CS322 Spring'14 Assignment 2: IR Code Generation

(Due Tuesday 5/13/14 @ 11:59pm)

This assignment is a follow-up to this week's lab (Lab 4). The lab focused on generating IR code for simple expressions and statements. In this assignment, you will continue working on IR code generation, but for other language features, including classes and objects, methods and invocations, declarations and types.

1 The Input Language

The input language is the same miniJava language you work on last term, although for this IR code-gen assignment, we'll use its AST representation. The full program representation of the AST is in the file `ast/Ast.java`. The following is its grammar:

```
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> ")"
     |  "(" "ArrayType" Type ")"              //**

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

Exp -> "(" "Binop" BOP Exp Exp ")"            //**
    |  "(" "Unop" UOP Exp ")"                 //**
    |  "(" "Call" Exp <Id> "(" {Exp} ")" ")"
    |  "(" "NewObj" <Id> "(" {Exp} ")" ")"
    |  "(" "NewArray" Type <IntLit> ")"       //**
    |  "(" "Field" Exp <Id> ")"
    |  "(" "ArrayElm" Exp Exp ")"             //**
    | "This"
    | <Id>
    | <IntLit>
    | <BoolLit>
    | <StrLit>
```

We are not implementing every aspect of the miniJava language in this assignment. In particular, we are skipping arrays and all operation forms of expressions. The AST nodes that are marked with `//**` in the above grammar listing do not need to be handled. Furthermore, we will skip new object initialization, *i.e.* we are not going to generate the `init` routines.

2 The Target Language

The target IR language is a register-machine based language named IR. It's an extension to the IR1 language used in Lab 3, with added support for static data sections and indirect function calls. Here is an itemized description of IR's features.

- A program consists of a list of global data sections and a list of function definitions.

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

- A data section has a global label (its name), an integer (its size), and a list of global labels.

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

  In our code-gen, data sections are used to represent class descriptors. Each class in an input program has a corresponding data section in the output IR program (except for the static class containing the "main" method). The list of global labels inside the data section is the class's v-table, which contains the class's method labels. (Since miniJava does not support static fields or methods, there is no other data in a data section.)

- A function definition contains the usual attributes, a global label (its name), parameter and local variable lists (the latter may be omitted if it's empty), and a list of instructions (its body). Note that there is no type information present in either the function or the variable declarations.

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

- 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"
```

  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 | BOP
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 `<IntLit>`, `<BoolLit>`, `StrLit`, and `<Id>` tokens are defined as usual. The rest are defined below:

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

- Finally, there are five pre-defined functions:

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

3 The Code-Gen Program Structure

A starting version of the code-gen program is provided to you (in `IRGen0.java`). The overall structure of the program is an `IRGen` class. Within it, there are two major parts. The first part consists of the definition of a major data structure, `ClassInfo`, which is used for storing information about class declarations, to be used in many places in the code-gen. It also contains definitions of a few smaller data structures for various needs, *e.g.* for carrying results back from a `gen` routine. The second part of the program consists of the collection of `gen/genAddr` routines for AST nodes.

3.1 The Class Information Data Structure

This data structure is defined in the form a nested class. It is for keeping all useful information about a class declaration to be used later in the `gen` routines.

```
static class ClassInfo {
  String name;                        // class name
  ClassInfo parent;                   // ptr to parent's record
  boolean isMainClass;                // true if class contains "main"
  Ast.ClassDecl classDecl;            // class source ast
  ArrayList<String> vtable;           // (virtual) method table
  HashMap<String,Integer> offsets;    // field variable offsets
  int objSize;                        // object size
}
```

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

3.2 Other Data Structures

A few other data structures are also defined.

- CodePack and AddrPack — For returning `<type,src,code>` and `<type,addr,code>` tuples from `gen` and `genAddr` routines. Note that we expended these packs from Lab 4's version to include a type component.

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

```
static class AddrPack {
  Ast.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> {}
```

3.3 The Top-Level Code-Gen Routine

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 tern, calls other `gen` routines. This top-level `gen(Ast.Program n)` routine also performs other tasks. In fact, 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.

3.4 The Rest of the Code-Gen Routines

The rest of code-gen routines are for other AST nodes. In the `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.

4 Your Task

Your task is to complete the implementation of all `gen` routines in this code-gen program. To start, download and unzip the file `hw2.zip`. You'll see a `hw2` directory with the following items:

- `ast/` — contains the AST definition file `Ast.java` and an AST parser `astParser.java`
- `ir/` — contains the IR definition file `IR.java`
- `tst/` — contains a set of AST programs and their corresponding reference IR code
- `IRGen0.java` — a starting version of the code-gen program
- `Makefile` — for building the code-gen
- `genir` — a script for running your `IRGen` program
- `runir` — a script for interpreting IR programs

Copy `IRGen0.java` to `IRGen.java` and edit the new program. This way, you'll always have the starting version available for reference. If you plan to use the CS Linux Lab machines for this assignment, check to make sure that you have Java 1.7 in your environment (use "`java -version`"). If not, you should add it in by running `addpkg` (and select `Java7`).

5 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. The more nodes your program handles correctly, the more points you'll receive. The provided test programs can be used to validate your program for individual cases. But by no means they provide a comprehensive coverage on all language features this code-gen program implements. 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.