Computer Science 3 - 2014

Programming Language Translation

Practical 6 for Week beginning 12 May 2014

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 pracs can take preference - 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):

- Electronic copies of your grammar files (ATG files) in a folder below `S:\csc301`
- Some examples of the output produced by your systems.
- 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!

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 by following the links at:

`http://www.scifac.ru.ac.za/` or from `http://www.ru.ac.za/`

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 `Command prompt`.
- 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    (Java)   or
      i:\csc301\trans\pra6c.zip    (C# version)
unzip  prac6.zip
```

This will create several other directories "below" the `prac6` directory, for example:

```
J:\prac6
J:\prac6\library
J:\prac6\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
```

Task 2 - A simple calculator

In the prac kit you will find an attributed grammar (`Calc.atg`) for a four function calculator, as discussed several times in lectures.

You can generate the Java (or C#) source files for this system and compile them with a command like

```
cmake  Calc
```

A command like

```
crun  Calc  data.txt   (Java)  or
Calc data.txt          (C#)
```

will run the calculator, taking input from the file `data.txt` and displaying the results on the screen.

As a (very daring) experiment, go on to modify the system to give you a five-function calculator, adding the moduls (%) operator and test it out. Also try running the system with data that is clearly incorrect, and observe the outcome. Can you improve the error reporting by using `SYNC` before the = sign that terminates each correct expression?

Task 3 - Checking on Tokens

In the prac kit you will find the following grammar in the file `TokenTests.atg` (Java below; the C# version in the C# kit is almost identical),

This grammar does nothing more than specify an application that will read a file of potential tokens and report on how repeated calls to the scanner would be able to report on what the tokens were. This is very similar to one that was displayed on the screen in the lecture a few days ago.

```
import library.*;

COMPILER TokenTests $CN
/* Test scanner construction and token definitions - Java 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, 2014 */

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

  static void display(Token token) {
  // Simply reflect the fields of token to the standard outpul
    IO.write("Line " + token.line + " Column");
    IO.write(token.col, 4);
    IO.write(":  Kind");
    IO.write(token.kind, 3);
    IO.writeLine("  Val |" + token.val.trim() + "|");
  } // display

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

  /* You may like to introduce others */

TOKENS

  ident      = letter { letter | digit } .
  integer    = digit { digit } .
  double     = digit { digit } "." digit { digit }  .
  string     = '"' { stringCh | backslash printable } '"' .
  char       = "'" ( charCh   | backslash printable ) "'" .

  /* You may like to introduce others */

IGNORE control

PRODUCTIONS

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

END TokenTests.
```

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. The C# version works in much the same way with `using Library;`.

- 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 10.7 of the text. Such editing is not really needed for tasks 3, 4, 5 and 6 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 lately.

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

```
cmake  TokenTests
```

A command like

```
crun  TokenTests  tokens.txt     (Java)   or
TokenTests  tokens.txt           (C#)
```

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.java` it might become clearer still.

If you give the command in the form

```
crun  TokenTests  tokens.txt  -L   (Java)   or
TokenTests  tokens.txt  -L         (C#)
```

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 in the way of input, 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 4 - Some other tokens

This little application might be useful if you need to describe awkward tokens. For example:

- The code as given might be having trouble recognizing `68.` as an integer number (68) followed by a period. We discussed the fix in class, you should remember. Try it out.

- Extend the definition of the `double` token to allow for numbers like `12.34, 12.34E+5, 1.2E4,` and `1234.0E-10`. Try out your application on a text file that contains tokens that it should recognize, and also incorrect ones like `12.34F+3` (use your imagination)

- Extend the definition of `ident` to allow you to recognize identifiers that can contain underscores and digits internally, as in `This_Is_2_Much` or `Whoops____I_did_it_again`, but cannot end with an underscore.

- Extend the number recognizers so that a number cannot begin with 0 (unless it is the number 0.xxx (double) or 0 (integer).

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.

Task 5 - Who are our most highly qualified staff?

You will remember meeting the staff in a previous practical (you can find this in `Staff1.txt`):

R. J. Foss, BSc, MSc, PhD.
Professor Philip Machanick, PhD.
Professor P. D. Terry, MSc, PhD.
George Clifford Wells, BSc(Hons), MSc, PhD.
Greg G. Foster, PhD.
James Connan, MSc.
Professor Edwin Peter Wentworth, PhD.
Dr Karen Lee Bradshaw, MSc, PhD.
Mrs Mici Halse, MSc.
C. Hubert H. Parry, BMus.
Professor Hannah Thinyane, BA, BSc, PhD.
Dr Barry V. W. Irwin, PhD.

Professor Alfredo Terzoli.
Ms Busi Mzangwa.
Yusuf M. Motara, MSc.
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, 2014 */

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" | "Dr" | "Mr" | "Mrs" | "Ms" | "Miss" .
  Degree    =   "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
              | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc" | "PhD" .
              /* and others like this if you really need them, */
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 R.J. Foss
Dr P. Machanick
Dr P.D. Terry
Dr G.C. Wells
Dr G.G. Foster
Dr E.P. Wentworth
Dr K.L. Bradshaw
Dr H. Thinyane
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 administrivia in our direction.

Oh, while you are about it, sometimes the degree qualifications mention where that degree was obtained. The file `Staff2.txt` in the kit is so annotated:

R. J. Foss, BSc (Natal), MSc(UNISA), PhD(Rhodes).
Professor Philip Machanick, PhD.
Professor P. D. Terry, MSc(Rhodes), PhD (Cantab).
George Clifford Wells, BSc(Hons) (Rhodes), MSc(Rhodes), PhD (Bristol).
Greg G. Foster, PhD(Rhodes).
James Connan, MSc(Stell).
Professor Edwin Peter Wentworth, PhD(UPE).
Dr Karen Lee Bradshaw, MSc(Rhodes), PhD(Cantab).
Mrs Mici Halse, MSc(Rhodes).
C. Hubert H. Parry, BMus(Cantab).
Professor Hannah Thinyane, BA, BSc (Adelaide), PhD(South Australia).
Dr Barry V. W. Irwin, PhD(Rhodes).

Professor Alfredo Terzoli.
Ms Busi Mzangwa.
Yusuf M. Motara, MSc.
Mrs Caroline A. Watkins, BSc (UCT).

How could you handle this sort of list *without adding to the already long list of possible qualifications?*

And by now you should know that "How would you" is PDT-speak for "write the code to do it"!

Hints:

(a) 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.

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, 2014 */

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 name if not already there, adds another line reference
  // What do you suppose is the purpose of the "declared" parameter?
  // Don't you hate it when people dont' write comments into their code?

  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 5. 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 - in the unlikely event that you do not have an up to date version of Java or C#, you will have to use the basic classes instead.