RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 2: week beginning 10 August 2020

Task 2 of this prac is due for submission by 10 pm on Thursday, 13th August 2020 through the first prac2 RUConnected (RUC) submission link. The remainder of the prac is due for submission through the second RUC submission link by 2 pm on your next practical day.

Objectives

In this practical you will

- become familiar 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.

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 (25 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 10pm Thursday 13th August.
- Electronic copies of your source code for tasks 3 [6 marks], 4 [6 marks], and 5 [8 marks] by the start of the next practical session.

As before, not all your tasks will be marked. For this practical, tutors will mark Tasks 2 and 4. I will mark either task 3 or 5.

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 on *all* cover sheets and with suitable comments typed into *all* listings.

Note however, that “working with” someone does not imply that you hand in the same solution. The interpretation used in this course (and which determines whether a submission is plagiarised or not) is that you may discuss how you would approach the problem, but not actually work together on one coded solution. You are still expected to hand in your own solution at all times.

You are 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.

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.

To open the correct terminal window, do one of the following:

If you are using the HAMILTON lab machines or have set up the C# environment correctly on your own laptop, open a DOS CMD window, by typing `CMD.exe` into Windows RUN or searching for “cmd.exe” using Windows search.

Else if you are using your own laptop without the C# environment configured correctly, open a *Developer Command Prompt for VS* window by finding it as an option in the app list under the Visual Studio folder.

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 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 / 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 for input/output files and tracing options
ASM2 lsmall.pvm
ASM2 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 (ASM1 only). 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 reads a list of integers and writes it in reverse.

```
void Main () {
// Read a zero-terminated list of numbers and write it backwards
// P.D. Terry, Rhodes University, 2015
  const max = 10;
  int[] list = new int[max];
  int i = 0, n;
  read(n);
  while ((n != 0) && (i < max)) {  // input loop
    list[i] = n;
    i++;
    read(n);
  }
  while (i > 0) {                  // output loop
    i--;
    write(list[i]);
  }
} // Main
```

You can compile and run this (`PARVA REVERSE.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]`-- (not within expressions, however).

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

```
 0 DSP      3                    42 LDA      1
 2 LDA      0                    44 LDA      1
 4 LDC      10                   46 LDV
 6 ANEW                          47 LDC      1
 7 STO                           49 ADD
 8 LDA      1                    50 STO
10 LDC      0                    51 LDA      2
12 STO                           53 INPI
13 LDA      2                    54 BRN      16
15 INPI                          56 LDA      1
16 LDA      2                    58 LDV
```

```
18 LDV                          59 LDC      0
19 LDC      0                   61 CGT
21 CNE                          62 BZE      84
22 LDA      1                   64 LDA      1
24 LDV                          66 LDA      1
25 LDC      10                  68 LDV
27 CLT                          69 LDC      1
28 AND                          71 SUB
29 BZE      56                  72 STO
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
41 STO                          82 BRN      56
                                84 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. (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 with the first of the two assemblers, ASM1).

The (modified and suitably commented) REVERSE.PVM file must be submitted for assessment as well as the answer to question (a) by 10 pm on Thursday 13th August 2020.

Task 3 Coding the hard way (6 marks)

Time to do some creative work at last. Task 3 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 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) FACT.PVM file must be submitted for assessment.

Task 4 Trapping overflow and other pitfalls (6 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 with both systems to watch disaster happen. (You may also have noticed that the factorial program from Task 3, also seems to run into issues when computing factorials greater than 12, but the Parva application provided in the kit, which includes a more user-friendly interpreter alerts you to this by giving a “range error”).

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 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 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.

When submitting the changed code for the PVM interpreters, 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, 1 or more marks will be deducted.

Task 5 Support for characters (8 marks)

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 UPPERCASE.
  // P.D. Terry, Rhodes University
    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
```

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 emulator and the assembler still further with opcodes CAP, INC and DEC.

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 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]++;

Please submit all changed files for the PVM interpreters and assembler.