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

Lab 1: Intermediate Representations

This lab is to practice programming in several IR languages. The purposes are two-fold:

- to get a hands-on feel of the differences between IR languages and high-level programming languages, and
- to get familiar with these IR languages in preparation for the up-coming interpretation and IR codegen projects.

You need Java 1.8 to run this lab. If you don't have it in your environment (run `java -version` to check), please use `addpkg` to add it. Download from D2L the file `lab1.zip` and unzip it.

1 Register-Machine IR

Consider a register-machine based language, LR1. Its grammar is shown below. Most of LR1's instructions qualify as "three address", except the `call` instruction, which includes more than three parameters: the call target, the arguments, and a potential return-value target.

IR1 Grammar

```
Program -> {Func}

Func    -> <Global> VarList [VarList] "{" {Inst} "}"

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

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

ArgList -> "(" [Src {"," Src}] ")"

Addr    -> [<IntLit>] "[" Dest "]"
Src     -> <Id> | <Temp> | <IntLit> | <BoolLit> | <StrLit>
Dest    -> <Id> | <Temp>

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

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

Footnotes:

- Formal parameters and local variables to a function need to be explicitly declared. As shown above, they are represented by two separate `VarLists`. (Only their names need to be declared; their associated types need not.) If there no local variables, the corresponding `VarList` can be omitted.
- The following functions are pre-defined:

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

1.1 A Sample IR1 Program

The file `sum.ir1` contains a sample IR1 program:

sum.ir1

```
# Return the sum of array elements (Input: array and its elm count)
#
_sum (a, n)               # function name and parameters
(sum, i)                  # local variables
{
  sum = 0                 # accumulated sum
  i = 0                   # loop idx
L0:
  if i >= n goto L1       # reached the end of array?
  t1 = i * 4              # compute addr of a[i]
  t2 = a + t1             #
  t3 = [t2]               # fetch a[i]
  sum = sum + t3          # add a[i] to sum
  i = i + 1
  goto L0
L1:
  return sum
}

_main ()
(a, sum)
{
  a = call _malloc(12)          # alloc and init array
  [a] = 1                       # store a[0] = 1
  4[a] = 2                      # store a[1] = 2
  8[a] = 3                      # store a[2] = 3
  sum = call _sum(a,3)          # call _sum()
  call _printStr("Array sum:")
  call _printInt(sum)           # print result
  return
}
```

Exercise Try running this program with the provided `runir1` script, which invokes an IR1 interpreter:

```
linux> ./runir1 sum.ir1
```

1.2 IR1 Programming Exercises

In the `lab3` directory, you'll see five additional `.ir1` program files, each contains a driver routine and a function for you to complete. The descriptions of the five functions are:

`average(int[] a, int n)` — Return the average value of array `a`'s elements.

`max(int[] a, int n)` — Return the largest element of the array `a`.

`midx(int[] a, int n)` — Return the index of array `a`'s largest element.

`reverse(int[] a, int n)` — Reverse the order of array `a`'s elements.

`sort(int[] a, int n)` — Sort array `a`'s elements into an ascending order.

In all cases, the second parameter to the function is the array length.

2 Stack-Machine IR

As studied in class, stack machine code is a form of intermediate code. It assumes the presence of an operand stack. Most operations take their operands from the stack and push their results back onto the stack. For example, an integer subtract operation would remove the top two elements from the stack and push their difference onto the stack. Neither the operands nor the result need be referenced explicitly in the subtract instruction. In fact, only a few instructions (e.g. `push` and `pop`) need to reference a *single* operand explicitly. Note that a separate operand stack is associated with each function in a program. Comparing to register-machine IR, stack IR has the advantage of being simple, compact, and easy to interpret.

2.1 The SC1 Language

SC1 is a stack-machine based IR language. For each function in a SC1 program, there is an implicit operand stack and a storage array for local variables. Globally, there is a heap memory. About half of SC1 instructions have no explicit operand; the other half take a single integer operand. The language's grammar is shown below.

SC1 Grammar

```
Program -> {StrDef} {FunDef}

StrDef -> "StrLit" <IntLit> ":" <StrLit>

FunDef -> "Func" <IntLit> ":" <IntLit>
          {Inst}

Inst   -> Inst0               // Inst with no explicit operand
        | Inst1 <IntLit>      // Inst with one operand

Inst0  -> "ALOAD" | "ASTORE" | "NEWARRAY" | "PRINT" | "NEG" | "ADD" | "SUB" |
          "MUL" | "DIV" | "AND" | "OR" | "SWAP" | "VRETURN" | "RETURN"

