Computer Science 3 - 2017

Programming Language Translation

Practical for Week 2, beginning 24 July 2017 - Solutions

There were some very good solutions submitted, and some energetic ones too - clearly a lot of students had put in many hours developing their code. This is very encouraging, but there was also evidence of "sharing" out the tasks, not really working together a proper group, and not developing an interpreter that was up to the later tasks. And 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 servers - `PRAC2A.ZIP`

Task 3 involved reading some Parva code for a simple algorithm and then adding suitable commentary. It is highly recommended that you adopt the style shown below, where the higher level code acts as commentary, rather than adopting a line by line explanation of each mnemonic/opcode.

```
                 ; Demonstrate de Morgan's Laws
                 ; P.D. Terry, Rhodes University, 2017

 0 DSP    2      ; bool x is v0, y is v1
 2 PRNS   "   X      Y    (X.Y)\' X\'+Y\'  (X+Y)\' X\'.Y\'\n\n"
 4 LDA    0      ;
 6 LDC    0      ;
 8 STO           ; x = false;
 9 LDA    1      ; repeat
11 LDC    0      ;
13 STO           ;   y = false;
14 LDA    0      ;   repeat
16 LDV           ;
17 PRNI          ;     write(x);
18 LDA    1      ;
20 LDV           ;
21 PRNI          ;     write(y);
22 LDA    0      ;
24 LDV           ;
25 LDA    1      ;
27 LDV           ;
28 AND           ;
29 NOT           ;
30 PRNI          ;     write(!(x && y));
31 LDA    0      ;
33 LDV           ;
34 NOT           ;
35 LDA    1      ;
37 LDV           ;
38 NOT           ;
39 OR            ;
40 PRNI          ;     write(!x || !y);
41 LDA    0      ;
43 LDV           ;
44 LDA    1      ;
46 LDV           ;
47 OR            ;
48 NOT           ;
49 PRNI          ;     write(!(x || y));
50 LDA    0      ;
52 LDV           ;
53 NOT           ;
54 LDA    1      ;
56 LDV           ;
57 NOT           ;
58 AND           ;
59 PRNI          ;     write(!x && !y);
60 PRNS   "\n"   ;     writeLine();
62 LDA    1      ;
64 LDA    1      ;
66 LDV           ;
67 NOT           ;
68 STO           ;     y = !y;
69 LDA    1      ;
71 LDV           ;
72 NOT           ;
73 BZE    14     ;   until (!y);
75 LDA    0      ;
77 LDA    0      ;
79 LDV           ;
80 NOT           ;
81 STO           ;   x = !x;
82 LDA    0      ;
84 LDV           ;
85 NOT           ;
86 BZE    9      ; until (!x);
88 HALT          ; System.Exit();
```

It is easy to see that this does not use short circuit evaluation of Boolean expressions, as it uses AND and OR, which are infix operators that requires their two operands both to have been evaluated and pushed onto the expression stack. However, it is easy to eliminate the AND and OR by introducing "jumping code" as it is sometimes called. We rely on the idea that for short-circuit semantics to hold we can write the following logical identities:

```
x AND y ≡  if x  then y    else false
x OR y  ≡  if x  then true else y                          ...
```

If we apply them to an analysis of various Boolean expressions in the algorithm we could also use

```
!x AND !y ≡  if !x  then !y   else false
!x OR !y  ≡  if !x  then true else !y                      ...
```

