Computer Science 3 - 2016

Programming Language Translation

Practical 3, Week beginning 1 August 2016

Hand in your solutions to this practical *before* lunch time on your next practical day, correctly packaged in a transparent folder with your cover sheets. **Unpackaged and late submissions will not be accepted - you have been warned.** Please do NOT come to a practical and spend the first hour printing or completing solutions from the previous week's exercises. 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.** 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 you with simple applications of the Coco/R parser generator, and
- write grammars that describe simple language features.

You will need this prac sheet and your text book. As usual, copies of the prac sheet are also available at `http://www.cs.ru.ac.za/courses/CSc301/Translators/trans.htm` .

Outcomes:

When you have completed this practical you should understand

- 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:

This week you are required to hand in, besides the cover sheet:

- Listings of your solutions to the grammar problems, produced on the laser printer by using the LPRINT utility Some of these listings will get quite "wide" so please set them out nicely.
- Electronic copies of your grammar files (ATG files).

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 web site.

Task 1 - 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`

- Immediately after logging on, get to the DOS command line level.

- Copy the prac kit into a newly created directory/folder in your file space - use the D: drive if you observe strange behaviour.

```
j:
md  prac3
cd  prac3
copy  i:\csc301\trans\prac3.zip
unzip  prac3.zip
```

  You will 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 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 62 of the text, 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 (ie 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.

Well, you did all that. Well done. What next?

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.

That's right - we have not finished yet. Modify the grammar so that you can use parentheses in your expressions, and can recognize numbers that include 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.

Warning. Language design and grammar design is easy to get wrong. Think hard about these next problems before you begin, and while you are doing them.

Task 3 - Take a logical approach to your practicals

Task 2 showed how one could parse a list of arithmetic expressions. It's not a very big conceptual step to write a similar grammar that will define a language for a list of Boolean assignments, as exemplified by (`Bool.txt`)

```
a = true;
b = false;
c = a and b or true;
d = !true;
e = ! not true;
f = x || !y && z;
g = a && (b or c) and d;
```

Remenber that NOT takes precedence over AND which takes precedence over OR.

Task 4 - So what if Parva is so restrictive - fix it!

Parva really is a horrid little language, isn't it? But its simplicity means that it is easy to devise Terry Torture on the lines of "extend it".

In the prac kit you will find the grammar for a version of Parva based on the one on page 89. Generate a program from this that will recognise or reject Parva programs, and verify that the program behaves correctly with two of the sample programs in the kit, namely `VOTER.PAV` and `VOTER.BAD`.

```
cmake Parva
Parva voter.pav
Parva voter.bad  -L
```

Now modify the grammar to add various features. Specifically, add (and check that the additions work):

- The `%` operator that you missed so much!
- All those fun arithmetic assignment operators: `+=`, `-=` `*=` and `/=`.
- A restriction that an identifier may contain underscore characters, but may not end with one, so that `My_Name_is_Pat` would be acceptable, but not `Pat_is_my_name___`.
- Optional *elsif* and *else* clauses for the *if* statement.
- Multiple assignments within one statement as in Python.
- A *do-while* loop, and *break* and *continue* statements.
- A *for* loop inspired by the one in Pascal (look it up!).

Here are two silly examples of code that should give you some ideas:

```
void Main() {
// Demonstrate various statements
  const too_Much = 55;
  int mark__1 = 5, mark__2 = 10, mark__3 = 20;
  mark__1, mark__2, mark__3 = mark__2, mark__1 + mark__3, 256;
  age = 0;
  while (age < 180) {
    age = age + 1;
    write("Happy birthday ");
    if (age == 75) {
      write("That's quite enough for one life!");
      break;
    }
    write("Have another good year\n");
  }
} // Main
```

```
void Main () {
// (not supposed to do anything useful!)
  int age;
  bool beenKissed;
  read("How old are you, and have you been kissed? ", age, beenKissed);
  if (age == 16) {
    write("sweet sixteen");
    if (! beenKissed) write(" and never been kissed");
  }
  elsif (age == 21) {
    write("party time!");
    int headache = 0, strain = 0;
    for beers = 20 downto 0 {
      strain += 1; headache += 2;
      if (strain % 8 == 0) {
        write("That\'s better"); strain = 0;
      }
    }
  }
  elsif ((age > 21) && (age < 40))
    write("over the hill, bru");
  elsif (age > 70)
    write("take a new lover");
  else
    write("life must be boring");
} // Main
```

