Computer Science 3 - 2013

Programming Language Translation

Practical for Week 20, beginning 9 September 2013

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

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 are slower than native code systems.

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, and your solutions to the programming exercises below. (Use LPRINT, please.)
- Preferably, electronic copies of your source code for those exercises, using the electronic submission system.
- Discussion of the experiments in Task 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 at:

`http://www.scifac.ru.ac.za/` or from `http://www.ru.ac.za/`

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 `PRAC20.ZIP`.

- Immediately after logging on, get to the DOS command line level by using the `Start -> Command prompt` option from the tool bar.

- Copy the prac kit into a newly created directory/folder in your file space

```
md  prac20
cd  prac20
copy  i:\csc301\trans\prac20.zip
unzip  prac20.zip
```

  This will create several other directories "below" the `prac20` directory:

```
J:\prac20
J:\prac20\Assem
J:\prac20\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.

- If UltraEdit is your editor of choice, the version in the lab can be configured to run various of the compilers easily, and it is possible to tweak it to run others in the same sort of way. *To get this to work properly, start UltraEdit from a command window by giving the command* `UEDIT32`*, rather than by clicking on the icon on the desktop.*

- You may prefer to work in C#. A version of the prac kit is available (`prac20c.zip`) with C# versions of the system and source files. It all works in much the same way.

Task 2

Start off by considering the following gem 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, 2013
  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 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 3 - Build the assembler

In the directory `prac20\Assem` you will find Java or C# files that give you a minimal assembler and emulator for the PVM stack machine (described in Chapter 4.7). The Java files have the names (similar ones for C#):

`PVMAsm.java`	a simple assembler
`PVM.java`	an interpreter/emulator very close to the one on page 63
`Assem.java`	a driver program

You can compile and make this assembler/interpreter system by issuing the batch command

```
MAKEASM
```

It takes as input a "code file" in the sort of format shown in the examples in section 4.5. There are three very simple example programs in the kit, so make up the minimal assembler/interpreter and try to run them with the `ASM` batch command:

```
ASM  hello.pvm
ASM  lsmall.pvm
ASM  divzero.pvm
```

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.

Task 4 - Coding the hard way

Time to do some creative work at last. Task 4 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 shall 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 be.

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

Task 5 - Trapping overflow

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` or `PVM.cs`) 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 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 assembler you will have to issue the `MAKEASM` *command to recompile it, of course. It's easy to forget to do this and then wonder why nothing seems to have changed.*

Task 6 - 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, 2013
  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 7 - 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, 2013
  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: 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 8 - 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 FREQCH.PVM file must be submitted for assessment.

Task 9 - Nothing like practice to make perfect!

Hand translate the following program into PVM code to produce a truth table for a simple Boolean expression.

```
void main () {
/* This won't compile, but the idea should be obvious
   P.D. Terry, Rhodes University, 2013 */

  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
}
```

---- The final BOOL.PVM must be submitted for assessment.

Task 10 - Safety first

We have already implied in Task 5 that interpreter systems should trap errors sensibly. In this task you are invited to make further modifications to the interpreter, **but first make a copy of the one produced so far, as it will be needed again in Task 11.**

You should have noticed that interpreting many of the opcodes involves a call to the auxiliary routines `Push()` and `Pop()`. Consider what would happen if calls to `Push()` and `Pop()` did not balance - for example if one were to assemble incorrect code like

```
LDA   1
STO
```

for which the interpreter would make only one call to `Push()` but two calls to `Pop()`, so that the stack would be corrupted, and chaos would ensue sooner or later. This sort of mess can be detected at runtime without too much trouble. Refer to Figure 4.3 (page 53), when it should be clear that `cpu.pc` should remain confined to the range `0 ... HeapBase` while `cpu.sp` should essentially remain confined to the range `cpu.hp ... cpu.fp`. Create a modified interpreter that incorporates these range checks. Chaos would also ensue if one tried to access an array element "out of range". This is discussed on pages 73 to 74; incorporate these checks into your interpreter.

Hint: You will have to think carefully about how to do this neatly. Avoid the urge to hack, or to cut and paste. Aim for elegance!

--- A listing of the final version of the assembler/interpreter must be submitted for assessment.

Task 11 - How do our systems perform?

In the kit you will find two versions of the infamous Sieve program written in PVM code. `S1.PVM` uses the original opcode set; `S2.PVM` uses the extended opcodes suggested in Task 8.

Run both versions through your two systems and obtain timings for a suitable upper limit (say 1000) and number of iterations (say 2000) for the combinations:

Original opcodes + interpreter with no bounds checks
Original opcodes + interpreter with the bounds checks of Task 10

Extended opcodes + interpreter with no bounds checks
Extended opcodes + interpreter with the bounds checks of Task 10

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

Comment on the results. Are they what you expect?

Interpreters are 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.