Computer Science 3 - 2011

Programming Language Translation

Practical for Week 24, beginning 10 October 2011 - solutions

As usual, the sources of full solutions for these problems may be found on the course web page as the file `PRAC24A.ZIP` or `PRAC24AC.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.

Tasks 3 and 4 - The Boolean calculator.

Several people had a little difficulty with this, and ledt their answers incomplete. One possibility is to use two "parallel" arrays, one of the actual values for the variables, and one to mark those that had not yet been assigned values. One cannot use a single boolean field to distinguish between three states - undefined, defined (true) and defined (false).

Note that the solution below uses the `Variable` production to extract the index of the variable, not its name, and the use of `toUpperCase()` ensures that the system ignores case completely (the `IGNORECASE` directive applies only to key words). Note also that one cannot set `defined[index] = true;` immediately after parsing *Variable*, but only after parsing and computing the va ule of the *Expression*.

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

COMPILER Bool $CN
/* Boolean expression calculator Java version
   P.D. Terry, Rhodes University, 2011 */

  static boolean[] mem = new boolean[26];
  static boolean[] defined = new boolean[26];

IGNORECASE

CHARACTERS
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
TOKENS
  variable   = letter .

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

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Bool                             (. int index = 0;
                                      boolean value = false;
                                      for (int i = 0; i < 26; i++) defined[i] = false; .)
  = { ( Variable<out index>
        "=" Expression<out value>  (. mem[index] = value;
                                      defined[index] = true; .)
      |
        "print"
        Expression<out value>      (. IO.writeLine(value); .)
      )
      SYNC ";"
    } EOF .

  Variable<out int index>
  = variable                       (. index = Character.toUpperCase(token.val.charAt(0)) - 'A'; .)
  .

  Expression<out boolean value>    (. boolean termValue; .)
  = Term<out value>
    { Or Term<out termValue>       (. value = value || termValue; .)
    } .

  Term<out boolean value>          (. boolean factValue; .)
  = Factor<out value>
    { [ And ] Factor<out factValue>  (. value = value && factValue; .)
    } .
  Factor<out boolean value>         (. value = false; .)
  =   Not Factor<out value>         (. value = ! value; .)
    | Primary<out value>
      { "'"                         (. value = ! value; .)
      }
  .

  Primary<out boolean value>        (. int index;
                                       value = false; .)
  =   True                          (. value = true; .)
    | False                         (. value = false; .)
    | Variable<out index>           (. if (!defined[index])
                                         SemError("variable not defined");
                                       value = mem[index]; .)

    | "(" Expression<out value> ")"
  .

  True   = "TRUE" | "1" .
  False  = "FALSE" | "0" .
  And    = "AND" | "&&" | "." .
  Or     = "OR" | "||" | "+" .
  Not    = "NOT" | "!" .
END Bool.
```

This has altered the grammar to demand that a semicolon follow each statement so that it can be used as a synchronization point.

Task 5 - Avoid the Muddle in the Middle and Meddle with the Medley

Given a Cocol grammar that described a medley event at a Highland Gathering, the exercise was to add actions so that the parser could disqualify any band that did not play at least two jigs, and would also report on (a) the total number of bands that had competed in the event (b) the name of the band that played the medley with the greatest number of tunes (of any sort).

Some dreadfully complicated solutions were submitted. Try always to find an elegant solution.

```
import library.*;

COMPILER Event $CN
/* Describe simple pipe band competition event.  (Java version)
   P.D. Terry, Rhodes University, 2011 */

  static int bandCount = 0, longestCount = 0, tuneCount, jigCount;
  static String longestBand, nextBand;

IGNORECASE

CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_".

TOKENS
  nameOfBand = letter { letter } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Event
  = Band { Band }                   (. IO.writeLine(bandCount + " bands competed");
                                       IO.writeLine("Longest medley played by " + longestBand); .) .

  Band                              (. bandCount++; .)
  = nameOfBand                      (. nextBand = token.val; /* must save it here for later */ .)
    Medley                          (. if (jigCount < 2)
                                         SemError(nextBand + " disqualified - too few jigs");
                                       /* maybe add "else */ if (tuneCount > longestCount) {
                                         longestBand = nextBand;
                                         longestCount = tuneCount;
                                       } .) .

