Computer Science 3 - 2009

Programming Language Translation

Practical for Week 24, beginning 12 October 2009

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,    *.PAV,    *.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 - Internet, anyone?

(This task is based on a problem set in an examination some years back. In the examination, candidates were given the basic grammar and asked to suggest the actions that had to be added.)

As you should be aware, IP addresses as used in Internet communication are typically expressed in "dotted quad" form, as exemplified by `146.231.128.6`. The IP address actually represents a 32 bit integer; the four numbers in the quadruple corresponding to successive 8 bit components of this integer. For humans, machine addressing is usually more memorable when expressed in "DNS" format, as exemplified by `terrapin.ru.ac.za`. Some systems maintain tables of matching addresses, for example

```
146.231.122.131   bungee.ru.ac.za       #comments appear like this
146.231.128.6     terrapin.ru.ac.za
146.231.56.10     thistle-sp.ru.ac.za
147.28.0.62       psg.com
```

When we moved our CS and IS departments to new premises in Hamilton on the famous 9/11 in 2001, a decision was made to rename and uniquely renumber all the many machines in our possession. Our system administrators tried to draw up a table like the one above, which was then merged with the existing table in the IT division. Unfortunately, a few mistakes were made, which caused havoc until they were ironed out. For example, there were lines reading

```
146.231.122.1123  pdt1.ict.ru.ac.za     #invalid IP address
146.231.122.156   pdt2.ict.ru.ac.za
146.231.122.156   pdt3.ict.ru.ac.za     #non-unique IP address
```

The ATG files below show how Coco/R might be used to develop a system that would enable a file in this format to be checked quickly, and the errors identified. One shows how parameters may be passed between parsing

methods, and the other how `static` variables in the parser class may be accessed from several methods:

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

COMPILER Check3 $CN
// Put your names and a description here
// This version uses parameters

IGNORECASE

CHARACTERS
  digit   = "0123456789" .
  letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  eol     = CHR(10) .

TOKENS
  number = digit { digit } .
  name   = letter { letter | digit | "." | "-" } .

COMMENTS FROM "#" TO eol

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Check3                        (. ArrayList<Integer> IPList = new ArrayList<Integer> (); .)
  =
  { Entry<IPList> }
  EOF                           (. if (Successful()) IO.writeLine("all correct"); .)
  .

  Entry<. ArrayList<Integer> IPL .>
  = IPNumber<IPL> name .

  IPNumber<. ArrayList<Integer> IPL .>
  =                             (. int n, m; .)
  Number<out n>
  "." Number<out m>             (. n = 256 * n + m; .)
  "." Number<out m>             (. n = 256 * n + m; .)
  "." Number<out m>             (. n = 256 * n + m;
                                   if (IPL.contains(n)) SemError("duplicate IP number");
                                   else IPL.add(n); .)
  .

  Number<out int n>
  =  number                     (. try {
                                     n = Integer.parseInt(token.val);
                                   } catch (NumberFormatException e) {
                                     n = 256;
                                   }
                                   if (n > 255) SemError("number too large"); .)
  .

END Check3.
```

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

COMPILER Check4 $CN
// Put your names and a description here
// This version uses a static Arraylist, but no parameters

static ArrayList <Integer> IPList = new ArrayList <Integer> ();

IGNORECASE  // as before
CHARACTERS  // as before
TOKENS      // as before
COMMENTS    // as before
IGNORE      // as before
PRODUCTIONS
  Check4
  =
  { Entry }
  EOF                           (. if (Successful()) IO.writeLine("all correct"); .)
  .
```

```
Entry
= IPNumber name .

IPNumber
=                              (. int n, m; .)
Number<out n>
"." Number<out m>              (. n = 256 * n + m; .)
"." Number<out m>              (. n = 256 * n + m; .)
"." Number<out m>              (. n = 256 * n + m;
                                  if (IPList.contains(n)) SemError("duplicate IP number");
                                  else IPList.add(n); .)
.


Number<out int n>
... as before

END Check4.
```

Start off by studying these grammars carefully, and then making and executing the program.

- Note the use of the `ArrayList` class that may prove useful in various other applications in this course. The above examples use the "generic" version of `ArrayList` and require Java 1.5 or 1.6. If you only have Java 1.4 you will have to use the original version of `ArrayList` - such code is to be found in the files `Check1.atg` and `Check2.atg`.

- Another simple standalone application using these classes - with and without generics - is provided in the kit (close C# equivalents are to be found in the C# prac kit for those who want to experiment with these).

- Note the declaration of `static ArrayList IPList` in the one grammar, which sets up the list of IP numbers that can be "globally" checked and added to by methods at the lower levels of the grammar.

As before, you should be able to use Coco/R to generate and then compile source for an application like this, as exemplified by

```
cmake  Check3  or  cmake  Check4
```

A command like

```
crun  Check3  ipdata.txt
```

will run the program `Check3` and try to parse the file `ipdata.txt`, sending error messages to the screen. Giving the command in the form

```
crun  Check4  ipdata.txt -L
```

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

Task 3 Extending the application

As you may know, multiple names may be given to a host (associated with one IP number) on the Internet. On the basis that we might wish to do this for the computers in our building, but still only assign each name once, extend the application so that it might react sensibly to data of a form exemplified below.

```
146.231.122.131   bungee.ru.ac.za       #comments appear like this
                  humfac.ru.ac.za       #alternative name
                  www.humfac.ru.ac.za   #and yet another name
146.231.122.75    bungee.ru.ac.za       #invalid - name already used
146.231.128.6     terrapin.ru.ac.za
146.231.56.10     thistle-sp.ru.ac.za
147.28.0.62       psg.com
147.28.0.67       psg.com               #invalid - name already used
146.231.122.1123  pdt1.ict.ru.ac.za     #invalid IP address
146.231.122.156   pdt2.ict.ru.ac.za
146.231.122.156   pdt3.ict.ru.ac.za     # non-unique IP address
```

Hint: This is not hard to do. Use the `ArrayList` class to create two different lists.

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 as you extended it in Prac 21, 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 is 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.

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 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 useful the symbol table entries:

```
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 `Parva\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

  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.

```
import library.*;

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

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

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

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
crun Calc calc.txt
crun 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". (`Calc.atg` is already "spread out" to help in this regard).

Hints:

- If you study page 227 of the textbook it should almost immediately become obvious how to do this.
- 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

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

Hints:

- Use the `ArrayList` 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

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

```
static final 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
```

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.

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 Calculator.

`CMAKE` is designed to make the process of executing (first) Coco/R and (then) the Java or C# compiler as easy as possible. Note that if the Coco/R phase is successful but the subsequent Java or 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 `tyep errors`. Remember that this usually requires you to edit the .ATG file - don't be tempted to edit the `Parser.java` or `Parser.cs` file instead!

For Java:

- `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.java` should be placed in the subdirectory `Base/Application`.
- These auxiliary classes should all be defined to belong to a package named `Application`.
- If you are on your own system, ensure that the `Base/library` subdirectory exists and has the class files for the local library (these should have been unpacked for you from the prac kit).
- The files `Parser.java`, `Scanner.java` and `Application.java` will be created in the `Base/Application` subdirectory.

For C#:

- `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.