Computer Science 3 - 2016

Programming Language Translation

Practical 2, Week beginning 25 July 2016

Hand in this prac sheet *before* lunch time on your next practical day, correctly packaged in a transparent folder with your solutions and the "cover sheet". **Unpackaged and late submissions will not be accepted - you have been warned.** Please do NOT come to a practical and spend the first hour printing or completing solutions from the previous week's exercises. Since the practical will have been done on a group basis, please hand in one copy of the cover sheet for each member of the group. These will be returned to you in due course, signed by the marker.

Objectives:

In this practical you are to

- become familiar you with the workings of two simple machine emulators for the PVM pseudo-machine that we shall use frequently in the course.
- gain some experience with the machines, writing machine code for them, comparing them and extending them.

You will need this prac sheet and your text book. Copies of the prac sheet and of the Parva report are also available at `http://www.cs.ru.ac.za/courses/CSc301/Translators/trans.htm`.

Outcomes:

When you have completed this practical you should understand

- the opcode set for the Parva Virtual Machine (PVM);
- how to write and debug machine level code for the PVM;
- how to extend the PVM to incorporate new opcodes;
- why, and by how much, interpretive systems can vary in execution overhead.

To hand in:

This week you are required to hand in, besides the cover sheet:

- Listings of the final version of the assembler/emulator system you produce or (preferably) extracts showing only the extensions clearly (to save paper!) , and your solutions to the programming exercises below. (Use LPRINT, please.) *One listing/group please.*
- Additionally, electronic copies of source code for those exercises, using the electronic submission system.
- Discussion of the experiments in Tasks 4 and 11.

Keep the prac sheet and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

You are referred to the rules for practical submission which are clearly stated in our Departmental Handbook. However, for this course pracs must be posted in the "hand-in" box outside the laboratory and not given to demonstrators.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another group's or student's work as your own. Attempts to do this will result in (at best) a mark of zero and (at worst) severe disciplinary action and the loss of your DP. You are allowed - even encouraged - to work and study with other students, but if you do this you are asked to acknowledge that you have done so. You are expected to be familiar with the University Policy on Plagiarism, which you can consult on the university web site:

Task 1 - Creating a working directory and unpacking the prac kit

There are several files that you need, zipped up this week in the file `PRAC2.ZIP`.

- Immediately after logging on, get to the DOS command line level and log onto your file space.
- Check that you can see the I: drive. If not, ask a demonstrator for help.
- Copy the prac kit into a new directory/folder in this file space, from I: or the web page.

```
md  prac2
cd  prac2
copy  i:\csc301\trans\prac2.zip
unzip  prac2.zip
```

Task 2 - Build the assemblers

In the working directory you will find C# files that give you two minimal assemblers and emulators for the PVM stack machine (described in Chapter 4.7). These files have the names

`PVMAsm.cs`	a simple assembler
`PVMPushPop.cs`	an interpreter/emulator, making use of auxiliary Push and Pop methods
`PVMInLine.cs`	an interpreter/emulator, with the pushing and popping "inlined"
`Assem.cs`	a driver program

`PVMPushPop` incorporates rather more constraint checking than is found in `PVMLine`, and also has an option for doing a line-by-line trace of the code it is interpreting.

You compile and make two nominally equivalent assembler/interpreter systems by issuing the batch commands

`MAKEASM1`	make up a system ASM1.EXE using `PVMPushPop` as the PVM
`MAKEASM2`	make up a system ASM2.EXE using `PVMInLine` as the PVM

These take as input a "code file" in the format shown in the examples in section 4.5 and in the prac kit. Make up the minimal assembler/interpreters and, as a start, run these using a supplied small program:

`ASM1  lsmall.pvm`	this will prompt you for input and output files and for tracing options
`ASM2  lsmall.pvm`	
`ASM2  lsmall.pvm immediate`	this will enter the emulator immediately after compiling

Wow! Isn't Science wonderful? Try the interpretation with and without the trace option, and familiarize yourself with the trace output and how it helps you understand the action of the virtual machine (ASM1 only).

