

Computer Science 3 - 2010

Programming Language Translation

Practical for Week 20, beginning 6 September 2010 - Solutions

Full source for the solutions summarized here can be found in the ZIP file on the Web page - PRAC20A.ZIP (Java) and PRAC20AC.ZIP (C#).

Task 2

Most people had seen at least one improvement that could be made to the frequency checker. Here is one simple suggestions (there are others, of course):

```

read("First number? ", item);
while (item > 0) {
    if (item < limit)
        count[item] = count[item] + 1;
    read("Next number (<= 0 stops) ", item);
}
// terminate input with a result <= 0
// if in range
// increment appropriate count

```

Task 4

Most people seemed to get to (or close to) a solution, or close to a solution. Here is one very simple one that matches the simple improvement above:

<pre> ; read a list of positive numbers, determine frequency of each ; P.D. Terry, Rhodes University, 2010 0 DSP 3 2 LDA 1 4 LDC 2000 limit = 2000 (toy problem) 6 ANEW 7 STO 8 LDA 2 10 LDC 0 12 STO 13 LDA 2 15 LDV 16 LDC 2000 18 CLT 19 BZE 42 while (i < limit) { 21 LDA 1 23 LDV 24 LDA 2 26 LDV 27 LDXA 28 LDC 0 30 STO count[i] = 0; 31 LDA 2 33 LDA 2 35 LDV 36 LDC 1 38 ADD i = i + 1; 39 STO 40 BRN 13 } 42 PRNS "First number? " 44 LDA 0 46 INPI read("First number? ", item); 47 LDA 0 49 LDV 50 LDC 0 52 CGT 53 BZE 89 while (item > 0) { 55 LDA 0 57 LDV 58 LDC 2000 60 CLT 61 BZE 82 if (item < limit) 63 LDA 1 65 LDV 66 LDA 0 68 LDV 69 LDXA 70 LDA 1 </pre>	<pre> 72 LDV 73 LDA 0 75 LDV 76 LDXA 77 LDV 78 LDC 1 80 ADD count[item] = 81 STO count[item] + 1; 82 PRNS "Next number (<= 0 stops) " 84 LDA 0 86 INPI read("Next Number", item); 87 BRN 47 } 89 LDA 2 91 LDC 0 93 STO i = 0; 94 LDA 2 96 LDV 97 LDC 2000 99 CLT 100 BZE 141 while (i < limit) { 102 LDA 1 104 LDV 105 LDA 2 107 LDV 108 LDXA 109 LDV 110 LDC 0 112 CGT 113 BZE 130 if (count[i] > 0) { 115 LDA 2 117 LDV 118 PRNI write(i); 119 LDA 1 121 LDV 122 LDA 2 124 LDV 125 LDXA 126 LDV 127 PRNI write(count[i]); 128 PRNS "\n" write("\n"); 130 LDA 2 } 132 LDA 2 134 LDV 135 LDC 1 137 ADD 138 STO i = i + 1; 139 BRN 94 } 141 HALT System.exit(0) </pre>
---	---

Notice the style of commentary - designed to show the algorithm to good advantage, rather than being a statement by statement comment at a machine level (which is what many people did). Some people changed the original algorithm considerably, which was acceptable, but perhaps they missed out on the intrinsic simplicity of the translation process.

Task 5 - Checking overflow

Checking for overflow in multiplication and division was not well done. You cannot multiply and then try to check overflow (it is too late by then) - you have to detect it in a more subtle way. Here is one way of doing it - note the check to prevent a division by zero. This does not use any precision greater than that of the simulated machine itself. I don't think anybody spotted that the `PVM.rem` opcode also involved division, and many people who thought of using a multiplication overflow check on these lines forgot that numbers to be multiplied can be negative as well as positive.

```

case PVM.mul:           // integer multiplication
    tos = pop(); sos = pop();
    if (tos != 0 && Math.abs(sos) > maxInt / Math.abs(tos)) ps = badVal;
    else push(sos * tos);
    break;
case PVM.div:           // integer division (quotient)
    tos = pop();
    if (tos == 0) ps = divZero;
    else push(pop() / tos);
    break;
case PVM.rem:           // integer division (remainder)
    tos = pop();
    if (tos == 0) ps = divZero;
    else push(pop() % tos);
    break;

```

Some students used an intermediate `long` variable (most of them forgot that they should use the `abs` function as well!)