Admittedly this has quite a lot more code than the binary operator code in the original. However, short-circuited Boolean evaluation is so much better that it is worth developing special opcodes to achieve it, as we shall see later in the course.

```
                  ; Demonstrate de Morgan's Laws
                  ; P.D. Terry, Rhodes University, 2017
                  ; using short-circuit semantics
  0 DSP    2      ; bool x is v0, y is v1
  2 PRNS   "   X      Y    (X.Y)\' X\'+Y\'  (X+Y)\' X\'.Y\'\n\n"
  4 LDA    0      ;
  6 LDC    0      ;
  8 STO           ; x = false;
  9 LDA    1      ; repeat
 11 LDC    0      ;
 13 STO           ;   y = false;
 14 LDA    0      ;   repeat
 16 LDV           ;
 17 PRNI          ;     write(x);
 18 LDA    1      ;
 20 LDV           ;
 21 PRNI          ;     write(y);
 22 LDA    0      ;
 24 LDV           ;
 25 BZE   32      ;
 27 LDA    1      ;
 29 LDV           ;
 30 BRN   34      ;
 32 LDC    0      ;
 34 NOT           ;
 35 PRNI          ;     write(!(x && y));
 36 LDA    0      ;
 38 LDV           ;
 39 NOT           ;
 40 BZE   46      ;
 43 LDC    1      ;
 44 BRN   50      ;
 46 LDA    1      ;
 48 LDV           ;
 49 NOT           ;
 50 PRNI          ;     write(!x || !y);
 51 LDA    0      ;
 53 LDV           ;
 54 BZE   60      ;
 56 LDC    1      ;
 58 BRN   63      ;
 60 LDA    1      ;
 62 LDV           ;
 63 NOT           ;
 64 PRNI          ;     write(!(x || y));
 65 LDA      0    ;
 67 LDV           ;
 68 BZE     74    ;
 70 LDC      0    ;
 72 BRN     78    ;
 74 LDA      1    ;
 76 LDV           ;
 77 NOT           ;
 78 PRNI          ;     write(!x && !y);
 79 PRNS    "\n"  ;     writeLine();
 81 LDA      1    ;
 83 LDA      1    ;
 85 LDV           ;
 86 NOT           ;
 87 STO           ;     y = !y;
 88 LDA      1    ;
 90 LDV           ;
 91 NOT           ;
 92 BZE     14    ;   until (!y);
 94 LDA      0    ;
 96 LDA      0    ;
 98 LDV           ;
 99 NOT           ;
100 STO           ;   x = !x;
101 LDA      0    ;
103 LDV           ;
104 NOT           ;
105 BZE      9    ; until (!x);
107 HALT          ; System.Exit();
```

It might be possible to manipulate these logical expressions to make for an even shorter solution, and you might like to puzzle out how this can be done. See also the discussion on the course web site.

Task 4 - Execution overheads - part one

See discussion of Task 10 below.

Task 5 - Coding the hard way

Task 5 was to hand-compile the Factorial program into PVM code. Most people got a long way towards this. Once again, look at how I have commented this, using "high level" code.

```
  0 DSP    3      ; n is v0, f is v1, i is v2
  2 LDA    0
  4 LDC    1
  6 STO           ; n = 1;
  7 LDA    0
  9 LDV
 10 LDC    20     ; // max = 20, constant
 12 CLE           ; while (n <= max) {
 13 BZE    78
 15 LDA    1
 17 LDC    1
 19 STO           ;   f = 1;
 20 LDA    2
 22 LDA    0
 24 LDV
 25 STO           ;   i = n;
 26 LDA    2
 28 LDV
 29 LDC    0
 31 CGT           ;   while (i > 0) {
 32 BZE    55
 34 LDA    1
 36 LDA    1
 38 LDV
 39 LDA    2
 41 LDV
 42 MUL
 43 STO
 44 LDA    2      ;     f = f * i;
 46 LDA    2
 48 LDV
 49 LDC    1
 51 SUB
 52 STO           ;     i = i - 1;
 53 BRN    26     ;   }
 55 LDA    0
 57 LDV
 58 PRNI          ;   write(n);
 59 PRNS   "! = " ;   write("! = ");
 61 LDA    1
 63 LDV
 64 PRNI          ;   write(f);
 65 PRNS   "\n"   ;   write("\n") (or use PRNL)
 67 LDA    0
 69 LDA    0
 71 LDV
 72 LDC    1
 74 ADD
 75 STO           ;   n = n + 1;
 76 BRN    7      ; }
 78 HALT
```

Note that `max` is a constant, not a variable. There is no need to assign it a variable location and store 20 into this - simply build the value of 20 into the instructions that need to use it. And why on earth redraft the whole algorithm into one that uses a *do-while loop* (or a *repeat-until* loop) in place of the suggested *while* loop?

