Computer Science 3 - 2012

Programming Language Translation

Practical for Week 20, beginning 3 September 2012 - 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 files on the servers - `PRAC20A.ZIP` (Java) and `PRAC20AC.ZIP` (C#).

Task 5

Task 5 was to hand-compile the first hailstone program into PVM code. Most people got a long way towards this.

Have a look at how I have commented this, using "high level" code, rather than detailed line by line commentary of the form "load address of X". Many of the submissions had "commentary" that was, frankly, almost useless. Try the following test for assembler code: Cover over the real code with a piece of paper and read only the comments. Does what you read make sense on its own? I maintain that it should. The easiest way to do this is by using a high level algorithmic notation.

```
; Finds the smallest initial term so that we get a
; run of more than limit terms in the hailstone series
    0 DSP   4    ; limit 0, start 1, length 2, term 3
    2 PRNS  "What is the length to exceed? "
    4 LDA   0
    6 INPI       ; read("What ... ", limit);
    7 LDA   1    ;
    9 LDC   0    ;
   11 STO        ; start = 0
   12 LDA   2    ;
   14 LDC   0    ;
   16 STO        ; length = 0
   17 LDA   2    ;
   19 LDV        ;
   20 LDA   0    ;
   22 LDV        ;
   23 CLE        ;
   24 BZE   101  ; while (length <= limit) {
   26 LDA   1    ;
   28 LDA   1    ;
   30 LDV        ;
   31 LDC   1    ;
   33 ADD        ;
   34 STO        ;   start = start + 1;
   35 LDA   3    ;
   37 LDA   1    ;
   39 LDV        ;
   40 STO        ;   term = start
   41 LDA   2    ;
   43 LDC   1    ;
   45 STO        ;   length = 1
   46 LDA   3    ;
   48 LDV        ;
   49 LDC   1    ;
   51 CNE        ;
   52 BZE   99   ;   while (term != 1) {
   54 LDA   2    ;   // perhaps direc to 17
   56 LDA   2    ;
   58 LDV        ;
   59 LDC   1    ;
   61 ADD        ;     length = length + 1
   62 STO        ;
   63 LDA   3    ;
   65 LDV        ;
   66 LDC   2    ;
   68 REM        ;
   69 LDC   0    ;
   71 CNE        ;
   72 BZE   88   ;     if (term % 2 != 0)
   74 LDA   3    ;
   76 LDC   3    ;
   78 LDA   3    ;
   80 LDV        ;
   81 MUL        ;
   82 LDC   1    ;
   84 ADD        ;
   85 STO        ;       term = 3 * term + 1;
   86 BRN   97   ;     else // perhaps direct to 46
   88 LDA   3    ;
   90 LDA   3    ;
   92 LDV        ;
   93 LDC   2    ;
   95 DIV        ;       term = term / 2
   96 STO        ;
   97 BRN   46   ;   // while
   99 BRN   17   ;
  101 PRNS  "To exceed"
  103 LDA   0
  105 LDV
  106 PRNI      ;  write("To exceed", limit);
  107 PRNS  " start from"
  109 LDA   1
  111 LDV
  112 PRNI      ;  write(" start from", start);
  113 PRNS  " to generate"
  115 LDA   2
  117 LDV
  118 PRNI      ;  write(" to generate", length, " terms");
  119 PRNS  " terms"
  121 HALT
```

Task 5 - Trapping overflow

Checking for overflow in multiplication and division was not always 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. Note that it is necessary to check for "division by zero" in the `rem` code as well!

```
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;
```

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;
```

Task 6 - 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 that are mentioned in several places).

```
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;
```

Note that the `PRNC` opcode outputs the character in a field width of 1, not 0 as most people tried. This has the effect that we can output characters without intervening spaces. Note also the way in which the value is forced "modulo 256" to become a valid ASCII value. I don't recall seeing anyone do this.

To build a really safe system there are further refinements we should make. It can be argued that we should not try to store a value outside ot 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 others to handle `STLC, STLC_0` 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;
```

Introducing opcodes to convert to lower case and check for a letter is simply done by using the methods from the Java `Character` wrapper class (notice the need for casting operations as well):

```
case PVM.low:           // toLowerCase
  push(Character.toLowerCase((char) pop()));
  break;
```

```
      case PVM.islet:         // isLetter
        tos = pop();
        push(Character.isLetter((char) tos) ? 1 : 0);
        break;
```

