Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 10:00-11:15 @ EB 103; Office Hour: TR 11:20-12:20.

# Assignment 2: Imperative Language Implementation

**(Due Thursday 11/3/22)**

This assignment deals with issues arose from implementing an imperative language. It is structured the same as Assignment 1, with the implementation of an interpreter and a compiler. For the compiler, the target is an register-machine IR.

## 1. ImpLang

ImpLang is a simple imperative expression language, with the following grammar:

```
Expr -> <num>                       // int const
     |  <id>                        // var
     |  (Op Expr Expr)              // binop
     |  (if Expr Expr Expr)         // if
     |  (:= <id> Expr)              // assign
     |  (write Expr)                // output
     |  (seq Expr Expr)             // sequence of two exprs
     |  (while Expr Expr)           // while loop
     |  (for <id> Expr Expr Expr) // for loop (*advanced feature*)
Op   -> + | - | * | / | %           // arith op
     |  < | > | ==                  // relational op
<num>: integer  // "wholeNumber"
<id>:  string   // "ident"
```

It is an imperative language because it has imperative features, e.g. assignment, output, loop, and side-effecting expressions. It is also an expression language, since all components are in expression form, each produces an integer value.

The following are the informal semantics for this language's constructs:

- A number `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 behave similarly, with one exception. For a `/` or `%` expression, if `e2`'s value is 0, the expression's value is undefined.
- Evaluating `(< e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the first is less than the second, the expression yields 1; otherwise it yields 0.

  The `(> e1 e2)` behaves similarly.
- Evaluating `(== e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the two values are the same, the expression yields 1; otherwise it yields 0.
- Evaluating `(if c t f)` evaluates `c`; if the result is non-zero, then expression `t` is evaluated, otherwise expression `f` is evaluated. The `if` expression yields the value of the selected expression, either `t` or `f`.
- Evaluating the assignment expression `(:= x e)` evaluates `e`, assigns the resulting value into variable `x`, and yields that value.

- 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 `(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.

The last construct of the language is the `for` loop; its semantics is rather complex. Read and understand the specifications carefully. You'll need the details for your interpreter and compiler implementation.

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

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

You may want to defer `for` loop's implementation until after completing all other parts of the language. In fact, the testing cases for this construct are placed in a separated test program. (See below.)

Note that any `for` expression can actually be expressed by an equivalent expression involving a combination of other simpler ImpLang constructs. In other words, the `for` construct is really just a "syntactic sugar." You may want to find such an equivalent expression, since it will give you an alternative way of implementing the `for` construct. If you go this way, you need to make sure that the semantics of the two expressions match exactly.

## 2. RegIR

RegIR is a register-machine IR language, with the following grammar:

```
Prog    -> {Inst} EndInst
Inst    -> <id> = Src AOP Src          // Binop
        |  <id> = Src                  // Move
        |  if Src ROP Src goto <lab>   // CJump
        |  goto <lab>                  // Jump
        |  <lab> :                     // Label decl
        |  print Src                   // Print
EndInst -> return Src                  // Return prog's value
Src     -> <num> | <id>
AOP     -> + | - | * | / | %
ROP     -> < | > | ==
<num>: integer  // integer literal
<id>:  string   // var or temp name
<lab>: integer  // label num
```

A RegIR program is a list of normal instructions (e.g. `Binop`, `Move`, `Jump`, etc.), followed by a final `return` instruction, which is included to handle expression languages such as ImpLang.

Compared to ImpLang, RegIR is clearly a lower-level language:

- no nested expressions, each instruction handles a single operation
- only two primitive control forms, jump and conditional jump

- relational operators are limited to appear only inside conditional jumps

Here is a sample RegIR program for computing the average of the values of three variables, `x`, `y`, and `z`:

```
t1 = x + y
t2 = t1 + z
t3 = t2 / 3
return t3
```

## Your Tasks

In addition to an interpreter and a compiler, there is an extra program for you to implement.

1. **Param-Evaluation Order**

   Following up on Exercise 4, write a Scala program to figure out Scala's choice on function arguments' evaluation order. Specifically, write a program, `EvalOrder.scala`, which, when executed, will print out either `"left-to-right"` or `"right-to-left"`, indicating the evaluation order of a multi-arg function's arguments. You should run this program with both Scala's interpreter and compiler to verify the result.