Task 6 - Trapping overflow and other pitfalls

Checking for overflow in multiplication and division was not always well done. You cannot safely 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 when handling multiplication. This does not use any precision greater than that of the simulated machine itself. I don't think many spotted that the `PVM.rem` opcode also involved division, and some people who thought of using a multiplication overflow check on these lines forgot that numbers to be multiplied can be negative.

An alternative, slightlier risky method is shown as a comment - risky because, if the emulator were written in a system that itself trapped multiplicative overflow, it would all blow up anyway.

```
        case PVM.mul:           // integer multiplication
          tos = Pop();
          sos = Pop();
          if (tos != 0 && Math.Abs(sos) > maxInt / Math.Abs(tos)) ps = badVal;
// riskier
//        if (tos != 0 && tos * sos / tos != sos) 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;
```

or for the "inline" assembler

```
        case PVM.mul:           // integer multiplication
          tos = mem[cpu.sp++];
          if (tos != 0 && Math.Abs(mem[cpu.sp]) > maxInt / Math.Abs(tos)) ps = badVal;
// riskier
//        if (tos != 0 && tos * mem[cpu.sp] / tos != mem[cpu.sp]) ps = badVal;
          else mem[cpu.sp] *= tos;
          break;
        case PVM.div:           // integer division (quotient)
          tos = mem[cpu.sp++];
          if (tos != 0) mem[cpu.sp] /= tos;
          else ps = divZero;
          break;
        case PVM.rem:           // integer division (remainder)
          tos = mem[cpu.sp++];
          if (tos != 0) mem[cpu.sp] %= tos;
          else ps = divZero;
          break;
```

It is possible to use an intermediate `long` variable (but don't forget the casting operations or the `Abs` function):

```
        case PVM.mul:           // integer multiplication
          tos = Pop();
          sos = Pop();
          long temp = (long) sos * (long) tos;
          if (Math.Abs(temp) > maxInt) ps = badVal;
          else Push(sos * tos);
          break;
```

If given too large an index for an array to handle, a PVM program will terminate with an array bounds error as correctly trapped by the Push/Pop assembler. The same error would not be trapped by the Inline system, which merrily allows the `LDXA` opcode to wander wheresoever it likes. To fix this requires the following changes to the PVMInline interpreter. This strategy is discussed in the textbook.

```
        case PVM.anew:          // heap array allocation
          int size = mem[cpu.sp];
          if (size <= 0 || size + 1 > cpu.sp - cpu.hp - 2)
```

```
          ps = badAll;
        else {
          mem[cpu.hp] = size;
          mem[cpu.sp] = cpu.hp;
          cpu.hp += size + 1;
        }
        break;

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

Task 7 - Arrays

The code as supplied for tracking students' attendance at a practical suffered from various defects - a "student number" of zero is useless, even though it would be accepted quite happily, a student is able to clock in more than once, the constant `StudentsInClass` has a misleading value, and if a large negative number is supplied the program crashes. A few simple changes will fix some or all of these. I was happy to accept just one or two of these changes, but here is a rather radical rewrite that embraces them all, and uses the value 0 to terminate the program, just so that you can have a look at how this would have been translated. (`STUDENTS1.PAV`):

