

# Computer Science 3 - 2017

## Programming Language Translation

### Practical for Week 6, beginning 21 August 2017

Hand in your solutions to this practical *before* lunch time on your next practical day, correctly packaged in a transparent folder with your cover and individual assessment sheets. 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 g13A1234.** Please resist the temptation to carve up the practical, with each group member only doing one task. The group experience is best when you work on tasks together.

#### Objectives:

Ja, well, no, fine. All you folks who thought this was a bit of a jorl so far, or have been thinking that IS projects are an excuse for back-sliding - think again. In this important practical you are to

- familiarize yourself with writing syntax-driven applications with the aid of the Coco/R parser generator, and
- study the use of symbol tables.
- develop a simple high-level compiler.

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>. You might also like to consult the web page at <http://www.cs.ru.ac.za/courses/CSc301/Translators/coco.htm>.

#### Outcomes:

When you have completed this practical you should understand

- how to attribute context-free grammars so as to allow a compiler generator to add semantic actions to a parser;
- the form of a Cocol description;
- how to construct and use symbol tables.
- something about high-level compilation.

#### To hand in:

This week you are required to hand in, besides the cover sheets (one per group member):

- Listings of your ATG files and the source of any auxiliary classes that you develop, produced on the laser printer by using the LPRINT utility. Listings get wide - take care not to go too wide!
- Electronic copies of your grammar files (ATG files), stored in a folder under one of the group's names.
- **Some examples of input files and the corresponding output produced by your systems.**

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 website.

## 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 `PRAC6.ZIP`.

- Immediately after logging on, get to the command line level in the usual way.
- Copy the prac kit into a newly created directory/folder in your file space

```
j:
md  prac6
cd  prac6
copy i:\csc301\trans\prac6.zip
unzip prac6.zip
```

- 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`    `*.TXT`    `*.BNF`    `*.FRAME`

## Task 2 - Checking on Tokens

In the prac kit you will find the following grammar in the file `TokenTests.atg`.

This grammar does nothing more than specify an application that will read a file of potential tokens and report on how the repeated calls to the scanner would allow information on the detected token to be displayed.

```
using Library;

COMPILER TokenTests $CN
/* Test scanner construction and token definitions - C# version
   The generated program will read a file of words, numbers, strings etc
   and report on what characters have been scanned to give a token,
   and what that token is (as a magic number). Useful for experimenting
   when faced with the problem of defining awkward tokens!

   P.D. Terry, Rhodes University, 2017 */

/* Some code that will be added to the parser class */

static void Display(Token token) {
    // simply reflect the fields of token to the standard output
    IO.Write("Line ");
    IO.Write(token.line, 4);
    IO.Write(" Column");
    IO.Write(token.col, 4);
    IO.Write(": Kind");
    IO.Write(token.kind, 3);
    IO.WriteLine(" Val |" + token.val.Trim() + "|");
} // Display

CHARACTERS /* You may like to introduce others */

sp          = CHR(32) .
backslash   = CHR(92) .
control     = CHR(0) .. CHR(31) .
letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
digit       = "0123456789" .
cstringCh   = ANY - "'" - control - backslash .
printable   = ANY - control .

TOKENS      /* You may like to introduce others */

ident       = letter { letter | digit } .
integer     = digit { digit }
            | digit { digit } CONTEXT ("..") .
double      = digit { digit } "." digit { digit } .
cstring     = "'" { cstringCh | backslash printable } "'" .
char        = '"' printable '"' .

IGNORE control
```

```

PRODUCTIONS

TokenTests
= { ( ANY
    | "."
    | ".."
    | "ANY"
  )
  } EOF      (. Display(token); .)
            (. Display(token); .)
.

END TokenTests.

```

Start off by studying this grammar carefully, and then making and executing the program.

- Note the `using` clause at the start. Clauses like these are needed so that the generated parser can make use of methods in the library namespaces mentioned.
- Coco/R requires various "frame files" to work properly. One of these by default is called `Driver.frame`. For some applications this is more conveniently copied to a file `GRAMMAR.frame` (where `GRAMMAR` is the name of the goal symbol) and then edited to add various extra features. This is discussed in Chapter 10 of the text. Such editing is not needed for task 3 but may be for the others.
- If there are any other aspects that you do not understand, please ask one of the tutors to explain them. But don't expect much help if you have not been coming to lectures lately.

