Computer Science 3 - 2014

Programming Language Translation

Practical 6 for week beginning 12 May 2014 - solutions

As usual, the sources of full solutions for these problems may be found on the course web page as the file `PRAC6A.ZIP` or `PRAC6AC.ZIP`.

While there were some splendid submissions, there were also some very weak ones, so please study the suggestions below, as the ability to add attributes and actions to grammars is crucially important if you are to use a tool like Coco. Furthermore, many people had not done as requested and provided specimen output, which at least might have given some indication of whether their systems worked.

Task 2 - The simple calculator

Adding the *mod* or % operator is very simple. Note again that one must check for division by zero

```
import library.*;

COMPILER Calc  $CN
/* Simple five function calculator - Java version
   P.D. Terry, Rhodes University, 2014 */

CHARACTERS
  digit      = "0123456789" .
  hexdigit   = digit + "ABCDEF" .

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calc                                  (. int expValue; .)
  = { Expression<out expValue>
      "="                               (. IO.writeLine(expValue); .)
    } EOF .

  Expression<out int expValue>          (. int termValue; .)
  = Term<out expValue>
    {   "+" Term<out termValue>         (. expValue = expValue + termValue; .)
      | "-" Term<out termValue>         (. expValue = expValue - termValue; .)
    } .

  Term<out int termValue>               (. int factValue; .)
  = Factor<out termValue>
    {   "*" Factor<out factValue>       (. termValue = termValue * factValue; .)
      | "/" Factor<out factValue>       (. if (factValue == 0)
                                             SemError("Quotient - Division by zero");
                                           else termValue = termValue / factValue; .)

***     | ( "%" | "mod" )
***           Factor<out factValue>     (. if (factValue == 0)
***                                          SemError("Modulus - Division by zero");
***                                        else termValue = termValue % factValue; .)
    } .

    Factor<out int factValue>           (. factValue = 0; .)
    =   decNumber                       (. try {
                                             factValue = Integer.parseInt(token.val);
                                           } catch (NumberFormatException e) {
                                           factValue = 0; SemError("number out of range");
                                           } .)
      | hexNumber                       (. try {
                                             factValue = Integer.parseInt(token.val.substring(1), 16);
                                           } catch (NumberFormatException e) {
                                             factValue = 0; SemError("number out of range");
                                           } .)
      | "(" Expression<out factValue>
        ")" .

END Calc.
```

Tasks 3 and 4 - Checking on Tokens

This is harder to get right than it at first appears. One has to be able to recognise and distinguish (assume spaces precede and follow the patterns below):

```
 30      (integer 30)
 0       (integer 0)
 030     (integer 0 followed by integer 30)
 30.     (integer 30 followed by period)
 0.      (integer 0 followed by period)
 9.      (integer 9 followed by period)
 .0      (period followed by integer 0)
 .9      (period followed by integer 9)
 0.3     (double 0.3)
 3.0     (double 3.0)
 3.3     (double 3.3)
```

Something like this seems to be required:

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

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, 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  /* You may like to introduce others */
  sp         = CHR(32) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit09    = "0123456789" .
  digit19    = "123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS      /* You may like to introduce others */
  ident      = letter { {"_"} (letter | digit09) } .

  integer    =   digit19 { digit09 }
               | digit19 { digit09 } CONTEXT(".")
               | "0"
               | "0" CONTEXT(".") .


  double     = ( digit19 { digit09 } "." | "0." ) digit09 { digit09 }
               [ "E" [ "+" | "-" ] digit09 { digit09 } ] .

  string     = '"' { stringCh | backslash printable } '"' .
  char       = "'" ( charCh   | backslash printable ) "'" .

IGNORE control

PRODUCTIONS

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

END TokenTests.
```

There may be a lurking bug in Coco/R that needs to be probed further. The alternative below does not work, although I see no reason why not!