Task 3 - A look at PVM code

Consider the following gem of a Parva program which creates a truth table for a simple Boolean function.

```
void Main () {
/* Tabulate a simple Boolean function
   P.D. Terry, Rhodes University, 2016 */

  bool X, Y, Z;

  write("   X     Y     Z     X OR !Y AND Z\n");
  X = false;
  repeat
    Y = false;
    repeat
      Z = false;
      repeat
        write(X, Y, Z, X || !Y && Z, "\n");
        Z = ! Z;
      until (!Z); // again
      Y = ! Y;
    until (!Y); // again
    X = ! X;
  until (!X); // again
} // Main
```

You can compile and run this (`PARVA BOOL.PAV`) at your leisure to make quite sure that it works.

The Parva compiler supplied to you this week is not the same as last week - it only allows a single `Main()` function, but it includes "else" and the modulo "`%`" operator, supports a "repeat" … "until" statement, and allows increment and decrement operations like `i++` and `array[j]--`.

In the prac kit you will also find a translation of this program into PVM code (`BOOL.PVM`). Study this code and complete the following tasks:

```
 0   DSP     3
 2   PRNS    "   X     Y     Z     X OR !Y AND Z\n"
 4   LDA     0
 6   LDC     0                                    46   LDA     2
 8   STO                                          48   LDA     2
 9   LDA     1                                    50   LDV
11   LDC     0                                    51   NOT
13   STO                                          52   STO
14   LDA     2                                    53   LDA     2
16   LDC     0                                    55   LDV
18   STO                                          56   NOT
19   LDA     0                                    57   BZE     19
21   LDV                                          59   LDA     1
22   PRNB                                         61   LDA     1
23   LDA     1                                    63   LDV
25   LDV                                          64   NOT
26   PRNB                                         65   STO
27   LDA     2                                    66   LDA     1
29   LDV                                          68   LDV
30   PRNB                                         69   NOT
31   LDA     0                                    70   BZE     14
33   LDV                                          72   LDA     0
34   LDA     1                                    74   LDA     0
36   LDV                                          76   LDV
37   NOT                                          77   NOT
38   LDA     2                                    78   STO
40   LDV                                          79   LDA     0
41   AND                                          81   LDV
42   OR                                           82   NOT
43   PRNB                                         83   BZE     9
44   PRNS    "\n"                                 85   HALT
```

(a) How can you tell that the translation has not used short-circuit Boolean operations?

(b) Add commentary to the code that "matches" the Parva code fairly closely. Have a look at the `LSMALL.PVM` code example in the prac kit to see a "preferred" style of commentary, where the high level code appears as commentary on the low level code.

(c) What would you need to change if you wanted to make use of short-circuit Boolean operations? (You should test your ideas with the first of the two assemblers, ASM1).

(d) Suppose you wanted to produce the truth table to display 0 and 1 in place of false and true. What (simple) change will allow you to do this?

---- The (modified and suitably commented) BOOL.PVM file must be submitted for assessment.

Task 4 - Execution overheads - part one

In the prac kit you will find a translation `SIEVE1.PVM` of a cut down version of a prime-counting program `SIEVE.PAV` based on last week's exercises (the source code is also there, but is not printed here to save paper).

Run `SIEVE1.PVM` through both versions of the assemblers and obtain timings for a suitable upper limit (say 4000) and number of iterations (say 100) for the combinations:

Hint: The lab computers are very fast. You may have to alter those suggestions quite a bit to produce measurably distinct timings. Do this simply by changing the appropriate constants in the PVM file before you assemble it

Comment on the results. Are they what you expect? If not, why not?

