Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

Assignment 1: Interpreter

(Due 1/29/15 @ 11:59pm)

In this assignment, you are going to implement an interpreter for the intermediate language, IR1. This assignment builds on top of Lab 2. If you have not attended the lab for any reason, you should go through its materials first. The assignment carries a total of 100 points.

Preparation

Download the zip file "`hw1.zip`" from D2L. After unzipping, you should see a `hw1` directory with the following items:

- `hw1.pdf` — this document
- `IR1Interp0.java` — a starter version of the interpreter
- `Makefile` — for compiling the lexer (*usage:* `make`)
- `run` — a script for running the tests (*usage:* `./run tst/*.ir`)
- `ir1/` — a subdirectory containing IR1's AST node definitions, and an IR1 parser
- `tst/` — a subdirectory containing a set of test programs

The IR1 Language

In Lab 1, you have programmed with IR1. For your convenience, IR1's grammar is shown below again:

IR1 Grammar

```
Program -> {Func}

Func    -> <Global> VarList [VarList] "{" {Inst} "}"
VarList -> "(" [<id> {"," <id>}] ")"

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

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

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

<Id:     <letter> (<letter>|<digit>)*>
<Temp:   "t" (<digit>)+>
<Global: "_" <Id>>
```

The following functions are pre-defined in IR1:

```
_malloc(size)    // memory allocation
_printInt(arg)   // print an int value (including address value)
_printBool(arg)  // print a boolean value
_printStr(str)   // print a string
```

IR1 has two scope levels, global and function. Only functions exist in the global scope. All other named objects, *i.e.* variables, temps, and labels, exist within a function scope.

For this assignment, you will not be dealing with IR1's grammar directly. Instead, you will be working with IR1's AST representation.

The IR1 AST Representation

IR1's AST representation is defined in `ir1/IR1.java`. This AST corresponds to IR1's grammar closely. Each nonterminal in the grammar is represented by an AST node, so are a selected set of terminals (*e.g.* the ID, temp, and literal tokens). You should spend some time familiarizing yourself with this AST representation, since you are going to write interpretation routines for these AST nodes.

The Interpreter Program

Your interpreter program should be called `IR1Interp.java`. A starter version is provided in `IR1Interp0.java`. Use this program as your starting template.

The issues concerning the interpreter program are listed below. All have been covered in this week's lecture and/or lab. Some have code readily available from the starter version or from the lecture/lab materials.

Value Representation

IR1 supports three types of values, integers, booleans, and string literals. We have seen that these values can be represented as subclasses of a common parent class:

```
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 unified value representation simplies storage and environment organization. This portion of code is included in the starter version.

Storage and Environment Organization

At the global level, we need

- a heap memory for storing 'malloc'ed data, and
- an (optional) function lookup table for easy finding of a function when it is called.

Possible data structures to use:

```
ArrayList<Val> heap;
HashMap<String,IR1.Func> funcMap;
```

At the function level, we need

- environment(s) for storing variables, parameters, and temps,
- an (optional) label lookup table for easy finding of a label instruction when a jump is executed.

Possible data structures to use:

```
HashMap<String,Integer> labelMap;
HashMap<Integer,Val> tempMap;
HashMap<String,Val> varMap;
```

Note that these data structures are to be used at per function level, so a mechanism for maintaining multiple copies of them needs to be established. When a function is called, a new set of these data structures needs to be created. At the same time, the caller's set still needs to be preserved so that once the call returns, the caller's execution can continue. (See more discussion on the `CALL/RETURN` instructions below.)

Execution of Individual Instructions

For each IR1 instruction, there is an `execute()` method defined. It executes the instruction according to its semantics.

Most instructions' semantics are self-evident, hence their `execute()` methods are straightforward. The following instructions need some special attention:

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

  - create a new set of data structures for the callee,
  - evaluate arguments and pass them to callee's data structures,
  - find callee's AST node and switch to execute it,
  - if a return value is expected, copy the return value to its destination.

  The details of parameter passing depends on your data structure implementation. One possibility is to pass the parameters' values to the callee's variable environment.

  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 it follows the return-value-passing mechanism the caller uses.

- `Jump` **and** `CJUMP` **Instructions**

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

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

Evaluation of Addresses and Operands

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 be of type `int`, representing an index into the `ArrayList`.

For the `Temp` and `Id` nodes, `evaluate()` should lookup their values from their corresponding storage environments and return accordingly.

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

Running and Testing

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.

Requirements and Grading

This assignment will be graded mostly on your interpreter's correctness. We may use additional IR1 programs to test.

The minimum requirement 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.