Computer Science 3 - 2015

Programming Language Translation

Practical 6 for week beginning 5 October 2015

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 g13A9876** 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 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 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
unzip  prac6.zip
```

- You will 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,    *.TXT    *.BAD    *.FRAME
```

- You will also find another directory "below" the `prac6` directory:

```
J:\prac6
J:\prac6\ASSEM
```

  containing some auxiliary classes for a later task.

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.

```
using Library;

COMPILER Calc  $CN
/* Simple four function calculator with 26 memory cells
   P.D. Terry, Rhodes University, 2015 */

  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[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 = Convert.ToDouble(token.val);
                                         } catch (Exception) {
                                           factVal = 0; SemError("number out of range");
                                         } .)
    | Variable                        (. int index = token.val[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 `using` clause at the start, needed so that the generated parser can make use of methods in the library namespace mentioned. In larger applications you may need several libraries to be quoted.

- 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 tasks 2, 3, 4 and 5 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 calculator. You do this most simply by

```
cmake   CALC
```

A command like

```
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

```
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?

Task 4 - 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:

```
using Library;
using System.Collections.Generic;

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                     (. List<int> IPList = new List<int> (); .)
  =
  { Entry<IPList> }
  EOF                        (. if (Successful()) IO.WriteLine("all correct"); .)
  .

  Entry<. List<int> IPL .>
  = IPNumber<IPL> name .
  IPNumber<. List<int> 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 = Convert.ToInt32(token.val);
                                   } catch (Exception) {
                                     n = 256;
                                   }
                                   if (n > 255) SemError("number too large"); .)
  .

END Check3.
```

```
using Library;
using System.Collections.Generic;

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

static List<int> IPList = new List<int> ();

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
  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>
  =  number                     (. try {
                                     n = Convert.ToInt32(token.val);
                                   } catch (Exception) {
                                     n = 256;
                                   }
                                   if (n > 255) SemError("number too large"); .)
  .

END Check4.
```

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

- Note the use of the `List` class that may prove useful in various other applications in this course. Code for a simple application of the `List` class in C# is to be found in the prac kit, and also on the course website.

- Note the declaration of `static List 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

```
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

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

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

Task 5 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 `List` class to create two different lists.

Task 6 - An assembler for the PVM

Warning: This is an extended exercise. Please complete it, as by so doing you should learn a great deal.

On pages 141 - 144 of the text you will find a discussion of how Coco/R can be used to write a simple assembler for PVM code that allows for named labels to be introduced and then used as the targets of branch instructions (don't you wish you had been able to use this when you did Practical 2?)

The code for such a system is supplied in the prac kit. This includes a familiar PVM interpreter that is invoked after successful assembly, and which handles the extended opcode set from Prac 2. As usual, you can build it with the command

```
cmake  Assem
```

after which a command like

```
Assem  fact1.pvm
```

will execute the assembler/interpreter.

There are several simple PVM programs in the prac kit, so experiment with them. When you have had your fun, extend this system so that it will allow you to

- assemble programs that also using the extended opcodes (like `INC, DEC, LDL n, LDC_0, LDA_1`)
- use names for your "variables", in instructions like `LDA List`, as an alternative to using absolute addresses in instructions like `LDL 4`. The assembly process can build up a list of these names as they are introduced, and automatically allocate the corresponding offset addresses for the variable.
- deal correctly with a string used as argument to the `PRNS` opcode that has "escape sequences" in it.

Hints: Before trying to hack out a solution, spend some time studying the grammar as supplied and the source of the support classes that you can find in the `Assem\Table.cs` file.

This system also makes use of the `List` class, which by now you should have realised, is a handy one for building simple lists of objects of any convenient type in an array-like structure which will adjust its size as appropriate. It is suggested that the use of the `List` class is adequate for storing the *symbol tables* - the lists of labels and variables used in the program being assembled.

Do test your system out thoroughly. There are simple variations on the factorial program that you might have met in an earlier prac in files named `fact1.pvm` through `fact4.pvm`, and it is easy to invent further examples of your own. But try it out on very simple examples to begin with!

Hints:

(a) It is a good idea to add to the basic `Driver.frame` file, and from this to create a customized `Assem.frame` file, which among other things can will allow the parser to direct output to a new file whose name is derived from the input file name by a change of extension. This has been done for you in the file `Assem.frame`.

(b) You should explore the use of the `SemError` and `Warning` facilities (see page 137) for placing error and other messages in the listing file at the point where you detect that users need a firm hand!

(c) You can limit your escape sequence handling to `\n \t \f \"` and `\'`.

Something to think about (In this case, however, you do not have to code it up, because we may do something like that next week): Suppose someone were to suggest to you that a more sophisticated assembler system might try automatically to substitute one-word opcodes like `LDC_3` for the two-word opcodes like `LDC 3` that a user might have used. Would this be a good idea? How could it be implemented?

