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 2: IR Code Optimization

(Due 2/12/15 @ 11:59pm)

In this assignment, you are going to implement the three optimizations that we studied in class: *constant folding*, *address optimization*, and *comparison embedding*. A baseline version of an IR code generator is provided as the implementation platform.

The assignment carries a total of 100 points.

Preparation

Download the zip file `"hw2.zip"` from D2L. After unzipping, you should see a `hw2` directory with the following items:

- `hw2.pdf` — this document
- `IR0Gen.java` — the baseline IR code generator
- `ast0/` — a directory containing the AST representation and its parser
- `ir0/` — a directory containing the target IR representation
- `tst/` — a subdirectory containing a set of test programs
- `IR0Interp.jar` — an IR0 interpreter
- `Makefile` — for compiling programs
- `gen`, `run` — scripts for testing programs

Copy the baseline program `IR0Gen.java` to a new file, `IR0GenOpt.java`, and implement the optimizations in this new program. (Don't forget to change the class name in the file accordingly.)

1. Constant Folding (50 points)

When seeing an AST expression `"(Binop + 2 4)"`, the baseline code generator faithfully generates an IR0 instruction `"t1 = 2 + 4"`.

Constant folding is to have the code generator evaluate constant expressions to their values, and/or to use the constant information to simplify the IR code. For the above example, the code generator would generate the instruction `"t1 = 6"`, instead.

Constant folding can appear in many different forms and can be used to simplify IR code for many different AST nodes. The following are some areas you should pay attention to. The test suite include more examples you can use to improve your implementation.

- Constants may appear at multiple levels in an expression. Your program need to perform the optimization recursively on an expression AST tree from bottom up, so that cases such as the following can be recognized:

```
(Binop + (Binop * 2 3) (Binop - 4 1)) => (Binop + 6 3) => 9
```

- Constant folding applies to all types of constants and operations. Here are some examples:

```
(Binop < 1 2)              => true
(Binop == (Binop + 1 2) 3) => (Binop == 3 3) => true
(Binop || true false)      => true
```

- In Boolean expression case, due to short-circuit semantics, constant folding can simplify expressions that contain non-constant components:

  ```
  (Binop || true x)              => true
  (Binop && false (Binop + x y)) => false
  (Binop && x false)             => (simpler IR code)
  ```

  The last example is an challenging case. It shows that even if a constant appears as the second operand, the code generator can use the information to simply the IR code:

  ```
  "if false == false goto L0" becomes "goto L0"
  ```

  (See `test10.ir.base` and `test10.ir.opt` for this difference.)

- Constant folding can also simplify If and While statements:

  ```
  If true Print 1                => Print 1
  If false Print 2 Else Print 3 => Print 3
  If (Unop ! true) Print 4      => (empty!)
  While false Print 1           => (empty!)
  ```

Advice: Not all cases of constant folding are equally easy to implement. You should start with the simple ones, *e.g.* arithmetic cases, and incrementally progress to more challenging cases.

2. Address Optimization (25 points)

The baseline code generator uses a simple template to generate addresses for `Load` and `Store` instructions. It does not use the offset feature of the `Ir0.Addr` instruction at all. Consider the following example,

```
# AST0 Program                   # IR0 Program (baseline)
Assign a (NewArray 2)            t1 = malloc (8)
Assign (ArrayElm a 0) 3          a = t1
Assign (ArrayElm a 1) 4          t2 = 0 * 4
                                 t3 = a + t2
                                 [t3] = 3
                                 t4 = 1 * 4
                                 t5 = a + t4
                                 [t5] = 4
```

What we'd like to have as output is the following IR code:

```
# IR0 Program (optimized)
t1 = malloc (8)
a = t1
[a] = 3
4[a] = 4
```

This is what we call *address optimization.* The implementation of this optimization is quite simple. When generating `IR0.Addr` in the `gen` routine for the `ArrayElm` node, the code generator checks to see if the `idx` component is an integer literal; if so, it can take advantage of the offset form of `IR0.Addr`.

Note that in order to handle implicit constant `idx` components, such as

```
(ArrayElm a (Binop + 1 1))
```

constant folding optimization must be performed first.

3. Comparison Embedding (25 points)

For bpth the `If` and `While` statements, the IR code generated by the baseline generator evaluates the `cond` expression first; it then compares the value against `false` to decide whether to jump or to fall through.

Consider the following example:

```
# Ast0:                     # IR0 Program (baseline)
If (Binop > n 0)            t1 = n > 0
 Assign x 1                 if t1 == false goto L0
                            x = 1
                            L0:
```

The IR code is fine funtion-wise. However, a more efficient version exists:

```
# IR0 Program (optimized)
if n <= 0 goto L0
x = 1
L0:
```

In this version, the comparison operation is embedded directly inside the `CJump` instruction.

This optimization is called *comparison embedding.*

To implement this optimization, the code generator needs to make changes in the `gen` routines for `Ast0.If` and `Ast0.While`. The idea is to perform a simple look-ahead on the `cond` expression before recursively invoking `gen` on it. If `cond` happens to be a relational expression, then the code generator will invoke `gen` on the expression's operands instead, and embed the relational operation directly into a `CJump` instruction.

Compiling and Testing

You may use the `Makefile` to compile your `IR0GenOpt.java` program, and use the `gen` script to run it on the test AST inputs:

```
linux> make
linux> ./gen tst/test*.ast
```

The `gen` script will compare your IR code with a reference version in the corresponding `.ir.opt` file. Note that even if you have implement everything correctly, the comparison may show differences in label numbers and in instruction order. These differences are fine.

Requirements, Grading, and What to Turn In

Two sets of reference IR code are provided, the baseline version (in `.ir.base` files) and the optimized version (in `.ir.opt` files). You should set your goal to match the code quality in the optimized version, but any improvements over the baseline version will earn you partial credit. The more cases your program can handle, the more points you will get.

The **minimum requirement** for receiving a non-F grade on this assignment is that your `IR0GenOpt.java` program compiles without error and performs correct optimization on at least one of the test programs.

Submit a single file, `IR0GenOpt.java`, through the "Dropbox" on the D2L class website.