```
void main () {
// Track students as they clock in and out of a practical - improved version
// P.D. Terry, Rhodes University, 2017
// Improved version

  const StudentsInClass = 100;
  bool[] atWork = new bool[StudentsInClass + 1];

  int student = 1;                        // students are numbered 1 .. 100
  while (student <= StudentsInClass) {
    atWork[student] = false;              // nobody is at the practical to start with
    student = student + 1;
  }

  read("Student? (> 0 clocks in, < 0 clocks out, 0 terminates) ", student);
  while (student != 0) {
    bool clockingIn = true;               // distinguish "in" and "out" easily
    if (student < 0) {
      clockingIn = false;
      student = -student;                 // fix the number
    }
    if (student > StudentsInClass)
      write("Invalid student number\n");
    else if (clockingIn)
      if (atWork[student])  write(student, " has already clocked in!\n");
      else atWork[student] = true;
    else
      if (!atWork[student]) write(student, " has not yet clocked in!\n");
      else atWork[student] = false;
    read("Student? (> 0 clocks in, < 0 clocks out, 0 terminates) ", student);
  } // while

  write("The following students have still not clocked out\n");
  student = 1;
  while (student <= StudentsInClass) {
    if (atWork[student]) write(student);
    student = student + 1;
  } // while

} // main
```

A translation into PVM code is a little tedious, and it is easy to leave some of the code out and get a corrupted solution:

```
                  ; Track students as they clock in and out pf a practical
                  ; P.D. Terry, Rhodes University, 2017
                  ; bool[] atwork is v0, int student is v1
   0 DSP      3   ;
   2 LDA      0   ;
   4 LDC      100 ;
   6 LDC      1   ;
   8 ADD          ;
   9 ANEW         ;
  10 STO          ; bool[] atWork =  new bool[...]
  11 LDA      1   ;
  13 LDC      1   ;
  15 STO          ; int student = 1;
  16 LDA      1   ;
  18 LDV          ;
  19 LDC      100 ;
  21 CLE          ;
  22 BZE      45  ; while (student <= 100) {
  24 LDA      0   ;
  26 LDV          ;
  27 LDA      1   ;
  29 LDV          ;
  30 LDXA         ;
  31 LDC      0   ;
  33 STO          ;   atWork[Student] = false;
  34 LDA      1   ;
  36 LDA      1   ;
  38 LDV          ;
  39 LDC      1   ;
  41 ADD          ;   student = student + 1;
  42 STO          ;
  43 BRN      16  ; }
  45 PRNS     "Student? (> 0 clocks in, < 0  ...
  47 LDA      1   ;
  49 INPI         ; read(student);
  50 LDA      1   ;
  52 LDV          ;
  53 LDC      0   ;
  55 CNE          ;
  56 BZE      166 ; while (student != 0) {
  58 LDA      2   ;
  60 LDC      1   ;
  62 STO          ;   bool clockingIn = true;
  63 LDA      1   ;
  65 LDV          ;
  66 LDC      0   ;
  68 CLT          ;
  69 BZE      83  ;  if (student < 0) {
  71 LDA      2   ;
  73 LDC      0   ;
  75 STO          ;    clockingIn = false;
