Computer Science 3 - 2008

Programming Language Translation

Practical for Week 20, beginning 1 September 2008 - Solutions

There were some very good solutions submitted, and some very energetic ones too - clearly a lot of students had put in many hours developing their code. This is very encouraging. Do learn to put your names into the introductory comments of programs that you write.

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 palindrome checker to ensure that the loop terminated as quickly as possible. Here are some suggestions (there are even more ways of course):

```
isPalindrome = true;                    // optimist
low = 0; high = n - 1;                  // initial indices
while (low < high) {                    // sweep through the list
  if (list[low] != list[high])
    isPalindrome = false;               // bad luck
  low = low + 1; high = high - 1;       // adjust indices
}

isPalindrome = true;                    // optimist
low = 0; high = n - 1;                  // initial indices
while (low < high && isPalindrome) {    // sweep through the list
  if (list[low] != list[high])
    isPalindrome = false;               // bad luck
  low = low + 1; high = high - 1;       // adjust indices
}

isPalindrome = true;                    // optimist
bool checking = true;                   // start search
low = 0; high = n - 1;                  // initial indices
while (checking) {                      // sweep through the list
  if (list[low] != list[high]) {
    isPalindrome = false;               // bad luck
    checking = false;                   // no need to search further
  }
  low = low + 1; high = high - 1;       // adjust indices
  if (low >= high) checking = false     // reached middle
}

isPalindrome = true;                    // optimist
low = 0; high = n - 1;                  // initial indices
while (low < high) {                    // sweep through the list
  if (list[low] != list[high]) {
    isPalindrome = false;               // bad luck
    low = high;                         // to abort the loop
  }
  low = low + 1; high = high - 1;       // adjust indices
}

isPalindrome = true;                    // optimist
low = 0; mid = n/2;                     // initial indices
while (low < mid) {                     // sweep through the list
  if (list[low] != list[n - 1 - low]) {
    isPalindrome = false;               // bad luck
    low = mid;                          // to abort the loop
  }
  low = low + 1;                        // adjust indices
}
```

Task 4

Most people seemed to get to a solution, or close to a solution. Here is one that matches the original Parva algorithm. 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 most 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.

```
; Read a sequence of numbers and report whether they form a palindromic
; sequence (one that reads the same from either end)
; Examples:   1 2 3 4 3 2 1  is palindromic
;             1 2 3 4 4 3 2  is non-palindromic
; P.D. Terry, Rhodes University, 2008
; Coded directly from Palin.pav without making the (obvious) improvements
; var n (0), low (1), high (2), item (3), isPalindrome (4), list (5)

  0  DSP   6
  2  LDA   5     ;
  4  LDC   100   ;
  6  ANEW        ;
  7  STO         ; int[] list = new int[100];
  8  LDA   0     ;
 10  LDC   0     ;
 12  STO         ; n = 0;
 13  LDA   3     ;
 15  INPI        ; read(item);
 16  LDA   3     ;
 18  LDV         ;
 19  LDC   0     ;
 21  CNE         ;
 22  BZE   49    ; while (item != 0) {
 24  LDA   5     ;
 26  LDV         ;
 27  LDA   0     ;
 29  LDV         ;
 30  LDXA        ;
 31  LDA   3     ;
 33  LDV         ;
 34  STO         ;    list[n] = item;
 35  LDA   0     ;
 37  LDA   0     ;
 39  LDV         ;
 40  LDC   1     ;
 42  ADD         ;
 43  STO         ;    n = n + 1;
 44  LDA   3     ;
 46  INPI        ;    read(item);
 47  BRN   16    ; } // while
 49  LDA   4     ;
 51  LDC   1     ;
 53  STO         ; isPalindrome = true;
 54  LDA   1     ;
 56  LDC   0     ;
 58  STO         ; low = 0;
 59  LDA   2     ;
 61  LDA   0     ;
 63  LDV         ;
 64  LDC   1     ;
 66  SUB         ;
 67  STO         ; high = n - 1;
 68  LDA   1     ;
 70  LDV         ;
 71  LDA   0     ;
 73  LDV         ;
 74  LDC   1     ;
 76  SUB         ;
 77  CLT         ;
 78  BZE   124   ; while (low < n - 1) {
 80  LDA   5     ;
 82  LDV         ;
 83  LDA   1     ;
 85  LDV         ;
 86  LDXA        ;
 87  LDV         ;
 88  LDA   5     ;
 90  LDV         ;
 91  LDA   2     ;
 93  LDV         ;
 94  LDXA        ;
 95  LDV         ;
 96  CNE         ;
 97  BZE   104   ;   if (list[low] != list[high]) {
 99  LDA   4     ;
101  LDC   0     ;
103  STO         ;     isPalindrome = false;
104  LDA   1     ;   }
106  LDA   1     ;
108  LDV         ;
109  LDC   1     ;
111  ADD         ;
112  STO         ;   low = low + 1;
113  LDA   2     ;
115  LDA   2     ;
117  LDV         ;
118  LDC   1     ;
120  SUB         ;
121  STO         ;   high = high - 1;
122  BRN   68    ; } // while
124  LDA   4     ;
126  LDV         ;
127  BZE   133   ; if (isPalindrome)
129  PRNS  "Palindromic sequence"
131  BRN   135   ; else
133  PRNS  "Non-palindromic sequence"
135  HALT        ; exit
```

