Computer Science 3 - 2011

Programming Language Translation

Practical for Week 24, beginning 10 October 2011

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 g09A1234.** 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/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` for some useful tips on using Coco.

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. Listings get wide - take care not to go too wide!
- Electronic copies of your grammar files (ATG files) in a folder below `S:\csc301`
- 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:

```
J:\prac24
J:\prac24\library
J:\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
```

- If UltraEdit is 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 `Calc.atg` (as discussed in lectures) and `Bool.atg`. This latter 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, 2011 */

  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 var>               (. index = var - 'A'; .)
      "=" Expression                  (. mem[index] = value;
                                         IO.writeLine(value); .)
      ";"
    } EOF .

  Variable<out char var>
  = variable                          (. var = token.val.charAt(0); .)
  .

  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" | "||" | "+" .
  Not        = "NOT" | "!" .
END Bool.
```

Start off by studying this grammar carefully.

- 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 139 of the text. Such editing is not really needed for the first few tasks in this practical.

- If there are any other aspects that you do not understand, please ask one of the tutors to explain them.

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  (Java)          or   Bool  bool.txt (C#)
```

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 semantic 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 SYNC and WEAK be introduced? (Terry-speak for "please include them in your grammar".)

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

Task 5 - Avoid the Muddle in the Middle and Meddle with the Medley

The Pipe Band world just keeps showing up, doesn't it? The Cocol grammar below (`Event.atg`) describes a medley event at a Highland Gathering, with which by now you must be familiar.

```
COMPILER Event $CN
/* Describe a pipe band competition event - the Medley
   Piper Pat Terry, Rhodes University, 2011 */

IGNORECASE

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_".

TOKENS
  nameOfBand = letter { letter } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Event   = Band { Band } .
  Band    = nameOfBand Medley .
  Medley  = OneTune { OneTune } "break" .
  OneTune =   "March" | "SlowMarch" | "Jig" | "Hornpipe" | "Reel"
            | "Strathspey" { "Strathspey" } "Reel" .
END Event.
```

The organisers have decided that this year each band must play at least two jigs.

What would you have to add to this grammar so that the parser could disqualify any band that did not play at least two jigs, and would also tell you (a) the total number of bands that have competed in the event (b) the name of the band that played the medley with the greatest number of tunes (of any sort).

Those of you interested in learning more about what a Medley performance entails might like to look at the flash videos I have put on the web site of a few very good pipe bands, or listen to the corresponding MP3s only. For the tone-deaf, there is a much quieter specimen data file in the kit as well.

Task 6 - Regular Expressions

The Cocol grammar below (`RE.atg`) describes a sequence of regular expressions (written one to a line).

```
COMPILER RE $CN
/* Regular expression grammar
   P.D. Terry, Rhodes University, 2011 */

CHARACTERS
  lf         = CHR(10) .
  control    = CHR(0) .. CHR(31) .
  noQuote1   = ANY - control - "'" .
  noQuote2   = ANY - control - '"' .
  meta       = "()*|[]-?+" .
  simple     = ANY - control - "'" - '"' - meta .


TOKENS
  atomic     = simple .
  escaped    = "'" noQuote1 "'" | '"' noQuote2 '"' .
  EOL        = lf .
```

```
IGNORE  control - lf

PRODUCTIONS
  RE         = { Expression EOL } EOF .
  Expression = Term { "|" Term } .
  Term       = Factor { Factor } .
  Factor     = Element [ "*" | "?" | "+" ] .
  Element    = Atom | Range | "(" Expression ")" .
  Range      = "[" OneRange { OneRange } "]" .
  OneRange   = Atom [ "-" Atom ] .
  Atom       = atomic | escaped .
END RE.
```

After studying this grammar. go on to add appropriate actions and attributes so that you could generate a program that would parse a sequence of regular expressions and report on the alphabets used in each one. For example, given input like

```
a | b c d | ( x y z )*
[a-g A-G] [x - z]?
a? "'" z+
```

the output should be something like

```
Alphabet = a b c d x y z
Alphabet = A B C D E F G a b c d e f g x y z
Alphabet = ' a z
```

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.bnf` and `Parva.bnf`. You can build an EBNF parser immediately and try it out on some of these.

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 (whether for EBNF productions or, indeed for source code in any particular language) can be extremely useful when you are asked to maintain very big sources. Such a listing, for the `PVMAsm.bnf` productions in the kit, might look like the one below (where the convention is that negative numbers denote the lines where the identifiers were "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 two places only, 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 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

that records the corresponding 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 23. 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) The way we have set up Coco requires that any "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`).

(e) For this application, direct the output to a file, whose name can be derived from the input file name. Copy the `Driver.frame` file to create one called `EBNF.frame` (in your work directory) and follow the suggestions contained in the comments in this file, which are hopefully fairly clear.

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 (Java) or `List` class (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. 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.