. Our interpreter will consist of three modules: a lexer, a parser, and an evaluator. The job of the lexer and parser is to read a SIMPL program from a text file and convert it into an OCaml data structure. The evaluator simulates the SIMPL program and returns the final memory state that results. The lexer and parser have been written for you; your job is to code the evaluator.

2 Overview of Files

2.1 Support Code

Obtain the starting code for the assignment by downloading these files from eLearning:

```
simpltypes.ml
simplparser.mly
simpllexer.mll
simpl.ml
```

File `simpltypes.ml` defines the OCaml data structures we will use to model SIMPL programs. You should read through the file to familiarize yourself with the types defined therein. Your evaluator code will be given a value of type `icmd`, which models the SIMPL program to be evaluated.

You do not need to look at the `simplparser.mly` or `simpllexer.mll` files at all to complete the assignment. They tell OCaml how to turn a text file into an `icmd` value. (Those interested how this is accomplished should consult Chapter 12 of the OCaml online manual.)

File `simpl.ml` is where all your code should go. Please do not modify any of the other files!

2.2 Build Procedure

Once you acquire the four files listed above, put them all in the same directory somewhere, and make an initial build of the software by executing the following commands in this order:

```
ocamlc -c simpltypes.ml
ocamlyacc simplparser.mly
ocamlc -c simplparser.mli
ocamlc -c simplparser.ml
ocamllex simpllexer.mll
ocamlc -c simpllexer.ml
ocamlc -c simpl.ml
ocamlc -o simpl.exe simpltypes.cmo simpllexer.cmo simplparser.cmo simpl.cmo
```

(When compiling on Unix, omit the `.exe` extension from `simpl.exe` in the final command.) Note that the OCaml programs `ocamlc`, `ocamlyacc`, and `ocamllex` must all be in your path in order for this to work. (They are all located in the same directory, so if `ocamlc` is in your path, the other two should be also.)

The first six of these commands should only need to be executed once. As you edit `simpl.ml`, you should only need to execute the last two commands to recompile your new code.

3 The Assignment

3.1 List of Tasks

Here is a list of the tasks you must complete to finish the assignment:

1. The `store` type (denoted σ in class) defined in `simpl.ml` models memory as a function from variable names to integers. (The type `varname` is just an alias for `string`; it is defined in `simpltypes.ml`.) You must write code for the `init_store` function in `simpl.ml`, which creates the initial memory store from a list ℓ of variable-integer pairs. That is, your `init_store` implementation should return a function f such that applying f to any variable name v returns the integer that is paired with v in list ℓ. When your function f is applied to any variable that is not in list ℓ, it may either return an arbitrary integer or it may raise an exception.

   Hint: You may wish to use the `List.assoc` library function in your implementation. Alternatively, you might wish to use the `List.fold_left` library function together with your solution to problem 2 of homework 1.

2. The evaluator in `simpl.ml` consists of three functions: `eval_arith` (for evaluating arithmetic expressions), `eval_bool` (for evaluating boolean expressions), and `exec_cmd` (for evaluating commands). All three take the current store as input, along with a value of type `iarith`, `ibool`, or `icmd`, respectively. Each of these types is defined in `simpltypes.ml`. Functions `eval_arith` and `eval_bool` return an OCaml integer or boolean value, respectively. Function `exec_cmd` returns the new store that results from evaluating the `icmd` given as input. The initial evaluator you're given does nothing; it just returns the store unaltered. Your assignment is to fill in the missing code to complete the evaluator.

   Hint: Start by implementing `eval_bool` and then `eval_arith`. Finally, implement `exec_cmd`. To implement the case for `Assign`, use your solution to problem 2 of homework 1.

The `main` function at the bottom of the `simpl.ml` file reads a SIMPL program from a text file into an `icmd` structure and then calls your code to execute it. You won't need to modify `main` to complete the assignment, but it may help to understand what it does in order to complete the tasks above:

The `main` function first invokes the lexer and parser defined in the files `simpllexer.mll` and `simplparser.mly` to convert the text file into an `icmd` structure. Next, it invokes your `init_store` code to create an initial store. The list it passes to `init_store` is constructed from the command-line arguments that the user typed when launching the interpreter. It next passes the initial store returned by `init_store` along with the SIMPL program to your `exec_cmd` code. Finally, if the SIMPL program terminates, the `main` function prints the final value of variable `ret` and terminates.

3.2 Testing

You should test your implementation using the two sample SIMPL programs provided with the assignment:

- `factorial.sim` computes the factorial of its input. To test it, first build the interpreter (see section 2.2) and then execute:

  ```
  simpl factorial.sim 5
  ```

  If your evaluator code is correct, then the program should print 120 (= 5!).

- `isprime.sim` yields 1 if its input is a prime number and 0 otherwise. To test it, execute:

  ```
  simpl isprime.sim 100
  ```

  If your evaluator code is correct, then the program should print 0 (because 100 is not a prime number).

You might wish to write more of your own SIMPL programs to test your code. These will come in handy in later assignments, since it will give you a suite of SIMPL programs with which you can test your enhancements to the interpreter.

