RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 6: week beginning 7th September 2020

All tasks in this practical should be submitted through the RUC submission link by 2 pm on Friday 25th September 2020.

Objectives

In this practical you will

- 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 might also like to consult the folder "Manuals and other documentation" on RUConnected 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 (30 marks)

- Electronic copies of your grammar files (ATG files) via RUConnected submission link.
- Electronic copies of the source of any auxiliary classes that you develop.

I do NOT require listings of any C# code produced by Coco/R.

For this practical, tutors will mark Task 2, and I will mark Task 3 or 4.

Feedback for the tasks marked will be provided via RUConnected. Check carefully that your mark has been entered into the Departmental Records.

You are expected to be familiar with the University Policy on Plagiarism and to heed the warning in previous practical handouts regarding "working with another student".

Task 0 Creating a working directory and unpacking the prac kit

Unpack the prac kit `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 (Assem) "below" the `prac6` directory containing some auxiliary classes for a later task.

Task 1 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 */

  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 133 of the course notes. Such editing is not really needed for task 2 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 2 A better calculator [10 marks] To be marked by tutors

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 `MAX` function, as exemplified by
  ```
  A = 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?

Please submit the code for your annotated grammar.

Task 3 An assembler for the PVM [10 marks]

On pages 157 – 160 of the course notes 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 3. 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 use the following extended opcodes

```
LDL N     STL N
LDA_0     LDA_1
LDC_0     LDC_1
INC       DEC
```

- 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.

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 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 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 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 152) for placing error and other messages in the listing file at the point where you detect that users need a firm hand!

Please submit the code for your annotated grammar and source code of any changed/new auxiliary routines you use.

Task 4 A cross reference generator for your assembler [10 marks]

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.

Please submit the code for your annotated grammar and the source code (.cs) for any of the auxiliary routines you have created/changed.