Computer Science 301

Programming Language Translation

Practical 3: week beginning 17th August 2020

All tasks in this practical should be submitted through the RUC submission link by 2 pm on your next practical day.

Objectives

In this practical you will

- gain some more experience extending the PVM,
- familiarize yourself with simple applications of the Coco/R parser generator, and
- write grammars that describe simple language features.

Copies of this handout and the Parva language report are available on the RUC course page.

Outcomes

When you have completed this practical you should understand:

- how to extend the PVM to incorporate new opcodes,
- how to develop context-free grammars for describing the syntax of various languages and language features;
- the form of a Cocol description;
- how to check a grammar with Coco/R and how to compile simple parsers generated from a formal grammar description.

To hand in (30 marks)

This week you are required to hand in via the link on RUConnected:

- Electronic copies of the changed .cs source code for task 1 [10 marks].
- Electronic copies of your grammar files (ATG files) for tasks 3 [10 marks] and 4 [10 marks].

You do NOT need to submit listings of any C# code produced by Coco/R.

As before, not all your tasks will be marked. For this practical, tutors will mark Task 3. I will mark either task 1 or 4.

Feedback for the tasks marked will be provided via RUConnected. 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. 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.

Note however, that “working with” someone does not imply that you hand in the same solution. The interpretation used in this course (and which determines whether a submission is plagiarised or not) is that you may discuss how you would approach the problem, but not actually work together on one coded solution. You are still expected to hand in your own solution at all times.

You are expected to be familiar with the University Policy on Plagiarism.

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 `PRAC3.ZIP`.

To ensure that all batch scripts provided in the kit execute correctly, copy the zipped prac kit from RUConnected into a new directory/folder in your file space (say `PRAC3`) and unzip the kit there.

Then run all batch scripts provided in the kit from within this new directory. Note these scripts need to be run from the command line – and in order to get used to doing this prior to the exam, I suggest you always use a command line terminal window when doing the practical work.

To open the correct terminal window, do one of the following:

If you are using the HAMILTON lab machines or have set up the C# environment correctly on your own laptop, open a DOS CMD window, by typing `CMD.exe` into Windows RUN or searching for “cmd.exe” using Windows search.

Else if you are using your own laptop without the C# environment configured correctly, open a *Developer Command Prompt for VS* window by finding it as an option in the app list under the Visual Studio folder.

In the kit for this week, besides the .cs and .bat files needed to create the PVM machine (as used in prac 2), you will also find the executable version of Coco/R and batch files for running it, frame files, and various sample programs and grammars, contained in files with extensions like

```
*.ATG, *.PAV, *.TXT *.BAD
```

Task 1 Improving the opcode set for the PVM [10 marks]

Section 4.10 of the text discusses the improvements that can be made to the system by adding new single-word opcodes like LDC_0 and LDA_0 in place of double-word opcodes for frequently encountered operations like LDC 0 and LDA 0, and for using load and store opcodes like LDL N and STL N (and, equivalently, opcodes like LDL_0 and STL_0 for frequently encountered special cases).

Enhance both versions of your PVMs to incorporate the following opcodes:

```
LDL N     STL N
LDA_0     LDA_1
LDL_0     LDL_1
STL_0     STL_1
LDC_0     LDC_1
```

Ensure that you recompile your ASM1 and ASM2 systems after making the changes above, as you did in Practical 2 using **`MAKEASM1`** and **`MAKEASM2`**.

Hint: Several of the above are very similar to one another. Note that the assemblers have already been primed with the mappings from these mnemonics to integers, but, once again, you must be careful to make sure you modify all the parts of the system that need extending - you will have to add quite a bit to various switch statements to complete the tasks. Do this for both versions of the PVM.

Try out your systems by developing an "improved" version of FACT.PVM, say FACNEW.PVM using the new opcodes.

The code for the final assembler/emulators must be submitted for assessment, as must FACNEW.PVM. It would help if you simply printed only those parts of the interpreters that you have modified in this task as large portions of the original will not need to change at all.

Task 2 Simple use of Coco/R - a quick task

In the kit you will find `Calc.atg`. This is essentially the calculator grammar on page 70 of the notes, with a slight (cosmetic) change of name.

Use Coco/R to generate a parser for data for this calculator. You do this most simply by giving the command

```
cmake Calc
```

The primary name of the file (`Calc`) is case sensitive. Note that the `.ATG` extension is needed, but not given in the command. Used like this, Coco/R will simply generate three important components of a calculator program - the parser, scanner, and main driver program. Cocol specifications can be programmed to generate a complete calculator too (that is, one that will evaluate the expressions, rather than simply check them for syntactic correctness), but that will have to wait for the early hours of another day.

(Wow! Have you ever written a program so fast in your life before?)

Of course, having Coco/R write you a program is one thing. But it might also be fun and interesting to run the generated program and see what it is capable of doing.

A command like

`Calc calc.txt` (or `Calc.exe calc.txt`)

will run the program `Calc` and try to parse the file `calc.txt`, sending error messages to the screen. Giving the command in the form

```
Calc calc.bad -L
```

will send an error listing to the file `listing.txt`, which might be more convenient. Try this out.

For some light relief and interest you might like to look at the code the system generated for you (three `.cs` files are created in the parent directory: `Calc.cs`, `Scanner.cs` and `Parser.cs`). You don't have to comment this week, simply gaze in awe. Don't take too long over this, because now you have the chance to be more creative.