Inst1  -> "CONST" | "LOAD" | "STORE" | "GOTO" | "IFZ" | "IFNZ" | "IFEQ" |
          "IFNE" | "IFLT" | "IFLE" | "IFGT" | "IFGE" | "SPRINT" | "CALL"
```

Footnotes:

- Each function is given an integer ID; the `main` function's ID is always 0. In a function header, the first integer is its ID, and the second integer represents the number of parameters the function has.
- In addition to functions, string literals also need to declared. Each string is given an integer ID.

- For instructions with an integer operand, the meaning of the integer varies depending on the instruction. (More info is on the next page.)

The SC1's instruction definitions are given in the following table:

Instruction	Sematics	Stack Top (before *vs* after)
`CONST n`	load constant `n` to stack	→ `n`
`LOAD n`	load `var[n]` to stack	→ `val`
`STORE n`	store `val` to `var[n]`	`val` →
`ALOAD`	load array element	`arrayref, idx` → `val`
`ASTORE`	store `val` to array element	`arrayref, idx, val` →
`NEWARRAY`	allocate new array	`count` → `arrayref`
`NEG`	`- val`	`val` → `result`
`ADD`	`val1 + val2`	`val1, val2` → `result`
`SUB`	`val1 - val2`	`val1, val2` → `result`
`MUL`	`val1 * val2`	`val1, val2` → `result`
`DIV`	`val1 / val2`	`val1, val2` → `result`
`AND`	`val1 & val2`	`val1, val2` → `result`
`OR`	`val1 | val2`	`val1, val2` → `result`
`SWAP`	swap top two stack elements	`val1, val2` → `val2, val1`
`GOTO n`	`pc = pc + n`	
`IFZ n`	`if (val==0) pc = pc + n`	`val` →
`IFNZ n`	`if (val!=0) pc = pc + n`	`val` →
`IFEQ n`	`if (val1==val2) pc = pc + n`	`val1, val2` →
`IFNE n`	`if (val1!=val2) pc = pc + n`	`val1, val2` →
`IFLT n`	`if (val1<val2) pc = pc + n`	`val1, val2` →
`IFLE n`	`if (val1<=val2) pc = pc + n`	`val1, val2` →
`IFGT n`	`if (val1>val2) pc = pc + n`	`val1, val2` →
`IFGE n`	`if (val1>=val2) pc = pc + n`	`val1, val2` →
`PRINT`	print `val`	`val` →
`SPRINT n`	print the `n`th string literal	
`CALL n`	call the `n`th function	
	caller:	`arg1,...,argk` →
	callee: args in `var[0]..var[k-1]`	
`RETURN`	return from procedure	
`VRETURN`	return a value from function	
	callee:	`val` →
	caller:	→ `val`

Footnotes:

- For the jump instructions, the operand `n` represents the *relative* displacement (in units of instructions) from the the current instruction position. `n` can be either positive or negative. For example, `GOTO +3` means jump ahread 3 instructions; `GOTO -2` means jump back 2 instructions. (Note that the positive sign, `+`, is optional.)
- For the `SPRINT` and `CALL` instructions, the operand `n` represents a string-literal's or a function's ID.
- To make a function call, you need to push all the arguments on to the operand stack (in the order the function expects) before issuing the `CALL` instruction. The `CALL` instruction will pop off all the arguments from caller's operand stack, and transfer them to callee's local variable storage, where callee's instructions can have access.
- There are two return instructions: `RETURN` for returning from a procedure (no return value), and `VRETURN` for returning from a function with a return value. In the latter case, the return value needs

to be at the top of callee's operand stack. The `VRETURN` instruction will move the value from callee's stack to caller's stack.

2.2 A Sample SC1 Program

The file `sum.sc1` contains a sample SC1 program:

sum.sc1

```
StrLit 0: "Array sum:"

Func 1: 2
0.  CONST 0
1.  STORE 2
2.  CONST 0
3.  STORE 3
4.  LOAD 3
5.  LOAD 1
6.  IFGE +12
7.  LOAD 2
8.  LOAD 0
9.  LOAD 3
10. ALOAD
11. ADD
12. STORE 2
13. LOAD 3
14. CONST 1
15. ADD
16. STORE 3
17. GOTO -13
18. LOAD 2
19. VRETURN
```

sum.sc1

```
Func 0: 0
0.  CONST 3
1.  NEWARRAY
2.  STORE 0
3.  LOAD 0
4.  CONST 0
5.  CONST 1
6.  ASTORE
7.  LOAD 0
8.  CONST 1
9.  CONST 2
10. ASTORE
11. LOAD 0
12. CONST 2
13. CONST 3
14. ASTORE
15. LOAD 0
16. CONST 3
17. CALL 1
18. STORE 1
19. SPRINT 0
20. LOAD 1
21. PRINT
22. RETURN
```

```
// A corresponding
// high-level
// C-like program

int sum(int[] a, int n) {
  int sum = 0;
  int i = 0;
  while (i < n) {
    sum = sum + a[i];
    i = i + 1;
  }
  return sum;
}

void main() {
  int[] a = new int[3];
  int sum;
  a[0] = 1;
  a[1] = 2;
  a[2] = 3;
  sum = sum(a,3);
  print("Array sum:");
  print(sum);
}
```

Exercise Try to find a mapping between the high-level program's statements and the SC1 program's instructions. For each statement, find the corresponding block of SC1 instructions, and add the statement as a comment to the first instruction of the block. Here is one example:

```
0.  CONST 0     # int sum = 0;
1.  STORE 2
```

2.3 SC1 Programming Exercises

Implement the five functions described in the previous section in SC1. Again, you'll find five `.sc1` files in the `lab3` directory. While it is possible to think and program directly in SC1, it would be easier to start with a high-level program, then "translate" it statement-by-statement to SC1 code. Also, note that you don't need to include an instruction number in front of each instruction; the number is optional. Once you have a SC1 program, you may verify it with the provided run scripts, `runsc1` or `runsc1-debug`. Both invoke an SC1 interpreter, with the latter providing additional execution trace information:

```
linux> ./runsc1 sum.sc1
linux> ./runsc1-debug sum.sc1
```

2.4 A Brief Look at Java Bytecode

Take a look inside the Java program, `Sum.java`, in the `lab3` directory. It implements the array sum function. Now compile it with `javac`, then disassemble the `.class` program with `javap`:

```
linux> javac Sum.java
linux> javap -c Sum.class > Sum.bcode
```

The resulting file, `Sum.bcode`, contains the instruction listing of Java bytecode for the source program. Compare this listing with `sum.ir1`. You should see strong similarities between the two.