Computer Science 3 - 2017

Programming Language Translation

Practical 7: Week beginning 4 September 2017

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Wednesday 20 September**, 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 g14A1234.** 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.

I shall try to get the marking done as soon as possible after 20 September, and may release solutions before then.

Objectives:

In this practical you are to

- familiarize yourself with a compiler largely described in chapters 12 to 14 that translates Parva to PVM code.
- extend this compiler in numerous ways, some a little more demanding than others.

This prac sheet is 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 by 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, or preferably, just edit out the changed sections and print those**.
- A few examples of very short test programs and the corresponding PVM code as generated by your compiler.
- Electronic copies of your complete solutions.

I do NOT require listings of any 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 on the university website.

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 may 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 Parva has been developed from the system described in Chapters 12 and 13, extended on the lines suggested in Chapter 14, so as incorporate functions with value parameters. The interpreter is a Push/Pop one similar to the one you (should have) studied way back in Practical 2, incorporating the extended opcodes implemented in theat practical.

The operator precedences in Parva as supplied use the "Pascal-like" operator precedences, as described in the book. Study these carefully and note how the compiler provides "short-circuit" semantics correctly (see page 183) 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, especially as they deal with operator precedence and the `char` type. 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, hacking often leads to wasting more time than it saves. Finally, *please* 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 141) to allow pragmas like those shown to have the desired effect (see the sample program above).

```
$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 `PRAC7.ZIP`.

- Immediately after logging on, get to the command line level as usual!

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

```
J:
md  prac7
cd  prac7
copy  i:\csc301\trans\prac7.zip
unzip  prac7.zip
```

  This will create another directory "below" the `prac7` directory:

```
J:\prac7
J:\prac7\Parva
```

  containing the C# classes for the code generator, symbol table handler and the PVM.

  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,     Examples\*.PAV
```

- As usual, you can use the `CMAKE` command to rebuild the compiler, and a command like `Parva EG.PAV` to run it.

You should attempt all of Tasks 2 through 12. Perfect answers to these would earn you a mark of 100% for this prac. Bonus marks can be earned for attempting any or all of Tasks 13 and 14. You are urged to do these - more for the insight they will give you in preparing for the final examination than for the marks!

Task 2 - Use of the debugging and other pragmas

We have already commented on the `$D+` pragma. How would you add to 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?

Suggested pragmas would be (to be included in the source being compiled at a convenient point):

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

while the effect of `$C+` might more usefully be achieved by using the compiler with a command like

```
Parva Myprog.pav -c          (produce .COD file)
Parva Myprog.pav -c -l       (produce .COD file and merge error messages)
```

Other useful debugging aids are provided by the `$ST` pragma, which will list out the current symbol table. Two more are the `$SD` and `$HD` pragmas, which will generate debugging code that will display the state of the run-time stack area and the runtime heap area at the point where they were encountered in the source code. Modify the system so that these are also dependent on the state of the `$D` pragma. In other words, the stack dumping code would only be 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: These additions are almost trivially easy. You will also need to look at (and modify) the `Parva.frame`

file, which is used as the basis for constructing the compiler proper (see page 154).

Task 3 - Sometimes it makes sense, as you start, to be told how you will stop

Extend the *HaltStatement* to to have an optional parameter that can issue a report before execution ceases (a useful way of indicating how a program has ended prematurely).

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. Here is another example that might warrant a warning

```
while (i > 10) {  }
```

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

Solutions to tasks 2, 3 and 4 are to be submitted by the end of the afternoon. Please do NOT print out the entire ATG file. Cut and paste the sections you have edited into a simple text file, type your group names (don't just write them by hand) at the top of this, and make an LPRINT listing.

The same goes for other submissions later. There is no real need to print out all the code for the modified PVM, code generator, table handler etc if all you do is change 10 lines or so. As you make modifications, "sign" them with a little comment like

// Change for task x

and then it will be easy to find them again. It will help further if you can use a light highlighter to emphasize changes and additions. Changes to the ATG file will be scattered throughout it, and it might be easiest to print that one almost completely when it is all finished (don't give listings for each task separately).

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.

```
write("This is a string that is way too long to fit on one source line comfortably so we "
      "simply write it in pieces as two adjacent shorter strings");
```

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.

Hint: Pay careful attention to how the system handles escape sequences in strings.