These little programs and some other like them are in the kit, and you can easily write some more of your own. Actually, the ones above are rather ambitious. Start on something really simple, like:

```
void Main () {
  if (a)
    c = d;
  else c = 12;
} // Main
```

and

```
void Main () {
  int i = 0;
  do
    i *= 2;
  while (i < 10);
} // Main
```

Note: Read that phrase again: "that should give you some ideas". And again. And again. Don't just rush in and write a grammar that will recognise only some restricted forms of statement. Think hard about what sorts of things you can see there, and think hard about how you could make your grammar fairly general.

Hint: All we require at this stage is the ability to *describe* these features. You do *not* have to try to give them any semantic meaning or write code to allow you to use them in any way. In later pracs we might try to do that, but please stick to what is asked for this time, and don't go being over ambitious.

Warning. Language design and grammar design is easy to get wrong. Think hard about these problems.

Task 5 - One for the Musicians in our Midst (but the rest of you should do it too)

After the musical introduction to this section of the course that introduced you to Tonic Sol-Fa, you will be intrigued to learn that the musicians in Scottish pipe bands often compete at events called Highland Gatherings where three forms of competition are traditionally mounted. There is the so-called "Slow into Quick March" competition, in which each band plays a single Slow March followed by a single Quick March. There is the so-called "March, Strathspey and Reel" competition, where each band plays a single Quick March, followed by a single Strathspey, and then by a single Reel; this set may optionally be followed by a further Quick March. And there is also the "Medley", in which a band plays a selection of tunes in almost any order. Each tune falls into one of the categories of Quick March, Strathspey, Reel, Slow March, Jig and Hornpipe but, by tradition, a group of one or more Strathspeys within such a medley is always followed by a group of one or more Reels.

Develop a grammar to describe the activity at a Highland Gathering at which a number of competitions are held, and in each of which at least one band performs. Competitions are held in one category at a time, with a short break between competitors, and between events, and with a suitable announcement being made to introduce the category of competition and the name of each band to the audience. Regard concepts like "March", "Reel", "AnnounceCompetition", "break" and so on as terminals - in fact there are many different possible tunes of each sort, of course, but you may have to be a piper to distinguish one tune from another.

Task 6 - Meet some of the staff of the Hamilton Building

Develop a Cocol grammar that will describe a list of the names of staff who have optional titles, first names and/or initials, surnames, and (usually) qualifications, for example (`Staff.txt`):

```
R. J. Foss, BSc, MSc, PhD.
Professor Philip Machanick, PhD.
Professor P. D. Terry, MSc, PhD.
George Clifford Wells, BSc(Hons), MSc, PhD.
Greg G. Foster, PhD.
James Connan, MSc.
Ms Michelle Coupe.
Phumezo Dukashe.
Professor Karen Lee Bradshaw, MSc, PhD.
Mx Mic Halse, MSc.
Vivian Kila  .
Dr Mos Tsietsi .
C. Hubert H. Parry, BMus.
Prof Barry V. W. Irwin, PhD.
Miss Busi Mzangwa.
```

Hint: notice the rather critical placing of commas and periods in this file. The exercise is partly one of being able to define tokens sensibly. Try to do this realistically so far as names are concerned, and limit yourself to a few examples of qualifications only.

Task 7 - Describing a set of EBNF productions in another way

The file `EBNF.ATG` contains a Cocol grammar that describes EBNF using EBNF conventions, which might be familiar from lectures. Try this out "as is" to begin with, for example:

```
cmake EBNF
EBNF EBNF.TXT
EBNF EBNF.BAD  -L
```

Next, use the grammar as a guide to develop a new system that will recognise or reject a set of productions like those in EBNF.TXT, but with your Cocol grammar written in such a way that it does not itself use any of the Wirth "meta brackets" { and } and [and]. As a hint, you will have to set up an equivalent grammar, using right-recursive production rules.

Appendix: Practical considerations when using Coco/R

I strongly recommend that you use a standalone ASCII editor to develop these grammars - like NotePad++, UltraEdit, 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 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 the next week, 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 text 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 demonstrators, who will hopefully be able to help you resolve all (or most) of them.