Computer Science 3 - 2009

Programming Language Translation

Practical for Week 21, beginning 21 September 2009

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 g03A1234.** 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/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 or UltraEdit in a small Courier font. 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 Java 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 at:

`http://www.scifac.ru.ac.za/plagiarism_policy.pdf`

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 `PRAC21.ZIP` (Java version) or `PRAC21C.ZIP` (C# version)

- Immediately after logging on, get to the DOS command line level by using the `Start -> Command prompt` option from the tool bar.

- Copy the prac kit into a newly created directory/folder in your file space

```
j:
md  prac21
cd  prac21
copy  i:\csc301\trans\prac21.zip
unzip  prac21.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
```

- UltraEdit is probably your editor of choice. The version in the lab is configured to run various of the compilers easily, and has been tweaked to run Coco/R in much the same sort of way (see below). *To get this to work properly, start UltraEdit from a command window by giving the command* `UEDIT32`*, rather than by clicking on an icon on the desktop.*

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 106 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 program and see that it works.

A command like

```
crun  Calc  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

```
crun  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 `.java`

files are created in a subdirectory `Calc`) - 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, can apply an `abs()` function, and can compute factorials, as in

```
3 + abs(4 * (6 + 5)) - 4! + (5-2)!
```

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 problems before you begin, and while you are doing them.

Task 3 - Happy families

A family can be described in a format that makes provision for listing the parents, grandparents and children and also giving the numbers of cars, houses, cats and dogs - for example

Family:
    Terry
Parents:
    Patrick David,
    Sally
Children:
    Kenneth David,
    Helen Margaret Alice
Grandparents:
    William David Terry (deceased),
    Pep Terry,
    John Fawcitt,
    Winifred Doreen Fawcitt
1 car
1 house 2 cats

Write a grammar to describe such a family, some of whose members may have passed on (deceased). Assume that the various sections can come in any order (so we are allowed to name the grandparents before the parents, for example). Make provision for various forms of name - for example O'Toole, McGregor, or Stacey-Ann.

Task 4 - One for the Musicians in our Midst (but the rest of you should do it too)

After such a musical introduction to this section of the course you will be intrigued to learn that it is possible to write grammars to describe the words of songs and the notes sung to those words expressed in "Tonic Solfa"

Here is an example of a jingle that is probably familiar:

```
Happy birthday to you
so so la   so  do te

Happy birthday to you
so so la   so  re do

Happy Birthday, Happy Birthday
so so so me     do do te   la

Happy birthday to you!
fa fa me   do  re do
```

This tune (and two others) can be found in the prac kit.

A song can consist of any number of lines, but each line is written in two parts (one with the words, which can contain any number of letters and which might be terminated with commas, full stops, question marks or exclamation points, and which can contain quotes or hyphens) and one with the words of the tonic solfa (do, re, me, fa, so, la, te). Each of these component lines is terminated by an end-of-line mark, and each pair is separated from the next one by a further end-of-line mark.

Write a grammar to describe songs written in this way.

Task 5 - 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 the first level of Parva, taken from page 164. Generate a program from this that will recognise or reject simple 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
crun Parva voter.pav
crun Parva voter.bad -L
```

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

- The `%` operator
- A *do-while* loop, and *break* and *continue* statements
- Increment and decrement statements like `Curse++;` `--Temper;` and `Bug[N]--;` (treat these as statements, not as components of expressions).
- A *for* loop like the one in Java.
- A restriction that an identifier cannot end with an underscore.
- An optional *else* clause for the *if* statement.
- The ability to express numbers in hexadecimal or binary form, as well as in decimal.
- A character type, and functions for converting from character values to integer values and *vice versa*.

Here are some silly examples of code that should give you some ideas of what this implies:

```
void main () {
// Demonstrate syntax for various loops
// (not supposed to do anything useful!)
  int i = 1, k = 0;
  while (i < 10) {
    do {
      k = k + 1;
      if (k > 5) break;
    } while (k < i);
    i = i + 1;
  }
  for (i = 1; i <= 10; i = i + 1) {
    read(k);
    write(i, i * k);
  }
  i = 0;
  for (bool b = true; b || i < 10; b = ! b) i = i + 1;
  do i++; while (i < 10);
} // main

void main () {
// Demonstrate syntax for constants and characters
// (not supposed to do anything useful!)
  int i = 01101%, j = 03FH;
  char ch = 'a';
  for (ch = ch; ch <= 'z'; ch++)
    write(char(ch + 4), int(ch), i + j * ch);
} // main
```

