CS 302 Homework 3 – due 3:30 p.m., Wed., May 19, 1999

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* anything to do with real numbers, and the `LOOP`, and `EXIT` statements. *Also* exclude anything to do with records. The abstract syntax for this subset of **PCAT** is described in `/u/cs302acc/3/ast.txt`.

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.

Your intermediate code generator should be an executable `irgen` that reads an `.ast` file from `stdin` and produces an intermediate code (`.ir`) file on `stdout`.

To ease the implementation task, you may make the following simplifying assumptions:

- The `WRITE` statement should output the strings "`1`" and "`0`" for booleans, instead of "`TRUE`" and "`FALSE`".
- Assume that `TRUE` and `FALSE` are never redefined within a PCAT program.

Array bounds violations should be handled by generating code to call a specified runtime library function that prints "`Array bounds violation`" to `stderr` and exits immediately.

As usual, a "correct" generator is available in `/u/cs302acc/3/irgen`. In addition, an interpreter that executes `.ir` files is available in `/u/cs302acc/3/irinterp`. 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. 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 `/u/cs302acc/3/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:

`ld`	[$a1$],$r2$	load from memory address $a1$ into register $r2$
st	$r1$,[$a2$]	store $r1$ into memory at address $a2$
call	*procedure_name*	do procedure call
return		return from procedure
br	*label*	unconditional branch
bg	*label*	branch if last compare said greater
bl	*label*	branch if last compare said less
be	*label*	branch if last compare said equal
bge	*label*	branch if last compare said greater or equal
ble	*label*	branch if last compare said less or equal
bne	*label*	branch if last compare said not equal
cmp	$op1$,$op2$	compare operands $op1$ and $op2$
mov	$op1$,$r2$	move operand $op1$ to register $r2$
neg	$op1$,$r2$	register $r2 := -op1$
add	$op1$,$op2$,$r3$	register $r3 := op1 + op2$
sub	$op1$,$op2$,$r3$	register $r3 := op1 - op2$
umul	$op1$,$op2$,$r3$	register $r3 := op1 * op2$
udiv	$op1$,$op2$,$r3$	register $r3 := op1/op2$(integer division)
inc	$r1$	register $r1 := r1 + 1$
dec	$r1$	register $r1 := r1 - 1$

Table 1: Intermediate code operators.

- 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)

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: all operands represent integers or addresses, and are assumed to have size 1.

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.

IO and heap memory allocation are performed by issuing ordinary procedure calls to the special built-in procedures `PCAT$read_int` (which returns an integer result value), `PCAT$write_int` (which takes an integer argument), `PCAT$write_string` (which takes the address of a string as argument), `PCAT$write_newline`, `PCAT$bounds_error` (which takes no arguments, issues the message `Array bounds violation` to `stderr`, and exits), and `PCAT$alloc` (which takes an integer size argument and returns the address of the allocated storage).

Label and temporary names should be unique within a procedure (including the initialization code for the procedure's locals).

Implementation

Use the files `/u/cs302acc/2/{ast.c,ast.h,parse_ast.y}` to handle AST trees.

Definitions and supporting code for generating operands and emitting code are in `/u/cs302acc/3/ir.[ch]`. Code can be emitted one line at a time as it is generated; there is no need to store it up. A skeleton for a working generator is in `irgen0.c`; feel free to use this as the basis for your generator if you wish.

It is *not* necessary to use a symbol table when generating intermediate code. This causes problems only when generating code for the boolean constants `TRUE` and `FALSE`; 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.c` 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.c`.

Arrays should be represented by pointers to contiguous heap-allocated memory; the first word can represent the size of the array, with the contents starting at the second word.

Submitting the Program

Prepare a `makefile` that builds your generator and produces an executable called `irgen`. Submit your program by `mailing` a `shar` "bundle" containing *all* the relevant files to `cs302acc@cs.pdx.edu`.