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 3: IR Code Generation

(Due Thursday 2/26/15 @ 11:59pm)

In this assignment, you are going to implement part of an IR code generator. The focuses are on classes and objects, methods and invocations, declarations and types.

The assignment carries a total of 100 points.

Preparation

Download and unzip the file `hw3.zip`. You'll see a `hw3` directory with the following items:

- `hw3.pdf` — this document
- `IRGen0.java` — a starter version of the code-gen program
- `ast/` — contains the AST definition file `Ast.java` and files of an AST parser
- `ir/` — contains the IR definition file `IR.java`
- `tst/` — contains a set of test programs
- `Makefile` — for building the code-gen
- `gen, run` — scripts for generating and running IR programs

Check to make sure that you have Java 1.8 in your environment (use "`java -version`"). If not, you should add it in by running `addpkg` (and select `java8`).

The Input Language

The input language to the code generator is the miniJava AST language. However, we are not implementing every aspect of the language in this assignment. In particular, we are skipping arrays and all operation forms of expressions. Furthermore, we will skip new object initialization (*i.e.* we are not going to generate the `init` routines). The following is the portion of the AST language that is relevant to this assignment:

```
Program    -> {ClassDecl}
ClassDecl  -> "ClassDecl" <Id> [<Id>] {VarDecl} {MethodDecl}
VarDecl    -> "VarDecl" Type <Id> Exp
MethodDecl -> "MethodDecl" Type <Id> "(" {Param} ")" {VarDecl} {Stmt}
Param      -> "(" Type <Id> ")"
Type -> "void" | "IntType" | "BoolType" | "(" "ObjType" <Id> ")"

Stmt -> "{" {Stmt} "}"
     |  "Assign" Exp Exp
     |  "CallStmt" Exp <Id> "(" {Exp} ")"
     |  "If" Exp Stmt [ "Else" Stmt ]
     |  "While" Exp Stmt
     |  "Print" (Exp | <StrLit>)
     |  "Return" (Exp | "(" ")")

Exp  -> "(" "Call" Exp <Id> "(" {Exp} ")" ")"
     |  "(" "NewObj" <Id> ")"
     |  "(" "Field" Exp <Id> ")"
     |  "This"
     |  <Id> | <IntLit> | <BoolLit>
```

The code generator program will work on the AST language's internal representation, where every AST node is represented by a Java class. These class definitions are in the file `ast/Ast.java`.

The Target Language

The target language is the full version of the register-machine IR languages that we've been using in labs and previous homeworks. This language is simply called IR. Its internal Java class representation can be found in the file `ir/IR.java`. Here is an itemized description of IR's features.

- A program consists of a set of data sections followed by a set of function definitions:

```
Program -> {Data} {Func}
```

- A data section has a name (represented by a `<Global>` label), a size, and a list of data items, which are also in the form of `<Global>` labels:

```
Data -> "data" <Global> "(" "sz=" <IntLit> ")" ":" [<Global> {"," <Global>}]
```

  For our input language, data sections are used to represent class descriptors. Each class in an input program has a corresponding data section in the output IR program. (Strictly speaking, the static class containing "main" does not need a class descriptor, but we include it anyhow for simplicity.) The data section's content is the class's method table (*i.e.* the v-table) — those `<Global>` labels are methods' labels. Since miniJava does not support static fields or methods, there is no other data in a class descriptor.

- A function definition contains the usual attributes, a name, parameter and local variable lists (both represented by `VarList`, the latter could be omitted), and a body (represented by a set of `Insts`):

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

  Note that there is no type information present in either the function declaration or the variable lists.

- IR has the usual set of instructions:

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

CallTgt -> <Global> | <Id> | <Temp>
ArgList -> "(" [Src {"," Src}] ")"
```

  The Call instruction is the most complex. It allows all of its arguments to be included in the instruction; it supports both direct and indirect (with the `"*"` on) calls; and it supports the return of a value if needed.

- IR supports three data types: Boolean (`:B`), integer (`:I`), and address pointer (`:P`):

```
Type -> ":B" | ":I" | ":P"
```

  Their corresponding sizes are 1, 4, and 8 bytes, respectively.

  Both the Load and the Store instructions include a type tag, which represents the data type of the item being loaded or stored.

- IR's operand forms and operators are shown below:

```
Addr -> [<IntLit>] "[" Dest "]"
Src  -> <Id> | <Temp> | <Global> | <IntLit> | <BoolLit> | <StrLit>
Dest -> <Id> | <Temp>
BOP -> AOP | ROP
AOP -> "+" | "-" | "*" | "/" | "&&" | "||"
ROP -> "==" | "!=" | "<" | "<=" | ">" | >="
UOP -> "-" | "!"
```

  Even though we are not implementing AST expressions with operators, there are needs to use IR operators to support the code-gen for some AST nodes.

- IR's labels, globals, and temps are defined below:

