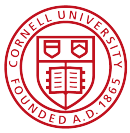
CS 4110 – Programming Languages and Logics
Homework #3



Instructions. This assignment may be completed alone or with a partner. You and your partner should submit a single solution on Gradescope. Please do not offer or accept any other assistance on this assignment. Apart from the automatic 24-hour extension, late submissions will not be accepted.

Submit these files on Gradescope: your completed `eval.ml`, an archive called `tests.zip` containing your 10 tests, and a text file `debriefing.txt`. Because we grade anonymously, please do not include your name or NetID in your submission.

Exercise 1. Consider the IMP language, extended with the following constructs:

$$
\begin{array}{lll}
a \in \textbf{Aexp} & a ::= & \cdots \mid \textbf{input} \\
b \in \textbf{Bexp} & b ::= & \cdots \mid a_1 \le a_2 \mid a_1 \neq a_2 \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 \ge a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2 \\
c \in \textbf{Com} & c ::= & \cdots \mid \textbf{print}\ a \mid \textbf{break} \mid \textbf{continue} \mid \textbf{test}\ b
\end{array}
$$

Informally, these constructs should behave as follows:

- The new boolean expressions are standard, except that they should not short-circuit. For example, in the expression b_1 **or** b_2, both b_1 and b_2 should be evaluated even if b_2 evaluates to **true**.
- The arithmetic expression **input** prints "> " and read an integer from the console.
- The command **print** a evaluates a and prints the result and a newline to the console.
- The command **test** b is a unit test. If b evaluates to **true**, it behaves like **skip**. Otherwise, it should print `TestFailed` and the location of the test in the file and halt the interpreter.
- The commands **break** and **continue** modify the flow of control in **while** loops. The **continue** command skips the remainder of the current loop iteration and proceeds with the next iteration, while the **break** command terminates the loop and proceeds with the next command after the loop. (These are the same semantics as in "real" imperative languages, such as Python.)

  A **break** or **continue** outside of the body of any **while** loop should be treated as an error. In a formal operational semantics, configurations of the form $\langle \sigma, \textbf{break} \rangle$ and $\langle \sigma, \textbf{continue} \rangle$ would not have any transitions—they would "get stuck." In your interpreter, such configurations should print `IllegalBreak` and `IllegalContinue` and halt.

Your task has two parts:

1. Implement an interpreter for this language in OCaml. Many of the aspects of the design, such as how to represent the store, are up to you. To help you get started, we have provided a number of files including a parser, pretty printer, and main module. The code is all in this `hw3-release.zip` file, including a README that explains the various source files and how to build the program. You only need to edit the `eval.ml` file. Submit your edited `eval.ml` as your solution to this part.

2. Develop a suite of 10 test IMP programs, each in a separate file. Your tests should not use the **input** command and should terminate successfully when run on a correct interpreter. Because your tests should focus on covering as much of the language's interesting behavior as possible, you should avoid redundancy between tests. We will give extra credit for any tests that expose a bug in our implementation. Please submit a single archive `tests.zip` with the tests in files named `test01.imp` through `test10.imp`.

To implement the **break** and **continue** commands, we need a way to keep track of the command that should be executed after one of these commands is encountered. Before you start coding, you may find it useful to sketch out the operational semantics for these constructs on paper. The following discussion may be helpful for getting started, but it is not the only way to implement the interpreter.

Consider the following program:

```
x := 1;
y := 1;
while (true) do {
  x := x + 1;
  if (4 < x) then { break } else { skip };
  continue;
  y := y + 1
};
x := x * 2
```

After the **break** is executed, execution should proceed with the command `x := x * 2`. We will call this command a *continuation* because it describes how execution should proceed after the control flow of the program has been altered, and we will extend the operational semantics to keep track of continuations during evaluation. Note that because the language has two control operators (**break** and **continue**) we will need to keep track of pairs of continuations, and because loops can be nested, we will need to keep track of a list of pairs.

A simple way to represent continuations is to change the configurations for commands to $\langle \sigma, c, c', \kappa \rangle$ where:

- σ is the store,
- c is the current command,
- c' is a continuation representing the next command to be executed after c, and
- κ is a list of continuation pairs (c_b, c_c) where c_b is the command that should be executed after a **break** and c_c is the command that should be executed after a **continue**.

Initially, c' is **skip** and κ is []. Executing a **break** or **continue** command should discard the continuation c' and proceed with a command retrieved from κ. Executing a **while** command should add a pair of commands to κ.

To help you get started, here are some rules that define the behavior of **skip** and sequence commands using continuations:

$$\textsc{SkipSkip}\ \frac{}{\langle \sigma, \textbf{skip}, \textbf{skip}, \kappa \rangle \Downarrow \sigma} \qquad \textsc{SkipContinue}\ \frac{c' \neq \textbf{skip} \qquad \langle \sigma, c', \textbf{skip}, \kappa \rangle \Downarrow \sigma'}{\langle \sigma, \textbf{skip}, c', \kappa \rangle \Downarrow \sigma'}$$

$$\textsc{SeqSkip}\ \frac{\langle \sigma, c_1, c_2, \kappa \rangle \Downarrow \sigma'}{\langle \sigma, (c_1; c_2), \textbf{skip}, \kappa \rangle \Downarrow \sigma'} \qquad \textsc{SeqContinue}\ \frac{\langle \sigma, c_1, (c_2; c'), \kappa \rangle \Downarrow \sigma'}{\langle \sigma, (c_1; c_2), c', \kappa \rangle \Downarrow \sigma'}$$

Note there are two cases for each. For **skip**, if the continuation is also **skip**, then we halt. Otherwise, we proceed with the continuation c'. Similarly, for sequences $c_1; c_2$, if the continuation is **skip** then we execute c_1 and use c_2 as the continuation. Otherwise, we execute c_1 and "shift" the remaining work, c_2, onto the continuation by using $c_2; c'$ as the continuation.

Debriefing

1. How many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did everyone in your study group participate?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. If you have any other comments, we would like to hear them! Please write them here or post a private note on Ed.