  76 LDA      1   ;
  78 LDA      1   ;
  80 LDV          ;
  81 NEG          ;
  82 STO          ;    student = - student
  83 LDA      1   ;  }
  85 LDV          ;
  86 LDC      100 ;
  88 CGT          ;
  89 BZE      95  ;  if (student > StudentsInClass)
  91 PRNS     "Invalid student number"
  93 BRN      159 ;
  95 LDA      2   ;
  97 LDV          ;
  98 BZE      130 ;  else if (clockingIn)
 100 LDA      0   ;
 102 LDV          ;
 103 LDA      1   ;
 105 LDV          ;
 106 LDXA         ;
 107 LDV          ;
 108 BZE      118 ;    if (atWork[student])
 110 LDA      1   ;
 112 LDV          ;      write (student)
 113 PRNI         ;
 114 PRNS     " has already clocked in!\n"
 116 BRN      128 ;
 118 LDA      0   ;    else
 120 LDV          ;
 121 LDA      1   ;
 123 LDV          ;
 124 LDXA         ;
 125 LDC      1   ;
 127 STO          ;      atWork[student] = true;
 128 BRN      159 ;
 130 LDA      0   ;   else
 132 LDV          ;
 133 LDA      1   ;
 135 LDV          ;
 136 LDXA         ;
 137 LDV          ;
 138 NOT          ;
 139 BZE      149 ;     if (!atWork[student]
 141 LDA      1   ;
 143 LDV          ;       write(student)
 144 PRNI         ;
 145 PRNS     " has not yet clocked in!\n"
 147 BRN      159 ;
 149 LDA      0   ;
 151 LDV          ;     else
 152 LDA      1   ;
 154 LDV          ;
 155 LDXA         ;
 156 LDC      0   ;
 158 STO          ;      atWork[student] = false];
 159 PRNS     "Student? (> 0 clocks in, < 0 ...
 161 LDA      1   ;
 163 INPI         ;   read(student)
 164 BRN      50   ; } // while (student != 0)
 166 PRNS     "The following students have still not ...
 168 LDA      1   ;
 170 LDC      1   ;
 172 STO          ; student = 1;
 173 LDA      1   ;
 175 LDV          ;
 176 LDC      100 ;
 178 CLE          ;
 179 BZE      206 ; while (student <= 100
 181 LDA      0   ;
 183 LDV          ;
 184 LDA      1   ;
 186 LDV          ;
 187 LDXA         ;
 188 LDV          ;
 189 BZE      195 ;   if (atWork[student])
 191 LDA      1   ;
 193 LDV          ;
 194 PRNI         ;     write(student);
 195 LDA      1   ;
 197 LDA      1   ;
 199 LDV          ;
 200 LDC      1   ;
 202 ADD          ;
 203 STO          ;     student = student + 1;
 204 BRN      173 ; } // while (student <= 100
 206 HALT         ; System.Exit()
```

Task 8 - Your lecturer is quite a character

To be able to deal with input and output of character data we need to add two new opcodes, modelled on the INPI and PRNI codes whose interpretation would be as below. All of the new opcodes require additions to the lists of opcodes in the assembler and interpreter (be careful of two-word opcodes; they crop up in several places).

Note that the output of numbers was arranged to have a leading space; this is not as pretty when you see i t a p p l i e d t o c h a r a c t e r s , i s i t - 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 make quite sure that only sensible ASCII characters will be printed:

