Computer Science 3 - 2010

Programming Language Translation

Practical for Week 24, beginning 4 October 2010

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/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 `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,    *.BNF,    *.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 - A simple calculator

In the kit you will find `Calc.atg`. This is an attributed grammar for a simple four function calculator that can store values in any of 26 memory locations, inspiringly named A through Z.

```
import library.*;
import java.util.*;

COMPILER Calc  $CN
/* Simple four function calculator with 26 memory cells
   P.D. Terry, Rhodes University, 2010 */

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

CHARACTERS
  digit      = "0123456789" .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .

TOKENS
  Number     = digit { digit } [ "." { digit } ] .
  Variable   = letter .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calc                             (. int index = 0; double value = 0.0;
                                      for (int i = 0; i < 26; i++) mem[i] = 0.0; .)
  = { Variable                     (. index = token.val.charAt(0) - 'A'; .)
      "=" Expression<out value>    (. mem[index] = value;
                                      IO.writeLine(value); .)
    } EOF .
```

```
Expression<out double expVal>      (. double expVal1 = 0.0; .)
= Term<out expVal>
  {   "+" Term<out expVal1>        (. expVal += expVal1; .)
    | "-" Term<out expVal1>        (. expVal -= expVal1; .)
  } .

Term<out double termVal>           (. double termVal1 = 0.0; .)
= Factor<out termVal>
  {   "*" Factor<out termVal1>     (. termVal *= termVal1; .)
    | "/" Factor<out termVal1>     (. termVal /= termVal1; .)
  } .

Factor<out double factVal>         (. factVal = 0.0; .)
=   Number                         (. try {
                                        factVal = Double.parseDouble(token.val);
                                      } catch (NumberFormatException e) {
                                        factVal = 0; SemError("number out of range");
                                      } .)
  | Variable                       (. int index = token.val.charAt(0) - 'A';
                                      factVal = mem[index]; .)
  | "(" Expression<out factVal> ")"
  .

END Calc.
```

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 the first applications 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 a complete calculator. You do this most simply by

```
cmake   CALC
```

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.

Task 3 - A better calculator

Make the following extensions to the system:

- Check and report on attempts to divide by zero (use the `SemError` method).
- Allow a user to make use of a `SQRT` function (to evaluate square roots) and also a `MAX` function, as exemplified by

```
A = sqrt(4 + 3 * 5) * max(E, F + 2)
```

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

```
Calc = { 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 0.0, 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 4 - Have fun playing trains again

The file `Trains.atg` contains a simple grammar describing trains, as discussed in Prac 22, 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, 2010 */

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 22 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 5 Eliminating metabrackets

During the translators 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:

(a) they be rewritten without using the Wirth brackets, but using right recursive productions and an explicit ε (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" .
```

(b) 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);

(c) 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`);

(d) 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-teminal which does not appear in any other production).

As a useful service to others who might take this course 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.

Here are some suggestions for deriving a complete system:

(a) In the kit you will find the skeleton of a grammar file `EBNF.atg` with suitable definitions of character sets and tokens similar to those you have seen before:

```
COMPILER EBNF $CN
/* Parse a set of EBNF productions
   P.D. Terry, Rhodes University, 2010 */

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  digit    = "0123456789" .
  noquote1 = ANY - "'" .
  noquote2 = ANY - '"' .

TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .

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

IGNORE  CHR(9) .. CHR(13)
```

```
PRODUCTIONS
   EBNF       = { Production } EOF .
   Production = nonterminal "=" Expression "." .
   Expression = Term { "|" Term } .
   Term       = [ Factor { Factor } ] .
   Factor     =   nonterminal
                | terminal
                | "[" Expression "]"
                | "(" Expression ")"
                | "{" Expression "}" .
END EBNF.
```

(b) It should be obvious that you will need to set up a "symbol table". In the kit is supplied the skeleton of table handler in the file `EBNF\Table.java` (see below).

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

```
COMPILER EBNF $CN
/* Do remember your name!
   And a comment about what this is supposed to do */


CHARACTERS
  ...
TOKENS
  ...
COMMENTS FROM "(*" TO "*)"  NESTED

IGNORE  CHR(9) .. CHR(13)

PRODUCTIONS
  EBNF
  = { 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);
                                    .)
    WEAK "=" Expression<name, i>
    SYNC "."                        (. Table.addText(".\n", i); .)

/* obviously a lot more needed here */

END EBNF.
```

(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 prac kit will be found some other example data and production sets to assist in your development of the system, and an executable derived from a model solution (`ebnfconverter.exe`). Try it out:

```
ebnfconverter.exe   home.bnf
```

(f) After converting a set of productions from EBNF to BNF, try converting the BNF productions once again to check that your system works consistently.

```
crun EBNF home.bnf > home.new
crun EBNF home.new > home.out
```

where `home.new` and `home.out` should effectively consist of the same productions (perhaps in a different order).

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

class Table {
                                    // You may need other static variables and methods
  public static final int
    LHS = 1,
    RHS = 2,
    BOTH = 3;

  static ArrayList<Entry> list = new ArrayList<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 rule in the table at position i

  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
  // each production, and at least once on the right side of each production

} // Table
```