

CS 322 Homework 3 – due 5 p.m., Friday, Feb. 25, 2005

(Copyright, Andrew Tolmach 2003-2005. All rights reserved.)

Generating Intermediate Code for PCAT

Working individually or in teams of two, write an intermediate code generator for (a subset of) the PCAT 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. Also, as before, the `WRITE` statement should output the string “1” for the boolean “`TRUE`” and “0” for the boolean “`FALSE`”. The behavior of function procedures that `RETURN` without a value can be arbitrary. You may assume that the intermediate language’s notion of integers and how to read them directly matches PCAT. Finally, you may assume that `TRUE`, `FALSE`, and `NIL` are never redefined within a PCAT program.

The intermediate code representation you will generate is similar in style to the SPARC assembly code, but with major simplifications, particularly in the treatment of variable addressing (addressing by name is supported), operand types (there is essentially only one type), and procedure call. In addition, nested procedure and variable declarations are included in the output format. Details of the intermediate code are given below.

The code generator should read the `.pcat` file from standard input, using the existing front end code 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 intermediate code on standard output, which is normally captured into an `.ir` file.

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

As usual, a “correct” generator is available in the jar file `irgen.jar`; it can be run on file `foo.pcat` by typing

```
java -classpath frontend.jar:irgen.jar IRGenDriver < foo.pcat
```

In addition, an interpreter that executes `.ir` files is available in the jar file `irinterp.jar`; to run it on file `foo.ir`, you can type

```
java -classpath irinterp.jar IRInterpDriver foo.ir
```

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

The intermediate code consists of declarations and code sequences, once for each procedure body and variable initialization, as well as one for the main program. Declarations are nested exactly

nop		no operation
ld	[<i>a1</i>], <i>r2</i>	load from memory address <i>a1</i> into register <i>r2</i>
st	<i>op1</i> , [<i>a2</i>]	store value <i>op1</i> into memory at address <i>a2</i>
call	<i>procedure_name</i>	do procedure call
scall	<i>system_call_name</i> , <i>op1</i>	do system call
return		return from procedure
ba	<i>label</i>	unconditional branch
bg	<i>label</i>	branch if last compare said greater
bl	<i>label</i>	branch if last compare said less
be	<i>label</i>	branch if last compare said equal
bge	<i>label</i>	branch if last compare said greater or equal
ble	<i>label</i>	branch if last compare said less or equal
bne	<i>label</i>	branch if last compare said not equal
bgu	<i>label</i>	branch if last compare said greater unsigned
blu	<i>label</i>	branch if last compare said less unsigned
bgeu	<i>label</i>	branch if last compare said greater or equal unsigned
bleu	<i>label</i>	branch if last compare said less or equal unsigned
cmp	<i>op1</i> , <i>op2</i>	compare operands <i>op1</i> and <i>op2</i>
mov	<i>op1</i> , <i>r2</i>	move operand <i>op1</i> to register <i>r2</i>
add	<i>op1</i> , <i>op2</i> , <i>r3</i>	register <i>r3</i> := <i>op1</i> + <i>op2</i>
sub	<i>op1</i> , <i>op2</i> , <i>r3</i>	register <i>r3</i> := <i>op1</i> - <i>op2</i>
smul	<i>op1</i> , <i>op2</i> , <i>r3</i>	register <i>r3</i> := <i>op1</i> * <i>op2</i>
sdiv	<i>op1</i> , <i>op2</i> , <i>r3</i>	register <i>r3</i> := <i>op1</i> / <i>op2</i> (integer division)
adda	<i>op1</i> , <i>op2</i> , <i>r3</i>	like add, but <i>op1</i> and <i>op3</i> represent addresses

Table 1: Intermediate code operators.

as in the AST, so that the correct scoping of each identifier can be determined just from the intermediate code. Each code sequence is a sequence of instructions, each with a SPARC-like operator and 0-3 operands. We assume a SPARC-like load/store architecture, with an unlimited number of temporary registers available. A full grammar for the intermediate representation is on-line in `ir.gram`. The operators are shown in Table 1.

Registers (*r*) are:

- register temporaries (written `%t n`) for some integer *n*
- the fixed registers `%o0` and `%i0` whose use is described below

Addresses (*a*) are:

- addresses of string constants (represented by the quoted string itself)
- addresses of named variables or labels (represented by the identifier)
- addresses of call arguments (`$a n`) whose use is described below

- addresses held in registers (r)
- the zero address (`$null`)

Operands (op) are:

- addresses
- integers held in registers (r)
- integer literals (any signed 32-bit integer value)

Labels are written as `L n` for some integer n .

Note that source operands are considerably more general than on SPARC. In particular, variables can be referenced by name without the need for explicit addressing. The intermediate code is essentially typeless: most operands represent integers or addresses, and are assumed to have size 1. However, address arithmetic should only be done with `adda`, and the zero address should be represented as `$null` rather than `0`. (Adhering to these rules will be important when we later translate the intermediate representation into real machine code.)

Branches and calls can be to any label without regard for its offset from the current pc. There are no delay slots, and thus no `annul` forms. As on SPARC, conditional jumps are formed from a `cmp` and an appropriate branch instruction.

Procedure calls use the following idiom: first, the actual values of procedure arguments 1, 2, ... are stored into the memory locations represented by `$a0`, `$a1`, ... Then a `call` instruction is executed. Within the procedure, the formal parameters can be accessed by name, just like local variables. Procedures return by executing a `return` instruction. If the procedure returns a value, it moves that value to special register `%i0` before returning. The calling procedure can fetch the returned value from special register `%o0`. Note that this protocol requires that nested procedure calls are performed before any arguments are stored into the `$a` locations. When the top-level procedure terminates, the value of `%o0` is returned as the overall status value of the program; this should be 0 for programs that terminate successfully.

IO and heap memory allocation are performed by issuing ordinary procedure calls to the special system calls `read_int` (which returns an integer result value), `write_int` (which takes an integer argument), `write_string` (which takes the address of a string as argument), `write_newline` (which takes no arguments), `bounds_error` (which takes no arguments, issues the message “Array bounds violation” to `stderr`, and exits), `nil_pointer` (which takes no arguments, issues the message “Nil pointer dereference” to `stderr`, and exits), and `alloc` (which takes an integer size argument in words and returns the address of the allocated storage). The argument, if any, is given as the second parameter to the `scall` instruction; if there is no argument, a dummy parameter value (e.g. 0) should be given.

Label and temporary names should be unique within a procedure (including the initialization code for the procedure’s locals). Code should never branch between two initialization code sequences or between an initialization code sequence and the main body.

Implementation

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

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

```
static void 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
```

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. Code can be emitted one line at a time as it is generated; there is no need to store it up.

It is *not* necessary to use a symbol table 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 easy way to see if these have been redefined, and no uniform way to handle the fact that they are constants rather than variables. So don't worry about the former, and expect the latter to lead to messy code: `IRGen0.java` shows one approach.

Use control flow form for booleans by default; when a value is needed, use a full-word integer. 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 layout as in Assignment 1.

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.pcat` by typing

```
java -classpath .:frontend.jar IRGenDriver < foo.pcat
```

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.