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 2: IR Code Generation (II)

(Due Thursday 2/11/16 @ 11:59pm)

In this assignment, you are going to implement a second IR code generator. This time, the focuses are on (1) classes and objects, and (2) methods and instance variables. The assignment carries a total of 10 points.

Download and unzip the file `hw2.zip`. You'll see a `hw2` directory with the following items:

`hw2.pdf` — this document

`ast/Ast.java, ast/<other>.java` — AST definition and its parser code

`ir/IR1.java` — IR definition

`IRGen0.java` — a starter version of the code-gen program

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

`tst/` — contains a set of tests

`Makefile` — for compiling your program

`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 full version of the miniJava AST language. However, we are skipping arrays and all operation forms of expressions. 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
     |  "Return" [Exp]

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

Furthermore, we make the following simplification assumptions:

- No static data or methods other than the `main` method.
- Methods are implemented with static binding. (Hence there is no need to create class descriptors in IR code.)
- No init routines for new objects. (Hence class fields' init values in source program are ignored.)
- In source program, base classes are defined before their subclasses. (Hence a simple sequential processing of class decls is sufficient.)

As in Assignment 1, the IR code-gen 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 simply called IR. It is an extension to the IR1 language used in Assignment 1. IR has the capacity to handle all the features of our source language, miniJava. Here are its highlights. (The features that are not needed for this assignment are noted.)

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

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

  The data sections are for holding static class information, such as the vtables. (*Note:* They are not needed for this assignment.)

- Function definitions and variable declarations are exactly the same as in IR1:

```
Func    -> <Global> VarList [VarList] "{" {Inst} "}"
VarList -> "(" [<id> {"," <id>}] ")"
<Global> = _[A-Za-z][A-Za-z0-9]*
```

- 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}] ")"
```

  Three instructions differ from the IR1's version: load, store, and call. The load and store instructions each have an extra `Type` tag, which represents the data type of the item being loaded or stored. The call instruction supports an extra indirect form with the `"*"` flag, which allows the `CallTgt` be stored in a variable or a temp. (*Note:* This form is not needed for this assignment.)

- 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, in abstract memory units.

- There is a new `_printBool()` built-in function. Boolean values need to be printed with this function, instead of with `_printInt()`.

IR's Java class representation can be found in the file `ir/IR.java`.

The Code-Gen Program Structure

A starter version of the IR 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, global variables, and 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.

Supporting Data Structures and Utilities

- **ClassInfo** — This data structure is for keeping useful information about a class declaration in one place. One instance of this class is created for each `Ast.ClassDecl` node in the input program.

```
static class ClassInfo {
  String name;                       // class name
  ClassInfo parent;                  // pointer to parent's record
  Ast.ClassDecl classDecl;           // class source AST
  HashMap<String,Integer> offsets;   // instance variable offsets
  int objSize;                       // class object size
}
```

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

  - `methodBaseClass()` — returns a method's base class record
  - `methodType()` — returns a method's return type
  - `fieldType()` — returns a field's type
  - `fieldOffset()` — returns a field's offset in object storage

- **CodePack** — For packing return items from expressions' `gen` routines.

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

  Note that we extended this class from Assignment 1's version to include a type component. The type information is needed in the load and store instructions.

- **Env** — A mapping structure for keeping track of local variables and parameters and their types.

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

  For each `Ast.MethodDecl` node, 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. This environment is then passed as an argument to all gen routines.

- **classEnv** — A global variable for keeping the whole collection of ClassInfo records.

```
static HashMap<String, ClassInfo> classEnv = new HashMap<String, ClassInfo>();
```

- **thisObj** — A global variable for representing the "current" object.

```
static IR.Id thisObj = new IR.Id("obj");
```

- **getClassInfo()** — This is an utility routine. 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 an `Ast.Exp` node representing an object to get its base class's `ClassInfo` record.
- **gen(Ast.Type)** — An utility routine for mapping `Ast.Type` to `IR.Type`.

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 needs to set up the global data structures. It processes the `ClassDecl` list in two passes:

1. It collects information from each `ClassDecl` and stores it in an `ClassInfo` record. All `ClassInfo` records need to be ready before any lower level `gen` routine gets called, because cross-class references may exist in a miniJava program.
2. This is the actual code-gen pass. In this pass, the `gen` routine is invoked on each `ClassDecl`. The return results are merged into a single list of functions, and an `IR.Program` node is constructed.

Your Task

Your task is to complete the implementation of all `gen` routines, including the setup of all data structures. 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 actual code.

Requirements and Grading

As usual, you can test your `IRGen` program with the provided test files using the `gen` and `run` scripts. (*Note:* The provided IR interpreter works only with Java 1.8.)

The IR programs in `.ir.ref` files are for reference purpose only. Your program's output *do not* need to match them exactly. However, it is possible (and not too difficult) to generate matching IR programs for this assignment. (This is a contrast to Assignment 1.) Regardless whether your your IR programs match the `ref` files or not, they 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 `IRGen` program's correctness. 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.

What to Turn in

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