CS 577 Homework 2 – Implementing Liveness Analysis – due 2pm, Wednesday, March 3, 2004

PCAT is a simple Pascal-like language sometimes used in PSU's undergraduate compiler course. There is a well-defined intermediate representation (IR) for PCAT programs, which resembles a simplified form of SPARC assembly code. On the course web page, you'll find source code for a Java program `ProcessIR` that reads an IR file into an internal representation, and computes and writes out the CFG for each procedure body defined in the IR. A collection of PCAT program files and the corresponding IR files are provided as test inputs.

Your assignment is to extend `ProcessIR` to compute and write out the set of live variables (and temporary registers) at the end of each basic block. This will involve computing various per-block sets of variable definitions and uses, and iterating to find the fixed-point of the appropriate dataflow equations.

Intermediate Code Details

The intermediate code representation you will process 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, PCAT supports nested procedure and variable declarations, and this nesting is preserved in the IR.

The intermediate code consists of declarations and code sequences, once for each procedure body and variable initialization expression, as well as one for the main program. Each code sequence is a sequence of labels and 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)

Operands (op) are:

- addresses
- integers held in registers (r)

`ld`	`[`$a1$`],`$r2$	load from memory address $a1$ into register $r2$
`st`	$op1$`,[`$a2$`]`	store value $op1$ into memory at address $a2$
`call`	*procedure_name*	do procedure call
`return`		return from procedure
`ba`	*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.

- 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`. IO and heap memory allocation are performed by issuing ordinary procedure calls to special built-in procedures (e.g., `PCAT$write_int`).

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

Mechanics

The following files are provided:

- `lexparse.jar` contains executable classes for a lexical analyzer and parser for .ir files. (These are produced by generators and rely on some additional libraries, so I haven't bothered to provide sources).
- `IR.java`, which defines an internal representation for IR files.
- `Cfg.java`, which defines a class of which each instance represents a CFG.
- `OperandSet.java`, which contains supporting code for manipulating sets of IR operands.
- `ProcessIR.java`, which is the top-level driver.

To compile and run the driver as it is, you simply put all the files in the current directory and say

```
javac -classpath .:lexparse.jar ProcessIR.java
```

Java's built-in make facility should take care of compiling all the `.java` files. You also need to use the same classpath when you run the program, e.g.

```
java -classpath .:lexparse.jar ProcessIR < prime.ir
```

Put your liveness analyzer in a new file `Liveness.java`, and modify `ProcessIR` to invoke the analyzer on each CFG and print out the results.

For example, the output for procedure `mark` in program `prime.ir` should look something like this:

```
0 { prime x IsPrime }
1 { }
2 { prime %t2 x IsPrime }
3 { prime %t2 x IsPrime }
4 { prime %t2 %t3 x }
5 { prime %t2 %t3 x }
6 { }
7 { }
8 { }
```

where the numbers on the left are basic block numbers as computed by Cfg, and the items in the sets are IR operands. (Order within the sets doesn't matter.)

Further notes:

- The driver provided reads a .ir file from standard input, and computes CFG's for each PCAT procedure body and for the top-level program. It processes each procedure *before* its subprocedures. It doesn't bother looking at the code fragments for variable initializations, and you should ignore these too.

- When computing live variables, you should consider only `AddrName` and `RegTemp` operands; ignore `%O0, %I0, $a`n, etc. (This restriction is already coded into `OperandSet.add`).
- You don't need to worry about the efficiency of your fixpoint computation (unless you want to!).
- Feel free to modify any of the provided files, so long as you submit your changed versions.

How to submit your homework.

Submit your homework *by email* to `apt@cs.pdx.edu` before the beginning of class on the due date. You should submit your `Liveness.java` code, and the code for any other files that you have modified, as plain-text *attachments* to your email. Also include any non-obvious information about how to run things.