Computer Science 3 - 2017

Programming Language Translation

Practical 2, Week beginning 24 July 2017

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 10.

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

Start off by considering the following gem of a Parva program which produces a truth table that illustrates de Morgan's famous laws.

```
void main () {
/* Demonstrate de Morgan's Laws
   P.D. Terry, Rhodes University, 2017 */

  bool X, Y;

  write("   X      Y    (X.Y)' X'+Y'  (X+Y)' X'.Y'\n\n");
  X = false;
  repeat
    Y = false;
    repeat
      write(X, Y, !(X && Y),  !X || !Y, !(X || Y), !X && !Y, "\n");
      Y = ! Y;
    until (!Y); // again
    X = ! X;
  until (!X); // again
} // main
```

You can compile this (`PARVA DEMORGAN.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 statements like `i++` and `array[j]--` (not within expressions, however).

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

```
                          ; try adding high level comments
                          ; and for a start include your names
 0   DSP     2
 2   PRNS    "   X      Y    (X.Y)\' X\'+Y\'  (X+Y)\' X\'.Y\'\n\n"
 4   LDA     0
 6   LDC     0
 8   STO
 9   LDA     1
11   LDC     0
13   STO
14   LDA     0
16   LDV
17   PRNB
18   LDA     1
20   LDV
21   PRNB
22   LDA     0
24   LDV
25   LDA     1
27   LDV
28   AND
29   NOT
30   PRNB
31   LDA     0
33   LDV
34   NOT
35   LDA     1
37   LDV
38   NOT
39   OR
40   PRNB
41   LDA     0
43   LDV
44   LDA     1
46   LDV
47   OR
48   NOT
49   PRNB
50   LDA     0
52   LDV
53   NOT
54   LDA     1
56   LDV
57   NOT
58   AND
59   PRNB
60   PRNS    "\n"
62   LDA     1
64   LDA     1
66   LDV
67   NOT
68   STO
69   LDA     1
71   LDV
72   NOT
73   BZE     14
75   LDA     0
77   LDA     0
79   LDV
80   NOT
81   STO
82   LDA     0
84   LDV
85   NOT
86   BZE     9
88   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 to evaluate the first two of the four Boolean expression in the `write` list? (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 commented) DEMORGAN.PVM file must be submitted by the end of the afternoon.

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 - Coding the hard way

Time to do some creative work at last. Task 5 is to produce an equivalent program to the Parva one below (`FACT.PAV`), but written directly in the PVM stack-machine language (`FACT.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.

```
void main () {
// Print a table of factorial numbers 1! ... 20!
// Your names here!
  const limit = 20;
  int n = 1;
  while (n <= limit) {
    int f = 1;
    int i = n;
    while (i > 0) {
      f = f * i;
      i = i - 1;
    }
    write(n, "! = ", f, "\n");
    n = n + 1;
  }
} // main
```

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 might improve it quite a bit. (Of course, if your machine-code programs are correct you won't need to do so. As has often 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 be.

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

Task 6 - 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 7 - Arrays

Start off by considering a further splendid exposition of the Parva programmer's art (`STUDENTS.PAV`)

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

  const StudentsInClass = 100;
  bool[] atWork = new bool[StudentsInClass];
  int student = 0;
  while (student < StudentsInClass) {
    atWork[student] = false;
    student = student + 1;
  }
  repeat {
    read("Student? (> 0 clocks in, < 0 clocks out, >= 100 terminates) ", student);
    if ((student > 0) && (student < StudentsInClass)) atWork[student] = true;
    if (student < 0)
      if (!atWork[-student]) write(student, " has not yet clocked in!\n");
      else atWork[-student] = false;
  } until (student >= StudentsInClass);
  write("The following students have still not clocked out\n");
  student = 0;
  while (student < StudentsInClass) {
    if (atWork[student]) write(student);
    student = student + 1;
  }
} // main
```

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

Next, decide why it is a rather bad program. If you can't see why, try running it - and see if you can break it. 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 produces meaningless results or even makes bomb out completely?" Go on to improve it - but keep the improvement quite simple and don't get carried away - remember my friend Dr Einstein's sage advice.

When you have done that, hand translate the improved `STUDENTS.PAV` ito PVM code (`STUDENTS.PVM`).

---- The (improved) STUDENTS.PAV and the (suitably commented) STUDENTS.PVM files must be submitted for assessment.

Task 8 - Your lecturer is quite a character

If the PVM and Parva could only handle characters as well as integers and Booleans, we could write a program like the exciting one below that reads a string of characters terminated with a period (full stop) and then writes it all in upper case SDRAWKCAB. (`SENTENCE.PAV`).

```
void main() {
// Read a piece of text terminated with a period and write it backwards in UPPER CASE.
// P.D. Terry, Rhodes University, 2017
  const
    limit = 256;                    // demonstration upper limit on sentence length
  char[]
    sentence = new char[limit];     // the number of times each appears
  int leng = 0;                     // read all characters
  repeat
    read(sentence[leng]); leng++;
  until (sentence[leng - 1] == '.');  // terminate input with a full stop
  while (leng > 0) {                // write characters in reverse order
    leng--;
    write(upper(sentence[leng]));
  }
} // main
```

Not a problem for the PVM and 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.

This program also assumes the existence of 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`

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.

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 9 - Improving the opcode set still further

Section 4.9 of the text discusses the improvements that can be made to the system by adding new single-word opcodes like `LDC_0` and `LDA_0` in place of double-word opcodes for frequently encountered operations like `LDC 0` and `LDA 0`, and for using load and store opcodes like `LDL N` and `STL N` (and, equivalently, opcodes like `LDL_0` and `STL_0` for frequently encountered special cases).

Enhance your PVM by incorporating the following opcodes:

```
LDL N     STL N
LDA_0     LDA_1     LDA_2     LDA_3
LDL_0     LDL_1     LDL_2     LDL_3
STL_0     STL_1     STL_2     STL_3
LDC_M1    LDC_0     LDC_1     LDC_2     LDC_3
```

Hint: Several of the above are very similar to one another, but, once again, you must be careful to make sure you modify all the parts of the system that need to be modified.

Try out your system by developing "improved" versions of `STUDENTS.PVM` and `SENTENCE.PVM`, say `STUDENTS1.PVM` and `SENTENCE1.PVM`, that uses these new opcodes.

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 10 - 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. Experiment to see whether this is true and, if so, how big this effect is.

In the prac kit you will find further translations `SIEVE2.PVM` and `SIEVE3.PVM` 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 in various ways.

Run `SIEVE2.PVM` and `SIEVE3.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: 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, can you suggest 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. Have fun, and good luck.