Computer Science 3 - 2010

Programming Language Translation

Practical for Week 24, beginning 4 October 2010 - 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.

Few people had done as requested and provided specimen output, which at least would have given some indication of how well their systems worked.

Task 3 - The extended calculator.

Most people had little difficulty with this. Many used two "parallel" arrays, one of `double` values and one of `boolean` values, to solve the problem of not using variables before they had been given values. An alternative solution just to show how it might be done using an array of objects representing the encapsulated properties of each variable would be as follows:

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

COMPILER Calc  $CN
/* Simple four function calculator with 26 memory cells - extended
   P.D. Terry, Rhodes University, 2010 */

// We can include the simple class definition "in-line" as here:

  static class MemCell {
    public double value = 0.0;
    public boolean defined = false;
  }

  static MemCell[] mem = new MemCell[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] = new MemCell(); .)
  = { (
          Variable                    (. index = token.val.charAt(0) - 'A'; .)
          WEAK "="
          Expression<out value>       (. mem[index].value = value;
                                         mem[index].defined = true; .)
        | "print"
          Expression<out value>       (. IO.writeLine(value); .)
      ) SYNC ";"
    } 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>      (. if (termVal1 == 0.0) SemError("division by zero");
                                         else termVal /= termVal1; .)
    } .
```

```
  Factor<out double factVal>          (. factVal = 0.0;
                                        double factVal2; .)
  =   Number                          (. try {
                                           factVal = Float.parseFloat(token.val);
                                         } catch (NumberFormatException e) {
                                           factVal = 0; SemError("number out of range");
                                         } .)
    | Variable                        (. int index = token.val.charAt(0) - 'A';
                                        if (!mem[index].defined) SemError("undefined value");
                                        else factVal = mem[index].value; .)
    | "(" Expression<out factVal> ")"
    | "sqrt"
      "(" Expression<out factVal>     (. if (factVal < 0) {
                                           SemError("square root of negative value");
                                           factVal = 0.0;
                                         }
                                        else factVal = Math.sqrt(factVal); .)
      ")"
    | "max"
      "(" Expression<out factVal>
          ","
          Expression<out factVal2>    (. if (factVal2 > factVal) factVal = factVal2; .)
      ")" .

END Calc.
```

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

An alternative way of introducing the `max` function would be to allow it to have multiple arguments. This is easily done:

```
    | "max"
      "(" Expression<out factVal>
          { WEAK ","
            Expression<out factVal2> (. if (factVal2 > factVal) factVal = factVal2; .)
          }
      ")" .
```

Task 4 - Have fun playing trains again

Some dreadfully complicated solutions were submitted. Try always to find an elegant solution. Here is one, using a single static Boolean field to handle the safety problem:

```
import library.*;

COMPILER Trains1 $CN
/* Grammar for railway trains with simple safety regulations (Java version)
   P.D. Terry, Rhodes University, 2010 */

  static final int  // type of train
    passenger = 0,
    freight   = 1,
    mixed     = 2,
    empty     = 3;
  static int type;
  static boolean danger, hasFreight, safe;

  public static OutFile output;

IGNORECASE

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

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Trains1 = { OneTrain } EOF .

  OneTrain
  =                                   (. danger = false;
                                         type = empty;
                                         safe = true;
                                         hasFreight = false; .)
    LocoPart
    [ [ GoodsPart                     (. hasFreight = true; .)
      ]
```

```
      HumanPart
    ]
    SYNC "."                        (. output.writeLine(" .");
                                       switch(type) {
                                         case passenger:
                                           output.write("passenger train"); break;
                                         case mixed:
                                           output.write("mixed freight/passenger train"); break;
                                         case freight:
                                           output.write("freight train"); break;
                                         case empty:
                                           output.write("empty train"); break;
                                       }
                                       output.write(" - safety regulations ");
                                       if (safe) output.writeLine("obeyed");
                                       else output.writeLine("contravened");
                                       output.writeLine(); .) .

  LocoPart
  = "loco"                          (. output.write(" " + token.val); .)
    { "loco"                        (. output.write(" " + token.val); .)
    } .

  GoodsPart
  = Truck                           (. if (danger) {
                                         safe = false;
                                         SemError("fuel truck may not follow loco");
                                       } .)
    { Truck } .

  HumanPart
  =   "brake"                       (. output.write(" " + token.val);
                                       type = freight; .)
    |                               (. if (danger) {
                                         safe = false;
                                         SemError("fuel truck may not precede coach");
                                       } .)
      { "coach"                     (. output.write(" " + token.val); .)
      } "guard"                     (. output.write(" " + token.val);
                                       if (hasFreight) type = mixed;
                                       else type = passenger; .) .

  Truck
  = (   (   "coal" | "closed"
          | "open" | "cattle" )     (. danger = false; .)
      | "fuel"                      (. danger = true; .)
    )                               (. output.write(" " + token.val); .) .

END Trains1.
```

Several people were guided into using a set of state variables remembering the last kind of rolling stock parsed. Here is a solution on those lines:

```
import library.*;

COMPILER Trains2 $CN
/* Grammar for railway trains with simple safety regulations (Java version)
   P.D. Terry, Rhodes University, 2010 */

  static final int  // type of train
    passenger = 0,
    freight   = 1,
    mixed     = 2,
    empty     = 3;
  static final int  // kind of last component
    safeTruck = 1,
    fuelTruck = 2,
    humans    = 3,
    loco      = 4;
  static int type, lastSeen;
  static boolean hasFreight, safe;

  public static OutFile output;

IGNORECASE

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

IGNORE CHR(0) .. CHR(31)
```

```
PRODUCTIONS
  Trains2 = { OneTrain } EOF .

