Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue & Thu 12:00-13:15 @ ASRC 230, Lab: Fri 10:00-11:15/11:30-12:45 @ FAB 170

Assignment 3: IR Interpreter

(Due Thursday 2/25/16 @ 11:59pm)

In this assignment, you are going to implement an interpreter for the intermediate language, IR1. Read the instructions below carefully, since for this assignment, you have freedom in deciding program details. This assignment carries a total of 10 points.

Download the zip file "`hw3.zip`" from D2L. After unzipping, you should see a `hw3` directory with the following items:

`hw3.pdf` — this document

`IR1Interp0.java` — a starter version of the interpreter

`ir/IR1.java` — IR1 definition

`tst/` — contains a set of tests

`Makefile` — for compiling your program

`run` — script running IR programs

The IR1 Language

IR1 is not a new language. We have used it in Assignment 1 and Lab 1. For your convenience, its grammar is attached at the end of this document, but you will mostly be working with IR1's Java class definitions in `ir/IR1.java`.

The Interpreter Program

Your interpreter program should be called `IR1Interp.java`. A starter version is provided in the file, `IR1Interp0.java`. The interpreter program has a standard two-part structure. There are data structures for supporting information storing and retrieving, and there are code for executing the instructions. At the program's top-level, the interpreter reads in an IR1 program through a parser; it then searches for the `IR1.Func` node representing the `_main()` function in the program. (Every valid IR1 program has one `_main()` function.) The interpretation process starts with the `_main()` function.

Each function is interpreted through a standard "fetch-and-execute" loop over its instructions. There is an `execute()` method for every `IR1.Inst` node, which executes the instruction according to its semantics. Collectively, the `execute()` methods have three possible return status:

- `CONTINUE` — the instruction is a "normal" instruction. The interpreter should continue to the next instruction in line.
- `RETURN` — the instruction is a `Return`. The interpreter should break out the current "fetch-and-execute" loop, and return from this function's `execute()` method.
- an index to an instruction (of this function). The interpreter should continue the "fetch-and-execute" loop from the index.

A simple approach for represent these three cases is to use integer as the return value type. Reserve the ≥ 0 range for instruction index cases; and use two negative values for `CONTINUE` and `RETURN`.

For most of the IR1 instructions, the `execute()` method is straightforward to implement. The important issues concerning the interpreter program are discussed below.

- **Value Representation**

  IR1 supports three types of values, integers, booleans, and strings. In the interpreter, we use a unified representation for them:

```
abstract static class Val {}
static class IntVal extends Val { int i; }
static class BoolVal extends Val { boolean b; }
static class StrVal extends Val { String s; }
static class UndVal extends Val {}
```

  This representation simplifies storage organizations.

- **Storage Organizations**

  IR1's storage need includes a heap memory for `malloc`'ed data, and space for individually named objects, including variables, temps, and parameters. Therefore, in the interpreter, we need two storage organizations, one for representing the heap memory, and one for keeping track of variables, temps, and parameters' values.

  The heap memory belongs to the global scope, since `malloc`'ed data needs to be kept alive until the end of program execution. All named objects (excluding functions), on the other hand, exist within a function's scope. Here is a possible storage arrangement:

```
// Global heap memory (a single copy for the whole program)
static ArrayList<Val> memory;
// Environment for vars, temps, and params (one copy per function invocation)
static class Env extends HashMap<String,Val> {}
```

- **Heap Memory Access**

  IR1's heap memory is defined at object size level, *e.g.* an integer stores in four units of memory, while a Boolean value stores in a single unit. For an interpreter, however, there is no requirement to implement a language's memory model with full details. As shown above, we use a unified value representation for all types, and represent the heap memory as an `ArrayList` of `Val`s. With this arrangement, it is easy to store and retrieve multi-unit data — there is no need to decompose or re-assemble.

  However, the interpreter must respect memory address calculation in IR1 programs. So for an integer value, four cells (instead of one) will be allocated in the interpreter's `memory` array. The actual value is store in the first cell, and the other three cells are not used. (They can be filled with the special undefined value, `UndVal`.) The read and store operations to the memory array are not affected, since there is never a need in IR1 to read or write part of an integer.

- **Supporting Data Structures**

  In interpreting a `Call` instruction, there is a need to locate the target function definition, and in interpreting a `Jump` or `CJump` instruction, there is a need to locate the instruction index of the target label declaration. While we can search through the program to find these locations every time there is a need, a better way is to pre-process the program, and create look-up tables for them. Here are possible data structures for this purpose:

```
// Mapping a function name to the function's definition
HashMap<String,IR1.Func> funcMap;
// Mapping a function name and a label name to the label decl's index
static HashMap<String,HashMap<String,Integer>> labelMap;
```

Note that the second look-up table is a nested mapping. This is because labels' scopes are functions. To look up for a label, we need to know which function it is defined in.