Use Coco/R to generate and then compile source for a complete system. You do this most simply by

```
cmake TokenTests
```

A command like

```
TokenTests tokens.txt
```

will run the program `TokenTests` and try to parse the file `tokens.txt`, displaying information about the tokens it might or might not recognize. Study the output carefully and convince yourself that the scanner is doing a good job. Unfortunately it displays the "kind" of token as a magic number, but with a little effort you should be able to work out what is what - and if you look at the generated file `Scanner.cs` it might become clearer still.

If you give the command in the form

```
TokenTests tokens.txt -L
```

it will send an error listing to the file `listing.txt`, which might be more convenient. Try this out. (Actually this application will accept anything, so you cannot really generate much in the way of syntax errors. But if you are perverse you might think of a way to do so.)

### Task 3 - Some other tokens

This little application might be useful if you need to describe awkward tokens. Try extending it to do the following:

- A simple one to start with. Write a token definition for a student number as used at Rhodes, and a token definition for a student e-mail address, specifically on the Rhodes mail server.
- Add a general definition for an e-mail address, inspired by ones that you may have seen.
- In Pascal a string is demarcated by single quotes. It cannot extend over a line break, and if you want to include a single quote within the string you represent it by two quotes in succession, as in

```
'That''s a great idea - don''t you agree?'
```

- Suppose, rather than ignore a comment like `/* a simple comment in C# */` you actually wanted to treat a comment as a token - for example if you were writing a program that extracted all the comments in a

C# program and threw the rest away. Such programs do exist, believe it or not. Can you come up with a suitable token definition? Careful - a comment token may have internal / and \* characters just to be awkward!

A point that I have not stressed in class (*mea culpa* as the Romans would have said, or "my bad" as you lot say) is that the TOKENS grammar does **not** have to be LL(1). The tokens are, in fact, specified by what amount to Regular Expressions, and they are handled not by a recursive descent algorithm, but using the sort of automata theory you may remember from Computer Science 202.

**Your solutions to tasks 3 and 4 must be submitted by the end of this afternoon, please.**

#### Task 4 - Who are our most highly qualified staff?

You will remember meeting the staff earlier in the course - for example (`staff.txt` has a longer list):

```
Professor P.D. Terry, MSc, PhD .
Dr Mos Tsietzi .
C. Hubert H. Parry, BMus.
Prof Barry V. W. Irwin, PhD.
Mrs Caroline A. Watkins, BSc.
```

and, perhaps, coming up with something like the following grammar for parsing such a list (`Staff.atg`):

```
COMPILER Staff $CN
/* Describe a list of academic staff in a department
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  uLetter = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter = "abcdefghijklmnopqrstuvwxyz" .
  letter  = uLetter + lLetter .

TOKENS
  name    = uLetter { letter | "'" uLetter | "-" uLetter } .
  initial = uLetter "." .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Staff = { Person } EOF .
  Person = [ Title ] FullName { "," Degree } SYNC "." .
  FullName = NameLast { NameLast } .
  NameLast = { initial } name .
  Title = "Professor" | "Prof" | "Dr" | "Mr" | "Mrs" | "Ms" | "Mx" | "Miss" .
  Degree = "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci" |
    | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc" | "PhD" .

END Staff.
```

The HR Division (who should already know all these things, but like to pretend they do not, just to make work for people like Mrs Caroline A. Watkins) have asked us to prepare them a list extracted from the above list (or another one like it - each department has been asked to do this, and you could Get Rich by putting your compiler course to good use). The extract list must simply list all the initials and surnames for each of the staff who has a PhD, namely be something like

```
Dr P.D. Terry
Dr M. Tsietzi
Dr B.V.W. Irwin
```

(What a clever bunch. Work hard and you might join them some day.)

Add to the Cocol grammar the appropriate actions needed to perform this task, and let's get HR off our backs until tomorrow, when doubtless they will throw more adminstrivia in our direction.

For a problem as simple as this one does not really need to parameterise the parsing routines - it will suffice to store the "semantic state" in a few static fields in the `Parser` class, which can be introduced at the start of the `ATG` file. Take particular care with the way in which you add the actions - it is easy to put them in slightly wrong places, and then to wonder why the system does not give the results you expect.