```
case PVM.inpc:          // character input
  adr = Pop();
  if (InBounds(adr)) {
    mem[adr] = data.ReadChar();
    if (data.error()) ps = badData;
  }
  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;
```

or for the "inline" assembler

```
case PVM.inpc:          // character input
  mem[mem[cpu.sp++]] = data.ReadChar();
  break;
case PVM.prnc:          // character output
    if (tracing) results.Write(padding);
  results.Write((char) (Math.Abs(mem[cpu.sp++]) % (maxChar + 1)), 1);
    if (tracing) results.WriteLine();
  break;
```

To build a really safe system there are further refinements we could make. It can be argued that we should not try to store a value outside of the range 0 .. 255 into a character variable. This suggests that we should have a range of `STO` type instructions that check the value on the top of stack before assigning it. One of these - `STOC` to act as a variation on `STO` - would be interpreted as follows; we would need another to handle `STLC` and so on (these have not yet been implemented in the solution kit).

```
case PVM.stoc:          // character checked store
  tos = Pop(); adr = Pop();
  if (inBounds(adr))
    if (tos >= 0 && tos <= maxChar) mem[adr] = tos; else ps = badVal;
  break;
```

or for the "inline" assembler, omitting the checking

```
case PVM.stoc:          // character (unchecked) store
  tos = mem[cpu.sp++]; mem[mem[cpu.sp++]] = tos;
  break;
```

Introducing opcodes to convert to lower or upper case is simply done by using the methods from the C# `Char` wrapper class (notice the need for casting operations as well, to satisfy the C# compiler):

```
case PVM.low:           // toLowerCase
  Push(Char.ToLower((char) Pop()));
  break;
case PVM.cap:           // toUpperCase
  Push(Char.ToUpper((char) Pop()));
  break;
```

or for the "inline" assembler - note that `cpu.sp` *is left unaltered.*

```
case PVM.low:           // toLowerCase
  mem[cpu.sp] = Char.ToLower((char) mem[cpu.sp]);
  break;
case PVM.cap:           // toUpperCase
  mem[cpu.sp] = Char.ToUpper((char) mem[cpu.sp]);
  break;
```

The `INC` and `DEC` operations are best performed by introducing opcodes that assume that an *address* has been planted on the top of stack for the variable (or array element) that needs to be incremented or decremented. This may not have been apparent to everyone, but consider (as hinted in the prac sheet) a statement like `a[i+j]++;`

```
case PVM.inc:           // ++
  adr = Pop();
  if (inBounds(adr)) mem[adr]++;
  break;
case PVM.dec:           // --
```

```
        adr = Pop();
        if (inBounds(adr)) mem[adr]--;
        break;
```

or for the "inline" assembler

```
      case PVM.inc:           // ++
        mem[mem[cpu.sp++]]++;
        break;
      case PVM.dec:           // --
        mem[mem[cpu.sp++]]--;
        break;
```

With all these in place the string reversal algorithm can be programmed as follows:

```
               ; Read a sentence and reverse in UPPER CASE
               ; P.D. Terry, Rhodes University, 2017
               ; char[] sentence is v0; leng is v1
               ; original opcode set

  0   DSP   2   ;
  2   LDA   0   ;
  4   LDC   256 ;
  6   ANEW      ;
  7   STO       ;  sentence = new char[256];
  8   LDA   1   ;
 10   LDC   0   ;
 12   STO       ;  leng = 0;
 13   LDA   0   ;  repeat {
 15   LDV       ;
 16   LDA   1   ;
 18   LDV       ;
 19   LDXA      ;
 20   INPC      ;    read(sentence[leng]);
 21   LDA   1   ;
 23   INC       ;    leng++;
 24   LDA   0   ;  }
 26   LDV       ;
 27   LDA   1   ;
 29   LDV       ;
 30   LDC   1  ;
 32   SUB      ;
 33   LDXA     ;
 34   LDV      ;
 35   LDC   46 ;    // '.' is 46 in ASCII table
 37   CEQ      ;
 38   BZE   13 ; until (sentence[leng-1] = '.');
 40   LDA   1  ;
 42   LDV      ;
 43   LDC   0  ;
 45   CGT      ; while (leng > 0) {
 46   BZE   63 ;
 48   LDA   1  ;
 50   DEC      ;   leng--;
 51   LDA   0  ;
 53   LDV      ;
 54   LDA   1  ;
 56   LDV      ;
 57   LDXA     ;
 58   LDV      ;
 59   CAP      ;
 60   PRNC     ;   write(upper(sentence[leng]);
 61   BRN   40 ; }
 63   HALT     ; System.Exit()
```

Task 9 - Improving the opcode set still further

Once again, adding the `LDL N` and `STL N` opcodes is very easy. Unfortunately, it is easy to leave some of the changes out and get a corrupted solution. The `PVMAsm` class requires modification in the *switch* statement that recognizes two-word opcodes:

```
   case PVM.brn:                          // all require numeric address field
   ...
   case PVM.ldc:
   case PVM.ldl:  // +++++++++++++++++  addition
   case PVM.stl:  // +++++++++++++++++  addition
     codeLen = (codeLen + 1) % PVM.memSize;
     if (ch == '\n')                      // no field could be found
       error("Missing address", codeLen);
     else {                               // unpack it and store
       PVM.mem[codeLen] = src.ReadInt();
       if (src.Error()) error("Bad address", codeLen);
     }
     break;
```

The `PVM` class requires several additions. We must add to the *switch* statement in the `Trace` and `ListCode` methods (several submissions missed this):