Task 6 - Your lecturer is quite a character

Reading and writing characters was trivially easy, being essentially a simple variation on the cases for numeric input and output. However, the output of numbers was arranged to have a leading space; this is not as pretty when you see it applied to characters, is it - which is why the call to `results.write` uses a second argument of 1, not 0 (this argument could have been omitted). Note the use of the modulo arithmetic to ensure that only sensible ASCII characters will be printed:

```

case PVM.inpc:          // character input
    mem[pop()] = data.readChar();
    break;
case PVM.prnc:          // character output
    if (tracing) results.write(padding);
    results.write((char) (Math.abs(pop()) % (maxChar + 1)), 1);
    if (tracing) results.writeLine();
    break;

```

With the aid of the `PVM.inpc` opcode the input section of the program changes to something like that shown below - note that we have to use the magic number 46 in the comparison (the code for "period" in ASCII):

```

44 INPC                read(ch)
45 LDA      0
47 LDV
48 LDC      46
50 CNE
51 BZE      77          while (ch != '.') {

```

Task 7 - Your lecturer - what's his case?

Extending the machine and the assembler still further with opcodes `CAP`, `INC` and `DEC` was also straightforward. However, many people had not considered the hint that one should not limit the `INC` and `DEC` opcodes to cases where they can handle only statements like `X++`. In some programs you might want to have statements like `List[N+6]++`.

Hence, the opcodes for the equivalent of a ++ or -- operation produced interesting answers. There are clearly two approaches that could be used: either increment the value at the top of the stack, or increment the variable whose address is at the top of the stack. I suspect the latter is more useful if you are to have but one of these (one could, of course, provide both versions of the opcodes). Here is my suggestion (devoid of precautionary checking):

```

case PVM.cap:           // toUpperCase
    push(Character.toUpperCase((char) pop()));
    break;
case PVM.inc:           // ++
    mem[pop()]++;
    break;
case PVM.dec:           // --
    mem[pop()]--;
    break;

```

Task 8 - Improving the opcode set still further

Once again, adding the LDL N and STL N opcodes is very easy. This required changes to be made to the assembler in PVMAsm.java as well as to the interpreter, which clearly confused several people considerably!

```

case PVM.ldl:           // push local value
    push(mem[cpu.fp - 1 - next()]);
    break;
case PVM.stl:           // store local value
    mem[cpu.fp - 1 - next()] = pop();
    break;

```

Some people forgot to introduce the LDL and STL wherever they could, did not incorporate CAP and INC/DEC and ran the last loop the wrong way! If one codes carefully, the character frequency checker reduces to the code shown below:

```

; read a string and display the frequency of each letter
; P.D. Terry, Rhodes University, 2010
; optimised instruction set for loading and storing
0 DSP      2
2 LDC      256      limit = 256 ASCII character set
4 ANEW
5 STL      1        count = new int[limit];
7 LDC      0
9 STL      0        ch = 0;
11 LDL     0
13 LDC      256
15 CLT
16 BZE     31      while (ch < limit) {
18 LDL     1
20 LDL     0
22 LDXA
23 LDC      0
25 STO
26 LDA      0        count[ch] = 0;
28 INC      ch++;
29 BRN     11      }
31 LDA      0
33 INPC
34 LDL     0        read(ch);
36 LDC      46
38 CNE
39 BZE     53      while (ch != '.') {
41 LDL     1
43 LDL     0
45 CAP
46 LDXA
47 INC      count[toUpperCase(ch)]++;
48 LDA      0
50 INPC
51 BRN     34      }
53 LDC      90
55 STL     0        ch = 'z';
57 LDL     0
59 LDC      65
61 CGE
62 BZE     92      while (ch >= 'A') {
64 LDL     1
66 LDL     0
68 LDXA
69 LDV
70 LDC      0
72 CGT
73 BZE     87      if (count[ch] > 0) {
75 LDL     0
77 PRNC      write(ch);
78 LDL     1
80 LDL     0
82 LDXA
83 LDV
84 PRNI      write(count[ch]);
85 PRNS     "\n"    write("\n");
87 LDA      0
89 DEC      ch--;
90 BRN     57      }
92 HALT      System.exit(0);