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

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

Add the *else* and *elsif* options to the *IfStatement*. Oh, yes, it is trivial to add them to the unattributed 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; }
```

Implement this extension (make sure all the branches are correctly set up). By now you should know that the obvious grammar leads to the dreaded dangling else and LL(1) warnings, but that these should not matter.

Task 7 - Something to do - while you wait for inspiration

Add a *DoWhile* loop to Parva, as exemplified by

```
do { a = b; c = c + 10; } while (c < 100);
```

Task 8 - 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 (since there is no *SwitchStatement*), but there might be several break statements inside a single loop, and loops can be nested inside one another.

And when your break is over, make provision to *Continue* as well.

Task 9 - Here we go loop - de - loop (You may b too young to remember that song?)

As an alternative way of writing loops, and a very easy one, add an indefinite loop to Parva, as exemplified by

```
loop
  writeLine("Keep it as simple as you can, but no simpler");
```

A loop like that is of limited use. This form of loop is usually combined in programs with a *BreakStatement*, to give a "middle exit" construct, exemplified by

```
int i, total = 0;
loop {
  read("Supply a number - 0 will terminate ", i);
  writeLine("you typed ", i);
  if (i == 0) break;
  total = total + i;
}
writeLine("The total is ", total);
```

Task 10 - Your professor is quite a character

One can get just so far with little programs limited to manipulating integers and booleans. To be able to write programs manipulating characters is a logical next step, enabling us all to develop classic programs like the following:

```
void Main () {
// Read a sentence and write it SDRAWKCAB (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(cap(sentence[i]));
  }
} // Main
```

Most of the opcodes needed in the PVM have been supplied to you, but the compiler will need some changes and additions to deal with functions like `cap(c)` and `low(ch)` . A major part of this exercise is concerned with the changes needed to apply various auto-conversions and constraints on operands of the `char` type, similar to those rather odd ones found in the C family of languages. In some respects, `char` ranks as an arithmetic type, so that expressions of the form

character + character	'a' + 'b'	(96 + 97 : 193 : integer)
character + integer	'a' + 10	(96 + 10 : 106 : integer)
character > integer	'a' > 12	(96 > 12 : true : bool)
character > character	ch > ' '	

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

integer = integer expression
integer = character expression
character = character expression

are all allowed, but

character = integer expression

is not allowed. Following 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 ch1 = (char) 1000;
char ch2 = (char) 2 * i;
```

should lead to run-time errors.

Task 11 - Make the change - upgrade now to Parva++

At last! Let's really make Parva useful and turn it into Parva++ by adding the increment and decrement *statement* forms (as variations on assignments, of course), 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, and the necessary opcodes are already waiting for you in the PVM. Be careful - only integer and character variables (and array elements) can be handled in this way. **Do not bother with trying to handle the ++ and -- operators within terms, factors, primaries and expressions**.

Task 12 - Let's operate like C#

Parva is looking closer to C/C++/C# with each successive long hour spent in the Hamilton Labs. Seems a pity not to get even closer. 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. 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).

Being a kindly old soul, and to save you some time on Google: the grammar for an *Expression* introduces more levels, and is more or less as follows (puzzle out any anomalies for yourself).

```
Expression = AndExp { "||" AndExp } .
AndExp     = EqlExp { "&&" EqlExp } .
EqlExp     = RelExp [ EqlOp RelExp ] .  // [ ] not { }
RelExp     = AddExp [ RelOp AddExp ] .  // [ ] not { }
AddExp     = MulExp { AddOp MulExp } .
MulExp     = Factor { MulOp Factor } .
Factor     = Primary | ( "+" | "-" | "!" ) Factor .
Primary    =   VariableDesignator
             | Function "(" Arguments ")"
             | Constant
             | "new" BasicType "[" Expression "]"
             | "("   Expression ")" .
AddOp      = "+" | "-" .
MulOp      = "*" | "/" | "%" .
EqlOp      = "==" | "!=" .
RelOp      = "<" | "<=" | ">" | ">=" .
```

You might like to think what advantages or disadvantages are to be gained by having fewer or more levels of precedence.

Task 13 - Generating tighter PVM code

Way back in Practical 2 we added some specialized opcodes like `LDA_2`, `LDC_3` 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, when requested by a suitable pragma like $O+. They are already to be found in the PVM interpreter.

Task 14 - If you survive this I'll pass you a reference so that employers will know your value

As supplied, the Parva compiler can only pass parameters "by value". Extend it on the lines of the approach adopted in C#, allowing you to write impressive methods like

```
void Swap(ref int i, ref int j) {
// Interchange values of i and j
  int k = i;
  i = j;
  j = k;
} // Swap

void Main() {
  int a, b;
  readLine(a, b);
  Swap(ref a, ref b);
  writeLine(a, b);
} // Main
```

Be careful - you must put in checks for compatibility and think carefully what this means in this context.