```
<Label>  = <Id>
<Global> = "_"<Id>
<Temp>   = "t"<IntLit>
```

  The rest of tokens, *i.e.*, `<IntLit>`, `<BoolLit>`, `<StrLit>`, and `<Id>` are defined as usual.

- Finally, there are four pre-defined functions:

```
_malloc(size), _printInt(arg), _printBool(arg), _printStr(str)
```

The Code-Gen Program Structure

A starter version of the code-gen program is provided to you in `IRGen0.java`. Within the overall `IRGen` class declaration, there are two major parts. The first part consists of definitions of several data structures and several utility routines, which are created to support information access and code generation. The second part consists of the collection of `gen` routines for individual AST nodes.

The Data Structures

- **ClassInfo** — This data structure is for keeping all useful information about a class declaration. These information are needed later by many `gen` routines.

```
static class ClassInfo {
  Ast.ClassDecl cdecl;      // classDecl AST
  ClassInfo parent;         // pointer to parent
  List<String> vtable;      // method-label table
  List<Ast.VarDecl> fdecls; // field decls (incl. inherited ones)
  List<Integer> offsets;    // field offsets
  int objSize;              // object size
}
```

  A set of utility routines are defined in this class for assisting accessing information in the data structure.

- **CodePack/AddrPack** — For returning `<type,src,code>` and `<type,addr,code>` tuples from `gen` and `genAddr` routines. Note that we expended these packs from Lab 3's version to include a type component. The type information is needed in the Load and Store instructions.

```
static class CodePack {
  IR.Type type;
  IR.Src src;
  List<IR.Inst> code;
}
```

```
static class AddrPack {
  IR.Type type;
  IR.Addr addr;
  List<IR.Inst> code;
 }
```

- **Env** — This data structure is for keeping track of local variables and parameters and for finding their types.

```
private static class Env extends HashMap<String, Ast.Type> {}
```

  For every method, an instance of `Env` is created and maintained. The parameters and local variables' type information are entered into the environment when their declarations are processed.

The Utility Routines

- **topoSort()** — This routine is for sorting `ClassDecls` based on parent-children relationship.
- **getClassInfo()** — This routine returns an object's base `ClassInfo`. When processing method calls or field accesses, there is a need to find the object component's class information. This routine can be invoked on the expression representing an object to get its base class's `ClassInfo` record.
- **createClassInfo()** — This routine creates a new `ClassInfo` record.

The Individual Code-Gen Routines

The code-gen program follows the standard syntax-directed translation scheme. The `main` method reads in an AST program through an AST parser; it then invokes the `gen` routine on the top-level `Ast.Program` node, which in turn, calls other `gen` routines.

In additional to generating code, the top-level `gen(Ast.Program n)` routine also sets up the global data structures. It processes the `ClassDecl` list in three passes:

1. It sorts `ClassDecl`s with a topological sort, so that a base class's decl always appear before its sub classes' decls. (Recall that this is to enable object size and instance variable offset calculation.)
2. It collects information from each `ClassDecl` and stores it in an `ClassInfo` record. This step is necessary to precede the actual code-gen, because cross-class references may exist in a miniJava program. The code-gen needs to have all `ClassInfo` records ready before any sub-level `gen` routine gets called.
3. This is the actual code-gen pass. In this pass, a recursive `gen` routine call will be invoked on each `ClassDecl`. The return results will be collected in two lists: a list of data sections and a list of functions. With them, an `IR.Program` node can be constructed.

Your Task

Your task is to complete the implementation of all `gen` routines and one utility routine (`createClassInfo()`. In the provided `IRGen0.java` file, you will find code-gen guideline for each individual AST node. Read these guidelines carefully. Make sure you understand them before writing program code.

Copy `IRGen0.java` to `IRGen.java` and edit the new program. This way, you'll have the starter version available for reference.

Requirements, Grading, and What to Turn In

This assignment will be graded mostly on your program's handling of the AST nodes. The point distribution is roughly 1/3 on declaration nodes, 1/3 on call handling, and 1/3 on the rest of statement nodes. Partial credits will be given if you don't have perfect implementation for all parts.

The provided test programs can be used to validate your program for individual cases. As usual, we may use additional programs to test.

The minimum requirement for receiving a non-F grade is that your `IRGen.java` program compiles without error, and it generates validate IR code for at least one simple AST program.

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