  Medley                            (. jigCount = 0; tuneCount = 0; .)
  = OneTune { OneTune }
    "break" .
    OneTune                           (. tuneCount++; /* at least one tune played here */ .)
    =   "March"
      | "SlowMarch"
      | "Jig"                         (. jigCount++;   /* count these separately */ .)
      | "Hornpipe"
      | "Reel"
      | "Strathspey"
        { "Strathspey"                (. tuneCount++; /* we can get several tunes here */ .)
        } "Reel"                      (. tuneCount++; /* and one more tune here too */ .) .

  END Event.
```

Notes:

- In a non-recursive system as simple as this one, the easiest way to ensure that all productions have access to all the variables is to declare them as static fields of the parser. Java does *not* have proper reference parameter passing (in this respect it can be quite awkward as a language) so you can pass values "down" but not "up", or in both directions (which you would really need to do here), unless you pass references to specially created objects of an appropriate class, and that is overkill for this problem.

- We have to count the total number of tunes and the total number of jigs separately, and these counts have to be reset to zero as each band begins to play.

- The `OneTune` production has a slightly deceptive name. A sequence of strathspeys and the mandatory reel contribute two or more tunes to the total.

- `nameOfBand` is the name of a terminal and is not (and cannot be) followed by `<parameters>`. Where it appears in the PRODUCTIONS it has to be followed *immediately* by an action to save the value of `token.val`. Every time you scan for a new token the value of `token.val` is refreshed - you cannot hope that it will magically remain unaltered until it is needed later.

- The semantic check on the number of jigs cannot be applied before the medley is complete - as can the check for the longest medley. Surely that is obvious, but in my experience there is a tendency to try applying the check as soon as a jig is detected.

- There is no need to use string variables to save strings like "March". We only have to *count* the tunes!

- Don't be tempted to play around with comparisons like `if (token.kind == Band_Sym)`. They require too deep a knowledge of the inner workings of Coco/R, and sooner or later will lead you astray.

- Several submissions did make use of parameters. Here is the important part of a (much more verbose) grammar developed in that way, for comparison:

```
  import library.*;

  COMPILER Event2 $CN

    static int bandCount = 0, longestCount = 0;
    static String longestBand, nextBand;

    static class Performance {
      public int jigCount = 0;
      public int tuneCount = 0;
    } // class Performance

  PRODUCTIONS
    Event2
    = Band { Band }                   (. IO.writeLine(bandCount + " bands competed");
                                         IO.writeLine("Longest medley played by " + longestBand); .) .

    Band                              (. Performance tunes = new Performance();
                                         bandCount++; .)
    = nameOfBand                      (. nextBand = token.val; /* must save it here for later */ .)
      Medley<tunes>                   (. if (tunes.jigCount < 2)
                                           SemError(nextBand + " disqualified - too few jigs");
                                         if (tunes.tuneCount > longestCount) {
                                           longestBand = nextBand;
                                           longestCount = tunes.tuneCount;
                                         } .) .
  Medley<Performance tunes>
  = OneTune<tunes>
    { OneTune<tunes> }
    "break" .

  OneTune<Performance tunes>         (. tunes.tuneCount++; /* at least one tune played here */ .)
  =   "March"
    | "SlowMarch"
    | "Jig"                          (. tunes.jigCount++;   /* count these separately */ .)
    | "Hornpipe"
    | "Reel"
    | "Strathspey"
      { "Strathspey"                 (. tunes.tuneCount++; /* we can get several tunes here */ .)
      } "Reel"                       (. tunes.tuneCount++; /* and one more tune here too */ .) .

END Event2.
```

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, 2011 */

  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.. However the implementation of `IntSet` 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 charactervalues,wheresettinganelementtotruedenotesmembershipofthatcharacter.Andinthiscaseonehastheadvantagethatonecan the "set" to produce the output very simpley 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, 2011 */

    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
// P.D. Terry, Rhodes University, 2011  (Java 1.5)

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>();
    }
  } // 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 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

    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

  } // Table
```

The `printTable` method above suffers from a possible disadvantage in that multiple occurrences of an non-terminal on one line, as in

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

create unnecessary duplicate entries. These could be eliminated in various ways; the simplest might be to do so at the output stage, rather than when they are added by the `addRef` method. Please yourself; here is my suggestion.

```
    public static void printTable() {
    // Prints out all references in the table (eliminate duplicates line numbers)
      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
        int last = 0;                                 // impossible line number
        for (int j = 0; j < e.refs.size(); j++) {      // work through the list of references
          int line = e.refs.get(j);
          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
          }
        }
        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
```

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. I apologize for the fact that I had included the wrong `Driver.frame` file in the kit, which must have confused the various groups who made no proper attempt to create an output file of the `OutFile` class.