As an example of using the new input/output opcodes, here is the encryption program. Notice that we have had to hard-code 46 as the integer equivalent of character '.', of course, and similarly hard-coded 97 as the integer equivalent of 'a'.

```
; rot13 encryption of a text terminated with a period
; P.D. Terry, Rhodes University, 2012
   0  DSP    1  ; ch at 0
   2  LDA    0  ; repeat {
   4  INPC      ;   read(ch);
   5  LDA    0  ;
   7  LDA    0  ;
   9  LDV       ;
  10  LOW       ;
  11  STOC      ;   ch = lowercase(ch);
  12  LDA    0  ;
  14  LDV       ;
  15  ISLET     ;
  16  BZE    36 ;   if (isletter(ch))
  18  LDA    0  ;
  20  LDC    97 ;
  22  LDA    0  ;
  24  LDV       ;
  25  LDC    97 ;
  27  SUB       ;
  28  LDC    13 ;
  30  ADD       ;
  31  LDC    26 ;
  33  REM       ;
  34  ADD       ;
  35  STOC      ;      ch = 'a' + (ch-'a'+13) % 26;
  36  LDA    0  ;
  38  LDV       ;
  39  PRNC      ;   write(ch)
  40  LDA    0  ;
  42  LDV       ;
  43  LDC    46 ;
  45  CEQ       ;
  46  BZE    2  ; } until (ch == '.');
  48  HALT      ; System.Exit(0);
```

Task 7 - Improving the opcode set

This is straightforward, if a little tedious, and 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 enumeration of the machine opcodes:

```
    public static final int // Machine opcodes
      ...
      ldl    = 63,     // ++++++++++++++++++  additions
      stl    = 64,
      lda_0  = 65,
      ...
```

We must add to the *switch* statement in the `trace` and `listCode` methods (several submissions missed this):

```
    static void trace(OutFile results, int pcNow) {
      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 be performed on the new opcodes too (several submissions missed this).

```
case PVM.ldc_m1:        // push constant -1
  push(-1);
  break;
case PVM.ldc_0:         // push constant 0
  push(0);
  break;
case PVM.ldc_1:         // push constant 1
  push(1);
  break;
...

case PVM.lda_0:         // push local address 0
  adr = cpu.fp - 1;
  if (inBounds(adr)) push(adr);
  break;
case PVM.lda_1:         // push local address 1
  adr = cpu.fp - 2;
  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.ldl_0:         // push value of local variable 0
  adr = cpu.fp - 1;
  if (inBounds(adr)) push(mem[adr]);
  break;
case PVM.ldl_1:         // push value of local variable 1
  adr = cpu.fp - 2;
  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.stl_0:         // pop to local variable 0
  adr = cpu.fp - 1;
  if (inBounds(adr)) mem[adr] = pop();
  break;
case PVM.stl_1:         // pop to local variable 1
  adr = cpu.fp - 2;
  if (inBounds(adr)) mem[adr] = pop();
  break;
```

We must add to the method that lists out the code (several submissions missed this). :

```
public static void listCode(String fileName, int codeLen) {
    ...
   case PVM.brn:
   case PVM.ldc:
   case PVM.ldl: // ++++++++++++++++++  addition
   case PVM.stl: // ++++++++++++++++++  addition
     i = (i + 1) % memSize; codeFile.write(mem[i]);
     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.

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

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

Finally we must add to the section that initializes the mnemonic lookup table:

```
    public static void init() {
      ...
      mnemonics[PVM.ldl]    = "LDL";    // ++++++++++++++++  additions
      mnemonics[PVM.stl]    = "STL";
      mnemonics[PVM.lda_0]  = "LDA_0";
      ...
```

