Computer Science 3 - 2015

Programming Language Translation

Practical for Week 2, beginning 7 September 2015 - 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.

```
; Read a zero-terminated list of numbers and write it backwards (say what it does)
; P.D. Terry, Rhodes University, 2015                          (and who was responsible)

   0 DSP      3   0 list, 1 i, 2 n      42 LDA      1
   2 LDA      0                         44 LDA      1
   4 LDC      10  const max = 10;       46 LDV
   6 ANEW         int[] list =          47 LDC      1
   7 STO            new int [max];      49 ADD
   8 LDA      1                         50 STO            i = i + 1;
  10 LDC      0                         51 LDA      2
  12 STO           i = 0;               53 INPI           read(n);
  13 LDA      2                         54 BRN      16  } // while
  15 INPI          read(n);             56 LDA      1
  16 LDA      2                         58 LDV
  18 LDV                                59 LDC      0
  19 LDC      0                         61 CGT          while (i > 0) {
  21 CNE           while (n ! 0         62 BZE      84
  22 LDA      1                         64 LDA      1
  24 LDV                                66 LDA      1
  25 LDC      10     // max             68 LDV
  27 CLT                                69 LDC      1
  28 AND             && i < max) {      71 SUB
  29 BZE      56                        72 STO            i = i - 1;
  31 LDA      0                         73 LDA      0
  33 LDV                                75 LDV
  34 LDA      1                         76 LDA      1
  36 LDV                                78 LDV
  37 LDXA                               79 LDXA
  38 LDA      2                         80 LDV
  40 LDV                                81 PRNI           write(list[i]);
  41 STO             list[i] = n;       82 BRN      56  } // while
                                        84 HALT
```

It is easy to see that this does not use short circuit evaluation of Boolean expressions, as it uses AND, which is an infix operator that requires its two operands both to have been evaluated and pushed onto the expression stack. However, it is easy to eliminate the AND (and, in the code on the right, even a comparison)"

```
16 LDA      2                         16 LDA      2                         16 LDA      2
18 LDV                                18 LDV                                18 LDV
19 LDC      0                         19 LDC      0                         19 BZE      54
21 CNE           while (n != 0        21 CNE           while (n != 0        21 LDA      1    while (n != 0
22 LDA      1                         22 BZE      57                        22 LDV
24 LDV                                24 LDA      1                         24 LDC      10     // max
25 LDC      10     // max             26 LDV             // max             26 CLT             && i < max)
27 CLT                                27 LDC      10                        27 BZE      54
28 AND             && i < max) {      29 CLT             && i < max) {     29 LDA      0
29 BZE      56                        30 BZE      57                           ...
31 LDA      0                         32 LDA      0
   ...                                   ...
```

(Other examples of short circuit coding are to be found in the solution to this week's test.)

Task 4 - Execution overheads - part one

See discussion of Task 9 below.

Task 5 - Palindromic sequences

Task 5 was to hand-compile the numerical palindrome checking into PVM code. Most people got a long way towards this.

Again, 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.

```
; Read a sequence of numbers and report
; whether they form a palindromic
; sequence (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, 2015
 0 DSP      6
 2 LDA      5
 4 LDC      100
 6 ANEW
 7 STO            int [] list = new int [10];
 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    // start while test
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             // list[low]
88 LDA      5
90 LDV
91 LDA      2
93 LDV
94 LDXA
95 LDV             // list[high[
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
```

Task 6 - Trapping overflow and other pitfalls

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;
```

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;
  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 palindrome checker program, when given too long a sequence of non-zero numbers for the array to handle, terminated with an array bounds error corectly trapped by the Push/Pop assembler. The same error was not 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 PVM Inline 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 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;
```

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;
```

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;
```

or for the "inline" assembler

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

Introducing opcodes to convert to lower case and check for a letter 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.islet:         // isLetter
  tos = Pop();
  Push(Char.IsLetter((char) tos) ? 1 : 0);
  break;
```

or for the "inline" assembler

```
case PVM.low:           // toLowerCase
  mem[cpu.sp] = Char.ToLower((char) mem[cpu.sp]);
  break;
case PVM.islet:         // isLetter
  mem[cpu.sp] = Char.IsLetter((char) mem[cpu.sp]) ? 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, 2015
   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  ;
  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 8 - 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 *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_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.stlc:          // character checked pop to local variable
        tos = Pop(); adr = cpu.fp - 1 - Next();
        if (InBounds(adr))
          if (tos >= 0 && tos <= maxChar) mem[adr] = tos;
          else ps = badVal;
        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;
      ...
```

or for the "inline" assembler

