# Computer Science 3 - 2008

## Programming Language Translation

### Practical for Weeks 25 - 26, beginning 13 October 2008

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Monday 27 October**, correctly packaged in a transparent folder with your cover sheet and individual assessment sheets. 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. **Please make it clear whose folder you have used for the electronic submission, for example g03A1234.** Please resist the temptation to carve up the practical, with each group member only doing one task. The group experience is best when you work on tasks together.

**The reason for requiring all submissions by 27th October is to free you up during swot week to prepare for the final examinations. I shall try to get the marking done as soon as possible after that.**

## Objectives:

In this practical you are to

- familiarize yourself with the simple compiler described in chapters 12 and 13 that translates Parva to PVM code
- extend this compiler in numerous ways, some a little more demanding than others.

You will need this prac sheet and your text book. As usual, copies of the prac sheet are also available at `http://www.cs.ru.ac.za/CSc301/Translators/trans.htm`.

## Outcomes:

When you have completed this practical you should understand

- several aspects of semantic constraint analysis in an incremental compiler
- code generation for a simple stack machine.

Hopefully after doing these exercises (and studying the attributed grammar and the various other support modules carefully) you will find you have learned a lot more about compilers and programming languages than you ever did before (and, I suspect, a lot more than undergraduates at any other university in this country). I also hope that you will have begun to appreciate how useful it is to be able to base a really large and successful project on a clear formalism - namely the use of attributed context-free grammars - and will have learned to appreciate the use of sophisticated tools like Coco/R.

## To hand in:

By the hand-in date you are required to hand in, besides the cover sheets (one per group member):

- Listings of your `Parva.atg` file and the source of any auxiliary classes that you develop, produced on the laser printer by using the LPRINT utility or UltraEdit. Please configure your (wide) listings nicely, and it would help if you could use a highlighter to show where you have made changes.
- Some examples of very short test programs and the code generated by your systems.
- Electronic copies of your solutions.

I do NOT require listings of any Java or C# code produced by Coco/R.

**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/plag2008.doc`

## Before you begin

The tasks are presented below in an order which, if followed, should make the practical an enjoyable and enriching experience. Please do not try to leave everything to the last few hours, or you will come horribly short. You must work consistently, and with a view to getting an overview of the entire project, as the various components and tasks all interact in ways that will probably not at first be apparent. Please take the opportunity of coming to consult with me at any stage if you are in doubt as how best to continue. By all means experiment in other ways and with other extensions if you feel so inclined.

This version of Parva has been restricted so as not to include functions. This means that there will be no practical work set on chapter 14 of the text. Because of the timing of our courses this is unavoidable, if highly regrettable. You should be warned that some of the material of that chapter may be examinable.

You are advised that it is in your best interests to take this opportunity of really studying the code in the Parva grammar and its support files. The exercises have been designed to try to force you to do that, but it is always tempting just to guess and to hack. With a program of this size that often leads to wasting more time than it saves. Finally, remember the advice given in an earlier lecture:

*Keep it as simple as you can, but no simpler.*

## A note on test programs

Throughout this project you will need to ensure that the features you explore are correctly implemented. This means that you will have to get a feel for understanding code sequences produced by the compiler. The best way to do this is usually to write some very minimal programs, that do not necessarily do anything useful, but simply have one, or maybe two or three statements, the object code for which is easily predicted and understood.

When the Parva compiler has finished compiling a program, say `SILLY.PAV`, you will find that it creates a file `SILLY.COD` in which the stack machine assembler code appears. Studying this is often very enlightening.

Useful test programs are small ones like the following. There are some specimen test programs in the kit, but these are deliberately incomplete, wrong in places, too large in some cases and so on. Get a feel for developing some of your own.

```
$D+ // Turn on debugging mode
void main (void) {
  int i;
  int[] List = new int[10];
  while (true) { // infinite loop, can generate an index error
    read(i);
    List[i] = 100;
  }
}
```

*The debugging pragma*

It is useful when writing a compiler to be able to produce debugging output - but sometimes this just clutters up a production quality compiler. The `PARVA.ATG` grammar makes use of the `PRAGMAS` option of Coco/R (see text, page 250) to allow pragmas like those shown to have the desired effect.

```
$D+ /* Turn debugging mode on  */
$D- /* Turn debugging mode off */
```

## Task 1 - Create a working directory and unpack the prac kit

There are several files that you need, zipped up in the file `PRAC25.ZIP` (Java) or `PRAC25C.ZIP` (C#).