```

Task 9 - Nothing like practice to make things perfect

This example aimed to demonstrate the use of the Boolean opcodes. Here is a solution, also making use of the new opcodes (a solution using the original opcodes would have been acceptable, of course). It suffices to use the AND and OR opcodes - there was no need to use short-circuit evaluation.

```
0 DSP 3 ; v0 is x, v1 is y, v2 is z 34 PRNB ; write(x || !y && z);
2 PRNS " X Y Z X OR !Y AND Z\n" 35 PRNS "\n" ; write("\n");
4 LDC 0 ; 37 LDL 2
6 STL 0 ; x = false; 39 NOT
8 LDC 0 ; repeat 40 STL 2 ; Z = ! Z;
10 STL 1 ; y = false; 42 LDL 2
12 LDC 0 repeat 44 NOT
14 STL 2 ; z = false; 45 BZE 16 ; until !Z;
16 LDL 0 ; repeat 47 LDL 1
18 PRNB ; write(x); 49 NOT
19 LDL 1 ; 50 STL 1 ; Y = ! Y;
21 PRNB ; write(y); 52 LDL 1
22 LDL 2 ; 54 NOT
24 PRNB ; write(z); 55 BZE 12 ; until !Y;
25 LDL 0 ; 57 LDL 0
27 LDL 1 ; 59 NOT
29 NOT ; (not y) 60 STL 0 ; X = !X;
30 LDL 2 ; 62 LDL 0
32 AND ; (not y and z) 64 NOT
33 OR ; x or (not y and z) 65 BZE 8 ; until !X;
67 HALT
```

Task 10 - Safety first

In this task you were invited to make further modifications to the interpreter to make it "safer". This part of the practical was not well done, however, and few groups had thought through how to trap all the disasters that might occur if very badly incorrect code found its way to the interpreter stage.

Several groups did follow the basic advice given. Noting that many of the opcodes involve calls to the auxiliary routines `push()` and `pop()`, it makes sense to do some checking there:

```
// Bumps stack pointer and pushes value onto stack
mem[--cpu.sp] = value;
if (cpu.sp < cpu.hp) ps = badMem;
}

static int pop() {
// Pops and returns top value on stack and bumps stack pointer
if (cpu.sp == cpu.fp) ps = badMem;
return mem[cpu.sp++];
}
```

Note that the system should not call on something like `System.out.println("error message")` when errors are detected, but should simply change the status flag `ps` to an appropriate value that will ensure that the fetch-execute cycle will stop immediately thereafter and invoke the `postMortem` method to clean up the mess. Many people had missed this point.

However, there are many other places where checking could and should be attempted. For example, the `cpu.pc` register might get badly corrupted. This can be checked by changing the start of the fetch-execute cycle as follows:

```
do {
pcNow = cpu.pc; // retain for tracing/postmortem
if (cpu.pc < 0 || cpu.pc >= codeLen) {
ps = badAdr;
break;
}
cpu.ir = next(); // fetch
...
}
```

It would be just as well to protect the BRN and BZE opcodes as well:

```

case PVM.brn:          // unconditional branch
    cpu.pc = next();
    if (cpu.pc < 0 || cpu.pc >= codeLen) ps = badAdr;
    break;

case PVM.bze:          // pop top of stack, branch if false
    int target = next();
    if (pop() == 0) {
        cpu.pc = target;
        if (cpu.pc < 0 || cpu.pc >= codeLen) ps = badAdr;
    }
    break;

```

There are many places where intermediate addresses are computed that really need to be checked. Several groups had read up in the text (or looked at solutions from previous years!) and introduced a further checking function on the lines of:

```

static boolean inBounds(int p) {
    // Check that memory pointer p does not go out of bounds. This should not
    // happen with correct code, but it is just as well to check
    if (p < heapBase || p > memSize) ps = badMem;
    return (ps == running);
}

