

# Computer Science 3 - 2008

## Programming Language Translation

### Practical for Week 24, beginning 6 October 2008

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 g03A1234.** 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 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 simple symbol tables.

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 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 simple symbol tables.

#### 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 or UltraEdit in small courier font (listings get wide - take care).
- Electronic copies of your grammar files (ATG files).
- Some examples of the output produced by your systems.

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/plag2008.doc>

## 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 PRAC24.ZIP (Java version) or PRAC24C.ZIP (C# version)

- Immediately after logging on, get to the command line level by using the Start -> All Programs -> Accessories -> Command prompt option from the tool bar.
- Copy the prac kit into a newly created directory/folder in your file space

```
j:
md  prac24
cd  prac24
copy i:\csc301\trans\prac24.zip
unzip prac24.zip
```

This will create several other directories "below" the prac24 directory:

```
L:\prac24
L:\prac24\Library
L:\prac24\EBNF
```

containing the Java classes for the IO Library and a skeleton table handler for a later task.

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,      *.PVM,      *.TXT      *.BAD      *.FRAME
```

- 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 - First steps to a Boolean calculator

In the kit you will find Bool.atg. This is a (partly) attributed grammar for a Boolean calculator, based on a fairly obvious grammar, and which can store values in any of 26 memory locations, inspiringly named A through Z.

```
import Library.*;
import java.util.*;

COMPILER Bool $CN
/* Boolean expression calculator
   P.D. Terry, Rhodes University, 2008 */

static boolean[] mem = new boolean[26];

IGNORECASE

CHARACTERS
    letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
TOKENS
    variable  = letter .

COMMENTS FROM "(" TO ")" NESTED
COMMENTS FROM "/" TO "/" NESTED

IGNORE CHR(0) .. CHR(31)
```

```

PRODUCTIONS
Bool
    (. int index = 0;
      boolean value = false;
      char var;
      for (int i = 0; i < 26; i++) mem[i] = false; .)
    = { Variable<out char var>
      "==" Expression
      ";"
    } EOF .

Variable<out char var>
    = variable
    .

Expression = Term { Or Term } .
Term       = Factor { [ And ] Factor } .
Factor     = "NOT" Factor | Primary { "" } .
Primary    = True | False | variable | "(" Expression ")" .
True       = "TRUE" | "1" .
False      = "FALSE" | "0" .
And        = "AND" | "&&" | "." .
Or         = "OR" | "|" | "+" .
END Bool.

```

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

- Note the `import` clauses at the start. 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 on page 267 of the text. Such editing is not really needed for this first application in this practical.
- 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 this week.

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

```
cmake Bool
```

A command like

```
crun Bool bool.txt
```

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

```
crun Bool bool.bad -L
```

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

### Task 3 - Complete the calculator

Of course, the calculator still only parses expressions, it does not evaluate them. Complete the attributes for the other non-terminals, so as to define a complete calculator.

### Task 4 - A better calculator

Now make the following extensions to the system:

- Modify the underlying grammar so that the basic production for the goal symbol is something like

```
Bool = { ( Variable "=" Expression | "print" Expression ) ";" } EOF .
```

that is, introduce two "statement" forms, one that assigns (without displaying the answer) and one that prints the value of an expression (without assigning it to a variable).

- Rather than assume that all memory locations are initially assigned known values of `false`, assume that they are initially "undefined", and flag as an error any attempts to use the value of a variable before it has been the target of an "assignment".
- The grammar as given attempts no "error recovery". How and where should this be introduced?

Test your system out thoroughly - give it both correct and incorrect data.

## Task 5 - Have fun playing trains again

The file `TRAINS.ATG` contains a simple grammar describing trains, as discussed in prac 21, 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, 2008 */

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 = "guard" | { "coach" } "brake" .
  Truck     = "coal" | "cold" | "open" | "cattle" | "fuel" .
END Trains.
```

In Prac 21 you were at pains to check the regulations syntactically. It may be easier sometimes to check quasi-syntactic features semantically. This is a good example. Without making any significant changes to the grammar, go on to add actions and attributes to the grammar so that it 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 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 264) for placing error and other messages in the listing file at the point where you detect that safety regulations have been disobeyed.

## Task 6 - A pretty-printer for PVM assembler code

A pretty-printer is a "compiler" that takes a source program and "translates" the source into the same language. That probably does not sound very useful! However, the "object code" is formatted neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

Develop a pretty-printer for programs written in the simple PVM assembler code (for which a grammar is given below, based on the one you met in the last test).

```
COMPILER PVMasm $CN
/* Grammar for subset of PVM assembler language
   P.D. Terry, Rhodes University, 2008 */
CHARACTERS
  control      = CHR(0) .. CHR(31) .
  Printable    = ANY - control .
  InString     = Printable - "'" .
  Digits       = "0123456789" .
  LF           = CHR(10) .
  CR           = CHR(13) .
TOKENS
  Number       = [ "-" ] Digits { Digits } .
  String       = "'" { InString } "'" .
  EOL          = LF .
  Comment      = ";" { Printable } CR .
IGNORE control - LF
PRODUCTIONS
  PVMasm      = { Statement } .
  Statement   = [ Number ] [ Instruction ] [ Comment ] SYNC EOL .
  Instruction = TwoWord | OneWord | PrintString .
  TwoWord     = ( "LDA" | "LDC" | "DSP" | "BRN" | "BZE" ) Number .
  OneWord     = (
    "ADD" | "AND" | "ANEW" | "CEQ" | "CGE" | "CGT" | "CLE" | "CLT"
    "CNE" | "DIV" | "HALT" | "INPB" | "INPI" | "LDV" | "LDXA" | "MUL"
    "NEG" | "NOP" | "NOT" | "OR" | "PRNB" | "PRNI" | "PRNL" | "REM"
    "STO" | "SUB" ) .
  PrintString = "PRNS" String .
END PVMasm.
```

The good news is that you will not have to develop any semantic analyzers, code generators or symbol table handlers in this project, but can assume that the source program is semantically correct if it is syntactically correct. One minor complication is that you cannot ignore comments, but fortunately PVM comments are very simply defined as tokens and can only appear in fixed positions.