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

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

```
J:
md  prac25
cd  prac25
copy  i:\csc301\trans\prac25.zip
unzip  prac25.zip
```

  This will create several other directories "below" the `prac25` directory:

```
J:\prac25
J:\prac25\Library
J:\prac25\Parva
```

  containing the Java classes for the I/O Library and the code generator and symbol table handler.

  You will also find the executable version of Coco/R and batch files for running it, frame files, and various sample data, code and the Parva grammar, contained in files with extensions like

```
*.ATG,     *.PAV
```

- UltraEdit is probably your editor of choice. The version in the lab is configured to run various of the compilers easily, and has been tweaked to run Coco/R in much the same sort of way (see below). *To get this to work properly, start UltraEdit from a command window by giving the command* `UEDIT32`*, rather than by clicking on an icon on the desktop*.

- As usual, you can use the `CMAKE` and `CRUN` commands to build and run the compiler.

## Task 2 - Better use of the debugging pragma

We have already commented on the `$D+` pragma. At present it is only used to request the printout of a symbol table. How would you change the system so that one would have to use a similar pragma or command line option if one wanted to obtain the assembler code file - so that the "`.COD`" file with the assembler code listing would only be produced if it were really needed?

```
$C+ /* Request that the .COD file be produced */
$C- /* Request that the .COD file not be produced */
```

Another useful (run-time) debugging aid is the undocumented `stackdump` statement. Compilation of this is also controlled by the $D pragma (in other words, the stack dumping code is only generated when in debug mode - much of the time you are testing your compiler you will probably be working in "debugging" mode, I expect).

Hint: This addition is almost trivially easy. You will also need to look at (and probably modify) the `Parva.frame` file, which is used as the basis for constructing the compiler proper (see page 268).

## Task 3 - Learning many languages is sometimes confusing

If, like me, you first learned programming in languages other than Java, you may persist in making silly mistakes when writing Parva programs. For example, Pascal programmers easily confuse the roles played by `==`, `!=` and `=` with the similar operators in Pascal denoted by `=`, `<>` and `:=`, and are prone to introduce words like `then` into *IfStatements*, giving rise to code like

```
if (a = b) then c := d;
if (x <> y) then p := q;
```

instead of

```
if (a == b) c = d;
if (x != y) p = q;
```

Can you think of (and implement) ways to handle these errors sympathetically - that is to say, to report them but then to "recover" from them without further ado?

(Confusing = with == in Boolean expressions is something C, C++ and Java programs can easily do as well. It is particularly nasty in C/C++, where a perfectly legal statement like

```
if (a = b) x = y;
```

does not compare `a` and `b`, but assigns `b` to `a`, as you probably know).

## Task 4 - Suppressing some error messages

Identifiers that are undeclared by virtue of mistyped declarations tend to be annoying, for they result in many subsequent errors being reported. Implement a strategy where each undeclared identifier is flagged as undeclared at the point of first reference, and then quietly entered as a variable of the `noType` pseudo-type in the symbol table. Can you extend the idea to recognize an undeclared array reference variable and thereby reduce the plethora of "unexpected subscript" messages that this non-declaration might otherwise generate?

The remaining tasks all involve coming to terms with the code generation process.

## Task 5 - Repeat these exercises until you get the idea

A very easy one: add the Pascal-like *Repeat* loop to Parva, as exemplified by

```
repeat  a = b; c = c + 10; until (c > 100);
```

## Task 6 - You had better do this one or else….

Add an *else* option to the *IfStatement*. Oh, yes, it is trivial to add it to the grammar. But be careful. Some *IfStatements* will have *else* parts, others may not, and the code generator has to be able to produce the correct code for whatever form is actually to be compiled. The following silly examples are all valid.

```
if (a == 1) { c = d; }
if (a == 1) {}
if (a == 1) {} else {}
if (a == 1) {} else { b = 1; }
```

In an earlier practical we suggested that an *IfStatement* might also incorporate optional `elsif` clauses - the grammar for this simply being on the lines of

```
IfStatement = "if" "(" Condition ")" Statement
              { "elsif" "(" Condition ")" Statement }
              [ "else" Statement ] .
```

Implement this extension (make sure all the branches are correctly set up). By now you should know that this will lead to LL(1) warnings, but if you get the system correct these will not really matter.