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 if one of the factors is 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. This code should not generate an error message either, as many people did. Leave the error reporting to the postmortem routine.

```
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 arrranged 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 `palin.pvm` changes to that shown below - note that we have to use the magic number 46 in the comparison (the code for "period" in ASCII):

```
13 LDA      3     ;
15 INPC           ; read(item);
16 LDA      3     ;
18 LDV            ;
19 LDC      46    ; // '.'
21 CNE            ;
22 BZE      49    ; while (item != '.') {
```

Task 7 - Even better palindromes

Extending the machine and the assembler still further with opcodes `CAP`, `ISLD`, `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, as one goup did). Here is my suggestion:

```
case PVM.cap:           // toUpperCase
  push(Character.toUpperCase((char) pop()));
  break;
case PVM.isld:          // isLetterOrDigit
  tos = pop();
  push(Character.isLetterOrDigit((char) tos) ? 1 : 0);
  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, but if one codes carefully the palindrome checker reduces to the code shown below:

```
; Read a sequence of characters and report whether they form a palindromic
; sentence (one that reads the same from either end) ignoring case and non letters
; and terminating sentence with a period (ASCII 46)
; Examples:   Madam I'm Adam. is palindromic
;             Pat Terry.      is non-palindromic
; This version uses the optimized opcode set for a PVM
; P.D. Terry, Rhodes University, 2008
;    var n (0), low (1), high (2), item (3), isPalindrome (4), str (5)

 0  DSP   6     ;
 2  LDC   100   ;
 4  ANEW        ;
 5  STL   5     ; char[] str = new char [100];
 7  LDC   0     ;
 9  STL   0     ; n = 0;
11  LDA   3     ;
13  INPC        ; read(item);
14  LDL   3     ;
16  LDC   46    ;
18  CNE         ;
19  BZE   47    ; while (item != '.') {
21  LDL   3     ;
23  CAP         ;
24  STL   3     ;   item = toUpperCase(item);
26  LDL   3     ;
28  ISLD        ;
29  BZE   42    ;   if (isLetterOrDigit(item)) {
31  LDL   5     ;
33  LDL   0     ;
35  LDXA        ;
36  LDL   3     ;
38  STO         ;     str[n] = item;
39  LDA   0     ;
41  INC         ;     n++;
42  LDA   3     ;   }
44  INPC        ;   read(item);
45  BRN   14    ; }
47  LDC   1     ;
49  STL   4     ; isPalindrome = true;
51  LDC   0     ;
53  STL   1     ; low = 0;
55  LDL   0     ;
 57  LDC   1     ;
 59  SUB         ;
 60  STL   2     ; high = n - 1;
 62  LDL   1     ;
 64  LDL   0     ;
 66  LDC   1     ;
 68  SUB         ;
 69  CLT         ;
 70  BZE   99    ; while (low < n - 1) {
 72  LDL   5     ;
 74  LDL   1     ;
 76  LDXA        ;
 77  LDV         ;
 78  LDL   5     ;
 80  LDL   2     ;
 82  LDXA        ;
 83  LDV         ;
 84  CNE         ;   if (str[low] != str[high])
 85  BZE   91    ;
 87  LDC   0     ;
 89  STL   4     ;     isPalindrome = false;
 91  LDA   1     ;
 93  INC         ;   low++;
 94  LDA   2     ;
 96  DEC         ;   high--;
 97  BRN   62    ; }
 99  LDL   4     ;
101  BZE   107   ; if (isPalindrome)
103  PRNS  "Palindromic string"
105  BRN   109   ; else
107  PRNS  "Non-palindromic string"
109  HALT        ; exit
```

Task 9 - 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:

```
static void push(int value) {
// 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 appopriate 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 at 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.anew:          // 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 had 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 10 - 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, or you had interpreted the `INC` and `DEC` opcodes in some other way, I doubt whether `S2.PVM` would have given you any meaningful results.

The timings I obtained on my 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.