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 1: IR Code Generation (I)

### (Due Thursday 1/28/16 @ 11:59pm)

This assignment is a follow-up to this week's lab (Lab 2). In Lab 2, we have worked on generating IR code for an AST program representation (AST0). In this assignment, we extend the lab by implementing an IR generator for a more powerful AST representation (AST1).

This assignment carries a total of 10 points.

## Preparation

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

`hw1.pdf` — this document

`ast/Ast1.java, ast/<other>.java` — AST1 definition and its parser code

`ir/IR1.java` — IR1 definition

`IR1Interp.jar` — an interpreter for the IR language

`tst` — a set of tests

`Makefile` — for compiling your program

`gen, run` — scripts for testing tests

## The Input and Output Languages

**AST1** is an extension of AST0, which was used in Lab 2. AST1 adds functions, declarations, types, and arrays. The following are descriptions of AST1's key features. For reference, the full grammar specification is included at the end of this document. However, you'll be working with the Java class representation of AST1's nodes, rather than the grammar directly.

- An AST1 program consists of a list of functions. A function consists of a name, a return type, parameter declarations, local variable declarations, and a list of statements.

```
Program -> {Func}
Func    -> "Func" ("void" | Type) <Id> "(" {Param} ")" {VarDecl} {Stmt}
```

- Both parameter and variable declarations consist of a name and a type. A variable declaration may also have an initialization expression.

```
Param   -> "(" "Param" Type <Id> ")"
VarDecl -> "VarDecl" Type <Id> [Exp]
```

- There are two basic types, integer and Boolean, and an array type constructor for building array types on top of other types. Array of arrays is allowed.

```
Type    -> "IntType" | "BoolType" | "(" "ArrayType" Type ")"
```

- There are two array-related expression forms, one for creating an array object and one for referencing an array element.

```
Exp     -> "(" "NewArray" Type <IntLit> ")"
        |  "(" "ArrayElm" Exp Exp ")"
```

- The assignment statement takes a more general form than the one in AST0. The LHS can be either an `Id` or an `ArrayElm`.

```
Stmt    -> "Assign" Exp Exp
```

- Function calls come in two forms, one with a return value, one without; hence one belongs to the `Exp` family, and the other to the `Stmt` family. Both forms take a list of expressions as arguments. There is also a return statement for returning from a function (with or without a value).

```
Exp     -> "Call" <Id> "(" {Exp} ")"
Stmt    -> "CallStmt" <Id> "(" {Exp} ")"
        |  "Return" [Exp]
```

- Every function should ends with a return instruction, even if there isn't one in the AST.
- String literal is technically defined as a member of the `Exp` family, but it is only allowed as an argument to the print statement; it cannot appear anywhere else.

```
Exp     -> <StrLit>
```

**IR1** is an extension of IR0 (ref. Lab 2). IR1's new features correspond to AST1's, except that it does not track variable's type. (This is to simply the task of IR generation. Proper treatment of types will be dealt with in a later assignment.) As a consequence, Boolean values are printed out as integer 0s and 1s. The following are IR1's main features. It's full grammar can be found at the end of this document.

- A program is a list of functions, and a function contains the usual information:

```
Program -> {Func}
Func    -> <Global> VarList [VarList] "{" {Inst} "}" <EOL>
<Global> = _[A-Za-z][A-Za-z0-9]*
```

  Function name is in the form of a `<Global>`, which is an alphanumeric sequence preceded by an underscore `"_"`.

- Both parameter and local variable declarations take the same `VarList` form, which is simply a list of `Id`s: (Recall that IR1 does not track variable's type.)

```
VarList -> "(" [<Id> {"," <Id>}] ")" <EOL>
```

- IR1 extends IR0's instruction set with two new instructions:

```
Inst    -> [Dest "="] "call" <Global> "(" {Src} ")" <EOL>
        | "return" [Src] <EOL>
```

  The call instruction can be in two forms, one with a `Dest` target for receiving a return value, and one without. Internally, they are represented by the same class, with an optional destination field.

- IR1's operand form is extended to include string literals to support strings in the print instruction:

```
Src     -> <Id> | <Temp> | <IntLit> | <BoolLit> | <StrLit>
```