## Task 7 - This has gone on long enough - time for a break

The *BreakStatement* is syntactically simple, but takes a bit of thought. Give it some! Be careful - breaks can only appear within loops, there might be several break statements inside a single loop, and loops can be nested inside one another.

## Task 8 - Break over, let's continue

The *ContinueStatement* is also syntactically simple. Once you have puzzled out the *BreakStatement* the *ContinueStatement* should be simple, surely?

## Task 9 - Make the change; enjoy life; upgrade now to Parva++ (Ta-ra!)

At last! Let's really make Parva useful and turn it into Parva++ by adding the increment and decrement statement forms exemplified by

```
int parva;
int [] list = new int[10];
...
parva++;
list[10]--;
--parva;
```

Suggestions for doing this - specifically by introducing new operations into the PVM - are made in section 13.5.1 of the text. Be careful - only integer variables (and array elements) can be handled in this way.

## Task 10 - Let's operate like C

The operator precedences in Parva as supplied resemble those in Pascal and Modula, where only three basic levels are supported. Modify Parva so that it uses a precedence structure based on that in C++ or Java (the basic grammar modifications were discussed in earlier practicals). Take special care to deal with "short circuit" semantics correctly (see section 13.5.2) and with type compatibility issues (see section 12.6.8)

From this point on you will find that the exercises are most easily done if you allow yourselves the liberty of adding a few instructions to the PVM, or modifying some of them, something which you experienced way back in the halcyon days of Practical 20.

## Task 11 - Better string handling

Strings only appear in Parva in read and write lists, and they have been implemented by stacking them in the top part of memory. This would get messy if we really wanted to add a string type to Parva. You may be relieved to learn that we don't want to do that at this stage (your predecessors in 2007 were faced with this exercise in the 24 hour exam). But as a prelude, consider how we might, instead, store literal strings in an "generic" ArrayList structure defined within the PVM and then modify the PRNS opcode to index this list.

## Task 12 - Some ideas are worse than a snake in the grass

If you know any Python you may have been impressed by a Python feature which allows one to write multiple assignments into a single statement, as exemplified by

```
A, B = X + Y, 3 * List[6];
A, B = B, A;                // exchange A and B
A, B = X + Y, 3, 5;         // incorrect
```

which can obviously be described by the context-free production

```
Designator { "," Designator } "=" Expression { "," Expression } ";"
```

Describing the syntax is one thing. Getting the semantics and code generation right should keep you out of mischief for an hour or two.

## Task 13 - Let's dive in where C programmers fear to go

In an earlier practical we suggested that Parva might be extended to provide a relational operator for determining whether the value of an expression matched one of a numbers in a list. Don't pretend you have forgotten - show how one could handle expressions of the form shown in the following example code

```
exp in (a, b, c)
itsASmallPrime = i in (2, 3, 5, 7, 11, 13);
if (list[i] in (i, 2 * i, 3 * i)) write("list[i] is a small multiple of its index");
```

## Task 14 - What are we doing all this for?

Many languages provide for a *ForStatement* in one or other form.

As your last fling in this practical, add a simple *for* loop to Parva, to allow statements with concrete syntax defined by

```
ForStatement = "for" Ident "in" "(" Expression { "," Expression } ")" Statement .
```

For example

```
int i;  // small primes
for (i in (2, 3, 5, 7, 11))
  write(i, " is a small prime");

bool a, b;  // truth tables
for (a in (false, true))
  for (b in (false, true))
    write(a, b, a || b, a && b, "\n");
```

Ensure that your loop also supports the *break* and *continue* statements, and that it checks for compatibility between the control variable and the elements of the list.

Do you think you should be allowed to change the value of this control variable within the loop - for example, what is your reaction to seeing code like

```
int i;  // small primes
for (i in (2, 3, 5, 7, 11)) {
  write(i, " is a small prime");
  i = 13;  // should this be allowed?
}
```

If you wanted to prevent this behaviour, how would you do it?