2. **An ImpLang Interpreter**

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

   There are two main issues in this interpreter implementation. The first is the handling of program states. For ImpLang, program state is just a collection of variable-value bindings, and is implemented by a mutable map in the interpreter:

   ```
   val store = collection.mutable.Map[String,Int]()
   ```

   To interpret a variable, a lookup into `store` is required. If the variable is found, it's value is returned; if not, a default value of 0 should be used (per ImpLang's semantics). You may use `store.get()` plus an `if` statement to deal with this situation, but there is a simpler solution. (*Hint:* Look up the `getOrElse()` method of Scala Map.) Similarly, to interpret an assignment construct, a variable-value binding (e.g. `x->v`) needs to be added into `store`. Look up info on how to do this. (*Hint:* It's very simple.)

   The second issue is dealing with constructs' semantics. You need to pay close attention to the semantic specifications given in the ImpLang section. For the `for` construct, you may either implement its semantics directly (make sure you get the order of evaluation right), or implement it using an equivalent expression.

   Two test programs, `TestILI.scala` and `TestILI2.scala`, are provided. The first one contains a set of tests for the simpler constructs of ImpLang, while the second one contains tests involving the `for` construct, including a prime-finding program. Run your interpreter with these tests and aim at passing all of them. Recall that to run the tests, you do

   ```
   linux> scala org.scalatest.run TestILI
   linux> scala org.scalatest.run TestILI2
   ```

3. **An ImpLang Compiler**

   The file `ILComp.scala` contains a skeleton of an ImpLang to RegIR compiler. Your task is to complete the compiler implementation.

   The main issue involving this compiler implementation arose from the differences between a stack-machine target and a register-machine target. Recall that in generating code for a stack machine, you only need to focus on generating instructions, not their operands. For each instruction, operands are automatically fetched from the (invisible) operand stack and the result is automatically stored back onto the stack. As an example, consider compiling the expression `(+ e1 e2)`. The recursive `compile()` routine in `ELComp.scala` gets called twice to generate code for the two operands, and their results are concatenated together, along with a new addition instruction: `compile(e1) ::: compile(e2) ::: (SAnd::Nil)`.

Now consider compiling the same expression to a register-machine target. The formula `compile(e1) ::: compile(e2) ::: <new add inst>` would not work. *Why*? Because the `<new add inst>` will be in the form of `t3 = t1 + t2`, where `t1` and `t2` represent the objects (vars, temps, or consts) that hold the values of `e1` and `e2`, and `t3` is a new temp created for holding the addition's result. *Where can we get the info about* `t1` *and* `t2`*?* Only from the recursive calls `compile(e1)` and `compile(e2)`. Thus, the `compile()` routine needs to return two pieces of data: a segment of code, and a `Src` object where the segment's value can be found. Here is the declaration of this routine in `ILComp.scala`:

```
def compile(e:Expr): (Program,Src) = ...  // return type is a pair
```

Now we can generate code for the addition construct as follows:

```
case Add(e1,e2) => {
  val tmp = newTemp()
  val (p1,s1) = compile(e1)
  val (p2,s2) = compile(e2)
  val p = p1 ::: p2 ::: (Bop(AOP.Add,tmp,s1,s2) :: Nil)
  (p, Name(tmp))
}
```

Similar to the interpreter case, two test programs are provided, `TestILC.scala` and `TestILC.scala`. They contain the same sets of tests.

**Notes:**

- As usual, the two language definition programs, `ImpLang.scala` and `RegIR.scala`, are *not* to be modified in any way. And, the given code structure in `ILInterp.scala` and `ILComp.scala`, must stay unchanged, although you are free to define additional variables and methods if needed.
- In addition to running the test programs, you may test your interpreter and compiler with more ImpLang programs of your own. Suppose you have a program in `myprog.il`, you may run the interpreter and the compiler directly:

  ```
  linux> scala ILInterp myprog.il [1|2]
  linux> scala ILComp myprog.il [1|2]
  ```

  The optional argument (`'1'` or `'2'`) at the end of the line is for printing out debugging info; `'2'` shows more info than `'1'`.

## Summary and Submission

Write a short summary (half page or one page) covering your experience with this assignment. Save this write-up in a text file, `assign2-notes.txt`. Zip this write-up, along with the three programs, `EvalOrder.scala`, `ILInterp.scala`, and `ILComp.scala`, into a single file, and submit it through the "Assignment 2" submission folder on the Canvas class website.