

Computer Science 3 - 2012

Programming Language Translation

Practical for Weeks 25 - 26, beginning 15 October 2012

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Monday 29 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 or two tasks. The group experience is best when you work on tasks together.

The reason for requiring all submissions by 29 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 compiler described in chapters 12 and 13 that translates Parva to PVM/PAM code;
- extend this compiler in numerous ways, some a little more demanding than others;
- if we have time next week, to modify the back end to load and run the code on special hardware - the PAM.

This prac sheet is also available at <http://www.cs.ru.ac.za/courses/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. Please print these on the laser printer using the LPRINT utility, as the listings get wide. Please ensure that the lines do all "fit" (into less than 120 characters) - lay out your grammars neatly! **It would also 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 by following the links at:

<http://www.scifac.ru.ac.za/>

or from <http://www.ru.ac.za/>

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.

Please resist the temptation simply to copy code from model answers issued in previous years.

This version of the Parva grammar has been restricted so as not to include Parva functions. However, the version of the PVM incorporates several opcodes that do support functions calls. This will allow us next week, hopefully, to generate code that can run on Michael Andersen's PAM (Parva Actual machine) which he has been working on this semester, and which is "microcoded" to execute the same opcodes set as we use in the interpreter. However, in the present prac there are no exercises 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.

The operator precedences in Parva as supplied use the expression precedence structure matching the "Pascal-like" ones in the book. Study these carefully and note how the compiler provides "short-circuit" semantics correctly (see page 167) and deals with type compatibility issues (see section 12.6.8).

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 128) 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 for 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
```

- If UltraEdit is your editor of choice, the version in the lab can be configured to run various of the compilers easily, and can be tweaked to run Coco/R in much 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 an icon on the desktop.*
- As usual, you can use the CMAKE and CRUN commands to build and run the compiler. The kit also supplied a PARVA.BAT file to allow you to give a command like `parva voter.pav` more easily than by using CRUN.

Task 2 - Use of the debugging and other pragmas

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 study and modify the `Parva.frame` file, which is used as the basis for constructing the compiler proper (see page 140).

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, programmers familiar with Pascal and Modula-2 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 "recover" from them without further ado?

(Confusing `=` with `==` in Boolean expressions is also something C, C++ and Java programmers can do. 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 - Things are not always what they seem

Although not strictly illegal, the appearance of a semicolon in a program immediately following the condition in an *IfStatement* or *WhileStatement*, or immediately preceding a closing brace, may be symptomatic of omitted code. The use of a so-called *EmptyStatement* means that the example below almost certainly will not behave as its author intended:

```
read(i);
while (i > 10);
{
    write(i);
    i = i - 1;
}
```

It should be possible to warn the user when this sort of code is parsed; do so.

Warnings are all very well, but they can become irritating. Use a `$W-` pragma or a `-w` command line option to allow advanced users to suppress warning messages.

Task 5 - How long is a piece of string?

Why do you suppose languages generally impose a restriction that a literal string must be contained on a single line of code?

In C++, two or more literal strings that appear in source with nothing but white space between them are automatically concatenated into a single string. This provides a mechanism for breaking up strings that are too long to fit on one line of source code. Add this feature to the Parva compiler. It is not needed in languages like C# and Java, which have proper strings, as the concatenation can be done with a `+` operator. Just for fun, allow this concatenation operator as an option between string literals that are to be concatenated.

Task 6 - "I am lost, but see in simple childhood something of the base" (Wordsworth: The Prelude)

Extend Parva so that you can express literal constants in decimal, hex (for example `0DEADH`) or binary (for example `0101111%`), and restrict decimal constants so that they may not begin with 0 unless the constant is zero.

Task 7 - Two of the three Rs - Reading and Writing

Extend the *WriteStatement* to allow a variation introduced by a new key word `writeLine` that automatically

appends a line feed to the output after the last *WriteElement* has been processed.

Extend the *ReadStatement* to allow a variation introduced by a new key word *readLine* that automatically ignores the remainder of the line after the last *ReadElement* has been processed.

Extend the *HaltStatement* to have an optional string parameter that will be output just before execution ceases (a useful way of indicating how a program has ended prematurely).

Task 8 - 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 9 - 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; }
```

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 10 - 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 currently only appear within *while* loops, but there might be several break statements inside a single loop, and loops can be nested inside one another.

Task 11 - Your professor is quite a character

Parva will get closer to C/C++/Java with each successive long hour spent in the Hamilton Labs. Seems a pity to stop now, so go right on and extend the system to allow for a character type as well as the integer and Boolean ones, enabling you to develop classic programs like the following:

```
void main () {
// Read a sentence and write it backwards
char[] sentence = new char[1000];
int i = 0;
char ch;
read(ch);
while (ch != '.') { // input loop
    sentence[i] = ch;
    i = i + 1;
    read(ch);
}
while (i > 0) { // output loop
    i = i - 1;
    write(sentence[i]);
}
}
```

Hint: A major part of this exercise is concerned with the changes needed to apply various constraints on operands of the *char* type. In some ways it ranks as an arithmetic type, so that *Expressions* of the form

```
character + character
character > character
character + integer
character > integer
```

are all allowable. However, assignment compatibility is more restricted. *Assignments* like

```
integer = integer;
integer = character;
character = character;
```

are all allowed, but

```
character = integer;
```

is not allowed. Following Java and C#, introduce a casting mechanism to handle the situations where it is necessary explicitly to convert integer values to characters, so that

```
character = (char) integer
```

would be allowed and, for completeness, so would

```
integer = (int) character
integer = (char) character
character = (char) character
```

But be careful. Parva uses an ASCII character set, so that executing code generated from statements like

```
int i = -90;
char ch = (char) 1000;
char ch2 = (char) 2 * i;
```

should lead to run-time errors.

