Computer Science 3 - 2012

Programming Language Translation

Practical for Week 20, beginning 3 September 2012

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, how 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 9.

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 by following the link from

`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 (`HAIL1.PAV`), the idea of which may be familiar:

```
void main () {
// Finds the smallest initial term so that we get a run of more than limit
// terms in the hailstone series
// P.D. Terry, Rhodes University, 2012

  int limit, start, length;
  read("What is the length to exceed? ", limit);
  start = 0; length = 0;
  while (length <= limit) {
    start = start + 1;
    int term = start;
    length = 1;
    while (term != 1) {
      length = length + 1;
      if (term % 2 != 0) term = 3 * term + 1; else term = term / 2;
    }
  }
  write("To exceed", limit, " start from", start , " to generate", length, " terms");
} // main
```

You can compile this (`PARVA HAIL1.PAV`) at your leisure to make quite sure that it works.

If you are observant you will note that this program has an "else" to the "if". The Parva compiler this week is not the same as last week - it only allows a single `main` function, but it includes "else" and the modulo "`%`" operator and it supports a "repeat" … "until" statement (see later examples).

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 given earlier (`HAIL1.PAV`), but written directly in the PVM stack-machine language (`HAIL1.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 as "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) HAIL1.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 simple text encryption program like the one below (`ENCODE.PAV`). In principle the Parva compiler could be extended to support it - later in the course, perhaps - but for the moment let us work towards this by adding some opcodes to the PVM and working at the assembler level. You will need ones for reading and writing a single character, and for converting an alphabetic character to lower case and for performing the Boolean operation implied by the `isLetter()` function:

```
void main() {
// rot13 encryption of a text terminated with a period
// P.D. Terry, Rhodes University, 2012

  char ch;
  repeat {
    read(ch);
    ch = lowerCase(ch);
    if (isLetter(ch)) ch = (char) ('a' + (ch - 'a' + 13) % 26);
    write(ch);
  }
  until (ch == '.');
}
```

After studying the code to see how the algorithm works, hand-compile this program into PVM code and get the system working. How would you decrypt messages that had been encrypted by this system?

---- The (suitably commented) ENCODE1.PVM file must be submitted for assessment.

Task 7 - 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
```

While you are at it, how about adding opcodes that would make for easier execution of statements like `parva++` or `hunger--`?

Hint: Adding "instructions" to the pseudo-machine is easy enough, especially as several of the above are very similar to one another, 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: Be careful. Think ahead! Don't limit your `INC` *and* `DEC` *opcodes to cases where they can handle only statements like* `X++`. *In some programs you might want to have statements like* `List[N+6]++`.

Try out your system by developing an "improved" version of `ENCODE.PVM` and `HAIL1.PVM`, say `ENCODE1.PVM` and `HAIL2.PVM`, that uses these new opcodes.

---- The final assembler/emulator must be submitted for assessment.

Task 8 - Nothing like practice to make perfect! - Arrays in the PVM

Hand translate the following program into PVM code to produce a system that will perform some array manipulations and work with Boolean types (`WORKERS.PAV`).

```
void main () {
// Track workers as they clock in and out of an organization
// P.D. Terry, Rhodes University, 2012

  const maxWorker = 100;
  bool[] atWork = new bool[maxWorker];
  int worker = 0;
  while (worker < maxWorker) {
    atWork[worker] = false;
    worker++;
  }
  repeat {
    read("Worker? (> 0 clocks in, < 0 clocks out, > 99 terminates) ", worker);
    if ((worker > 0) && (worker < maxWorker)) atWork[worker] = true;
    if (worker < 0)
      if (!atWork[-worker]) write(worker, " has not yet clocked in!\n");
      else atWork[-worker] = false;
  } until (worker >= maxWorker);
  write("The following workers have still not clocked out\n");
  worker = 0;
  while (worker < maxWorker) {
    if (atWork[worker]) write(worker);
    worker++;
  }
} // main
```

What happens if you supply too large a negative value as data?

---- The final `WORKERS.PVM` file must be submitted for assessment.

Task 9 - Do "improvements" necessarily make things "better"?

You might think it is pretty obvious that using as many one-word opcodes as possible should make your programs smaller, faster, better. Carry out some experiments to see how big this effect is.

In the kit you will find two versions of the program below written in PVM code. `H1.PVM` uses the original opcode set; `H2.PVM` uses the new opcodes and is consequently rather shorter. Run both versions through your system and obtain timings for a suitable upper vlue of `maxLimit` (say 100 - 200).

Your emulator will have had to assign enumeration values to the new opcodes. If you study the original source you will see that the original opcodes have been mapped onto the numbers 30 .. 62. You could map your new opcodes onto a set of numbers below 30, or above 62. Try the emulator both ways, and time the programs both ways. Write a paragraph or two presenting the results of this experiment, and try to explain the effects you observe.

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!

```
void main () {
// Draws up a table of starting values for the hailstone series that guarantee that a
// sequence will exceed a stipulated length
// P.D. Terry, Rhodes University, 2012

  int maxLimit, limit = 0;
  read("What is the largest length to exceed?", maxLimit);
  write("Exceed Start  Length\n");
  while (limit <= maxLimit) {
    int start = 0, length = 0;
    while (length <= limit) {
      start = start + 1;
      int term = start;
      length = 1;
      while (term != 1) {
        length = length + 1;
        if (term % 2 != 0) term = 3 * term + 1; else term = term / 2;
      }
    }
    write(limit, "\t", start , "\t", length, "\n");
    limit = limit + 1;
  }
} // main
```

Task 10 - Improving the algorithm can make things much better

The program above is capable of a great deal of improvement if you think about it. The variation below uses a technique sometimes called "memoization" (spelled that funny way) to try to improve its performance. You can find the PVM code for this program in the file `H3.PVM` (which uses the extended opcode set). Assemble the program, compare performance with the performance of `H1.PVM` and `H2.PVM` and comment on your findings.

```
void main () {
// Draws up a table of starting values for the hailstone series that guarantee that a
// sequence will exceed a stipulated length
// P.D. Terry, Rhodes University, 2012

  int maxLimit;
  read("What is the largest length to exceed? ", maxLimit);
  int[] startTable  = new int[maxLimit + 1];
  int[] lengthTable = new int[maxLimit + 1];
  int i = 0;                // set up initial lookup table
  while (i <= maxLimit) {
    startTable[i] = 0;
    i++;
  }

  write("Exceed Start  Length\n");

  int limit = 0;
  while (limit <= maxLimit) {
    int start = 0, length = 0;
    if (startTable[limit] != 0) {
      start = startTable[limit]; length = lengthTable[limit];
    } else {
      while (length <= limit) {
        start++;
        int term = start;
        length = 1;
        while (term != 1) {
          length++;
          if (term % 2 != 0) term = 3 * term + 1; else term = term / 2;
        }
      }
      i = limit;            // predict next entries in lookup table
      while ((i < length) && (i <= maxLimit)) {
        startTable[i]  = start;
        lengthTable[i] = length;
        i++;
      }
    }
    write(limit, "\t", start , "\t", length, "\n");
    limit++;
  }
} // main
```

Have fun, and good luck.