**The file `Staff1.atg` in the kit is laid out for ease of editing.**

**Your solutions to tasks 3 and 4 must be submitted by the end of this afternoon, please.**

## Task 5 - Have fun playing trains again

The file `Trains.atg` contains a simple grammar describing trains, as discussed in Prac 3, though devoid of the "safety" regulations, which gave you such trouble a few weeks back. The file `Trains.txt` has some simple train patterns.

```
COMPILER Trains $CN
/* Grammar for simple railway trains
   P.D. Terry, Rhodes University, 2017 */

IGNORECASE

COMMENTS FROM "(" TO ")" NESTED

IGNORE CHR(0) .. CHR(31)

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

In Prac 3 you were at pains to check the regulations syntactically. It may be easier sometimes to check quasi-semantic features semantically. This is a good example. Without making any significant changes to the syntax of the grammar above, go on to add actions and attributes to the grammar so that Coco/R can generate an application that will

- Parse the data file and print out the train pattern to an output file
- Report on the type of train - passenger, freight, mixed freight/passenger, or empty (locos only)
- Check that the safety regulations ("static semantic constraints") have been obeyed (no fuel trucks immediately behind the locomotives or immediately in front of any coaches).

*Hints:*

- (a) It will be a good idea to copy the `Driver.frame` file, and from this to create a customized `Trains.frame` file that will allow the parser to direct its output to a new file whose name is derived from the input file name by a change of extension.
- (b) For a problem as simple as this one does not really need to parameterise the parsing routines - it will suffice to store the "semantic state" in a few static fields in the parser class, which can be introduced at the start of the ATG file. Take particular care with the way in which you add the actions - it is easy to put them in slightly wrong places, and then to wonder why the system does not give the results you expect. You may wish to explore the use of the `SemError` and `Warning` facilities (see page 152) for placing error and other messages in the listing file at the point where you detect that safety regulations have been disobeyed.

## Important - arrangement of files when using Coco/R to build applications

Coco/R as used with the `CMAKE` batch file used in these exercises requires that files be located as described below. In this description, "Application" is used to denote the name of the application, for example `Parva` or `Calc`.

`CMAKE` is designed to make the process of executing (first) Coco/R and (then) the C# compiler as easy as possible. Note that if the Coco/R phase is successful but the subsequent C# compilation fails, the compiler error messages will have been redirected to a file named `errors`, where they may be inspected by editing the file (you will have to keep reloading it!) or simply displaying it with the command `type errors`. Remember that this usually requires you to edit the `.ATG` file - don't be tempted to edit the `Parser.cs` file instead!

- `Driver.frame` (or the modified `Application.frame`), `Scanner.frame`, `Parser.frame` and `Application.atg` should all be in the `Base` directory (the directory into which you have unpacked the prac kit).

- Any auxiliary source files such as `Table.cs` should be placed in the subdirectory `Base/Application`
- These auxiliary classes should all be defined to belong to a namespace called `Application`.
- If you are on your own system, ensure that the `Library.cs` file with the sources for the local library is in the `Base` directory
- The files `Parser.cs`, `Scanner.cs` and `Application.cs` will be created in the `Base` subdirectory.

The remainder of this prac - and incorporating work that you should plan to do over the vacation - is a non-trivial task based on one of my "24 hour examinations". You are strongly urged to give it your best shot, as they say, and it should also give you an idea of the standard you must aim to reach confidently.

## Task 6 - Eliminating metabrackets in productions

In this course we have made frequent use of the Cocol/EBNF notation for expressing production rules, and in particular the use of the `{}` and `[]` meta-brackets introduced by Wirth in his 1977 paper. An example of a system of productions using this notation is as follows:

```

Home      = Family { Pets } [ Vehicle ] "house" .
Pets      = "dog" [ "cat" ] | "cat" .
Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
Family    = Parents { Children } .
Parents   = "Dad" "Mom" | "Mom" "Dad" .
Parents   = "Dad" | "Mom" .
Child     = "Margaret" | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .

```

In analysing such productions and/or testing the grammar it has often been suggested that:

- they be rewritten without using the Wirth brackets, but using right recursive productions and an explicit  $\epsilon$  (in Cocol this is just omitted), as for example