```
integer    =   ( digit19 { digit09 } | "0" )
             | ( digit19 { digit09 } | "0" ) CONTEXT(".")
```

Most submissions fell down on one or more of those examples, possibly without noticing, and probably because they had not thought of exhaustive test data.

Task 5 - Who are our most highly qualified staff?

This was intended to be quite simple. A point many people missed was that first names had to be replaced by initials. Here is one (delinerately not the best) solution:

```
import library.*;

COMPILER Staff $CN
/* Describe a list of academic staff in a department, and extract those with PhD degrees
   P.D. Terry, Rhodes University, 2014 */

  static StringBuilder sb;
  static String lastName;

CHARACTERS
  uLetter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter   = "abcdefghijklmnopqrstuvwxyz" .
  letter    = uLetter + lLetter .
  control   = CHR(0) .. CHR(31) .
  varsity   = ANY - control - ")" .

TOKENS
  name      = uLetter { letter | "'" uLetter | "-" uLetter } .
  initial   = uLetter "." .
  university= '(' { varsity } ')' .


IGNORE control

PRODUCTIONS
  Staff     = { Person } EOF .

  Person                                          (. sb = new StringBuilder(); .)
  = [ Title ] FullName { "," Degree } SYNC "."  .

  FullName  = NameLast { NameLast } .

  NameLast  = { initial                           (. sb.append(token.val); .)
              } name                              (. sb.append(token.val.charAt(0));
                                                     sb.append('.');
                                                     lastName = token.val; .) .

  Title
  = "Professor" | "Dr" | "Mr" | "Mrs" | "Ms" | "Miss" .

  Degree
  = (  "BA" | "BSc" | "BCom" | "BMus" | "BEd"
     | "BSocSci"    | "BSc(Hons)"
     | "BCom(Hons)" | "MA" | "MSc"
     | "PhD"                                      (. String initials = sb.toString();
                                                     initials = initials.substring(0, initials.length() - 2);
                                                     IO.writeLine("Dr "  + initials + " " + lastName); .)
    )
    [ university ] .

END Staff.
```

The solution above appends an initial corresponding to the surname (the very last name) to the string builder for the initials, and then removes it again before the final name is printed. This might strike you as a bit inelegant. Indeed it should! Terry Theorem 2: There is always a better way:

```
PRODUCTIONS

  FullName  = NameLast {                          (. sb.append(lastName.charAt(0));
                                                     sb.append('.'); .)
              NameLast } .

  NameLast  = { initial                           (. sb.append(token.val); .)
              } name                              (. lastName = token.val; .)  .

  Degree
  = (  "BA" | "BSc" | "BCom" | "BMus" | "BEd"
     | "BSocSci"   | "BSc(Hons)"
     | "BCom(Hons)" | "MA" | "MSc"
     | "PhD"                                      (. String initials = sb.toString();
                                                     IO.writeLine("Dr "  + initials + " " + lastName); .)
    ) [ university ] .
```

Note the ability to introduce actions *before* a non-terminal is parsed, as shown in the *FullName* production. Kind of cute, don't you think? (see also the "trains" problem in the test solutions for another example).

And of course one can do still better (Theorem 2 again):

```
PRODUCTIONS

  FullName  = NameLast {                          (. sb.append(token.val.charAt(0));
                                                     sb.append('.'); .)
              NameLast }                          (. sb.append(" " + token.val); .) .


  NameLast  = { initial                           (. sb.append(token.val); .)
              } name .

  Degree
  = (  "BA" | "BSc" | "BCom" | "BMus" | "BEd"
     | "BSocSci"   | "BSc(Hons)"
     | "BCom(Hons)" | "MA" | "MSc"
     | "PhD"                                      (. IO.writeLine("Dr "  + sb.toString()); .)
    )
    [ university ] .
```

There was a tendency to write this sort of code:

```
COMPILER Staff $CN
/* When will some of you learn to put your names and a brief description at the start of your files? */

**    static StringBuilder sb = new StringBuilder();                                  // initialise

      ....