Here is the hailstone program recoded using these new opcodes (and using the `INC` code where appropriate).

```
; Finds the smallest initial term so that we get a
; run of more than limit terms in the hailstone series
    0 DSP   4    ; limit 0, start 1, length 2, term 3
    2 PRNS  "What is the length to exceed? "
    4 LDA_0
    5 INPI       ; read("What ... ", limit);
    6 LDC_0      ;
    7 STL_1      ; start = 0;
    8 LDC_0      ;
    9 STL_2      ; length = 0;
   10 LDL_2      ;
   11 LDL_0      ;
   12 CLE        ;
   13 BZE   51   ; while (length <= limit) {
   15 LDA_1      ;
   16 INC        ;   start++;
   17 LDL_1      ;
   18 STL_3      ;   term = start;
   19 LDC_1      ;
   20 STL_2      ;   length = 1;
   21 LDL_3      ;
   22 LDC_1      ;
   23 CNE        ;
   24 BZE   49   ;   while (term != 1) {
   26 LDA_2      ;
   27 INC        ;     length++;
   28 LDL_3      ;
   29 LDC_2      ;
   30 REM        ;
   31 LDC_0      ;
   32 CNE        ;
   33 BZE   43   ;     if (term % 2 != 0)
   35 LDC_3      ;
   36 LDL_3      ;
   37 MUL        ;
   38 LDC_1      ;
   39 ADD        ;
   40 STL_3      ;       term = 3 * term + 1;
   41 BRN   47   ;     else
   43 LDL_3      ;
   44 LDC_2      ;
   45 DIV        ;
   46 STL_3      ;       term = term / 2;
   47 BRN   21   ;   // while
   49 BRN   10   ;
   51 PRNS  "To exceed"
   53 LDL_0
   54 PRNI       ;  write("To exceed", limit);
   55 PRNS  " start from"
   57 LDL_1
   58 PRNI       ;  write(" start from", start);
   59 PRNS  " to generate"
   61 LDL_2
   62 PRNI       ;  write(" to generate", length, " terms");
   63 PRNS  " terms"
   65 HALT       ;  System.exit(0);
```

Task 9 - Do "improvements" necessarily make things "better"?

Surprisingly, no. In the prac worksheet the suggestion was made that you study the original source to see that the original opcodes had been mapped onto the numbers 30 .. 62. This meant that you could map the new opcodes onto a set of numbers below 30, or above 62. In the prac solution kit you will find four versions of the interpreter in which these ideas are tried out.

The following table shows various timings obtained on two translations of the `HAIL2.PAV` program, which was run for an upper limit of 250. This is not an exhaustive test - observing the output one is soon aware that most of the number crunching happens for the high values of limit, and the programs slow quite dramatically as this limit is approached (the underlying algorithm is very much "brute force").

The "checks removed" figures were obtained using variations of the interpreter source in which all the checks that CPU.SP remained in bounds had been suppressed, as well as the calls to next, push and pop (their effect was achieved by "inlining" the equivalent code. The solution kits show this in detail.) In all cases, suppressing the checks improved performance - one can see that an insistence on safety results in a considerable loss of run-time efficiency.

But apart from that, the behaviour may seem anomalous. `H1.PVM` did not use any of the "new" opcodes, and it is not at all obvious why there should be such a marked difference between the timings when these new opcodes were mapped to different ranges. It must depend quite heavily on the way in which the switch statement is implemented in the JVM. `H2.PVM` shows a sort of inverse behaviour - better performance when the new opcodes were mapped onto lower numbers.

This behaviour - when we added new opcodes "before" any ones that are used by `H1.PVM` the system slowed down, whereas when we added new opcodes "before" existing ones and then used these new ones (as we did in `H2.pvm`) the system sped up - is consistent with a suggestion that the JVM implementation of the *switch* statement that forms the heart of the interpretation must incorporate some sort of possibly linear search through a branch table - and the sooner an opcode is found, the better.

In a really serious implementation of an interpreter it would be worth carrying out further experiments to determine the optimal mapping, based, for example, on benchmarks carried out on a variety of programs and keeping statistics of which opcodes were actually the most heavily used. I was delighted some years agot tofind that one group had actually tried something like this on a simple program in which they had deliberately used a very limited set of opcodes. No such initiative this year, however.

The timings here were obtained a few years back on a 32 bit XP system. They were done fairly roughly with a stopwatch; one should really have run the simulations many times over and probably excluded the IO operations, but the fact that performance is inherently dependent on the efficiency of the interpreter is plain to see. Thanks to one group who provided a useful batch file for timing programs, which will be added to the arsenal of tools for future courses.