```

which can and should be invoked in situations like the following:

```

case PVM.dsp:          // decrement stack pointer (allocate space for variables)
    int localSpace = next();
    cpu.sp -= localSpace;
    if (inBounds(cpu.sp)) // initialize
        for (loop = 0; loop < localSpace; loop++)
            mem[cpu.sp + loop] = 0;
    break;
case PVM.lda:          // push local address
    adr = cpu.fp - 1 - next();
    if (inBounds(adr)) push(adr);
    break;
case PVM.ldl:          // push local value
    adr = cpu.fp - 1 - next();
    if (inBounds(adr)) push(mem[adr]);
    break;
case PVM.stl:          // store local value
    adr = cpu.fp - 1 - next();
    if (inBounds(adr)) mem[adr] = pop();
    break;
case PVM.inc:          // ++
    adr = pop();
    if (inBounds(adr)) mem[adr]++;
    break;

```

Several people had incorporated the refinements in the text for protecting the ANEW and LDXA opcodes:

```

case PVM.anev:          // heap array allocation
    int size = pop();
    if (size <= 0 || size + 1 > cpu.sp - cpu.hp - 2)
        ps = badAll;
    else {
        mem[cpu.hp] = size;
        push(cpu.hp);
        cpu.hp += size + 1;
    }
    break;

case PVM.ldxa:          // heap array indexing
    adr = pop();
    int heapPtr = pop();
    if (heapPtr == 0) ps = nullRef;
    else if (heapPtr < heapBase || heapPtr >= cpu.hp) ps = badMem;
    else if (adr < 0 || adr >= mem[heapPtr]) ps = badInd;
    else push(heapPtr + adr + 1);
    break;

```

Few, if any, thought to check that input operations might succeed or had succeeded:

```

case PVM.inpi:          // integer input
    adr = pop();
    if (inBounds(adr)) {
        mem[adr] = data.readInt();
        if (data.error()) ps = badData;
    }
    break;

```

For completeness we should check the PRNS opcode (the terminating NUL character might have been omitted by a faulty assembler):

```

case PVM.prns:          // string output
    if (tracing) results.write(padding);
    loop = next();
    while (ps == running && mem[loop] != 0) {
        results.write((char) mem[loop]); loop--;
        if (loop < stackBase) ps = badMem;
    }
    if (tracing) results.writeLine();
    break;

```

Task 11 - How do our systems perform?

In the kit you were given two versions of the infamous Sieve program written in PVM code. `S1.PVM` used the original opcode set; `S2.PVM` used the extended opcodes suggested in Task 8.

There were some intriguing claims made, several of which lead me to suspect their authors clearly think I am naive. If your interpreters were incorrect, I doubt whether `S2.PVM` would have given you any meaningful results.

The timings I obtained on an elderly 1.4GHz laptop for an upper limit of 1000 in the sieve and 2000 iterations were as follows:

	Java	C#
Original opcodes + interpreter with no bounds checks	10.30	10.60
Original opcodes + interpreter with the bounds checks of Task 9	15.57	13.04
Extended opcodes + interpreter with no bounds checks	9.47	7.07
Extended opcodes + interpreter with the bounds checks of Task 9	12.80	8.69

Although the Java and C# systems use effectively exactly the same source code for each, it is interesting to see that the ratios of these times are not the same. They all show a reasonable speedup when the extended opcode set is used (more for the C# versions than for the Java ones) but a considerable slow down when the error checks are introduced.

General comments

There were a few good solutions submitted, and some very energetic ones too - clearly some students had put in many hours developing their code. This is very encouraging. But there was also evidence of load shedding and lack of co-operation. I am looking for proper team efforts, not disjoint contributions that clearly show that some of you did not know what the other team members were doing.

Do learn to put your names into the introductory comments of programs that you write - and to comment your code properly!

And please learn to use LPRINT, which will save your lots of paper and printing bills.