CS 322 Homework 3 – due 1:30 p.m., Tuesday, May 29, 2012 [NOTE NEW DEADLINE!]

(Copyright, Andrew Tolmach 2003-2012. All rights reserved.)

Generating Intermediate Code for fab

Working individually or in teams of two, write an intermediate code generator for (a subset of) the **fab** language. As before, *exclude* coverage of anything to do with real numbers; your generator can do anything it wants with programs that use real number features. You may assume that the intermediate language's notion of integers, their operations, and how to read them directly matches **fab**.

There is one small change in the **fab** language compared to the Reference Manual. The manual says that all the expressions in a `write` statement should be evaluated before any of them are printed. You should instead implement the statement so that each expression is evaluated and then printed immediately, before the next is evaluated. In fact, the reference interpreter from assignment 1 already implemented this (technically incorrect) behavior. The only observable difference this new semantics can make is that more printing might occur before the program issues a runtime error or goes into an infinite loop.

The intermediate code representation you will generate is a 3-address code with broad similarities to x86 assembly code, but with major simplifications, particularly in the treatment of variable addressing (addressing by name is supported), operand types, and function call. Details of the intermediate code are given below.

The code generator should read the `.fab` file from standard input, using the existing front end code (which has some changes for this assignment) to perform lexical analysis, parsing, and checking. The generator itself can assume that the AST being compiled has successfully passed the checker. The generator should produce an internal representation of the intermediate code using constructor methods of the provided `IR` class. The resulting code can be printed to standard output, which is normally captured into an `.ir` file. It can also be directly executed using the provided interpreter, as described below.

Runtime errors such as array bounds violation should be handled by generating code to call the appropriate built-in system error call, e.g. `bounds_error`, as detailed below.

As usual, a "correct" generator is available in the jar file `ir.jar`; it can be run on file `foo.fab` to produce an output file `foo.ir` by typing

```
java -classpath frontend.jar:ir.jar IRGenDriver foo.fab foo.ir
```

In addition, an interpreter that executes IR is available in the same jar file; to run it on the "official" IR resulting from file `foo.fab`, you can type

```
java -classpath frontend.jar:ir.jar IRInterpDriver foo.fab
```

The interpreter takes user input from standard input and writes program output to standard output in the usual way. It is not necessary that the code you generate be identical to what my `IRgen` generates, but it must behave the same way as my code does when fed to `IRinterp`.

Intermediate Code Representation

The intermediate code representation (IR) consists of a sequence of top-level function declarations. Each function carries a list of formal parameters, a list of local variables, and a sequence of instructions. One of the functions (with no parameters or free variables) must be called "`$MAIN`"; it serves as the entry point to the program. When `$MAIN` terminates, it returns the overall status value of the program; this should be 0 for programs that terminate successfully.

We can think of the IR as operating on an abstract machine with the following state components:

- A global list of top-level function definitions $f_1, \ldots, f_n$.

- Named variables $nm_1, \ldots, nm_n$ refering to the parameters and locals declared for the current function.
- An unbounded number of temporaries `$T0`, `$T1`, ... local to the current function.
- A global byte-addressed memory referenced by locations *loc*.
- An unbounded number of outgoing-argument registers `$A<`c`>0`,`$A<`c`>1`,... holding arguments about to be passed to a function in the call labelled c.
- A single register `$RET` holding a function return value just before return. From the caller's perspective, the return register is tagged with its associated call number and looks like `$RET<`c`>`.

The IR manipulates typed values; its three types are `BOOL`, `INT`, and `PTR` (which includes locations, function names, and strings). For the most part, the IR treats all these as having uniform size; in particular, named variables, temporaries, and other registers can all contain any kind of value. However, when values are stored in memory, they occupy different numbers of bytes depending on their types: 1 byte for `BOOL`, 4 for `INT`, and 8 for `PTR`. In addition, values stored in memory must be aligned to addresses divisible by their size (e.g, an `INT` must be stored on a 4-byte boundary).

We can describe the IR instructions using the following grammar which matches the pretty-printed generated by the IR classes. But note that there is no parser provided; IR is built by calling node constructors directly; the mapping between grammar and constructors should be fairly obvious. Beware: the `IR` constructors do a certain amount of sanity checking, and `IRInterp` incorporates many dynamic checks, but there is no systematic static checker; it is quite possible to construct (and probably even interpret) nonsensical IR programs.

