Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 12:00-13:15 @ KMC 185; Office Hour: TR 11:00-11:55.

Assignment 3: Imperative Language and Semantics

(Due Thursday 10/18/18)

This week's assignment deals with imperative language's implementation and formal semantics.

1. EL1

EL1 is a simple imperative expression language. Its grammar is shown below:

```
Expr -> Int
     |  String
     |  (+ Expr Expr)
     |  (- Expr Expr)
     |  (* Expr Expr)
     |  (/ Expr Expr)
     |  (% Expr Expr)
     |  (<= Expr Expr)
     |  (:= String Expr)
     |  (while Expr Expr)
     |  (if Expr Expr Expr)
     |  (write Expr)
     |  (seq Expr Expr)
     |  (skip)
     |  (for String Expr Expr Expr)
```

An informal semantics for this language is as follows. The evaluation of each expression yields a single integer result.

- An integer `i` yields itself.
- A variable `x` yields its current value. Every variable is implicitly initialized to 0 at the beginning of program execution.
- Evaluating `(+ e1 e2)` evaluates `e1` and then `e2`, and yields the sum of their values. The other arithmetic operations are similar.
- Evaluating `(<= e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the first is less than or equal to the second, the expression yields 1; otherwise it yields 0.
- Evaluating the assignment expression `(:= x e)` evaluates `e`, assigns the resulting value into variable `x`, and yields that value.
- Evaluating `(while c b)` evaluates expression `c`; if the result is non-zero, expression `b` is evaluated and the entire `while` expression is evaluated again; otherwise the evaluation of the `while` is complete. A while expression always yields the value 0.
- Evaluating `(if c t f)` evaluates `c`; if the result is non-zero, then expression `t` is evaluated and the resulting value is yielded as the value of the `if` expression; otherwise expression `f` is evaluated and the resulting value is yielded as the value of the `if` expression.
- Evaluating `(write e)` evaluates `e`, prints the resulting value (followed by a `newline`) to standard output, and yields that value.
- Evaluating `(seq e1 e2)` evaluates `e1` and then `e2`, and yields the value of `e2`.

- Evaluating `(skip)` yields 0.
- Evaluating `(for x e1 e2 e3)` first evaluates `e1` to a value `v1` and stores into `x`; then repeats the following steps
  - Evaluate `e2` to a value `v2`
  - Fetch the value of `x` (call that `vx`)
  - If `vx > v2` then terminate evaluation of the for loop yielding the value 0
  - Otherwise, evaluate `e3` and discard the yielded result; then fetch the value of `x`, add 1 to it, and store the result back into `x`.

  For example, `(for i 1 10 (write i))` writes the numbers from 1 to 10 and yields the value 0.

2. EL1 Interpreter

The file `Interp1.scala` contains a skeleton of an EL1 interpreter. Your task is to complete the implementation.

- For the `for` expression, make sure you get the order of evaluation right. For example, the bizarre expression

```
(for i
     0
     (seq (:= i (+ i 2))
          10)
     (seq (write i)
          (:= i (+ i 3))))
```

  writes the numbers 2 and 8 and yields the value 0.

A test program, `TestInterp1.scala`, is provided. It contains a set of small tests, as well as an EL1 example program that computes prime numbers. Run your interpreter with these tests and make sure it passes them all.

2. A Second Interpreter

In evaluating binary operations (`+`, `-`, `*`, `/`, `%`, and `<=`), EL1's semantics is the same as most of other programming languages'; *i.e.*, it evaluates the operands from left-to-right. (Your interpreter implementation in Part 1 should reflect this.) Now we want to experience a different semantics.

Copy `Interp1.scala` to `Interp1b.scala`, and change the `object` name inside the program to `Interp1b`. Modify the interpreter code for the binary operators, so that the operands will now be evaluated right-to-left.

To test your new interpreter, you may run `TestInterp1b.scala`, which is identical to `TestInterp1.scala`, except for it invokes the new interpreter. Does the new interpreter passes all tests (hence behaves exactly the same as the first interpreter on these tests)?

Inspect the tests inside the test program. Do you think the two interpreters should behave the same on these tests?

Additional Task Find an example EL1 expression, not involving a write expression, that produces different answers depending on the order of evaluation of these operands. Test it with both interpreters to confirm.

3. Operational Semantics

We want to turn the informal semantics of EL1 given above into formal semantics, more specifically, into big-step operational semantics.

In this week's lectures, you've seen an example of big-step semantics for a language called IMP. EL1 has similar constructs, so you may use IMP's semantics as a blueprint. However, you need to note the following:

- EL1 has a different syntax, *e.g.* `(:= x e)` for assign and `(while c b)` for while. You need to use valid EL1 syntax in your specification.
- EL1 does not have Boolean expressions; and there is no `true` or `false`.
- With IMP, expressions reduce to a value, while statements reduce to a (new) state. In EL1, there is no distinction between expressions and statements; all constructs are expressions, and they each evaluate to a value. However, some expressions may have *side-effects*, *e.g.* they may change program state through their evaluation. To capture semantics in full, you need to use a value-state pair $\langle n, \sigma \rangle$ to represent the result of a reduction in the specification. Here is an example for integer:

  $$\frac{}{\langle \mathtt{n}, \sigma \rangle \Downarrow \langle n, \sigma \rangle}$$ (*Note:* `n` is an EL1 expression, while n is an integer value.)

Your Tasks

1. Give a big-step operational semantics for each of the following EL1 expressions:

   variable, add, leq, assign, while, if, seq, and skip.

2. (*Optional*) Give a big-step operational semantics for the for expression `(for x e1 e2 e3)`. Carefully study the informal semantics given above, and see how to turn it into an inference rule. (This part is optional, and carries a small extra credit.)

   Hint: You may reference other EL1 expressions in the recursive part of the specification.

4. Submission

Submit both versions of your interpreter, `Interp1.scala` and `Interp1b.scala`, and a file containing the requested example expression from Part 2, as well as the operational semantics from Part 3. Acceptable file formats are plain text and pdf. Combine the three files into a single zip file, and submit it through the "Assignment 3" submission folder on the D2L class website (under the "Activities/Assignments" tab). Keep your original files untouched in case there is a need to show their timestamp.