A starting point is to enhance the grammar with actions that simply write each terminal as it is parsed, intermingled with actions that insert line feeds and spaces at appropriate points. As a refinement, arrange for the pretty printer to replace any of the optional "labels" that are allowed before an opcode with a correctly computed sequence. An example should clarify. Presented with an input file reading

```
; Demonstrate division by zero
LDC 100
2 LDC      0      ; push 0
5 DIV      ; divide
PRNI
7 HALT      ; terminate
```

a pretty printer might generate output like

```
          ; Demonstrate division by zero
0 LDC 100
2 LDC 0    ; push 0
4 DIV      ; divide
5 PRNI
6 HALT      ; terminate
```

*Hints:*

- (a) It will be a good idea to the `Driver.frame` file, and from this to create a customized `PVMasm.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) On page 255 there is a discussion of how one may write a production that associates an action with an

empty option, and you may find this very useful. For example, we might have a section of grammar that amounts to

```
A = B [ C ] D .
```

which might be suitably attributed

```
A = B [
      C ( . action when C is present . )
    ] D .
```

but in some situations we might prefer or need

```
A = B (   C ( . action when C is present . )
        |   ( . action when C is absent . )
    ) D .
```

Something to think about. Suppose we had changed the definitions of some parts of this grammar as summarized below, and then used the ANY keyword in the description of comments (ANY is discussed on page 255). Could you write a pretty-printer following on from this idea?

```
CHARACTERS
...
Letters    = 'A' .. 'Z' + 'a' .. 'z' .
LF         = CHR(10) .
CR         = CHR(13) .

TOKENS
...
Identifier = Letters { Letters | Digits } .
EOL        = CR LF .

PRODUCTIONS
...
Statement  = [ Number ] [ Instruction ] [ ";" { ANY } ] SYNC EOL .
```

## Task 7 - A cross reference generator for EBNF

In the prac kit you will find (in `EBNF.atg`) a familiar unattributed grammar describing EBNF productions, and some simple sets of EBNF productions with names like `PVMasm.ebnf` and `Parva.ebnf`. You can build a EBNF parser quite easily and try it out immediately.

A cross reference generator for EBNF is a program that analyses a set of productions and prints a list of the non-terminals in it, along with the line numbers where they were used. A cross reference listing can be extremely useful when you are asked to maintain very big programs. Such a listing, for the `PVMasm.ebnf` productions in the kit, might look like the one below (where negative numbers denote the lines where the identifier was "declared"):

PVMasm	-1	
Statement	1	-2
Number	2	4
Instruction	2	-3
Comment	2	
SYNC	2	
EOL	2	
TwoWord	3	-4
OneWord	3	-5
PrintString	3	-10
String	10	

The following are terminals, or undefined non-terminals

Number Comment SYNC EOL String

Modify the EBNF grammar and provide the necessary support routines (essentially add a simple symbol table handler) to be able to generate such a listing.

### Hints:

- (a) Hopefully this will turn out to be a lot easier than it at first appears. You will notice that the EBNF grammar makes references to non-terminals in only two places, so with a bit of thought you should be able to see that there are in fact very few places where the grammar has to be attributed. When you have thought about the implications of that hint, check out your ideas with a tutor, so as not to spend fruitless hours writing far more code than you need.
- (b) You will, however, have to develop a symbol table handler. This can make use of the `ArrayList` class in two ways. You will need a list of records of the identifiers. For each of these records you will also need a list of records of the line numbers. A class like the following might be a useful one for this purpose:

```
class Entry {
    public String name;
    public ArrayList<Integer> refs;
    public Entry(String name) {
        this.name = name;
        this.refs = new ArrayList<Integer>();
    }
} // Entry
```

and your `Table` class (a skeleton of which is supplied in the file `EBNF\Table.java`) could be developed from the following ideas:

```
class Table {

    public static void clearTable() {
        // Reset the table

    public static void addRef(String name, boolean declared, int lineRef) {
        // Enters ident if not already there, adds another line reference

    public static void printTable() {
        // Prints out all references in the table

    } // Table
```

You will be relieved to hear that each of these methods can be implemented in only a few lines of code, provided you think clearly about what you are doing.

- (c) This exercise can also be used to highlight a further application of the `StringBuilder` class that you used in Practical 24. Use an object of this type to build up the list of non-terminals that turn out to be "undefined", as shown in the example above.
- (d) What may have escaped your attention is that "support" classes (like `Table.java`) needed by an application (like `EBNF`) must be stored in a sub-directory whose name matches the goal symbol (like `EBNF`).

Yes, I know, this exercise can be done using classes other than `ArrayList` and `StringBuilder`. However, these classes are used in various illustrations in the course, and it is as well for you to be properly familiar with them.

## Appendix - simple use of the `ArrayList` class

The prac kit contains a simple example (also presented on the course web page) showing how the generic `ArrayList` class can be used to construct a list of records of people's names and ages, and then to display this list and interrogate it. You can compile and run the program at your leisure. It requires that you have Java 1.5 or later, or C# 2.0 or later - if not you will have to use the basic classes instead.