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

Assignment 1: Interpreter and Compiler

(Due Thursday 10/20/22)

This assignment continues the work in Exercise 2. (You should complete that exercise first.) It aims at implementing an interpreter and a compiler for a slightly bigger language. In addition to the issues you have encountered in the exercise, this assignment deals with some new issues, e.g. type-checking and more cases of source-target instruction mismatch. This assignment carries a total of 10 points. It will be graded mostly based on correctness through testing.

Unzip the file `assign1.zip`, and you'll see an `assign1` folder with this handout, a `Makefile`, and several program files.

1. ExpLang

ExpLang is a simple expression language, defined by the following grammar:

```
Expr -> t                    // true
     |  f                    // false
     |  <num>                // int literal
     |  (not Expr)           // negation
     |  (Op Expr Expr)       // binop expr
     |  (if Expr Expr Expr)  // if expr
Op   -> and | or | xor       // Boolean op
     | + | - | * | / | %     // integer op (% is remainder)
     | < | > | ==            // relational op
```

It supports two data types, Boolean and integer.

- The Boolean portion of the language, `t`, `f`, `not`, `and`, `or` and `xor`, works exactly the same as BoolLang from Exercise 2; they can be used to build arbitrary Boolean expressions. Integer values can not be used in Boolean expressions. ExpLang follows the standard evaluation rule, *not* the short-circuit rule: for a binary operation, both operands are always evaluated.
- The integer portion of the language, `<num>`, `+`, `-`, `*`, `/` and `%`, can be used to build arbitrary integer expressions. Boolean values can not be used in integer expressions.
- Each of the two comparison operators, `<` and `>`, takes two integer expressions as operands, and produces a Boolean result.
- The equality operator, `==`, takes two operands of the same type (i.e. two Booleans or two integers), and produces a Boolean result.
- The if expression, `(if c t f)`, takes a Boolean expression and two integer expressions as its operands. Depending on the Boolean expression `c`'s value being true or false, only one of the two integer operands `t` and `f` is evaluated; the `if` expression yields the value of the selected expression.

The file `ExpLang.scala` contains the language's AST definition and a parser.

Your Task 1: An ExpLang Interpreter

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

The key different between this interpreter and the one from Exercise 2 is that the current interpreter needs to handle two data types. Notice that the `interp()` method is defined with the following signature:

```
def interp(e:Expr): Either[Boolean,Int] = e match {
  ...
}
```

It may return either a Boolean or an integer, wrapped in a `Left()` or `Right()` constructor, as shown in the following sample code:

```
case True => Left(true)
case False => Left(false)
case Num(n) => Right(n)
```

(Recall that Week 1's Scala intro lecture also contains an example of `Either`.)

Requirements:

- The given code structure in `ELInterp.scala` must stay unchanged, but you are free to define additional variables and methods if needed.
- Strictly follow ExpLang's semantics, e.g. Boolean operators only operate on Boolean operands; integer operators only on integer operands; relational operators and the if expression also need to follow their semantic rules. If a violation occurs, your interpreter should raise an `InterpException` with a proper message.

  Hint: The type-checking in the interpreter is value-driven. An expression `e`'s type can be derived from the return value of a call `interp(e)`. Scala's pattern matching feature can be handy for this, e.g.

  ```
  interp(e) match {
    case Left(b) => ...   // e is Boolean
    case Right(i) => ...  // e is integer
  }
  ```
- For both `Div` and `Rem` operations, the interpreter needs to check that the divisor is non-zero, otherwise it needs to raise an "divide by zero" `InterpException`.

