RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 3: week beginning 29 July 2019

Hand in this prac sheet before the start of your next practical, correctly packaged in a transparent folder with your solutions and the "cover sheet". 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 simple applications of the Coco/R parser generator, and
- write grammars that describe simple language features.

Copies of this handout, the cover sheet and the Parva language report are available on the RUConnected course page.

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 (40 marks)

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

- Listings of your solutions to the grammar problems, produced 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) uploaded on RUConnected.

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

Keep the cover 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 inside the laboratory before the next practical session and not given to demonstrators during the session.

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.

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`.

Immediately after logging on, get to the DOS command line level and log onto your file space.

Copy the zipped prac kit into a new directory/folder in your file space, either from the server (I:) or RUConnected.

```
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 1 Simple use of Coco/R - a quick task [5 marks]

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.

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, recognize hex numbers in the other common format (start with 0, sequence of hex digits, trailing H), allow the computation of factorials like 6! and allow an element for an absolute function call like abs(6-12).

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 Task 1 must be handed in (via the hand-in box) by the end of the prac afternoon (today that means 4:30pm). Remember to include the names of the group members as a comment at the top!

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 2 Take a logical approach to practical work [5 marks]

Task 1 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 3 So what if Parva is so restrictive -- fix it! [20 marks]

Parva really is a horrid little language, isn't it? But its simplicity means that it is easy to "extend it".

In the prac kit you will find the grammar for a version of Parva based on the one on page 100 of the notes. 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 in Prac 1!
- An optional *else* clause for the *if* statement.
- A *repeat-until* loop, a *for* loop and the *break* statement as in the examples.
- A restriction that numbers cannot begin with 0 unless this is the only digit.
- The character type.
- *readLine* and *writeLine* as well as *read* and *write* statements.
- Increment and decrement statements like `fun++; ++effort;` and `bugcount[n]--;` (statements only; do not incorporate these into general expressions)
- The set type, the *in* operator, and set handling expressions inspired by those found in Pascal and Modula-2 (look them up), in the examples below and in the kit.

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

```
  void Main() {
  // Demonstrate various statements
    int age;
    readline("How old are you? ". age);
    if ((age < 5) || (age > 75))
      writeLine("I find that hard to believe");
    else {
      repeat
        ++age;
        writeLine("Happy birthday ");
        if (age == 75) {
          write("That's quite enough for one career");
          break;
        }
      until (false);
      writeLine(" - time to retire gracefully");
    }
  } // Main

  void Main () {
  // Demonstrate some set manipulations (not supposed to do anything
useful!)
    set empty, odd, all, even;
    set[] c = new set[200];
    empty = { };
    odd   = {1, 3, 5, 7, 11, 13 };
    all   = { 0 .. 14 };
    even  = all - odd;
    if (200 in all) writeLine("something funny here");
    incl(even, 6);
```

```
    excl(odd, 13);
    set a, b, full = all;
    a = {1 .. 12, 15, 20 .. 30};
    b = odd * a;  // intersection
    int i = 0;
    while (i < 100) {
      excl(all, i);
      incl(c[i], i * i);
      i++;
    }
  } // 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
```

```
void Main () {
  int i = 0;
  do
    i++;
  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 4 Railway management needs some help! [10 marks]

In some parts of the world, railway trains fall into three categories - passenger trains, freight trains, and mixed passenger and freight trains. A train may have one (or more) locomotives (or engines). Behind the locomotive(s) in freight or mixed trains come one or more freight trucks. A freight train is terminated by a brake van after the last freight truck; a mixed train, on the other hand, follows the freight trucks with one or more passenger coaches, the last (or only one) of which must be a so-called guard's van. A passenger train has no freight trucks; behind the locomotive(s) appears only a sequence of coaches and the guard's van. Freight trucks come in various forms - open trucks, closed

trucks, fuel trucks, coal trucks and cattle trucks. Sometimes locomotives are sent along the line with no trucks or coaches if the intention is simply to transfer locomotives from one point to another.

Here is a Cocol grammar that describes a list of correctly formed trains (`Trains.atg`):

```
COMPILER Trains $CN
/* Grammar for simple railway trains
   P.D. Terry, Rhodes University */

IGNORECASE

COMMENTS FROM "(*" TO "*)" NESTED

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Trains    = { OneTrain } EOF .
  OneTrain  = LocoPart [ [ GoodsPart ] HumanPart ] SYNC "." .
  LocoPart  = "loco" { "loco" } .
  GoodsPart = Truck { Truck } .
  HumanPart = "brake" | { "coach" } "guard" .
  Truck     = "coal" | "closed" | "open" | "cattle" | "fuel" .
END Trains.
```

As an interesting variation, and in the interests of safety, modify the grammar to build in the restrictions that fuel trucks may not be placed immediately behind the locomotives, or immediately in front of a passenger coach.

There are, of course, several ways in which these extensions might be done. You may find that you come up with grammars that look plausible but which are, in fact, not quite correct; the problem may be harder than it first appears. A sample set of correct and incorrect trains has been provided in the source kit (`Trains.txt`), and it is easy to make up examples of your own.

Develop a better grammar and generate and test a parser that will accept safe trains and reject incorrect trains, either because the rolling stock is marshalled in the wrong order, or because the safety regulations are violated.

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 demonstrators, who will hopefully be able to help you resolve all (or most) of them.