Task 12 - 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 (don't try to extend *expressions*), exemplified by

```
int parva;
int [] list = new int[10];
char ch = 'A';
...
parva++;
--ch;
list[10]--;
```

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 and character variables (and array elements) can be handled in this way.

Task 13 - All assignments have equals, some have more equals than others

The C family of languages have all those cute augmented assignment operators. Time for Parva to join the family - so extend your system to allow statements like

```
int parva;
int [] list = new int[100];
...
parva *= 2;
list[10] += parva;
```

Once again, section 13.5.1 should give you some ideas of how to proceed. And, once again, remember that variables have types that require compatibility with the operators to be checked.

Task 14 - Beg, borrow and steal ideas from other languages

Suppose we extend Parva to adopt an idea used in Pascal, where a statement like

```
write(X : 5, X + A : 12, X - Y : 2 * N);
```

will write the values of X, X+A and X-Y in fields of widths 5, 12 and 2*N respectively.

Note that a statement like

```
write(3 + 4, A < B);
```

should produce numeric output for the first element, and Boolean output for the second one. But one like this

```
write(3 + 4 : A < B)
```

should not be allowed, because the (optional) formatting expression is meaningless unless it is numeric.

Hint: This is best handled by producing new opcodes based on the PRNI and PRNB opcodes already in the PVM, and these can make use of the formatted output routines which are in the library supplied with the distribution kits. But leave the original opcodes in the system as well!

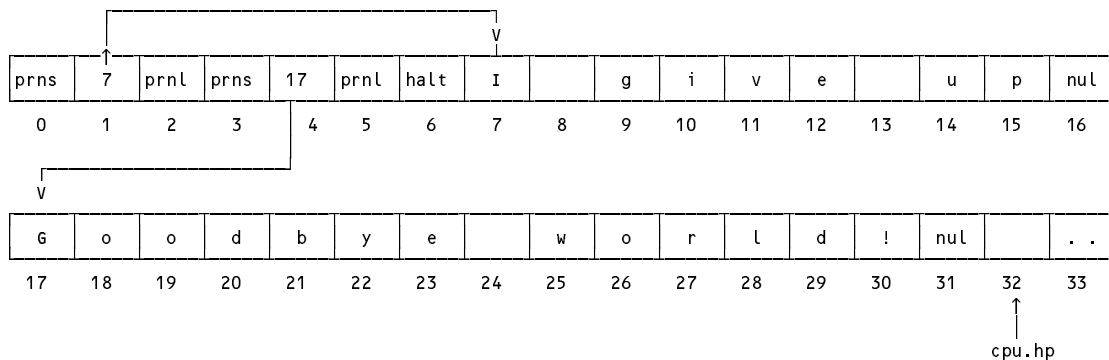
Task 15 - Let's untangle that heap of string

By now you should be well aware that any strings encountered in a Parva program are stacked at the top end of memory whence they are extracted by the PRNS opcode in the PVM.

There is an alternative model, one that is used in the PAM (Parva Actual Machine). In this model the strings are stored above the program code - effectively in space that the PVM would otherwise have used for its heap. A very simple program like

```
void main () {  
  // A fatalistic approach to the oncoming examinations?  
  writeLine("I give up");  
  writeLine("Goodbye world!");  
}
```

would be stored in memory like this



Modify the code generator and the PVM to use this model.

Hint: You might like to associate a label with each PRNS operation, since at the time you want to generate the opcode you won't know exactly where the string will be stored. If you like developing tight code suited to a small machine like the PAM, you might think of optimising the analysis and code generation so that if the same string appears in the program in two or more places, only one copy is loaded in memory.

Task 16 - Generating tighter PVM code

Way back in Practical 20 we added some specialized opcodes like `LDC_1`, `LDA_2` and so on to the PVM. It might seem a shame not to use them, so modify the code generator to use them where possible. They are already to be found in the PVM interpreter.

Have fun, and good luck. There are all sorts of other extensions one might think of, so if you have time to spend

Advance warning - next week

There will be a practical session next week, on 25 October as usual. In place of the usual test there will be a different sort of test - we shall hopefully be testing Michael Andersen's PAM board - a small computer that is programmed in PVM bytecodes. To do this we shall use the Parva compiler you have been working on as a "cross compiler" (one that runs on one machine but generates code for a different machine - go and look at those T-diagrams again).

This has been an interesting project, and I am very grateful to Michael for his initiative, for the work he has done and the fact that we (and future classes) will be able to benefit from it. You may have noticed that the "magic numbers" associated with the set of PVM opcodes have some omissions, and don't form a simple sequence. This is because the PAM has some extra opcodes that we do not need in this practical.

The PAM hardware also allows one to program a set of LEDs on board, and to read the state of a set of simple switches, and some excellent effects are possible.

Further details will be given in due course in an extra handout, and this could be a splendid end to the practical course. So don't leave all the tasks I have set you to the very last minute (but you never do that, do you?)