Computer Science 3 - 2016

Programming Language Translation

Practical for Week 2, beginning 25 July 2016 - 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.

```
 0   DSP     3     ; v0 is x, v1 is y, v2 is z
 2   PRNS    "   X     Y     Z     X OR !Y AND Z\n"
 4   LDA     0     ;
 6   LDC     0     ;                                  46   LDA     2     ;
 8   STO           ; x = false;                       48   LDA     2     ;
 9   LDA     1     ; repeat                           50   LDV           ;
11   LDC     0     ;                                  51   NOT           ;
13   STO           ;   y = false;                     52   STO           ;         z = ! z;
14   LDA     2     ;   repeat                         53   LDA     2     ;
16   LDC     0     ;                                  55   LDV           ;
18   STO           ;     z = false;                   56   NOT           ;       until (!z);
19   LDA     0     ;       repeat                     57   BZE     19    ;
21   LDV           ;                                  59   LDA     1     ;
22   PRNB          ;         write(x);                61   LDA     1     ;
23   LDA     1     ;                                  63   LDV           ;
25   LDV           ;                                  64   NOT           ;
26   PRNB          ;         write(y);                65   STO           ;       y = !y;
27   LDA     2     ;                                  66   LDA     1     ;
29   LDV           ;         write(z);                68   LDV           ;
30   PRNB          ;                                  69   NOT           ;
31   LDA     0     ;;;                                70   BZE     14    ;     until (!y);
33   LDV           ;;;                                72   LDA     0     ;
34   LDA     1     ;;;                                74   LDA     0     ;
36   LDV           ;;;     not(y)                     76   LDV           ;
37   NOT           ;;;                                77   NOT           ;
38   LDA     2     ;;;                                78   STO           ;     x = !x;
40   LDV           ;;;                                79   LDA     0     ;
41   AND           ;;;     (not y and z)              81   LDV           ;
42   OR            ;;;      x or (not y and z)        82   NOT           ;
43   PRNB          ;         write(x || !y && z);    83   BZE     9     ;   intil (!x);
44   PRNS    "\n"  ;         write("\n");             85   HALT          ;
```

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 the suggested test function we get

```
x OR (NOT y  AND z)   ≡  if x  then true    else ( NOT y AND z )
                      ≡  if x  then true    else ( if NOT y then z   else false)
```

The code that must be executed has to leave the truth value of `x or !y and z` on the top of the stack according to the little algorithm shown here thus becomes (see highlighted words 31 through 53).

Admittedly this has more code (22 words) than the binary operator code in the origina (12 words). However, short-circuited Boolean evalueations is so much better that it is worth developing special opcodes to achieve it, as we shall see later in the course.

```
 0   DSP     3      ; v0 is x, v1 is y, v2 is z
 2   PRNS    "   X     Y     Z     X OR !Y AND Z\n"
 4   LDA     0      ;
 6   LDC     0      ;
 8   STO            ; x = false;
 9   LDA     1      ; repeat
11   LDC     0      ;
13   STO            ;   y = false;
14   LDA     2      ;   repeat
16   LDC     0      ;
18   STO            ;     z = false;
19   LDA     0      ;     repeat
21   LDV            ;
22   PRNB           ;       write(x);
23   LDA     1      ;
25   LDV            ;
26   PRNB           ;       write(y);
27   LDA     2      ;
29   LDV            ;       write(z);
30   PRNB           ;
31   LDA     0      ;;;
33   LDV            ;;;     (x?)
34   BZE    40      ;;;     if x false goto 40
36   LDC     1      ;;;     then push (true)
38   BRN    53      ;;;     and short circuit
40   LDA     1      ;;;
42   LDV            ;;;     (y?)
43   NOT            ;;;     (!y)
44   BZE    51      ;;;     if (false) short circuit
46   LDA     2      ;;;
48   LDV            ;;;     (z)
49   BRN    53      ;;;     z nails it - push z
51   LDC     0      ;;;     else push (false)
53   PRNB           ;;;     write(x || !y && z);
54   PRNS    "\n"   ;       write("\n");
56   LDA     2      ;
58   LDA     2      ;
60   LDV            ;
61   NOT            ;
62   STO            ;       z = ! z;
63   LDA     2      ;
65   LDV            ;
66   NOT            ;     until (!z);
67   BZE    19      ;
69   LDA     1      ;
71   LDA     1      ;
73   LDV            ;
74   NOT            ;
75   STO            ;     y = !y;
76   LDA     1      ;
78   LDV            ;
79   NOT            ;
80   BZE    14      ;   until (!y);
82   LDA     0      ;
84   LDA     0      ;
86   LDV            ;
87   NOT            ;
88   STO            ;   x = !x;
89   LDA     0      ;
91   LDV            ;
92   NOT            ;
93   BZE     9      ; until (!x);
95   HALT           ;
```

It is 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.

Task 4 - Execution overheads - part one

See discussion of Task 11 below.

Task 5 - We can always improve, while still keeping it simple

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, some very much better):

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

Task 6 - Coding the hard way

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. Note that *limit* was a literal constant, not a variable, which is what many people translated it. No damage was done, of course.

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, and which is rarely helpful to a reader). 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 list of positive numbers, determine frequency of each
; P.D. Terry, Rhodes University, 2016
    0 DSP      3
    2 LDA      1
    4 LDC      2000             limit = 2000 (toy problem)
    6 ANEW
    7 STO                       count = new int[limit];
    8 LDA      2
   10 LDC      0
   12 STO                       int i = 0;
   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
   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)
```

Task 7 - 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 commen - 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;
```

The original program, if given too long a sequence of non-zero numbers for the array to handle, would terminate with an array bounds error corectly trapped by the Push/Pop assembler. The same error would not be trapped by the Inline system, which gaily allows the `LDXA` opcode to wander wheresoever it likes. To fix this resuires 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 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 that are mentioned 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 ensure 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 should 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

```
case PVM.stoc:          // character checked store
  tos = mem[cpu.sp++]; mem[mem[cpu.sp++]] = tos;
  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 9 - Your lecturer - what's his case this time?

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

```
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.

```
      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;
```

Task 10 - 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 for you to add them yourselves).

```
        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

```
        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
```

Modifying the code on the lines shown here 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.stl:           // store local value
  mem[cpu.fp - 1 - mem[cpu.pc++]] = mem[cpu.sp++];
  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
case PVM.stl:           // store local value
  mem[cpu.fp - 1 - mem[cpu.pc++]] = mem[cpu.sp++];
  break;
```

In improving the character frequency counter, 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, 2016
; 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                   count[ch] = 0;
   26 LDA      0
   28 INC                   ch++;
   29 BRN      11         }
   31 LDA      0
   33 INPC                read(ch);
   34 LDL      0
   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                  read(ch);
   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 11 - 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 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 between 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 withing 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 have thought.

```
 550 primes                 550 primes

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%)


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
```