Task 3 Extending the calculator [10 marks]

Modify the calculator grammar so that you can:

1. use parentheses in your expressions,
2. use leading unary + or - signs,
3. raise quantities to a "power" (as in 12^2 + 5^6, meaning $12^2 + 5^6$),

4. give the calculator a square-root capability (as in 4 + sqrt(5 * 12)), and
5. allow it to recognize numbers that incorporate a decimal point, as in 3.4 or 3. or .45

Of course, the application does not have any real "calculator" capability -- it cannot calculate anything (yet). It only has the ability to recognise or reject expressions at this stage. Try it out with some expressions that use the new features, and some that use them incorrectly.

A listing of your grammar for this task must be submitted. Remember to include your name as a comment at the top!

Task 4 When all else fails, look up what you need in the index [10 marks]

Warning. Language design and grammar design is easy to get wrong. Think hard about this problem before you begin, and while you are doing it.

In the prac kit you will find an extract from the index to the next edition of an all-time bestseller "Hacking out a Degree", in the file `Index.txt`. It looks something like this:

```
abstract class 12, 45
abstraction, data 165
advantages of Java and Modula-2 1-99, 100-500, Appendix 4
aegrotat examinations -- see unethical doctors
aggregate pass, chances of 0
class attendance, intolerable 15, 745
class members 31
deadlines, compiler course -- see sunrise
horrible design (C and C++) 34, 45, 56-80
lectures, missed 1, 3, 5-9, 12, 14-17, 21-25, 28
loss of DP certificate 2014
no chance of supp 2014
prac tests 10, 25, 27, 30
probable exclusion from Rhodes 2015
recursion -- see recursion
senility, onset of 21-24, 105
subminimum 40
supplementary exams (first courses only) 45 - 49
wasted years 2011 - 2014
```

Develop a grammar `index.atg` that describes this and similar indexes and create a program that will analyse an index and accept or reject it (syntactically). Answer this as far as possible in the sprit in which it is intended -- clearly a real index could have almost any combinations of characters making up tokens, so restrict yourself to permitting relatively simple words and numbers that might come between punctuation marks.

A listing of your grammar for this task must be submitted. Remember to include your name as a comment at the top!

Appendix: Practical considerations when using Coco/R

I strongly recommend that you use a standalone ASCII editor to develop these grammars -- like NotePad++, etc. Steer clear of MS-Word and Visual Studio. Keep it simple. For the examination at the end of the course it will be assumed that you are competent at using a standalone editor. Notepad++ and Notepad will be available, and the various command scripts used in the practicals will be provided if necessary.

Avoid using use folder names (directory names) with spaces in them, such as "Prac 3"

Use a *fairly short* name (say 5 characters) for your goal symbol (for example, `Gram`);

Remember that this name must appear after `COMPILER` and after `END` in the grammar itself;

Store the grammar in a file with the same short primary name and the extension `.atg` (for example `Gram.ATG`).

If required, store ancillary source code files in the subdirectory named `Gram` beneath your working directory. (Nothing like this should be needed this week.)

Make sure that the grammar includes the "pragma" `$CN`. The `COMPILER` line of your grammar description should thus always read something like

```
       COMPILER Gram $CN
```

Free standing use of Coco/R

You can run the C# version of Coco/R in free standing mode with a command like:

```
cmake Gram
```

which will produce you a listing of the grammar file and associated error messages, if any, in the file `LISTING.TXT`.

If the Coco/R generation process succeeds, the C# compiler is invoked automatically to try to compile the application.

If that (second) compilation does not succeed, a C# compiler error listing is redirected to the file `ERRORS`, where it can be viewed easily by opening the file in your favourite editor.

Error checking

Error checking by Coco/R takes place in various stages. The first of these relates to simple syntactic errors - like leaving off a period at the end of a production. These are usually easily fixed. The second stage consists of ensuring that all non-terminals have been defined with right hand sides, that all non-terminals are "reachable", that there are no cyclic productions, no useless productions, and in particular that the productions satisfy what are known as **LL(1) constraints**. We shall discuss LL(1) constraints in class in some detail, and so for this practical we shall simply hope that they do not become tiresome. The most common way of violating the LL(1) constraints is to have alternatives for a nonterminal that start with the same piece of string. This would mean that a so-called LL(1) parser (which is what Coco/R generates for you) could not easily decide which alternative to take - and in fact will run the risk of going badly astray. Here is an example of a rule that violates the LL(1) constraints:

```
          assignment =   variableName ":=" expression
```

```
                      | variableName index ":=" expression.
         index      =   "[" subscript "]" .
```

Both alternatives for `assignment` start with a `variableName`. However, we can easily write production rules that do not have this problem:

```
         assignment =   variableName [ index ] ":=" expression .
         index      =   "[" subscript "]" .
```

A moment's thought will show that the various expression grammars that are discussed in the notes in Chapter 6 - the left recursive rules like

```
         expression = term | expression "-" term .
```

also violate the LL(1) constraints, and so have to be recast as

```
         expression = term { "-" term } .
```

to get around the problem.

For the moment, if you encounter LL(1) problems, please speak to the long suffering tutors, who will hopefully be able to help you resolve all (or most) of them.