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

Start off by considering the following masterpiece of a Parva program (COUNT0.PAV)

```
void main() {
// Read a list of positive numbers and determine the frequency
// of occurrence of each
// P.D. Terry, Rhodes University, 2016
  const
    limit = 2000;
  int
    item;                           // data item
  int[]
    count = new int[limit];         // the number of times each appears
  int i = 0;                        // loop to clear counts
  while (i < limit) {
    count[i] = 0;
    i = i + 1;
  }
  read("First number? ", item);
  while (item > 0) {                // terminate input with a result <= 0
    count[item] = count[item] + 1;  // increment appropriate count
    read("Next number (<= 0 stops) ", item);
  }
  i = 0;                            // loop to output numbers and counts
  while (i < limit) {
    if (count[i] > 0) write(i, count[i], "\n");
    i = i + 1;
  }
}
```

You can compile this (PARVA COUNT0.PAV) at your leisure to make quite sure that it works.

Firstly, decide why it is not a masterpiece, but an awful program. If you can't see why, try running it - run it anyway, and see if you can break it. What happens? (By "break" I mean "can you run it with some sort of data that allows it to work, and then run it with some data that makes it bomb out?")

Secondly, produce a better version of the program (COUNT1.PAV). Keep it simple! There is no need to change much.

Task 6 - Coding the hard way

Time to do some creative work at last. Task 6 is to produce an equivalent program to the Parva one (COUNT1.PAV), but written directly in the PVM stack-machine language (COUNT1.PVM). In other words, "hand compile" the Parva algorithm directly into the PVM machine language. You may find this a bit of a challenge, but it really is not too hard, just a little tedious, perhaps.

Health warning: if you get the logic of your program badly wrong, it may load happily, but then go beserk when you try to interpret it. You may discover that the interpreter is not so "user friendly" as all the encouraging remarks in the book might have led you to believe interpreters all to be. Later we may improve it quite a bit. (Of course, if your machine-code programs are correct you won't need to do so. As it has been said: "Any fool can write a translator for source programs that are 100% correct".)

The most tedious part of coding directly in PVM code is computing the destination addresses of the various branch instructions.

Hint: As a side effect of assembly, the ASM system writes a new file with a .COD extension showing what has been assembled and where in memory it has been stored. Study of a .COD listing will often give you a good idea of what the targets of branch instructions should have been.

---- The (suitably commented) COUNT.PVM file must be submitted for assessment.

Task 7 - Trapping overflow and other pitfalls

Several of the remaining tasks in this prac require you to examine the machine emulator to learn how it really works, and to extend it to improve some opcodes and to add others.

In the prac kit you will discover two programs deliberately designed to cause chaos. `DIVZERO.PVM` bravely tries to divide by zero, and `MULTBIG.PVM` embarks on a continued multiplication that soon goes out of range. Try assembling and interpreting them with both systems to watch disaster happen.

Now we can surely do better than that! Modify the interpreters (`PVMPushPop.cs` and `PVMInLine.cs`) so that they will anticipate division by zero or multiplicative overflow, and change the program status accordingly, so that users will be told the errors of their ways and not left wondering what has happened.

You will have to be subtle about this - you have to detect that overflow is going to occur *before* things "go wrong", and you must be able to detect it for negative as well as positive overflow conditions.

Hint: After you edit any of the source code for the assemblers you will have to issue the `MAKEASMx` *commands to recompile them, of course. It's easy to forget to do this and then wonder why nothing seems to have changed.*

Task 8 - Your lecturer is quite a character

If the PVM could only handle characters as well as integers and Booleans, we could write a variation on the program above that could count the frequency of occurrence of each capital letter in a piece of text. Something like this, if only the Parva compiler were extended to support it (later in the course, perhaps?)

```
void main() {
// Read a piece of text terminated with a period and determine the frequency
// of occurrence of each letter
// P.D. Terry, Rhodes University, 2016
  const
    limit = 256;                    // 256 characters in ASCII set
  char
    ch;                             // general data character
  int[]
    count = new int[limit];         // the number of times each appears
  int i = 0;                        // loop to clear counts
  while (i < limit) {
    count[i] = 0;
    i = i + 1;
  }
  read(ch);
  while (ch != '.') {               // terminate input with a full stop
    count[ch] = count[ch] + 1;      // increment appropriate count
    read(ch);
  }
  ch = 'A';                         // loop to output characters and counts
  while (ch <= 'Z') {
    if (count[ch] > 0) write(ch, count[ch], "\n");
    ch = (char) (ch + 1);
  }
}
```

Not a problem for the assembler system. All we need to do is add appropriate opcodes to our virtual machine -for example, `INPC` for reading a character and `PRNC` for writing a character - to open up exciting possibilities. Do this, and modify the earlier "integer" program to produce the equivalent of the code just given (`FREQCH.PVM`).

