RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 2: week beginning 28 February 2022

Task 2 of this prac is due for submission by 9 am on Friday, 4th March 2022 through the first RUConnected (RUC) Prac 2 submission link. The remainder of the prac is due for submission through the second RUC submission link by 8 pm on the Wednesday before your next practical day.

Objectives

In this practical you will

- become familiar with the workings of a simple machine emulator for the PVM pseudo-machine that we shall use frequently in the course,
- gain some experience with the emulator, writing machine code for it, and extending it.

Copies of this handout and the Parva language report are available on the RUC course page.

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.

To hand in (max 35 marks)

This week you are required to hand in via the links on RUConnected:

- Electronic copy of your source code and commentary for task 2 [5 marks] by 9 am Friday 4th March.
- Electronic copies of your source code for tasks 3 [10 marks], 4 [4 marks], 5 [8 marks] and 6 [8 marks] by 8 pm Wednesday 9th March.

As before, some of your tasks may not be marked. For this practical, tutors will mark Tasks 2 and 4. I will mark some/all of tasks 3, 5, 6. Feedback for the tasks marked will be provided via RUConnected. 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. A rule not stated there, but which should be obvious, is that you are not allowed to hand in another student's or group'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 with suitable comments typed into *all* listings. Please ensure that all the names of the group members appear at the top of any source file handed in.

You are also expected to be familiar with the University Policy on Plagiarism.

Task 0 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`.

To ensure that all batch scripts provided in the kit execute correctly, copy the zipped prac kit from RUConnected into a new directory/folder in your file space (say `PRAC2`) and unzip the kit there.

This will create several other directories "below" the `prac2` directory:

```
J:\prac2
J:\prac2\Assem
J:\prac2\library
```

containing the Java classes for the I/O Library, and the Java sources for an assembler/interpreter system equivalent to the C# one described in Chapter 4. The differences between C# and Java are very minimal and it is hoped that you will have no problems in this regard.

Then run all batch scripts provided in the kit from within this new directory. Note these scripts need to be run from the command line – and in order to get used to doing this prior to the exam, I suggest you always use a command line terminal window when doing the practical work.

You can open a terminal window by typing `CMD.exe` into Windows RUN or searching for "cmd.exe" using Windows search.

Once you have a command line window open, move to the correct working directory (i.e. PRAC2) by executing the command:

```
CD prac2
```

Task 1 Build the assembler

In the working directory you will find Java files that give you a minimal assembler and emulator for the PVM stack machine (described in Chapter 4.7). These files have the names

`PVMAsm.java`	a simple assembler
`PVM.java`	an interpreter/emulator, making use of auxiliary Push / Pop methods
`Assem.java`	a driver program

You compile the assembler/interpreter system by issuing the batch command

`MAKEASM`	creates the required `class` files for the system with `assem.class` as the entry point

The compiled system takes 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/interpreter and, as a start, run this using a supplied small program by using the batch file ASM.bat:

`ASM lsmall.pvm`	this will prompt for input/output files and tracing options
`ASM lsmall.pvm immediate`	this will enter emulator immediately after compiling

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. When prompted for I/O files, hitting return means the program will use the default input/output, namely, STDIN and STDOUT.

Task 2 A look at PVM code (1+2+2 = 5 marks)

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 */
  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 at your leisure to make quite sure that it works. To compile, use the `Parva.exe` file and run it as: `Parva bool.pav`

The Parva compiler supplied to you this week is not the same as last week -- it only allows a single `Main()` function (no other functions), but it includes "else" and the modulo "%" operator, supports a "repeat" ... "until" statement, and allows increment and decrement operations like `i++` and `array[j]--` (but not within expressions).

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

(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. (Also see files `4-5-1.pvm` to `4-5-4.pvm` for examples of how NOT to comment your code and files `4-5-5.pvm` and `4-5-6.pvm` for additional examples of good ways of commenting PVM code.)

(c) What would you need to change if you wanted to make use of short-circuit Boolean operations? (You should test your ideas using the assembler, ASM). Make the changes to the PVM code.

The (modified and suitably commented) BOOL.PVM file must be submitted for assessment as well as the answer to question (a) by 9 am on Friday, 4th March 2022.

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

Task 3 Coding the hard way (10 marks)

Time to do some creative work at last. Task 3 is to produce an equivalent program to the Parva one below (`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.

```
  void main() {
// Read a list of positive numbers and determine the frequency
// of occurrence of each
// P.D. Terry, Rhodes University
  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
    if (item < limit)                 // if in range
      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;
  }
}
```

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 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 really be.

The (suitably commented) `COUNT1.PVM` file must be submitted for assessment.

Task 4 Trapping overflow and other pitfalls (4 marks)

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 to watch disaster happen.

Now we can surely do better than that! Modify the interpreter (`PVM.java`) so that it 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 problems are 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 assembler you will have to issue the `MAKEASM` command to recompile them, of course. It's easy to forget to do this and then wonder why nothing seems to have changed.

When submitting the changed code for the PVM interpreter, please ensure that you have commented the sections that you have changed, so that the markers can find the changes more easily. If you do not do this, marks will be deducted.

Task 5 Support for characters (8 marks)

If the PVM could only handle characters as well as integers and we could write a variation on the program above that could count the frequency of occurrence of *each letter* (ignoring case) 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 - see file FREQCH.PAV
// P.D. Terry, Rhodes University
  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 fullstop
    count[upper(ch)] = count[upper(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 a start, `INPC` for reading a character and `PRNC` for writing a character) to open up exciting possibilities. This example has also implied the availability of a method (`upper`) for converting characters to uppercase, which is easily added to the PVM by introducing a special opcode (say `CAP`). After making these extensions to the assembler, modify the earlier "integer" PVM program (task 4) to produce the equivalent of the code given above (FREQCH.PVM) and execute it.

Hint: Adding "instructions" to the PVM emulator 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 assembler has 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.

Please submit all changed files for the PVM interpreter and assembler as well as the `FREQCH.PVM` file. Please ensure that you have commented those parts of the system files that you have modified.

Task 6 Improving the opcode set still further (10 marks)

Section 4.10 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 both versions of your PVMs to incorporate the following opcodes:

```
LDL N     STL N     STLC N  (for storing char)
LDA_0     LDA_1
LDL_0     LDL_1
STL_0     STL_1
LDC_0     LDC_1
```

Ensure that you recompile your ASM system after making any changes to accommodate the new opcodes.

Hint: Several of the above changes are very similar to one another. 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.

Try out your system by running the files `sieve2.pvm` and `sieve3.pvm` which both use some of the enhanced set of opcodes and which calculate prime numbers using the Sieve of Eratosthenes algorithm. Note that `sieve1.pvm` uses the non-enhanced set of opcodes.

Please submit all changed files for the PVM interpreter and assembler. Please ensure that you have commented those parts of the system files that you have modified. (Note only one set of files needs to be submitted for Tasks 5 and 6).