RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practicals 7 and 8: weeks beginning 2nd and 9th September 2019

The practicals for the last two weeks of the course are both outlined in this document. However**,** ***tasks 1 to 6 are for Prac 7 and must be handed in before your next practical session***. The ***remaining tasks relate to Prac 8*** and must be ***handed in at Reception before 12:30pm on Thursday 19th September.*** Each prac submission must be correctly packaged in a transparent folder with your cover sheet and individual assessment sheets. Unpackaged and late submissions will not be accepted. Since the practical is 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. **Lastly, please resist the temptation to carve up the practical, with each group member only doing one task.** The group experience is best when you discuss each task together.

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.

Copies of this handout and the cover sheet are available on the RUConnected course page.

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. 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 (Prac 7: 25 marks; Prac 8: 40 marks + 20 bonus marks)

By the various hand-in dates 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 else just edit out the changed sections and print those**.
- Some 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.

You should attempt all of tasks 1 through 10. Perfect answers for these will earn you a mark of 100% for Pracs 7 and 8. Bonus marks can be earned for attempting any or all of tasks 10, 11 and 12. You are urged to do these -- more for the insight they will give you in preparing for the final examination than for the marks!

Tasks 1 to 5 must be handed in before your next practical session on Thursday, 12th September.

The prac test on Thursday, 12th September will cover tasks 1 through 5. This test will be run in the same way as the final examination.

Please submit Prac 8 (tasks 6 to 9 plus any bonus tasks) in the "hand-in" box at Reception by 12:30pm on Thursday, 19th September.

There is also 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).

Keep the cover sheets and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another student's or group'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 on *all* cover sheets and with suitable comments typed into *all* listings. You are expected to be familiar with the University Policy on Plagiarism.

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 of the notes, and extended on the lines suggested in Chapter 14.4, so as to incorporate void 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 that practical.

The operator precedences in Parva as supplied use the "Pascal-like" operator precedences, as described in the notes. Study these carefully and note how the compiler provides "short-circuit" semantics correctly (see Section 13.5.2) 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 variable types. 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 notes, page 156) 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 0 Creating a working directory and unpacking the prac kit

There are several files that you need, zipped up this week in the file `PRAC7.ZIP`.

- 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
```

- You will also find 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 Myfile.PAV` to run it.

Task 1 Use of the debugging and other pragmas (5 marks)

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 `$HD` pragmas, which will generate debugging code that will display the state of the runtime stack area and the runtime heap area at the point where they were encountered in the source code. Modify the system so that these are also dependent on the state of the `$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 Section 10.6 of the course notes).

Task 2 Sometimes it makes sense to be told how you will stop (3 marks)

Extend the `HaltStatement` 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 3 Things are not always what they seem (4 marks)

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.

Task 4 How long is a piece of string? (5 marks)

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 5 You had better do this one or else.... (8 marks)

Add the *else* and *elseif* 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; }
```

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

Task 6 Something to do - while you wait for inspiration (6 marks)

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

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

Task 7 This has gone on long enough -- time for a break (9 marks)

The *BreakStatement* is syntactically simple, but takes a bit of thought. Give it some! Be careful -- breaks can currently 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.

Task 8 Some characters to keep you occupied (12 marks)

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 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 9 What are we doing this for? (13 marks)

Many languages provide for a *ForStatement* in one or other form. Although most people are familiar with these, their semantics and implementation can actually be quite tricky.

As an alternative to the more familiar C-style *ForStatement*, implement one modelled on a syntax suggested by Python:

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)
        writeLine(a, b, a || b, a && b);
```

Hint: Once again, you might like to add some more operations to the PVM to assist in this regard. As before, ensure that your loop also supports the break statement. And, as always, insist on type compatibility between the control variable and the elements of the list.

Task 10 (Bonus) Upgrade to Parva++ (10 marks)

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), for example:

```
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 notes, 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 11 (Bonus) Do you ever feel threatened by change? (5 marks)

Do you think you should be allowed to change the value of the control variable associated with a *ForStatement* within the loop body - 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?
}
```

or, if you were using a Pascal-like form of the loop - and making the point more strongly:

```
for i = 1 to 10 {
  for i = 1 to 10  // nested loops are allowed, but surely not this one?
    write(i);
  writeLine();
}
```

See if you can find a way to prevent "threatening" (tampering with) the control variable within the body of the loop. And how, if at all, might this facility interact with what you developed for Task 10?

Task 12 (Bonus) Going to any length to complete the task (5 marks)

This one should be easy. Add a little function into your expression parsers to allow you to determine the length of an array, which might be useful in writing code like

```
void DisplayList(int [] list) {
// Display all the values in list[0] .. list[length - 1]
  int i = 0;
  while (i < length(list)) (
    write(list[i]);
    ++i;
  }
} // DisplayList
```

Hint: You may have forgotten that the heap manager, when allocating space for an array, also records the length of that array. As a further hint, you shouldn't need any new opcodes in the PVM.