Hint: Adding "instructions" to the pseudo-machine is easy enough, but you must be careful to make sure you modify all the parts of the system that need to be modified. Before you begin, study the code in the definition of the stack machine carefully to see where and how the opcodes are defined, how they are mapped to the mnemonics, and in which switch/case statements they are used.

Task 9 - Your lecturer - what's his case this time?

If Parva were even less of a toy we might try to improve on that program still further. For example, we could treat upper and lowercase letters as equivalent, and, just for fun, write out the counts in reverse alphabetic order:

```
void main() {
// Read a piece of text terminated with a period and determine the frequency
// of occurrence of each letter
// P.D. Terry, Rhodes University, 2016
  const
    limit = 256;                    // 256 characters in ASCII set
  char
    ch;                             // general data character
  int[]
    count = new int[limit];         // the number of times each appears
  int i = 0;                        // loop to clear counts
  while (i < limit) {
    count[i] = 0;
    i++;
  }
  read(ch);
  while (ch != '.') {               // terminate input with a full stop
    count[toUpperCase(ch)]++;       // increment appropriate count
    read(ch);
  }
  ch = 'Z';                         // loop to output characters and counts
  while (ch >== 'A') {
    if (count[ch] > 0) write(ch, count[ch], "\n");
    ch--;
  }
}
```

This uses a method for converting characters to uppercase which is easily added to the machine by introducing a special opcode. It also uses the infamous `++` and `--` operators, which can be handled by special opcodes that take less space (and should take less time to execute) than the tedious sequences needed for code corresponding directly to code like `n = n + 1`.

Extend the machine and the assembler still further with opcodes `CAP`, `INC` and `DEC`, and modify your frequency checker program to use them.

Hint: Note that the assemblers have already been primed with the mappings from these mnemonics to integers, but, once again, you must be careful to make sure you modify all the parts of the system that need extending - you will have to add quite a bit to various switch statements to complete the tasks. Do this for both versions of the PVM.

Hint: Be careful. Think ahead! Don't limit your `INC` *and* `DEC` *opcodes to cases where they can handle statements like* `X++` *only. In some programs - even in this one - you might want to have statements like* `List[N+6]++`.

Task 10 - Improving the opcode set still further

Section 4.10.2 of the text discusses the improvements that can be made to the system by using load and store opcodes like `LDL  N` and `STL  N`.

Once again, these are almost trivially easy to add to the system. Do so, and fine tune the character frequency checker program still further.

---- The final assembler/emulators must be submitted for assessment, as must FREQCH.PVM. It would help if you simply printed only those parts of the interpreters that you have modified - large portions of the original will not need to change at all. Be careful to include the sections that deal with the run-time error trapping, however.

Task 11 - Execution overheads - part two

You might think it is pretty obvious that using as many `STL` and `LDL` opcodes as possible should make your programs smaller, faster, better. Carry out some experiments to see whether this is true and, if so, how big this effect is.

In the prac kit you will find 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.

Run `SIEVE2.PVM` through both versions of your modified assemblers and obtain timings for the same limit (say 4000) and number of iterations (say 100) as in Task 4.

Hint: The lab computers are very fast. You may have to alter those suggestions quite a bit to produce measurably distinct timings.

Comment on the results. Are they what you expect? If not, why not?

Hopefully by now you will have found that interpreters are quite easy to develop, but this prac should show you that they are not necessarily very "efficient". What changes could one make to improve the efficiency of the interpreter for the PVM still further? (If you are very keen you might try out some of your ideas, but I suppose that is wishful thinking. Sigh …)

Think carefully about all this. Please don't think you can write two lines of utter rubbish three minutes after you were supposed to hand the prac in, and try to bluff me that you know what is going on!

Have fun, and good luck.