```

Home      = Family AllPets Transport "house" .
AllPets   = Pets AllPets | .
Transport = Vehicle | .
Pets      = "dog" PussyCat | "cat" .
PussyCat  = "cat" | .
Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
Family    = Parents Offspring .
Offspring = Offspring Children | .
Parents   = "Dad" "Mom" | "Mom" "Dad" .
Parents   = "Dad" | "Mom" .
Child     = "Margaret" | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .

```

- a check be made to see that all the non-terminals have been defined (this does not occur in the above case - `Children` is undefined);
- a check be made to see that each non-terminal is defined only once (this does not occur in the above case - there are two rules for `Parents`);
- a check be made to see that each non-terminal (other than the goal) must appear on the right side of at least one production (this does not occur in the above case; `Child` defines a non-terminal which does not appear in any other production).

As a useful service to others who might study this material in future years (and hopeful that, although you might encourage others to do so, you are not forced to do so yourself!), develop a system using `Coco/R` that will carry out the above transformations and checks.

The prac kit contains an executable version of this system for you to use to help understand better what is required. You can run this with a command like `ConvertEBNF 09.bnf` (and similarly for other examples).

Here are some suggestions for deriving a complete system of your own:

- In the kit you will find the skeleton of a Cocol grammar (`EBNFConverter.atg`) with suitable

definitions of character sets and tokens.

- (b) It should be obvious that you will need to set up a system for building up a list of modified productions (a sort of "symbol table"). In the source kit is supplied the skeleton of such a system, as suggested below:

```
class Entry {
    public String name;
    public List<String> text;

    public Entry(String name) {          // You may need other fields
        this.name = name;               // constructor
        this.text = new List<String>();
    }
} // Entry

class Table {

    // You may need other static variables and methods

    public const final int
        LHS   = 1,
        RHS   = 2,
        BOTH  = 3;

    static List<Entry> list = new List<Entry>();

    public static int Add(String name, int where)
    // Search for, and possibly add a nonterminal "name" to the table, according as
    // to "where" in a production it was found, and return the position as the
    // index where the name was either added previously or added by this call

    public static void AddText(String str, int i)
    // Add str to the list of strings for the i-th rule in the list

    public static void ListProductions()
    // List all productions to an output file

    public static void TestProductions() {
    // Check that all non-terminals have appeared once on the left side of
    // a production, and at least once on the right side of a production (other than
    // the first, which is assumed to define the goal symbol of the grammar)

    } // Table
```

- (c) To help you on your way, the first part of the attribution of the grammar might take the form:

```
COMPILER EBNFConverter $CN
/* Do remember your names! Type them, don't scribble them on a printout later.
   And include a comment about what this is supposed to do */

CHARACTERS
...

TOKENS
...

COMMENTS FROM "(" TO ")" NESTED

IGNORE control

PRODUCTIONS
EBNFConverter
= { Production }
EOF                                (. if (Successful()) {
                                   Table.ListProductions();
                                   Table.TestProductions();
                                   } .) .

Production
= SYNC nonterminal                (. string name = token.val;
                                   int i = Table.Add(name, Table.LHS);
                                   Table.AddText(name, i);
                                   Table.AddText("=", i); .)

    "=" Expression<name, i>
    SYNC "."                      (. Table.AddText(".\n", i); .) .

/* obviously a lot more needed here - it won't compile until you fix it! */

END EBNFConverter.
```

- (d) Inventing additional non-terminal names (without clashing with others that might already exist) might be done with the aim of deriving, for example

```
Home = Family HomeSeq1 HomeOpt2 "house" .
```

- (e) In the source kit will be found some other example data and production sets to assist in your development of the system.
- (f) After converting a set of productions from EBNF style to BNF style, try converting the BNF style productions once again to check that your system works consistently.
- (g) The example output give earlier retained the use of the simple ( parentheses ) for grouping elements of a sentential form. You might prefer to try to eliminate these parentheses as well as the [ option ] and { repetition } ones.

## Appendix - simple use of the List class

The prac kit contains `ListDemo.cs`, a simple example (also presented on the course web page) showing how the generic `List` class of C# can be used to construct a list of records of people's names and ages, and then to display this list and interrogate it. If you are uncertain on how to manipulate "generic data structures", you might like to study, compile and run the program at your leisure.

```
csharp ListDemo.cs
ListDemo
```