2. StackM1

StackM1 is a stack machine with the following instructions:

Instruction	Semantics
Const(n)	load constant n to stack
And	$val_1 \,\vert\, val_2$, push result to stack (bitwise and)
Or	$val_1 \,\&\, val_2$, push result to stack (bitwise or)
Add	$val_1 + val_2$, push result to stack
Mul	$val_1 * val_2$, push result to stack
Divrem	$val_1 \,/\, val_2$, push div and rem results to stack
Pop	pop off val
Dup	replicate val
Swap	swap val_1 and val_2
Label(i)	nop, marking a label position in program
Goto(i)	branch to Label(i)
Ifgt(i)	if $val > 0$ branch to Label(i)
Ifz(i)	if $val = 0$ branch to Label(i)

- n represents an integer value.
- i represents a label number; it is an instruction attribute, not an operand.
- val and val_2 represent the top element of the stack, while val_1 represents the second-to-top element.

Note that StackM1 operates only with integer values.

StackM1 is implemented in `StackM1.scala`. To avoid name conflicts with ExpLang's AST nodes, several instructions' internal class names are prefixed with an `S`. An `exec()` method is defined to run a given StackM1 program.

Your Task 2: An ExpLang Compiler

The file `ELComp.scala` contains a skeleton of an ExpLang to StackM1 compiler. Your task is to complete the compiler implementation. The given code structure must stay unchanged, but you are free to define additional variables and methods if needed.

While the program structure is similar to that of the interpreter, the contents are substantially different.

- Since StackM1 only operates with integer values, the compiler will map Boolean values true and false to integer values 1 and 0.
- Since there is no declarations in ExpLang, there is no type-checking in this compiler.
- Many ExpLang's operations do not have direct corresponding instructions in StackM1. You need to find a solution for each such case. Here are some examples and some hints:
  - `Not` – Since true and false are mapped to 1 and 0, think about how to flip between the two numerical values
  - `And`, `Or` – Can we use the bitwise and and or to implement them?
  - `Xor` – Can you express this operation by using other Boolean operators?
  - `Sub` – Can you express this operation by using other integer operators?
  - `Div`, `Rem` – The `Divrem` instruction seems to be a good match for them, but details need to be taken care of: `Divrem` pushes two result values on to the stack, we only need to keep one (which one?)
- Relational operations are typically not supported as independent instructions on a target machine; instead, they only appear in conditional branch instructions. StackM1 is no exception. Therefore, to implement `<`, `>`, `==`, and `if`, there is a need to introduce labels and branches. As an example, here is a pseudo-code block for implementing `(if c t f)`:

  ```
  <c code>
  Ifz(lab1)
  <t code>
  Goto(lab2)
  Label(lab1)
  <f code>
  Label(lab2)
  ```

  Make sure you understand the connection between the if expression and this code block. To compile the if expression, you just need to convert this code block to actual Scala code. The relational operators can be handled similarly.

  Hint: In constructing the above code block, you'll need to concatenate lists and instructions. The following two Scala operators are very useful for this purpose:

  `:::` (list-list concatenation) – e.g. `list1 ::: list2`

  `::` (prepending element to list) – e.g. `elm :: list`

  To append an element to the end of a list, you may do

  ```
  list ::: (elm :: Nil)
  ```

  An utility routine, `newLabel()`, is provided to you for creating unique new label numbers. (Label numbers are preset to start at 100.) To create a label, such as `lab1`, you can do:

  ```
  val lab1 = newLabel()
  ```

Testing Your Programs

Two test programs are provided: `TestELI.scala` and `TestELC.scala`. Each contains a set of basic tests. Use them to validate your interpreter and compiler implementation:

```
linux> scala org.scalatest.run TestELI
linux> scala org.scalatest.run TestELC
```

These test programs do not cover all cases. You could create your own test programs to cover additional cases. (It can be helpful to create a set of very simple tests, each for a single AST node.)

Summary and Submission

Write a short summary (half page or one page) covering your experience with this assignment:

- Status of your programs. If there are still issues remaining, describe them as clearly as possible.
- Experience and lessons. What issues did you encounter? How did you resolve them?

Save this write-up in a text file, `assign1-notes.txt`, Zip the interpreter and compiler programs and the write-up into a single file, and submit it through the "Assignment 1" submission folder on the Canvas class website.