RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practicals 7 and 8: weeks beginning 21st and 28th September 2020

Note that this handout covers practicals 7 and 8.

Tasks 1--3 must be submitted through the prac 7 RUC submission link by 2 pm on Thursday 1st October 2020.

The last weekly test of Thursday 1st October will cover all theory till end of lecture 34 and Prac 7 tasks.

Tasks 4--6 must be submitted through the prac 8 RUC submission link by 2 pm on Thursday 8th October 2020.

There will be no weekly test on the 8th October. A makeup test will be organised at a date to be determined. This test may be used to replace the 2 lowest test marks you received in the weekly tests.

Objectives

In this practical you will

- familiarize yourself with a compiler largely 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.

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 (15 marks + 20 marks)

- An electronic copy of your grammar file (Parva.atg).
- Electronic copies of the auxiliary source files (i.e. files in the Parva subdirectory) if these have been changed.

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

For practical 7, tutors will mark Tasks 1 and 2, and I will mark Task 3.

For practical 8, tutors will mark Task 4, and I will mark Tasks 5 and 6.

Feedback for the tasks marked will be provided via RUConnected. Check carefully that your mark has been entered into the Departmental Records.

You are expected to be familiar with the University Policy on Plagiarism and to heed the warning in previous practical handouts regarding “working with another student”.

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. If you do, you may find that they do not solve the problems posed in this practical, and you will then be given zero for the work as plagiarism will be clearly apparent.

This version of Parva has been developed from the system described in Chapters 12 and 13 of the notes. The operator precedences in Parva as supplied use a precedence structure based on that in C++, C# or Java, rather than the "Pascal-like" one used 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). The supplied compiler also incorporates the `char` type.

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

Unpack the prac kit `PRAC7.ZIP`.

- You will find another directory "below" the `prac7` directory:

  D:\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 testfile.PAV` to run it.

Task 1 Use of the debugging and other pragmas [4 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.

Hint: These additions are almost trivially easy. You will also need to look at (and modify) the `Parva.frame` file (see Section 10.6 of the course notes).

Task 2 Learning many languages is sometimes confusing [4 marks]

If C or C# was not your first programming language, 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 in C# by `==`, `!=` and `=` with the similar operators in Pascal denoted by `=`, `<>` and `:=`, and are prone to introduce words like `then` into an *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 report them, but then "recover" from them without further ado?

(Confusing = with == in Boolean expressions is also something that 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` and then assigns y to x if a is non-zero, as you probably know).

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

Task 3 Two of the three Rs - Reading and Writing [7 marks]

Extend the *WriteStatement* to allow a variation introduced by a new keyword `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 keyword `readLine` that causes the rest of the current data line to be discarded, so that the next *ReadStatement* will start reading from the next line of data.

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 4 Repeat discussions [5 marks]

As a very easy exercise, add a Pascal-like *Repeat* loop to Parva, as exemplified by

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

Task 5 You had better do this one or else.... [6 marks]

Add the *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; }
```

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 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, but there might be several break statements inside a single loop, and loops can be nested inside one another.