```
      case PVM.ldc_0:         // push constant 0
        mem[--cpu.sp] = 0;
        break;
      case PVM.ldc_1:         // push constant 1
        mem[--cpu.sp] = 1;
        break;
      ...
      case PVM.lda_0:         // push local address 0
        mem[--cpu.sp] = cpu.fp - 1;
        break;
      case PVM.lda_1:         // push local address 1
        mem[--cpu.sp] = cpu.fp - 2;
        break;
      ...
      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
        mem[--cpu.sp] = mem[cpu.fp - 1];
        break;
      case PVM.ldl_1:         // push value of local variable 1
        mem[--cpu.sp] = mem[cpu.fp - 2];
        break;
      ...
      case PVM.stl:           // store local value
        mem[cpu.fp - 1 - mem[cpu.pc++]] = mem[cpu.sp++];
        break;
      case PVM.stlc:          // store local value
        mem[cpu.fp - 1 - mem[cpu.pc++]] = mem[cpu.sp++];
        break;
      case PVM.stl_0:         // pop to local variable 0
        mem[cpu.fp - 1] = mem[cpu.sp++];
        break;
      case PVM.stl_1:         // pop to local variable 1
        mem[cpu.fp - 2] = mem[cpu.sp++];
        break;
      ...
```

We must add to the method that lists out the code (several submissions may have 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;
```

or for the "inline" assembler

```
      case PVM.inc:           // ++
        mem[mem[cpu.sp++]]++;
        break;
      case PVM.dec:           // --
        mem[mem[cpu.sp++]]--;
        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 are the encoding program and the character palindrome programs recoded using these new opcodes.

The palindrome program has also been optimised so as to terminate the checking loop as quickly as possible:

```
; Read a sequence of characters terminated by a period and report whether
; they form a palindrome (one that reads the same from each end)
;
; Examples:   too hot to hoot.  is palindromic
;             1234432.          is non-palindromic
; P.D. Terry, Rhodes University, 2015

    0  DSP      6
    2  LDC      100
    4  ANEW
    5  STL      5     char [] str = new char [100];
    7  LDC_0
    8  STL_0          n = 0;
    9  LDA_3
   10  INPC           read(ch)
   11  LDL_3
   12  LDC      46
   14  CNE
   15  BZE      36    while (ch != ',') {
   17  LDL_3
   18  LDC      32
   20  CGT
   21  BZE      32      if (ch > ' ') {
   23  LDL      5
   25  LDL_0
   26  LDXA
   27  LDL_3
   28  LOW
   29  STOC              str[n] = lower(ch);
   30  LDA_0
   31  INC               n++;
   32  LDA_3           }
   33  INPC            read(ch);
   34  BRN      11    } // while
   36  LDC_1
   37  STL      4     isPalinDrome = true;
   39  LDC_0
   40  STL_1          low = 0;
   41  LDL_0
   42  LDC_1
   43  SUB
   44  STL_2          high = n -1;
   45  LDL      4
   47  NEG
   48  BZE      77    while (isPalindrome
   50  LDL_1
   51  LDL_2
   52  CLT
   53  BZE      77      && low < high) {
   55  LDL      5
   57  LDL_1
   58  LDXA
   59  LDV                // str[low]
   60  LDL      5
   62  LDL_2
   63  LDXA
   64  LDV                // str[high]
   65  CNE              if (str[low] != str[high])
   66  BZE      71
   68  LDC_0              isPalindrome = false;
   69  STL      4
   71  LDA_1
   72  INC              low++;
   73  LDA_2
   74  DEC              high--;
   75  BRN      45    } // while
   77  LDL      4
   79  BZE      85    if (isPalindrome)
   81  PRNS     "Palindromic string"
   83  BRN      87    else
   85  PRNS     "Non-palindromic string"
   87  HALT
```

The encoding program has been optimized in several respects - can you see them all?

```
; rot13 encryption of a text terminated with a period
; P.D. Terry, Rhodes University, 2015
  0  DSP    1  ; ch at 0
  2  LDA_0     ; repeat {
  3  INPC      ;   read(ch);
  4  LDL_0
  5  LOW
  6  STLC   0  ;   ch = lowercase(ch);
  8  LDL_0
  9  ISLET
 10  BZE    25 ;   if (isletter(ch))
 12  LDC    97        // 'a'
 14  LDL_0
 15  LDC    84 ;      // 'a' - 13
 17  SUB
 18  LDC    26
 20  REM
 21  ADD
 22  I2C
 23  STLC   0  ;    ch = (char) ('a' + (ch-'a'+13) % 26);
 25  LDL_0
 26  PRNC      ;   write(ch)
 27  LDL_0
 28  LDC    46
 30  CEQ
 31  BZE    2  ; } until (ch == '.');
 33  HALT      ; System.Exit(0);
```

Task 9 - 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.

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		Sieve2.pvm	
ASM1 (Push/Pop - original)	0.73		-	
ASM2 (Inline - original)	0.30	(0.41)	-	
ASM3 (Push/Pop - extended)	0.72		0.55	
ASM4 (Inline - extended)	0.33	(0.45)	0.15	(0.27)

Laptop machine (XP-32)	Sieve1.pvm		Sieve2.pvm	
ASM1 (Push/Pop - original)	1.34		-	
ASM2 (Inline - original)	0.51	(0.38)	-	
ASM3 (Push/Pop - extended)	1.16		0.86	
ASM4 (Inline - extended)	0.51	(0.45)	0.26	(0.30)

The Desktop times were about 55-65% of those on the Laptop

The Inline times were between 38-45% of the Push/Pop system with the original limited opcode set.

The Inline times were between 27-30% of the Push/Pop system with the extended opcode set,

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

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


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