CS322 Winter'14 Lab 4 Handout
IRs and IR Code Generation

Learning Objectives

Upon successful completion, students will be able to:

- write simple three-address IR programs;
- implement an IR code generator for simple expressions and statements.

Lab Organization

This lab is divided into two parts. In the first part, you'll try to write some simple programs in an three-address-style IR language; and in the second part, you'll try to write an IR generator for a small AST language.

Part I. IR Programming

You are going to practice programming at IR level with the following IR language:

IR1:

```
Program -> {Func}

Func    -> <Global> VarList [Type] [VarList] "{" {Inst} "}"
VarList -> "(" [<id> Type {"," <id> Type}] ")"
Type    -> ":B" | ":I" | ":P"

Inst -> Dest "=" Src AOP Src                    // Binop
     |  Dest "=" UOP Src                        // Unop
     |  Dest "=" Src                            // Move
     |  Dest "=" Addr Type                      // Load
     |  Addr Type "=" 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

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

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

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

This language is similar to the one shown in this week's lecture, but a little simpler. It has the standard three-address instructions, plus a higher level function call instruction — all arguments to a call are included in the instruction. It also has three explicit types, integer (`:I`), Boolean (`:B`), and pointers (`:P`). Formal parameters and local variables to a function need to be explicitly declared with type information. Both the load and the store instructions are tagged with a type tag, to indicate the type of value to be fetched from or stored into memory. The following is a sample program written in this language:

```
# Return the sum of array elements (Input: array and its size)
#
_sum (a:P, n:I):I        # name, parameters, and return type
(sum:I, i: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]:I            # fetch a[i]
  sum = sum + t3         # add a[i] to sum
  i = i + 1
  goto L0
 L1:
  return sum
}

_main ()
(a:P, s:I)
{
  a = call _malloc(12)   # alloc space for array
  t1 = a                 # initialize array elements
  [t1]:I = 1             # a[0] = 1
  t2 = a + 4
  [t2]:I = 2             # a[1] = 2
  t3 = 2 * 4
  t4 = a + t3
  [t4]:I = 3             # a[2] = 3
  s = call _sum(a, 3)    # call _sum()
  call _print(s)         # print result
  return
}
```

Recall that in Lab 1, you practiced assembly programming with a set of functions. Today, you'll try to write the same set of functions in our IR language. These functions are:

`length(int[] a)` — Return the length of the array `a`.

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

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

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

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

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

In the `prog` sub-directory, you'll see six program files, each contains a driver function and a function for you to complete. Once you complete a program, you can test it with an interpreter by running the `run` script:

```
linux> ./run program.ir
```

Part II. IR Code Generation

In this week's lecture, we discussed IR code generation for simple expressions and statements using two example input and output languages, `AST0` and `IR0`. In this part, you are going to implement an IR code generator using the discussed approach. Here are the two languages' grammars again:

AST0:

```
Program -> {Stmt}

Stmt -> "Assign" Exp Exp
     |  "If" Exp Stmt ["Else" Stmt]
     |  "While" Exp Stmt
     |  "Print" Exp

Exp  -> "(" "Binop" BOP Exp Exp ")"
     |  "(" "Unop" UOP Exp ")"
     |  "(" "NewArray" <IntLit> ")"
     |  "(" "ArrayElm" Exp Exp ")"
     |  <Id>
     |  <IntLit>
     |  <BoolLit>

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

IR0:

```
Program -> {Inst}

Inst -> Dest "=" Src AOP Src
     |  Dest "=" UOP Src
     |  Dest "=" Src
     |  Dest "=" "malloc" "(" Src ")"
     |  Dest "=" "[" Dest "]"
     |  "[" Dest "]" "=" Src
     |  "print" "(" Src ")"
     |  "if" Src ROP Src "goto" <Label>
     |  "goto" <Label>
     |  <Label> ":"

Src  -> <Id> | <Temp> | <IntLit> | <BoolLit>   # Operand forms
Dest -> <Id> | <Temp>                          # LHS target forms

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

Go to the `irgen` sub-directory. You'll see a program file, `IR0Gen.java`. This is the file you'll be working on. Use an editor to open this file, and you'll see the general setup of a IR code generator. The `main` method reads in an AST0 program through an AST0 parser, and invokes the `gen` method on the top-level AST0 node.

The rest of the program is a collection of (overloaded) `gen` methods, one for each type of AST0 nodes. There is a group of `gen` methods for AST0 `Stmt` nodes; each of them returns a list of IR0 instructions (`List<IR0.Inst>`), corresponding to the `Stmt.c` attribute discussed in class. Another group are for AST0 `Exp` nodes; each of those returns a `ValPack` object, which consists of a list of IR0 instructions and an `IR0.Src` object holding the corresponding expression's value. The two components in a `ValPack` object correspond

to the two attributes, `Exp.c` and `Exp.v`, discussed in class. The third group of `gen` methods map AST0 operators to IR0 operators.

As discussed in class, we have a IR code template in the form of attribute definitions for each AST0 node. Now, your task is to use the template information as a guidance, and complete the `gen` method for each AST0 node.

After finishing coding, you can compile and test your `IR0Gen` program with some test `AST0` programs in the `tst` sub-directory. You can further verify the correctness of the generated `IR0` programs with an interpreter by running the `run0` script.