The instruction set:

`mov[B|I|P]` *src*`,` *dst*	move value *src* to *dst*
`addI` *src*$_1$`,` *src*$_2$`,` *dst*	*dst* := *src*$_1$ + *src*$_2$ (integer addition)
`addP` *src*$_1$`,` *src*$_2$`,` *dst*	*dst* [P] := *src*$_1$ [P] + *src*$_2$ [I] (pointer addition)
`subI` *src*$_1$`,` *src*$_2$`,` *dst*	*dst* := *src*$_1$ - *src*$_2$ (integer subtraction)
`mulI` *src*$_1$`,` *src*$_2$`,` *dst*	*dst* := *src*$_1$ * *src*$_2$ (integer multiplication)
`divI` *src*$_1$`,` *src*$_2$`,` *dst*	*dst* := *src*$_1$ / *src*$_2$ (integer division)
`modI` *src*$_1$`,` *src*$_2$`,` *dst*	*dst* := *src*$_1$ % *src*$_2$ (integer mod)
`ld[B|I|P]` *addr*`,` *dst*	load value from memory at address *addr* into *dst*
`st[B|I|P]` *src*`,` *addr*	store value from *src* into memory at address *addr*
`jmp L`n	unconditional jump
`cmp[B|I|P]` *src*$_1$`,` *src*$_2$	compare value *src*$_1$ against value *src*$_2$ (in "normal" order)
`je L`n	jump if last compare said equal
`jne L`n	jump if last compare said not equal
`jg L`n	jump if last compare said greater than (signed)
`jge L`n	jump if last compare said greater than or equal (signed)
`jl L`n	jump if last compare said less than (signed)
`jle L`n	jump if last compare said less than or equal (signed)
`ja L`n	jump if last compare said above (unsigned)
`jae L`n	jump if last compare said above or equal (unsigned)
`jb L`n	jump if last compare said below (unsigned)
`jbe L`n	jump if last compare said below or equal (unsigned)
`call <`*c*`,` *arg-count*`,` *returns-value?*`> *(`r`)`	call #c to function whose name is in r
`calls <`*c*`,` *arg-count*`,` *returns-value?*`>` s	call #c to system function named s
`L`n`:`	label declaration

Each function's list of instructions must begin and end with a label declaration. Function execution begins at the first instruction. Note that there is no `return` instruction; the function returns when it "falls off the end" of the last instruction.

General-purpose registers (r) are:

- named registers (nm) representing local variables and parameters

- temporary registers (written `$T`i for some integer i)

Sources (*src*) are:

- general-purpose registers (r)
- the caller's view of the return register (written `$RET<c>` for some unique call number c).
- 32-bit integer constants (i)
- the boolean constants `true` and `false`
- the nil address constant (`nil`)
- quoted string literals ("s")
- function names (s)

Destinations (*dst*) are:

- general-purpose registers (r)
- caller's argument registers (written `$A<c>`i for some unique call number c and argument number i)
- the callee's fixed return register `$RET`.

Addresses (a) are:

- addresses contained in source registers `[`r`]`
- addresses computed as the sum of the contents of a source register plus a fixed integer offset i`[`r`]`.

Local labels are written as `L`n for some non-negative integer n. Label numbers should be unique within a function. All names (parameters and locals) should be unique within a function. Top-level function names should be globally unique.

Regular (non-system) calls to IR functions use the following protocol. First, the arguments to the function are copied to the argument registers `$A0`, `$A1`, etc. Then the `call` instruction is executed giving the function label as target. Once inside the function, the formal parameters can be accessed by name, just like local variables. A function can return a value by moving it to special register `$RET` before returning by falling off the end. The caller can then fetch the returned value from `$RET`.

In practice, IR code generated from `fab` will generate *closure* records for all functions, as described below. However, there is no special support for closures in the IR itself.

IO and heap memory allocation are performed by issuing "system" function calls with the `calls` instruction, to one of the following special functions:

- `read_int` (which returns an integer result value)
- `write_int` (which takes an integer argument)
- `write_bool` (which takes a boolean argument)
- `write_string` (which takes a string argument)
- `write_newline` (which takes no arguments)
- `bounds_error` (which takes no arguments, issues the message "`Array bounds violation`" to `stderr`, and simulates an error exit)

- `nil_pointer` (which takes no arguments, issues the message "`Nil pointer dereference`" to `stderr`, and simulates an error exit)
- `alloc` (which takes an integer size argument in bytes and returns the address of the allocated storage)

Arguments and result for these functions are passed in the `$A` and `$RET` registers in the same way as for regular calls.

