

Computer Science 3 - 2015

Programming Language Translation

Practical 2, Week beginning 7 September 2015

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 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 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 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 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
ASM2 lsmall.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 (ASM1 only).

Task 3 - A look at PVM code

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]--`.

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	

- How can you tell that the translation has not used short-circuit Boolean operations?
- 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.
- 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.

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.

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 (`PALIN.PAV`), but written directly in the PVM stack-machine language (`PALIN.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 sequence of numbers and report whether they form a palindromic
// sequence (one that reads the same from either end)
// Examples:  1 2 3 4 3 2 1  is palindromic
//           1 2 3 4 4 3 2  is non-palindromic
// P.D. Terry, Rhodes University, 2015

    int
        n,                // number of items
        low, high,       // indices of items to be compared
        item;            // latest item read
    bool
        isPalindrome;    // Boolean flag
    int [] list = new int [10]; // the list of items

    n = 0;
    read(item);
    while (item != 0) {
        list[n] = item;
        n = n + 1;
        read(item);
    }
    isPalindrome = true; // optimist
    low = 0; high = n - 1; // initial indices
    while (low < n - 1) { // sweep through the list
        if (list[low] != list[high])
            isPalindrome = false; // bad luck
        low = low + 1; high = high - 1; // adjust indices
    }
    if (isPalindrome) write("Palindromic sequence");
    else write("Non-palindromic sequence");
} // 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) PALIN.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 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.

The suggested program in Task 5 has a very small working array. What happens if you try to execute PALIN.PVM with a list of, say, 12 numbers? Try this with both assemblers - and then fix the broken one!

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 - 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 encrypts it in lower case, using the "rot 13" algorithm. (ENCOD.PAV).

```
void Main() {
// rot13 encryption of a text terminated with a period
// P.D. Terry, Rhodes University, 2015

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

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.

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.

This example has implied the availability of a method for converting characters to lowercase, which is easily added to the PVM by introducing a special opcode. We have also hinted at the desirability of supporting 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 an assignment statement like `n = n + 1`; as seen before.

Extend the machine and the assembler still further with opcodes ISLET, LOW, INC and DEC, and hand compile the rot13 program to use them.

Hint: Be careful. Think ahead! Don't limit your INC and DEC opcodes to cases where they can handle assignment statements like `X++`; only. In some programs you might want to have assignment statements like `List[N+6]++`; . Regard these as statements, and not as components of expressions, as they are in C#.

How do you decode a message that has been encrypted by this program?

Task 8 - 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 both versions of your PVMs to incorporate 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_0	LDC_1	LDC_2	LDC_3

Hint: Several of the above 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. Do this for both versions of the PVM.

Try out your systems by developing an "improved" version of PALIN.PVM, say PALINC.PVM that handles

sentences, not numbers, on the lines of

```
void Main () {
// Read a sequence of characters terminated by a period and report whether
// they form a palindrome (one that reads the same from either end)

// Examples:  too hot to hoot.  is palindromic
//            1234432.         is non-palindromic
// P.D. Terry, Rhodes University, 2015

    int
    n,                                // number of characters
    low, high;                        // indices of characters to be compared
    char
    ch;                                // latest character read
    bool
    isPalindrome;                    // Boolean flag
    char [] str = new char [100];     // the string to be checked

    n = 0;
    read(ch);
    while (ch != '.') {
        if (ch > ' ') {                // effectively ignore spaces etc
            str[n] = lower(ch);        // convert to lower case
            n++;
        }
        read(ch);
    }
    isPalindrome = true;              // optimist
    low = 0; high = n - 1;            // initial indices
    while (low < n - 1) {              // sweep through the string
        if (str[low] != str[high])
            isPalindrome = false;     // bad luck
        low++; high--;                // adjust indices
    }
    if (isPalindrome) write("Palindromic string");
    else write("Non-palindromic string");
} // Main
```

---- The final assembler/emulators must be submitted for assessment, as must PALINC.PVM. 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 9 - Execution overheads - part two

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 whether this is true and, if so, how big this effect is.

In the prac kit you will find a second translation SIEVE2.PVM of a cut down version 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.

Run SIEVE2.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: The lab computers are very fast. 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, 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. 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.