Task 7 - A cross reference generator for your assembler

A cross reference generator for the PVM assembly language would be a program that analyses a PVM assembler program and prints a list of the variables and labels 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 a factorial program like

```
ASSEM
BEGIN
        DSP    3            ; arg is v0, nFact is v1, i is v2
        LDC    1
        STL    arg          ; arg = 1;
WHILE1: LDL    arg
        LDC    20           ; // max = 20, constant
        CLE                 ; while (arg <= max) {
        BZE    EXIT1
        LDC    1
        STL    nFact        ;   nFact = 1;
        LDL    arg
        STL    i            ;   i = arg;
WHILE2: LDL    i
        LDC    0
        CGT                 ;   while (i > 0) {
        BZE    EXIT2
        LDL    nFact
        LDL    i
        MUL
        STL    nFact          ;     nFact = nFact * i;
        LDL    i
        LDC    1
        SUB
        STL    i              ;     i = i = 1;
        BRN    WHILE2         ;   }
EXIT2:  LDL    0
        PRNI                  ;   write(arg);
        PRNS   "! = "         ;   write("! = ");
        LDL    nFact
        PRNI                  ;   write(nFact);
        PRNL                  ;   writeLine();
        LDA    arg
        INC                   ;   arg++;
        BRN    WHILE1         ; }
EXIT1:
EXIT3:  HALT
END.
```

might look like this (where a negative number denotes the line where the label was "defined"):

```
Labels:

while1     (defined)      -6   35
exit1      (defined)       9  -36
while2     (defined)     -14   26
exit2      (defined)      17  -27
exit3      (defined)     -37

Variables:

arg       - offset  0     5    6   12   33
nfact     - offset  1    11   18   21   30
i         - offset  2    13   14   19   22   25
```

Modify the `Assem.atg` grammar, `Assem.frame` and `Table.cs` to provide the necessary support to be able to generate such a listing.

Hints: Hopefully this will turn out to be a lot easier than it at first appears. The `LabelEntry` class can be modified to incorporate a list of references to line numbers, and a simple method can be used to add to this list where necessary.

```
       class LabelEntry {

         public string name;
         public Label label;
#        public List<Int32> refs = null;

#        public LabelEntry(string name, Label label, int lineNumber) {
           this.name  = name;
           this.label = label;
#          this.refs  = new List<Int32>();
#          this.refs.Add(lineNumber);
         }

#        public void AddReference(int lineNumber) {
#          this.refs.Add(lineNumber);
#        } // AddReference

       } // end LabelEntry
```

Similarly, it is not hard to add a method to the `LabelTable` class that can display these numbers at the end of assembly. Furthermore, much the same strategy applies to the list of variables.

Finally, 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 further to construct these tables. 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.

Task 8 - A pretty-printer for PVM assembler code

A pretty-printer is a "compiler" that takes a source program and "translates" the source into the same language. That probably does not sound very useful! However, the "object code" is formatted neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

An example should clarify. Presented with an input file reading

```
; Demonstrate division by zero
      LDC 100
          LDC         0    ; push 0
    DIV   ; divide
      PRNI
           HALT   ; terminate
```

a pretty printer might generate output with the fields in fixed widths, for example:

```
            ; Demonstrate division by zero
LDC  100
LDC  0      ; push 0
DIV         ; divide
PRNI
HALT        ; terminate
```

Extend your assembler so as to incorporate a pretty-printer for PVM assembler code, as well as compiling it down to "real" PVM code ready for interpretation.

A starting point is to enhance the grammar with actions that simply write each terminal as it is parsed, intermingled with actions that insert line feeds and spaces at appropriate points.

Hints:

(a) One minor complication is that you cannot now ignore comments, but fortunately PVM comments are very simply defined as tokens and can only appear in fixed positions.

(b) Now the `Assem.atg` grammar in the kit, like the one in the text, has a description of the statements of the language, including

```
COMMENTS FROM "{" TO "}"
COMMENTS FROM ";" TO lf

IGNORE CHR(9) .. CHR(13)


PRODUCTIONS
  Assem     = "ASSEM" "BEGIN" { Statement } "END" "." .
  Statement = OneWord | TwoWord | WriteString | Label | Branch .
```

which might intrigue you - you may not have noticed that a `Label` is treated as a statement! (How on earth does that work?) But, if you are to retain comments, you might need to work more on the lines of

```
CHARACTERS
  ...
  lf = CHR(10) .

TOKENS
  ...
  Comment = ";" { printable }
  EOL = lf .

IGNORE CHR(9) .. CHR(13) - lf


PRODUCTIONS
  Assem     = "ASSEM" "BEGIN" { Statement } "END" "." .
  Statement = [ Label ] [ OneWord | TwoWord | WriteString | Branch ] [ Comment ]  EOL .
```

(c) Does this cater for completely blank lines?

(d) Remember that the library routines provided for your use allow for the field width to be specified as a second argument, for example

```
int i = 12;
output.Write(i, 15);       // writes 12 right-justified in a field of 15 characters
output.Write(2 * i, -8);   // writes 24 left-justified in a field of 8 characters.
```

(e) On page 131 there is a discussion of how one may write a production that associates an action with an empty option, and you may find this very useful. For example, we might have a section of grammar that amounts to

```
A = B [ C ] D .
```

which might sometimes be suitably attributed

```
A = B [
      C      (. action when C is present .)
    ] D .
```

while in other situations we might prefer or need

```
A = B (   C    (. action when C is present .)
        |      (. action when C is absent .)
      ) D .
```

This might make you think back to an earlier tutorial, where I pointed out that sometimes one might write a nullable component of a grammar like `{ Something }` as `[ Something { Something } ]`.

Task 9 - (Bonus question) Outperform the Emeritus Professor

I have often mentioned how much I enjoy it when students come up with better ideas than I have had. The kit contains suggestions and skeleton code for creating two symbol tables - one for labels and one for variables. As experts in Object Oriented Programming (OOP) you may think this is really a bit silly, and as with the high-level `Parva2ToCSharp` exercise a few weeks ago, "not the sort of code we would write". Surely we could set up some sort of base classes and inherit from these and come up with code that is much more elegant? Remember the Golden Rule - *Make it as simple as you can, but no simpler.*

If you have time, show me how to do it (that is, when you hand in your solution, let it incorporate this suggestion in the most elegant way possible, don't waffle on!) Extra marks if you try this, and a lollipop and two free entry passes to DJ Pat's forthcoming retro-disco for the best solution.