```
void main () {
// Demonstrate various statements
// (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");
  }
  else
    if (age == 21) {
      write("party time!");
      int headache = 0, strain = 0;
      for (int beers = 20; beers > 0; beers--) {
        strain++; ++headache;
        if (strain % 8 == 0) {
          write("That\'s better"); strain = 0;
        }
      }
    }
  else if ((age > 21) && (age < 40)) write("over the hill, bru");
  else if (age > 70)
    write("take a new lover");
  else
    write("boring");
} // main
```

These little programs are in the kit (`test*.pav`), and you can easily write some more of your own.

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.

Task 6 - XML

XML (eXtensible Markup Language) is a powerful notation for marking up data documents in a portable way. XML code looks rather like HTML code, but has no predefined tags. Instead a user can create customized markup tags, similar to those shown in the following extract.

```
<!-- comment - a sample extract from an XML file -->
<personnel>
  <entry>
    <name>John Smith</name>
  </entry>
  <entry_2>
    <name>Joan Smith</name>
    <address/>
    <gender>female</gender>
  </entry_2>
</personnel>
```

An *element* within the document is introduced by an *opening tag* (like `<personnel>`) and terminated by a *closing tag* (like `</personnel>`), where the obvious correspondence in spelling is required. The name within a tag must start with a letter or lowline character (_), and may then incorporate letters, lowlines, digits, periods or hyphens before it is terminated with a > character. Between the opening and closing tags may appear a sequence of free format *text* (like `John Smith`) and further *elements* in arbitrary order. The free format text may not contain a < character - this is reserved for the beginning of a tag. An *empty element* - one that has no internal members - may be terminated by a closing tag, or may be denoted by an *empty tag* - an opening tag that is terminated by `/>` (as in `<address/>` in the above example). Comments may be introduced and terminated by the sequences `<!--` and `-->` respectively, but may not contain the pair of characters `--` internally (as exemplified above).

Develop a Cocol specification, incorporating suitable CHARACTER sets and TOKEN definitions for

(a) opening tags,
(b) closing tags,
(c) empty tags,
(d) free format text

and give PRODUCTIONS that describe complete documents like the one illustrated. You may do this conveniently on the page supplied at the end of the examination paper.

Incidentally, it should be noted that the full XML specification defines far more features than those considered here!

Appendix: Practical considerations when using Coco/R

For ease of use with the Java file and directory naming conventions, please

- **Do not use folder names (directory names) with spaces in them, such as "Prac 21"**
- 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
```

The laboratory installation of UltraEdit has been configured to link to Coco/R by using an option in the "advanced" pull down menu. To apply Coco/R to the file in the "current window", invoke UltraEdit from your working directory, for example

```
UEDIT32 Gram.atg
```

It is easy enough to configure a copy of UltraEdit you might have installed on your own computer to incorporate the option to invoke Coco/R. Use the Advanced ->; Tool Configuration pull down, and set up an option to read as in the example below

- The Command line should read **cmake %n**
- The Path line should read **%p**
- The Menu Item Name should be **Coco**
- Select the "Output to list box" option
- Tick the "Capture Output" option
- Remember to **Insert** the command into the menu

If the Coco/R generation process succeeds, the Java compiler is invoked automatically to try to compile the application. If this does not succeed, a Java compiler error listing is redirected to the file ERRORS, where it can be viewed easily by opening the file in UltraEdit.

Free standing use of Coco/R

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

```
cmake Gram
```

Like that, error messages are a little cryptic. In the form

```
cmake Gram -options m
```

the system will produce you a listing of the grammar file and the associated error messages, if any, in the file `LISTING.TXT`. If the Coco/R generation process succeeds the Java compiler is invoked automatically to try to compile the application. If this does not succeed, a Java compiler error listing is redirected to the file `ERRORS`, where it can be viewed easily by opening the file in UltraEdit.

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 means that a so-called LL(1) parser (which is what Coco/R generates for you) cannot 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 we have discussed in class - 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.