There is one further detail: each `call` and `calls` within a function is tagged with a unique identifying number c. In the caller's code, the outgoing argument registers and the return register associated with that call are tagged with the call number c, and are written `$A<`c`>0`, `$RET<`c`>`, etc. These call number tags are irrelevant for this assignment—the IR interpreter ignores them—but they will be useful when we come to do code generation in the next assignment. Note that there is only one set of argument registers and one return register in the IR machine state. Register names that differ only in their call number tag all refer to the same actual register. This means that if the actual arguments to a function themselves contain nested function calls, these nested calls must be performed before any arguments are stored into the `$A` locations for the outer call.

Implementation

As before, the file `frontend.jar` contains a complete `fab` front-end, which parses and type-checks `.fab` files and produces an AST data structure. File `Ast.java` documents the AST.

File `IR.java` documents the IR. Your code generator must be defined by a class `IRGen` that compiles and links with the `Ast`, `IR`, `IRGenDriver`, and `IRInterpDriver` classes provided here; you must not change these classes. Thus, your `IRGen` class must implement the method

```
static IR.Program gen(Ast.Program p)
```

Also, you will want to use the `Visitor` classes and `accept` methods in `Ast` to traverse declarations, statements, expressions, etc. Your generator class should be placed in a separate file `IRGen.java`, which can be compiled using

```
javac -classpath .:frontend.jar IRGen.java
```

You can then generate IR code for a file `foo.fab` into output file `foo.ir` by typing

```
java -classpath .:frontend.jar IRGenDriver foo.fab foo.ir
```

To run the IR interpreter using your IRGen, you can type

```
java -classpath .:frontend.jar:ir.jar IRInterpDriver foo.fab
```

assuming that your IRGen classes are in the current directory (and will thus take precedence over the reference versions in `ir.jar`).

There is a skeletal implementation in file `IRGen0.java`. Feel free to build your generator by extending this skeletal version. The top of this file contains definitions and supporting code for generating operands and emitting code.

It is *not* necessary to use a symbol table or environment when generating intermediate code. This causes problems only when generating code for the constants `true`, `false`, and `nil`; since there is no symbol table, there is no uniform way to handle the fact that they are constants rather than variables. `IRGen0.java` shows one approach to solving this problem.

Use control flow form for booleans by default. You'll need to write code to convert from control flow form to value form when storing a boolean; the converse code, to generate control flow from a value, is already in `IRGen0.java`.

Arrays and records should be represented by pointers to contiguous heap-allocated memory with the same general layout as in Assignment 1. Record layout will need to be refined to deal with the potentially different sizes and alignments of the various fields.

All **fab** functions should be represented using *closures*. Each function declaration should lead to generated IR code that allocates and initializes a closure record for the function. The closure record is an (ordinary) IR record whose first field contains the function's name, and whose subsequent fields contain the values of function's free identifiers. (Recall that these must be `const`s, so their values can be copied into the closure record at the point of function declaration without worrying about how they might subsequently change.) Within the environment where a function is declared, the function is represented by a pointer to its closure record.

A **fab** function call should generate IR code to (i) load the function's name from the first closure field; and (ii) `call` to that name, passing the closure record itself as a (new) initial argument. Compiling the body of a **fab** function should produce an IR function with an extra initial argument with a fresh name (e.g. `$CLOSURE`) that does not conflict with any of the existing argument names. Within the body of the function, references to **fab** free identifiers should compile to IR code that fetches the corresponding value field from `$CLOSURE`.

Submitting the Program

Place your `IRGen` class in the single file `IRGen.java`, and mail it to `cs322-01@cs.pdx.edu`, with `HW3` in the subject line. You must mail this file as a plain text *attachment*; the contents of the message itself don't matter. We should then be able to compile your code by creating a fresh directory, saving your attachment, copying in the provided `frontend.jar` and typing

```
javac -classpath .:frontend.jar IRGen.java
```

If we also copy in the provided file `IRGenDriver.class`, we should then be able to execute your generator on file `foo.fab` by typing

```
java -classpath .:frontend.jar IRGenDriver  foo.fab foo.ir
```

Note that we will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is important! You may lose points if you fail to submit your program in the correct way.

If you are working in a team of two, only one team member should submit a solution, which must have the names of both team members in a comment at the top; the other team member should send mail identifying him- or her-self as a team member. Please submit this information even if your team is unchanged from assignment 1.