Several submissions suggested that the differences could be explained away by the longer list of opcodes and the (relatively) slow lookup process that forms the basis of the `opCode` method in the `PVM.java` file (at least, that is what I think the authors were trying to say; some explanations were very badly expressed!). But this has nothing to do with it - that method is used by the *assembly* process when the source code is read in, and not at all by the *interpretation/execution* process when the program is "run".

```
maxLimit = 250

                              H1.PVM        H2.PVM

Opcode set                    Original      Optimized
High numbers                    9.93          10.44     105%
High numbers, checks removed    5.03           7.35     145%
Low numbers                    14.83           9.67      65%
Low numbers, checks removed    10.44           6.58      63%
```

I ran the simulations again using C# implementations of the system - the source code is to all intents and purposes identical:

```
                              H1.PVM        H2.PVM

Opcode set                    Original      Optimized
High numbers                   14.57          10.18      69%
High numbers, checks removed    6.58           4.26      65%
Low numbers                    14.83          10.44      70%
Low numbers, checks removed     7.09           4.00      56%
```

Interestingly, the C# system is a bit "faster" than the Java one when the opcodes are mapped onto low numbers, and there is far less variation in timing between the "high" and "low" number mappings of the opcodes.

The effects of using a better algorithm (`HAIL3.PAV`) are quite dramatic. I used a value of `maxLimit` of 300 and obtained the following timings

```
                              Java          C#

High numbers                    4.78          4.52
High numbers, checks removed    3.74          2.19
Low numbers                     4.52          4.78
Low numbers, checks removed     3.40          2.45
```

Once again suppressing runtime checking speeds the process up (more so for C# than for Java), but the C# system is not always faster than the Java one.

Task 8 - Nothing like practice to make things perfect! - Arrays in the PVM

This example aimed to demonstrate the use of the Boolean and array handling opcodes. Here is a solution, also making use of the new opcodes. Note that there is no need to introduce another variable to store `maxWorker` as many submissions did. And I had rather hoped you would try this with the extended opcode set!

```
; Track workers as they clock in and out of work
; atWork 0; worker 1
  0 DSP    2            ;
  2 LDC    100          ;
  4 ANEW                ; bool[] atWork =
  5 STL_0               ;   new bool[maxWorker];
  6 LDC_0               ;
  7 STL_1               ; int worker = 0
  8 LDL_1               ;
  9 LDC    100          ;
 11 CLT                 ;
 12 BZE    23           ; while (worker<maxWorker) {
 14 LDL_0               ;
 15 LDL_1               ;
 16 LDXA                ;
 17 LDC_0               ;
 18 STO                 ;   atWork[worker] = false;
 19 LDA_1               ;
 20 INC                 ;   worker++;
 21 BRN    8            ; }
                        ; repeat {
 23 PRNS   "Worker? ... ;
 25 LDA_1               ;
 26 INPI                ;   read("...' , worker);
 27 LDL_1               ;
 28 LDC_0               ;
 29 CGT                 ;
 30 LDL_1               ;
 31 LDC    100          ;
 33 CLT                 ;
 34 AND                 ;   if ((worker > 0)
 35 BZE    42           ;     && (worker<maxWorker))
 37 LDL_0               ;
 38 LDL_1               ;
 39 LDXA                ;
 40 LDC_1               ;
 41 STO                 ;     atWork[worker] = true;
 42 LDL_1               ;
 43 LDC_0               ;
 44 CLT                 ;
 45 BZE    67           ;   if (worker < 0)
 47 LDL_0               ;
 48 LDL_1               ;
 49 NEG                 ;
 50 LDXA                ;
 51 LDV                 ;
 52 NOT                 ;
 53 BZE    61           ;     if (!atWork[-worker])
 55 LDL_1               ;       write(worker)
 56 PRNI                ;
 57 PRNS   " ...        ;       write(" has ...
 59 BRN    67           ;   } else
 61 LDL_0               ;
 62 LDL_1               ;
 63 NEG                 ;
 64 LDXA                ;
 65 LDC_0               ;
 66 STO                 ;    atWork[-worker] = false;
 67 LDL_1               ;
 68 LDC    100          ;
 70 CGE                 ; } until
 71 BZE    23           ;    (worker>=maxWorker);
 73 PRNS   "The ...     ; write("The following ...
 75 LDC_0               ;
 76 STL_1               ; worker = 0;
 77 LDL_1               ;
 78 LDC    100          ;
 80 CLT                 ;
 81 BZE    95           ; while (worker<maxWorker) {
 83 LDL_0               ;
 84 LDL_1               ;
 85 LDXA                ;
 86 LDV                 ;
 87 BZE    91           ;   if (atWork[worker])
 89 LDL_1               ;     write(worker);
 90 PRNI                ;
 91 LDA_1               ;
 92 INC                 ;    worker++;
 93 BRN    77           ; } // while
 95 HALT                ; System.exit(0);
```