```
    static void Trace(OutFile results, int pcNow, bool traceStack, bool traceHeap) {
      switch (cpu.ir) {
        ...
        case PVM.ldl:  // +++++++++++++++++  addition
        case PVM.stl:  // +++++++++++++++++  addition
      }
      results.WriteLine();
    }
```

and we must provide case arms for all the new opcodes. A selection of these follows; the rest can be seen in the solution kit. Notice that for consistency all the "inBounds" checks should really be performed on the new opcodes too (several submissions missed this, and they have been left out here too so that you can add them yourselves). Firstly the basic two-word ones:

```
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;
```

or for the "inline" assembler `where we can code it all into one statement:`

```
case PVM.ldl:           // push local value
  mem[--cpu.sp] = mem[cpu.fp - 1 - mem[cpu.pc++]];
  break;
case PVM.stl:           // store local value
  mem[cpu.fp - 1 - mem[cpu.pc++]] = mem[cpu.sp++];
  break;
```

A great many submissions made a rather bizarre error. Part of the original kit read as follows - where the action for all the "missing" opcodes was to trap an error if they were encountered (by accident?)

```
case PVM.lda_2:         // push local address 2
case PVM.lda_3:         // push local address 3
case PVM.ldl:           // push local value
case PVM.ldl_0:         // push value of local variable 0
case PVM.ldl_1:         // push value of local variable 1
case PVM.ldl_2:         // push value of local variable 2
case PVM.ldl_3:         // push value of local variable 3
case PVM.stl:           // store local value
```

Incompletely modifying the code on the lines shown below would have had the effect of adding `PVM.lda_2`, `PVM.lda_3` as "extra" labels to the `PVM.ldl` clause (and similarly for other cases)!

```
case PVM.lda_2:         // push local address 2
case PVM.lda_3:         // push local address 3
case PVM.ldl:           // push local value
  mem[--cpu.sp] = mem[cpu.fp - 1 - mem[cpu.pc++]];
  break;
case PVM.ldl_0:         // push value of local variable 0
case PVM.ldl_1:         // push value of local variable 1
case PVM.ldl_2:         // push value of local variable 2
case PVM.ldl_3:         // push value of local variable 3
case PVM.stl:           // store local value
  mem[cpu.fp - 1 - mem[cpu.pc++]] = mem[cpu.sp++];
  break;
```

In improving the string reversal program, 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, this program reduces to the code shown below:

```
             ; Read a sentence and reverse in UPPER CASE
             ; P.D. Terry, Rhodes University, 2017
             ; char[] sentence is v0; leng is v1
             ; extended opcode set