      Degree
      = (  "BA" | "BSc" | "BCom" | "BMus" | "BEd"
         | "BSocSci"   | "BSc(Hons)"
         | "BCom(Hons)" | "MA" | "MSc"
         | "PhD"                                  (. String initials = sb.toString();
                                                     IO.writeLine("Dr "  + initials + " " + lastName);
**                                                   sb = new StringBuilder();   // re-initialise .)
        ) [ university ] .
```

which strikes me as odd - have you really been taught to end *while* loop bodies with "initialisation" code?

Mind you, on that theme, somebody clearly teaches students to write `a = (-1) * b;` rather than more simply `a = -b`, not to mention that other horror, `if (someBoolExpression == true) ...` rather than simply `if (someBooleanExpression) ...`

Task 6 - Regular expressions

The problem provided a Cocol grammar that described a sequence of regular expressions (written one to a line), and asked that actions be added to generate a program that can 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
```

Once again, this problem is easily solved by making use of static (global) fields in the parser:

```
import library.*;

COMPILER RE $CN
/* Regular expression grammar - determine the underlying alphabet (Java version)
   P.D. Terry, Rhodes University, 2014 */

  static boolean[] alphabet = new boolean[256];

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                                (. int i; .)
  = {                               (. for (i = 0; i < 256; i++) alphabet[i] = false; .)
      Expression EOL                (. IO.write("Alphabet = ");
                                       for (i = 0; i < 256; i++)
                                         if (alphabet[i]) IO.write( (char) i, 2);
                                       IO.writeLine(); .)
    } EOF .

  Expression
  = Term { "|" Term } .

  Term
  = Factor { Factor } .

  Factor
  = Element [ "*" | "?" | "+" ] .

  Element                           (. char ch; .)
  =   Atom<out ch>                  (. alphabet[ch] = true; .)
    | Range | "(" Expression ")" .

  Range
  = "[" OneRange { OneRange } "]" .

  OneRange                          (. char ch, ch1, ch2; .)
  = Atom<out ch1>                   (. alphabet[ch1] = true; .)
    [ "-" Atom<out ch2>             (. if (ch2 < ch1)
                                         SemError("invalid range");
                                       else
                                         for (ch = ch1; ch <= ch2; ch++)
                                           alphabet[ch] = true; .)
    ] .

  Atom<out char ch>                 (. ch = '\0'; .)
  = (   atomic                      (. ch = token.val.charAt(0); .)
      | escaped                     (. ch = token.val.charAt(1); .)
    ) .

END RE.
```

Notes:

- In principle we need to add a character to the alphabet only the first time it is detected, so …

- The philosophical way to solve this sort of problem is to use a set (an alphabet is a set of symbols, the Good Book will tell you on page 86) so you might have been among those tempted to use the `IntSet` class or some other such class. However the implementation of `IntSet` and its friends is somewhat complex, and since a set and an array of booleans are conceptually the same, I think the simplest and fastest solution here is just to use a simple array, indexed by the character values, where setting an element to true denotes membership of that character. And in this case one has the advantage that one can "iterate" through the "set" to produce the output very simply as well.

- Once again, a simple global static structure will suffice, since the grammar is non-recursive.

- Note how we have handled the `OneRange` action. It is easy to overlook the fact that a range like `[a-g]` means `abcdefg`.

- The solution above assumes that the character encoding for the source file is ASCII. How would the Cocol grammar need to be modified to handle Unicode?

Task 7 - The EBNF cross reference generator.

Once again, this is capable of a very simple elegant solution (hint: most Pat Terry problems admit to a simple elegant solution; the trick is to find it, so learn from watching the Expert in Action, and pick up the tricks for future reference). There are only two places in the basic grammar where non-terminals appear, and it is here that we must arrange to insert them into the table:

```
import library.*;

COMPILER EBNF $CN
/* Parse a set of EBNF productions
   Generate cross reference table
   P.D. Terry, Rhodes University, 2014 */

  public static OutFile output;

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  lowline  = "_" .
  control  = CHR(0) .. CHR(31) .
  digit    = "0123456789" .
  noquote1 = ANY - "'" - control .
  noquote2 = ANY - '"' - control .

TOKENS
  nonterminal = letter { letter | lowline | digit } .
  terminal    = "'" noquote1 { noquote1 } "'" | '"' noquote2 { noquote2 } '"' .

COMMENTS FROM "(*" TO "*)"  NESTED

IGNORE control

PRODUCTIONS
   EBNF                           (. Table.clearTable(); .)
   =
     { Production
     }                            (. Table.printTable(); .)
     EOF .

   Production
   =
     SYNC nonterminal             (. Table.addRef(token.val, true, token.line); .)
     WEAK "="
     Expression
     SYNC "." .

   Expression
   =
   Term
   { WEAK "|" Term
   } .
```

```
  Term
  =
  [ Factor
    { Factor
    }
  ] .

  Factor
  =
      nonterminal                (. Table.addRef(token.val, false, token.line); .)
    | terminal
    | "[" Expression "]"
    | "(" Expression ")"
    | "{" Expression "}" .
END EBNF.
```