- There are three 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)
```

  For memory allocation, we assume an integer (or a Boolean) has a size of 4. (In IR1, memory size is an abstract value; it is not defined in terms of physical units, such as byte.)

## Your Task: IR Generation

Implement an IR generator to translate AST1 programs to IR1 programs. Call your program `IR1Gen.java`. You may use your `IR0Gen.java` program of Lab 2 as a starter version for this assignment. Copy that program over, and change its contents to match AST1's and IR1's class names. (If you haven't finished `IR0Gen.java`, it might be a good idea to do that first, since it is a simpler version.)

The IR generator uses the syntax-directed translation approach. It traverses an AST tree, and generate IR code for each each node using local information and the recursive results from its children. You need to get familiar with the class definitions in both `ast/Ast1.java` and `ir/IR1.java`. More specifically, you need to know the field names of AST1 classes in order to process them, and you need to know the IR1 class constructors in order to create IR1 program objects.

The following are some hints for dealing with AST1 nodes that do not appear in AST0.

- *Program and functions.* IR generation for these nodes is mostly straightforward. There are corresponding IR1 constructs for both. There is only one issue. IR1 requires that every function ends with a return statement, so if an AST1 function does not have a return statement, `IR1Gen` needs to insert one at the end of the instruction list. (*Hint:* For AST1 functions with a non-void return type, you may assume there is always already a return statement (otherwise it would be a type error); for AST1 functions with a void return type, you may just insert a return statement without checking, since an extraneous return statement at the end does no harm.)
- *Variable declarations.* IR1's version of variable declaration does not include type nor initialization expression. Type can be simply ignored, but initialization must be handled. For an AST1 `VarDecl` node with initialization, `IR1Gen` should perform the following work (in addition to generating a corresponding `Var` node): (1) generate instructions to evaluate the initialization expression; (2) generate a move instruction to assign the expression's value to the variable in the `VarDecl`; (3) insert these instructions at the top of the instruction list for the corresponding function's body. Here is an example:

```
# AST1 Program
Func void main ()
 VarDecl IntType i (Binop * 2 4)
 Print  i
```

```
# IR1 Program
_main ()          # empty param list
(i)               # local var list
{
 t1 = 2 * 4       # inserted inst
 i = t1           # inserted inst
 printInt (i)
}
```

- *Assignment's LHS expressions.* The LHS and RHS of an assignment need to be treated differently in IR generation. For the LHS expression, `IR1Gen` needs to generate its location, rather than its value. Note that the overloaded `gen` routines for all expression nodes are defined for generating expressions' values. They are suitable for generating code for the RHS expression, but not for the LHS expression. You need to figure out a way to handle this situation. However, since there are only two expression forms that can appear as LHS, `Id` and `ArrayElm`, you only need to handle this situation for the `ArrayElm` node. (*Hint:* You may implement a `genAddr` routine for generating code for `ArrayElm`'s location, or use the `gen` routine, but remove some instruction(s) from the instruction list it generates.)
- *Malloc and Print system calls.* In IR0, these two system calls are treated as two individual instructions. In IR1, they are handled by the general call instructions. If you have implemented `IR0gen.java`, modify the corresponding code to use the `IR1.Call` instruction. Note that `IR1.Call` takes a *list* of arguments.

## Requirements and Grading

The IR1 programs in `.ir.ref` files are for reference purpose only. Your program's output *do not* need to match them. However, your IR1 programs should run successfully with the provided interpreter and generate matching output to those in `.out.ref` files.

This assignment will be graded mostly on your `IR1Gen` program's correctness. We may use additional programs to test. The minimum requirement for receiving a non-F grade is that your `IR1Gen.java` program compiles without error, and it generates validate IR1 code for at least one simple AST1 program.

## Running and Testing

The provided IR1 interpreter works only with Java 1.8. You should check to make sure that you have Java 1.8 in your environment by running:

```
linux> java -version
```

If not, you could add it in by running `addpkg` (and select `Java8`).

## What to Turn in

Submit a single file, `IR1Gen.java`, through the "Dropbox" on the D2L class website.

**AST1 and IR1 Grammars:**

"AST1"

```
Program -> {Func}
Func    -> "Func" ("void"|Type) <Id> "(" {Param} ")" {VarDecl} {Stmt}
Param   -> "(" "Param" Type <Id> ")"
VarDecl -> "VarDecl" Type <Id> [Exp]         // Exp could be null
Type    -> "IntType"
        |  "BoolType"
        |  "(" "ArrayType" Type ")"
Stmt    -> "{" {Stmt} "}"
        |  "Assign" Exp Exp
        |  "CallStmt" <Id> "(" {Exp} ")"
        |  "If" Exp Stmt ["Else" Stmt]
        |  "While" Exp Stmt
        |  "Print" Exp                       // Exp could be null or StrLit
        |  "Return" [Exp]                    // Exp could be null
Exp     -> "(" "Binop" BOP Exp Exp ")"
        |  "(" "Unop" UOP Exp ")"
        |  "(" "Call" <Id> "(" {Exp} ")" ")"
        |  "(" "NewArray" Type <IntLit> ")"
        |  "(" "ArrayElm" Exp Exp ")"
        |  <Id> | <IntLit> | <BoolLit> | <StrLit>
        |  "()"                              // null
BOP     -> "+"  | "-"  | "*" | "/"  | "&&" | "||"
        |  "==" | "!=" | "<" | "<=" | ">"  | ">="
UOP     -> "-" | "!"
```

"IR1"

```
Program -> {Func}
Func    -> <Global> VarList [VarList]            // Name, Params, Locals
           "{" {Inst} "}" <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
           ) <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]+
```