A slight improvement on readability is to introduce a type name for the per-function label map:

```
static class LabMap extends HashMap<String,Integer> {}
static HashMap<String,LabMap> labelMap;
```

Note that these data structures are for improving an interpreter's performance. You can still implement an interpreter without using any of them. (But you have to implement searching functions for function definitions and label declarations instead.)

- **Call and Return Instructions**

  For the `Call` instruction, the interpreter first needs to distinguish pre-defined functions from user-defined functions. For pre-defined functions, implement them according to their definitions, *e.g.* for a `_printInt()` call, you may use `System.out.println()` to implement.

  For a user-defined function, the interpreter should take the following steps:

  - evaluate arguments to values;
  - create a new `Env` for the callee; pair function's parameter names with arguments' values, and add them to the `Env`;
  - find callee's `Func` node and switch to execute it;
  - if a return value is expected, copy the return value from its storage location to its destination.

  For passing a return value from callee to caller, a simple approach is to use a global variable, which can be accessed by both parties.

  For the `Return` instruction, the interpreter just needs to make sure that the return value is set to designated storage location (*e.g.* the global variable).

- **Jump and CJump Instructions**

  For the unconditional `Jump` instruction, its `execute()` method should return the instruction index of the jump target label.

  For the conditional `CJump` instruction, its `execute()` method has two possible return values: if the condition is true, it should return the instruction index of the jump target label, otherwise, it should return `CONTINUE`.

- **Addresses and Operands**

  For each of the IR1 address and operand nodes, there is an `evaluate()` method.

  For the `Addr` node, the `evaluate()` method should return an address to the heap memory. If you implement the heap memory as an `ArrayList` (as suggested), then the return value should the index to an array cell.

  For the `Temp` and `Id` nodes, `evaluate()` should look up their values from the current `Env` (*i.e.* this function invocation's `Env`).

  For an literal operand node, *i.e.*, `IntLit`, `BoolLit`, or `StrLit`, `evaluate()` should just return its value.

Requirements and Grading

You should test your interpreter program with provided tests in `tst`:

```
linux> ./run tst/test*.ir
```

You should also test your interpreter with the IR1 programs from Lab 1.

This assignment will be graded mostly on your interpreter's correctness. We may use additional programs to test. The minimum requirement for receiving a non-F grade is that your program compiles on the CS Linux system without error and correctly interprets at least one test program.

What to Turn in

Submit a single file, `IR1Interp.java`, through the "Dropbox" on the D2L class website.

IR1 Grammar

```
Program -> {Func}
Func    -> <Global> VarList                        // Target Params
           [VarList]                               // Locals
           "{" {Inst <EOL>} "}" <EOL>              // Body
VarList -> "(" [<Id> {"," <Id>}] ")" <EOL>

Inst         -> Dest "=" Src BOP Src                    // Binop
           | Dest "=" UOP Src                        // Unop
           | Dest "=" Src                            // Move
           | Dest "=" Addr                           // Load
           | Addr "=" Src                            // Store
           | [Dest "="] "call" <Global> ArgList      // Call
           | "return" [Src]                          // Return
           | "if" Src ROP Src "goto" <Label>         // CJump
           | "goto" <Label>                          // Jump
           | <Label> ":"                             // LabelDec

Src           -> <Id> | <Temp> | <IntLit> | <BoolLit> | <StrLit>
Dest          -> <Id> | <Temp>
Addr          -> [<IntLit>] "[" Src "]"
ArgList -> "(" [Src {"," Src}] ")"

BOP          -> AOP | ROP
AOP          -> "+" | "-" | "*" | "/" | "&&" | "||"
ROP          -> "==" | "!=" | "<" | "<=" | ">" | >="
UOP          -> "-" | "!"

<Global>  = _[A-Za-z][A-Za-z0-9]*
<Label>   = [A-Za-z][A-Za-z0-9]*
<Id>      = [A-Za-z][A-Za-z0-9]*
<Temp>    = t[0-9]+
```

The following functions are pre-defined in IR1:

```
_malloc(size)    // memory allocation
_printInt(arg)   // print an integer (or a boolean as 0 or 1)
_printStr(arg)   // print a string literal (arg could be null)
```