Of course, the bulk of the effort has to be spent in deriving a suitable table handler. In this case we can just write very simple code like the following. Running the search loop from the bottom makes for a very simple `addRef` method. Note that this handler allows us to enter an identifier that has been "used" before it has been "declared". While this may be "wrong" for some applications, it prevents crashes of the cross-referencer itself.

```
// Handle cross reference table for EBNF productions (C# version)
// P.D. Terry, Rhodes University, 2014

package EBNF;

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

  class Entry {                      // Cross reference table entries
    public String name;              // The identifier itself
    public ArrayList<Integer> refs;  // Line numbers where it appears

    public Entry(String name) {
      this.name = name;
      this.refs = new ArrayList<Integer>();
    } // constructor
  } // Entry

  class Table {
    static ArrayList<Entry> list = new ArrayList<Entry>();

    public static void clearTable() {
    // Clears cross-reference table
      list = new ArrayList<Entry>();
    } // clearTable

    public static void printTable() {
    // Prints out all references in the table (eliminate duplicates line numbers)
      StringBuilder missing = new StringBuilder();
      for (Entry e : list) {
        boolean isDeclared = false;                    // haven't seen a definition yet
        Parser.output.write(e.name, -18);              // left justify in 18 spaces
        int last = 0;                                  // impossible line number
        for (int line : e.refs) {                      // work through the list of references
          isDeclared = isDeclared || line < 0;
          if (line != last) {                          // a new line reference
            Parser.output.write(line, 5);              // justify in 5 spaces
            last = line;                               // remember we have printed this line ref
          }
        }
        Parser.output.writeLine();
        if (!isDeclared) missing.append(e.name + " "); // build up list of undeclared nonterminals
      }
      Parser.output.writeLine();
      if (missing.length() > 0) {                      // no need if there were none
        Parser.output.writeLine("The following are terminals, or undefined non-terminals");
        Parser.output.writeLine();
        Parser.output.writeLine(missing.toString());
      }
    } // printTable
```

```
/* A version that will print out duplicate line references if required is shown below,
   For interest this has used traditional for loops rather than the neater/later "for each" ones

    public static void printTable() {
    // Prints out all references in the table
      StringBuilder missing = new StringBuilder();
      for (int i = 0; i < list.size(); i++) {
        boolean isDeclared = false;                    // haven't seen a definition yet
        Entry e = list.get(i);
        Parser.output.write(e.name, -18);              // left justify in 18 spaces
        for (int j = 0; j < e.refs.size(); j++) {      // work through the list of references
          int line = e.refs.get(j);
          Parser.output.write(line, 5);                // justify in 5 spaces
          isDeclared = isDeclared || line < 0;
        }
        Parser.output.writeLine();
        if (!isDeclared) missing.append(e.name + " "); // build up list of undeclared nonterminals
      }
      Parser.output.writeLine();
      if (missing.length() > 0) {                      // no need if there were none
        Parser.output.writeLine("The following are terminals, or undefined non-terminals");
        Parser.output.writeLine();
        Parser.output.writeLine(missing.toString());
      }
    } // printTable

  */


    public static void addRef(String name, boolean declared, int lineRef) {
    // Enters name if not already there, adds another line reference (negative if at
    // a declaration point in the original set of productions)
      int i = 0;
      while (i < list.size() && !name.equals(list.get(i).name)) i++;
      if (i >= list.size()) list.add(new Entry(name));
      list.get(i).refs.add(new Integer(declared ? -lineRef : lineRef));
    } // addRef


  } // Table
```

Several solutions revealed that people either had not thought that far or were confused about the point of the `declared` argument to the `addRef` method.

Note that the members of the `Table` class are all static. There is no need to instantiate an object of this class, and it just makes it more complicated to try to do so, as some people did.

Few groups made a proper attempt to create an output file of the `OutFile` class.