 0   DSP   2   ;
 2   LDC   256 ;
 4   ANEW      ;  sentence = new char[256];
 5   STL_0     ;
 6   LDC_0     ;
 7   STL_1     ;  leng = 0;
 8   LDL_0     ;  repeat {
 9   LDL_1     ;
10   LDXA      ;
11   INPC      ;    read(sentence[leng]);
12   LDA_1     ;
13   INC       ;    leng++;
14   LDL_0     ;  }
15   LDL_1     ;
16   LDC_1     ;
17   SUB       ;
18   LDXA      ;
19   LDV       ;
20   LDC   46  ;   // '.' is 46 in ASCII table
22   CEQ       ;
23   BZE   8   ; until (sentence[leng-1] = '.');
25   LDL_1     ;
26   LDC_0     ;
27   CGT       ; while (leng > 0) {
28   BZE   40  ;
30   LDA_1     ;
31   DEC       ;   leng--;
32   LDL_0     ;
33   LDL_1     ;
34   LDXA      ;
35   LDV       ;
36   CAP       ;
37   PRNC      ;   write(upper(sentence[leng]);
38   BRN   25  ; }
40   HALT      ; System.Exit();
```

Task 10 - Execution overheads - part two

In the prac kit you were supplied with a second translation SIEVE2.PVM of a cut down version of the same prime-counting program SIEVE.PAV as was used in Task 4, but this time using the extended opcode set developed in the last task. The kit also included the code that could be executed if the PVM were extended still further on the lines of the suggestions on page 44 of the textbook.

Running SIEVE1.PVM through both of the original and modified assemblers, and SIEVE2.PVM and SIEVE3.PVM through both of the modified assemblers gave the following timings for the same limit (4000) and number of iterations (100) on my machines, one a laptop running Windows XP and one a desktop running Windows 7-32.

Desktop Machine (Win 7-32)	Sieve1.pvm	(1.00)	Sieve2.pvm	(0.78)	Sieve3.pvm	(0.75)
ASM1 (Push/Pop)	0.73		0.57		0.55	
ASM2 (Inline)	0.30	(0.41)	0.20	(0.36)	0.13	(0.24)

Laptop machine (XP-32)	Sieve1.pvm	(1.00)	Sieve2.pvm	(0.75)	Sieve3.pvm	(0.67)
ASM1 (Push/Pop)	1.14		0.85		0.76	
ASM2 (Inline)	0.52	(0.46)	0.30	(0.35)	0.27	(0.35)

The Desktop times were about 65% of those on the slower Laptop. The Inline times were about 40% of the Push/Pop system with the original limited opcode set. The Inline times were about 30% of the Push/Pop system with the extended opcode set,

The reasons are not hard to find. The InLine emulator makes very few function calls within the fetch-execute cycle, whereas the Push/Pop one makes a very large number, each carrying an extra overhead. Similarly, the introduction of the `LDL` and `STL` codes allowed for fewer opcodes to be interpreted to achieve the desired result.

If one wishes to improve the performance of the interpreter further it might make sense to get some idea of which opcodes are executed most often. Clearly this will depend on the application, and so a mix of applications might need to be analysed. It is not difficult to add a profiling facility to the interpreter, and this has been done in yet another interpreter that you can find in the solution kit. Running this on the Sieve files yielded some interesting results. For a start, there were enormous numbers of steps executed - probably more than you might think.

```
  Original opcodes          Extended opcode set         Extended opcode set
                            LDL and STL used            LDL, STL, LDL_x STL_x

39 494 323 operations.    27 070 118 operations. (68%)    (same op count)

  LDA      10824405         LDL       8186502           LDL_2    3821200
  LDV       9386302         LDC       4148705           LDL_1    2582600
  LDC       4948605         BZE       2182801           BZE      2182801
  STO       3165703         CLE       1782901           LDC_0    1910701
  BZE       2182801         BRN       1727700           LDC      1782902
  CLE       1782901         LDXA      1727600           CLE      1782901
  ADD       1782701         STO       1327700           BRN      1727700
  BRN       1727700         STL       1038103           LDL_0    1727600
  LDXA      1727600         ADD        982801           LDXA     1727600
  CGT        982800         AND        982800           STO      1327700
  AND        982800         CGT        982800           ADD       982801
  HALT            1         LDA        799900           STL_2     982800
  ANEW            1         INC        799900           CGT       982800
  PRNS            1         LDV        399900           AND       982800
  PRNI            1         DSP             1           INC       799900
  DSP             1         PRNI            1           LDA_1     799800
                            PRNS            1           LDC_1     454902
                            ANEW            1           LDV       399900
                            HALT            1           STL        55101
                                                        LDL        55001
                                                        LDC_2        200
                                                        STL_1        200
                                                        LDL_3        101
                                                        LDA_3        100
                                                        STL_3          1
                                                        STL_0          1
                                                        HALT           1
                                                        ANEW           1
                                                        PRNS           1
                                                        PRNI           1
                                                        DSP            1
```