Assignment 8

Fri May 6 (125 points)

This is an extension of Assignment 7. We're building a compiler for the Calculator language. Previously, we supported only straight-line code, so we could accumulate every instruction into one basic block. Now we will extend the language with `if` statements and `while` loops. There are several new test cases in `src/test/resources/examples/run-expect`.

Source language

The changes to the source language are all in the `stmt` grammar:

```
    | 'while' expr 'do' stmt                      #WhileStmt
    | 'if' expr 'then' stmt                       #IfStmt
    | 'if' expr 'then' stmt ';' 'else' stmt       #IfElseStmt
    | 'begin' prog 'end'                          #BlockStmt
```

Here are a few simple examples of their usage:

```
var x = 3;
if x < 4 then print "Yes";  #expect Yes
print x;                    #expect 3
if x < 2 then print "Cool"; #no output
print "Ok";                 #expect Ok
```

An example with compound statements (`begin`/`end`):

```
var d = 8;
if d > 5 then           #true
  begin
    print "Okay";       #expect Okay
    if d < 10 then      #true
      begin
        print d;        #expect 8
        print "Yup";    #expect Yup
        d = d - 2;
      end;
    print d;            #expect 6
    if d > 7 then       #false
      print "Woo";      #no output
  end;
print d+1;              #expect 7
```

And a program that calculates the factorial of 20:

```
var n = 20;
var k = 1;
while n > 0 do
  begin
    k = k * n;
    n = n - 1;
  end;
print k;        #expect 2432902008176640000
```

The begin/end introduces a new scope, so here's an example where the variable x gets shadowed twice, but then is restored after each end.

```
var x = "Hello";
begin
  var x = 5;
  begin
    var x = 3.14;
    print x;        #expect 3.140000e+00
  end;
  print x;          #expect 5
end;
print x;            #expect Hello
```

IR implementation

To support implementing these conditional and loop statements, we have extended the intermediate representation as follows.

In addition to constants, variables, and temporaries, the `Address` class can now also represent *labels* to which control can be transferred. To construct a label, just call the constructor with no arguments:

```
Address label = new Address();
```

Like temporaries, labels are represented by a serial number, and they appear prefixed by an `L` in the output. Below is an example of one of the sample programs, translated into the IR.

```
var x = 3;                          L391:
if x < 4 then print "Yes";            INT vx30 = STORE(INT 3)
print x;                              BOOL t392 = LT(INT vx30, INT 4)
if x < 2 then print "Cool";           BRANCH(BOOL t392, L394, L393)
```

```
print "Ok";
```

```
L393:
  PRINT(STRING Yes)
  JUMP(L394)
L394:
  PRINT(INT vx30)
  BOOL t395 = LT(INT vx30, INT 2)
  BRANCH(BOOL t395, L397, L396)
L396:
  PRINT(STRING Cool)
  JUMP(L397)
L397:
  PRINT(STRING Ok)
  END()
```

That example also demonstrates several new instructions: BRANCH, JUMP, and END:

- The BRANCH instruction takes a boolean address and two labels. If the boolean is zero it jumps to the first address, if it's one it jumps to the second.
- The JUMP instruction is an unconditional jump, so it takes just one label and transfers control there.
- The END instruction indicates the end of the program. Previously, the program ended when we reached the end of the basic block. Requiring END gives us flexibility in the ordering of basic blocks in the program… we don't need to arrange for the last block to appear last.

Unlike in most assembly languages, we don't allow control to "fall through" the end of one block into the start of the next one. Every basic block should end with either BRANCH, JUMP, or END so there is no ambiguity about what to do next. Again, this is to allow flexibility in block ordering.

The ConvertToIR visitor now keeps an object of type IRGraph rather than BasicBlock. The IRGraph is a map from label numbers to basic blocks — here is the essence of the class:

```
class IRGraph implements Iterable<Integer> {
    private final TreeMap<Integer, BasicBlock> graph = new TreeMap<>();
    private BasicBlock block;

    public void newBlock(Address label) {
        block = new BasicBlock();
        add(label, block);
    }

    void add(Address label, BasicBlock bb) {
        assert label.kind == Address.Kind.LABEL;
        graph.put(label.serialNum, bb);
```

```
    }

    void add(Instruction instr) {
        block.add(instr);
    }
}
```

It keeps track of the current block, to which you can add new instructions. But it also allows you to create a new block, making that one the current block. Here's an example of using the methods of `IRGraph` to terminate the current block with a jump to the next one:

```
    // We're already in some current block
    // Generate label for the new block
    Address label = new Address();
    // Add an instruction to jump to that label
    graph.add(new Instruction(Instruction.Kind.JUMP, label));

    // Now we switch to the new block
    graph.newBlock(label);
    // Further instructions will be in the new block
    graph.add(new Instruction(Instruction.Kind.PRINT,
        new Address(Address.Kind.CONST, Type.INT, "42")));
    graph.add(new Instruction(Instruction.Kind.END));
```

And here's a representation of what that Java code would generate:

```
  L100:
    JUMP(L101)
  L101:
    PRINT(INT 42)
    END()
```

Finally, to support nested scopes of variables in `begin`/`end` blocks, we now use a `SymbolTable` data structure, rather than a simple map to keep track of variable names. We covered this representation of a symbol table in the online session on March 23rd (recording available). It features `enter()` and `leave()` methods to support a stack of scopes.