4 Submission of Solutions

You should submit your solution to this assignment through eLearning as a single file named `lastname.ml`. As usual, please put an OCaml comment at the top of your `.ml` file that includes your name and citations of any sources or collaborators. You may modify anything in `simpl.ml`, but do not modify any of the other files without permission from the instructor. To test your solutions, I will compile them using the same steps as described in section 2.2.

5 Grading

Your assignments will be graded on a scale of 0–10, where each grade is assigned as follows:

10: *a perfect solution; elegant code and no bugs found*
9: *somewhat inelegant, but it seems to work*
7–8: *mostly correct but one or two small bugs*
5–6: *major flaws for one or two cases*
2–4: *solution compiles, but significant code missing*
1: *did not compile, or no significant student-written code*
0: *no submission / late submission / no relevant student-written code*

Although the grading is out of only 10 points, all homeworks in the course will be weighted equally when computing your final grade.

Commenting your code can help earn you partial credit. In particular, if you cannot figure out how to solve part of the problem, include comments that describe what you were trying to do and why you couldn't get it to work. Be sure that your code compiles even if you know it doesn't correctly evaluate some SIMPL programs, since code that does not compile is an automatic grade of 0 or 1. If you know that your code fails on some inputs, you should say so in your comments. This can sometimes make the difference between grades within each range, since knowing that there is a bug means you were closer to a correct solution than if you didn't realize there was an error.

6 Language Reference

The following rules define the syntax and large-step operational semantics of SIMPL. These should be the basis for your evaluator implementation.

6.1 Syntax of SIMPL

commands	$c ::= \texttt{skip} \mid c_1\texttt{;}c_2 \mid v\texttt{ := }a \mid \texttt{if }b\texttt{ then }c_1\texttt{ else }c_2 \mid \texttt{while }b\texttt{ do }c$
boolean expressions	$b ::= \texttt{true} \mid \texttt{false} \mid a_1\texttt{<=}a_2 \mid b_1\texttt{ \&\& }b_2 \mid b_1\texttt{ || }b_2 \mid \texttt{!}b$
arithmetic expressions	$a ::= n \mid v \mid a_1\texttt{ + }a_2 \mid a_1\texttt{ - }a_2 \mid a_1\texttt{ * }a_2$
variable names	v
integer constants	n

6.2 Operational Semantics of SIMPL

6.2.1 Commands

$$\langle \texttt{skip}, \sigma\rangle \Downarrow \sigma \tag{1}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma_2 \qquad \langle c_2, \sigma_2\rangle \Downarrow \sigma'}{\langle c_1\texttt{;}c_2, \sigma\rangle \Downarrow \sigma'} \tag{2}$$

$$\frac{\langle a, \sigma\rangle \Downarrow n}{\langle v\texttt{ := }a, \sigma\rangle \Downarrow \sigma[v \mapsto n]} \tag{3}$$

$$\frac{\langle b, \sigma\rangle \Downarrow T \qquad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if }b\texttt{ then }c_1\texttt{ else }c_2, \sigma\rangle \Downarrow \sigma'} \tag{4}$$

$$\frac{\langle b, \sigma\rangle \Downarrow F \qquad \langle c_2, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if }b\texttt{ then }c_1\texttt{ else }c_2, \sigma\rangle \Downarrow \sigma'} \tag{5}$$

$$\frac{\langle \texttt{if }b\texttt{ then }(c\texttt{;while }b\texttt{ do }c)\texttt{ else skip}, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{while }b\texttt{ do }c, \sigma\rangle \Downarrow \sigma'} \tag{6}$$

6.2.2 Boolean Expressions

$$\langle \texttt{true}, \sigma\rangle \Downarrow T \tag{7}$$

$$\langle \texttt{false}, \sigma\rangle \Downarrow F \tag{8}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1\texttt{<=}a_2, \sigma\rangle \Downarrow n_1 \leq n_2} \tag{9}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow p \qquad \langle b_2, \sigma\rangle \Downarrow q}{\langle b_1\texttt{ \&\& }b_2, \sigma\rangle \Downarrow p \wedge q} \tag{10}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow p \qquad \langle b_2, \sigma\rangle \Downarrow q}{\langle b_1\texttt{ || }b_2, \sigma\rangle \Downarrow p \vee q} \tag{11}$$

$$\frac{\langle b, \sigma\rangle \Downarrow p}{\langle \texttt{!}b, \sigma\rangle \Downarrow \neg p} \tag{12}$$

6.2.3 Arithmetic Expressions

$$\langle n, \sigma\rangle \Downarrow n \tag{13}$$

$$\langle v, \sigma\rangle \Downarrow \sigma(v) \tag{14}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1\texttt{ + }a_2, \sigma\rangle \Downarrow n_1 + n_2} \tag{15}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1\texttt{ - }a_2, \sigma\rangle \Downarrow n_1 - n_2} \tag{16}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1\texttt{ * }a_2, \sigma\rangle \Downarrow n_1 n_2} \tag{17}$$