

Lab 1: Intermediate Representations

Learning Objectives Upon successful completion, students will be able to:

- Describe the main characteristics of IR languages in comparison to high-level languages.
- Write simple programs in both register-machine IRs and stack-machine IRs.

Preparation

Download and unzip the file `lab1.zip`. You'll see a `lab1` directory with the following contents:

```
sum.[java|ir1|sc1] — a sample program in Java, IR1, and SC1 form
hello.[ir1|sc1] — another sample program
avegage.*, max.*, reverse.* — a set of exercise programs
IR1Interp.jar, SC1Interp.jar — IR1 and SC1 interpreters
runir1, runsc1, runsc1-debug — scripts for testing programs
```

IR1: A 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 destination.

"IR1 Grammar"		
Program	-> {Func}	
Func	-> <Global> VarList <EOL>	// Name and Params
	[VarList <EOL>]	// Locals (can be omitted)
	"{" {Inst <EOL> } "}" <EOL>	// Body
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
	"return" [Src]	// Return
	"if" Src ROP Src "goto" <Label>	// CJump
	"goto" <Label>	// Jump
	<Label> ":"	// LabelDec
) <EOL>	
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]+
<IntLit> = [0-9]+
<BoolLit> = true|false
<StrLit> = \"(~\"\",\\n))*\"

```

Pre-Defined Functions:

```

_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)

```

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
}

```

Exercises

1. Try to run the above program with the provided `runir1` script, which invokes an IR1 interpreter:

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

2. Complete the three IR1 programs, `average.ir1`, `max.ir1`, and `reverse.ir1`, by providing an implementation for the following functions:

- `average(int[] a, int n)` — return the average value of array `a`'s elements (`n` is `a`'s length)
- `max(int[] a, int n)` — return the largest element of the array `a` (`n` is `a`'s length)
- `reverse(int[] a, int n)` — reverse the order of array `a`'s elements (`n` is `a`'s length)

The three provided program files are the starter version, each contains a driver for testing the corresponding function.

Compile and test these programs after you finish.

SC1: A 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.

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> <EOL>	// <IntLit> is strLit's ID
FunDef -> "Func" <IntLit> ":" <IntLit> <EOL>	// 1st <IntLit> is func's ID,
{Inst <EOL>}	
// 2nd <IntLit> is #params	
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 be 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	Semantics	Stack Top (before <i>vs</i> after)
CONST <i>n</i>	load constant <i>n</i> to stack	→ <i>n</i>
LOAD <i>n</i>	load var[<i>n</i>] to stack	→ <i>val</i>
STORE <i>n</i>	store <i>val</i> to var[<i>n</i>]	<i>val</i> →
ALOAD	load array element	arrayref, idx → <i>val</i>
ASTORE	store <i>val</i> to array element	arrayref, idx, <i>val</i> →
NEWARRAY	allocate new array	count → arrayref
NEG	- <i>val</i>	<i>val</i> → result
ADD	<i>val1</i> + <i>val2</i>	<i>val1</i> , <i>val2</i> → result
SUB	<i>val1</i> - <i>val2</i>	<i>val1</i> , <i>val2</i> → result
MUL	<i>val1</i> * <i>val2</i>	<i>val1</i> , <i>val2</i> → result
DIV	<i>val1</i> / <i>val2</i>	<i>val1</i> , <i>val2</i> → result
AND	<i>val1</i> & <i>val2</i>	<i>val1</i> , <i>val2</i> → result
OR	<i>val1</i> <i>val2</i>	<i>val1</i> , <i>val2</i> → result
SWAP	swap top two stack elements	<i>val1</i> , <i>val2</i> → <i>val2</i> , <i>val1</i>
GOTO <i>n</i>	pc = pc + <i>n</i>	
IFZ <i>n</i>	if (<i>val</i> ==0) pc = pc + <i>n</i>	<i>val</i> →
IFNZ <i>n</i>	if (<i>val</i> !=0) pc = pc + <i>n</i>	<i>val</i> →
IFEQ <i>n</i>	if (<i>val1</i> == <i>val2</i>) pc = pc + <i>n</i>	<i>val1</i> , <i>val2</i> →
IFNE <i>n</i>	if (<i>val1</i> != <i>val2</i>) pc = pc + <i>n</i>	<i>val1</i> , <i>val2</i> →
IFLT <i>n</i>	if (<i>val1</i> < <i>val2</i>) pc = pc + <i>n</i>	<i>val1</i> , <i>val2</i> →
IFLE <i>n</i>	if (<i>val1</i> <= <i>val2</i>) pc = pc + <i>n</i>	<i>val1</i> , <i>val2</i> →
IFGT <i>n</i>	if (<i>val1</i> > <i>val2</i>) pc = pc + <i>n</i>	<i>val1</i> , <i>val2</i> →
IFGE <i>n</i>	if (<i>val1</i> >= <i>val2</i>) pc = pc + <i>n</i>	<i>val1</i> , <i>val2</i> →
PRINT	print <i>val</i>	<i>val</i> →
SPRINT <i>n</i>	print the <i>n</i> th string literal	
CALL <i>n</i>	call the <i>n</i> th function caller: callee: args in var[0]..var[<i>k</i> -1]	arg1,...,argk →
RETURN	return from procedure	
VRETURN	return a value from function callee: caller:	<i>val</i> → → <i>val</i>

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 ahead 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.

A Sample SC1 Program

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

"sum.sc1"	"sum.sc1"	
<pre> 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 </pre>	<pre> 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 </pre>	<pre> // 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); } </pre>

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

```

Exercises

Implement the same three functions in SC1. Again, you are provided with the starter version of the three programs.

Hint: 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 an 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

```