Computer Science 3 - 2016

Programming Language Translation

Practical for Week 6, beginning 22 August 2016

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 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`.

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), 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,    *.PAV,   *.TXT    *.BAD   *.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 repeated calls to the scanner would be able to report on what the tokens were.

```
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, 2016 */

/* 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 ");
    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" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS      /* You may like to introduce others */

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

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 `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 really needed for the tasks 3 and 4 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 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. For example:

- The code as given might be having trouble recognizing `68.` as an integer number (68) followed by a period. The way to get round this is discussed in the textbook. Look it up and incorporate the fix.

- 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 4 - A cross reference generator for Parva

In the prac kit you will find (in `Parva.atg`) an unattributed grammar for the Parva language more or less as you extended it in Prac 3, and some simple Parva programs with names like `voter.pav` and `voter.bad`. You can build a Parva parser quite easily and try it out immediately.

A cross reference generator for Parva would be a program that analyses a Parva program and prints a list of the identifiers in it, along with the line numbers where the identifiers were found. A cross reference listing can be extremely useful when you are asked to maintain very big programs. Such a listing, for the `voter.pav` program in the kit, might look like this (where negative numbers denote the lines where the identifier was "declared"):

```
main             -1
votingAge        -2   8
age              -3   6   7   8  11  15
eligible         -3  11  12  12  13  17  17
total            -3  13  13  17
allEligible      -4   9   9  18
voters           -5  11  13
canVote          -8   9  10
```

Modify the Parva grammar and provide the necessary support routines (essentially add a simple symbol table handler) to be able to generate such a listing on the standard output file (IO).

Hints: Hopefully this will turn out to be a lot easier than it at first appears. You will notice that the Parva grammar has been organised so that all the references to identifiers are directed through a non-terminal `Ident`, 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.

You will, however, have to develop a symbol table class. This can make use of the `List` 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 useful for the symbol table entries:

```
class Entry {
  public string name;
  public List<int> refs;
  public Entry(string name) {
    this.name = name;
    this.refs = new List<int>();
  }
} // Entry
```

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

```
class Table {

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

  public static void AddRef(string name, bool declared, int lineRef) {
  // Enters name 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.

Task 5 - Towards marketing a versatile calculator

All you have ever wanted - turn an enormous Wintel machine into a calculator!

The next few tasks aim to reinforce the attribute grammar concept by constructing a calculator that can (a) deal with Boolean or Integer expressions (b) store and retrieve values in variables of the user's choice (c) guard against mixed-mode errors (such as dividing Boolean values or applying the NOT operator to an integer value).

Health warning. This is the most complex prac yet in this course, so do it in stages. But DO it!

A grammar describing the input to such a calculator (`Calc.atg`) appears below. The hierarchy of the set of productions for `Expression` imposes a precedence ordering on operators similar to the one used in C++/C#/Java rather than the simpler set we have used previously in Parva.

```
using Library;

COMPILER Calc $NC
//  Put your names and a description here

static int ToInt(bool b) {
// return 0 or 1 according as b is false or true
  return b ? 1 : 0;
} // ToInt

static bool toBool(int i) {
// return false or true according as i is 0 or 1
  return i == 0 ? false : true;
} // ToBool

CHARACTERS
  digit      = "0123456789" .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  number     = digit { digit } .
  identifier = letter { letter | digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calc       = { Print | Assignment } "quit" .

  Assignment = Variable "=" Expression SYNC ";" .
  Print      = "print" Expression { WEAK "," Expression } SYNC ";" .

  Expression = AndExp { "||" AndExp } .
  AndExp     = EqlExp { "&&" EqlExp } .
  EqlExp     = RelExp { EqlOp RelExp } .
  RelExp     = AddExp [ RelOp AddExp ] .
  AddExp     = MultExp { AddOp MultExp } .
  MultExp    = UnaryExp { MulOp UnaryExp } .
  UnaryExp   = Factor | "+" UnaryExp | "-" UnaryExp | "!" UnaryExp .
  Factor     = Variable | Number | "true" | "false" | "(" Expression ")" .
  Variable   = identifier .
  Number     = number .
  MulOp      = "*" | "/" | "%"  .
  AddOp      = "+" | "-" .
  RelOp      = "<" | "<=" | ">" | ">=" .
  EqlOp      = "==" | "!=" .

END Calc.
```

To make up a parser proceed in the usual way

```
cmake Calc
Calc calc.txt
Calc calc.bad -L
```

If you simply do that the results will not be very interesting - you will be able to parse expressions, and detect syntactically incorrect expressions, but not much more.

Add actions to this grammar so that it will be able to evaluate the expressions. To start with, assume that expressions will not make use of any "variables".

Hints:

- If you study Chapter 9.3 of the textbook it should almost immediately become obvious how to do this.
- Experience has shown that it really does help to lay out Cocol descriptions with the "syntax" on the left and the "actions/attributes" on the right. Follow the examples in the book and in the example grammars. `Calc.atg` is already "spread out" to help in this regard.
- As a first approximation, represent Boolean values *false* and *true* by integer 0 and 1 (respectively). Introducing small `static` methods into the parser class may help in this regard. The code for these methods has already been incorporated into `Calc.atg` for you.
- Note the use of the `SYNC` and `WEAK` directives that will allow Coco/R to introduce a measure of error recovery into the system.

Bonus: Can anything go wrong in evaluating simple expressions? If so, how do you detect the fact and report it?

Task 6 - Fun with Variables

Extend the system so that values can be stored in and retrieved from variables.

Hints:

- Use the `List` class to set up a "symbol table" to record the names, status and values of the variables.
- Variables can be "declared" at the point where they are first encountered. Correctly, this should be on the "left" of an assignment, but you should also be able sensibly to handle the situation where undeclared variables appear within expressions before they have been declared and when, of course, their associated values will still be undefined.

Task 7 - Type checking

Just because in C++ you can mix integer and Boolean operands in almost any way in expressions does not mean that this is sensible - and of course you cannot really do that in Java or C# either. Extend your system so that it keeps track of the types of operands and expressions, and forbids mixing of the wrong types.

Hint: You might like to define an enumeration

```
const int
  noType   = 0,
  intType  = 1,
  boolType = 2;
```

to represent the various types that might be encountered. `intType` and `boolType` can be associated with correctly formed and defined operands and expressions, while `noType` can be associated with expressions and operands where the type is incorrect. Thus for example

```
4 + 5               is of type  intType
true || false       is of type  boolType
4 == 5              is of type  boolType
4 && 5              is of type  noType
(4 == 5) && !false  is of type  boolType
```

Task 8 - Short circuit evaluation of Boolean expressions

We spent some time earlier discussing short-circuit Boolean evaluation of expressions. Recall the idea that for short-circuit semantics to hold we can make use of the following logical identities:

```
x AND y ≡  if x  then y    else false
x OR y  ≡  if x  then true else y
```

If you did not do so earlier, try to refine your system so that it incorporates short-circuit semantics.

If you not think that this can be achieved by adding to the code you have developed so far, explain why not, and suggest what other approach you would need to adopt to incorporate short-circuit semantics.

Appendix - 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.

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 aon how to manipulate "generic data structures", you might like to study, compile and run the program at your leisure.

```
csharp ListDemo.cs
ListDemo
```