  OneTrain
  =                                 (. type = empty;
                                       safe = true;
                                       hasFreight = false; .)
    LocoPart
    [ [ GoodsPart                   (. hasFreight = true; .)
      ]
      HumanPart
    ]
    SYNC "."                        (. output.writeLine(" .");
                                       switch(type) {
                                         case passenger:
                                           output.write("passenger train"); break;
                                         case mixed:
                                           output.write("mixed freight/passenger train"); break;
                                         case freight:
                                           output.write("freight train"); break;
                                         case empty:
                                           output.write("empty train"); break;
                                       }
                                       output.write(" - safety regulations ");
                                       if (safe) output.writeLine("obeyed");
                                       else output.writeLine("contravened");
                                       output.writeLine(); .) .

  LocoPart
  = "loco"                          (. output.write(" " + token.val); .)
    { "loco"                        (. output.write(" " + token.val); .)
    }                               (. lastSeen = loco; .) .

  GoodsPart
  = Truck { Truck } .

  HumanPart
  =   "brake"                       (. output.write(" " + token.val);
                                       type = freight; .)
    |                               (. if (lastSeen == fuelTruck) {
                                         safe = false;
                                         SemError("fuel truck may not precede coach");
                                       }
                                       lastSeen = humans; .)
      { "coach"                     (. output.write(" " + token.val); .)
      } "guard"                     (. output.write(" " + token.val);
                                       if (hasFreight) type = mixed;
                                       else type = passenger; .) .

  Truck
  = (   (   "coal" | "closed"
          | "open" | "cattle" )     (. lastSeen = safeTruck; .)
      | "fuel"                      (. if (lastSeen == loco) {
                                         safe = false;
                                         SemError("fuel truck may not follow loco");
                                       }
                                       lastSeen = fuelTruck; .)
    )                               (. output.write(" " + token.val); .) .

END Trains2.
```

Task 5 - Eliminating metabrackets

With one or two exceptions, the solutions submitted were sorely incomplete!

The basic idea is quite simple - simply copy the text in the productions to a list of strings storing the tokens and metasymbols for each production. However, when the { `RepeatedOption` } or [`SingleOption`] construction is encountered, invent a name, store this in the list instead, and then start a stylised production with that name, whose right hand side incorporates the `RepeatedOption` or `SingleOption`. The fact that the whole system is recursive handles the problem of options within options and so on very effectively!

There are a few subtleties that one might not think of at first. For example, a production like

```
One = { B C | D E } "end" .
```

must be transformed to

```
One = Seq "end" .
Seq = ( B C | D E ) Seq | .
```

(including parentheses) and not to

```
One = Seq "end" .
Seq = B C | D E Seq | .
```

which would have been equivalent to

```
Seq = ( B C )    |    ( D E Seq )    | .
```

(I'll 'fess up - I also got this wrong when I first tried writing this system!)

```
COMPILER EBNF $CN
/* Convert a set of EBNF productions to use BNF conventions, and carry out
   some rudimentary checks on their being properly defined.  (Java version)
   P.D. Terry, Rhodes University, 2010 */

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
  = { Production } EOF              (. if (Successful()) {
                                         Table.listProductions();
                                         Table.testProductions();
                                       } .) .

  Production
  = SYNC nonterminal                (. String name = token.val;
                                       int i = Table.add(name, Table.LHS);
                                       Table.addText(name, i);
                                       Table.addText("= ", i); .)
    WEAK "=" Expression<name, i>
    SYNC "." .

  Expression<String s, int i>
  = Term<s, i> { WEAK "|"           (. Table.addText("|", i); .)
    Term<s, i> } .

  Term<String s, int i>
  = [ Factor<s, i> { Factor<s, i> } ] .

  Factor<String s, int i>           (. String name;
                                       int j; .)
  =   nonterminal                   (. name = token.val;
                                       j = Table.add(name, Table.RHS);
                                       Table.addText(name, i); .)
    | terminal                      (. name = token.val;
                                       Table.addText(name, i); .)
    | "["                           (. name = Table.newName(s, "Opt");
                                       Table.addText(name, i);
                                       j = Table.add(name, Table.BOTH);
                                       Table.addText(name, j);
                                       Table.addText("= ", j); .)
        Expression<s, j>
      "]"                           (. Table.addText("|", j); .)
```

```
    | "("                          (. Table.addText("(", i); .)
        Expression<s, i>
      ")"                          (. Table.addText(")", i); .)
    | "{"                          (. name = Table.newName(s, "Seq");
                                      Table.addText(name, i);
                                      j = Table.add(name, Table.BOTH);
                                      Table.addText(name, j);
                                      Table.addText("= ", j);
                                      Table.addText("(", j); .)
        Expression<s, j>           (. Table.addText(")", j);
                                      Table.addText(name, j);
                                      Table.addText("|", j); .)
      "}" .

END EBNF.
```

The table handler to match this might be coded as follows. Note the simple device of counting how many times each non-terminal is added to the left hand and right hand side of a production. This makes checking for undefined and redefined and unreachable non-terminals quite easy.

As the second thing that one might not think of - if an attempt is made to redefine a non-terminal, as in

```
One = "x" | "y" B .
B = "p" | "q" .
One = "x" One |  B "y" .
```

then to get the productions displayed neatly we have to be prepared to enter `One` into the table a second time. Remember that the non-terminals can be introduced into the grammar in any order (conventionally we take the first one to be the "goal", however).

```
// Table handler for EBNF -> BNF style converter
// P.D. Terry, Rhodes University, 2010 (Java version)

package EBNF;

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

  class Entry {
    public int left, right;
    public String name;
    public ArrayList<String> text;
    public Entry(String name) {
      this.left   = 0;
      this.right  = 0;
      this.name   = name;
      this.text   = new ArrayList<String>();
    }
  } // Entry

  class Table {

    public static final int
      LHS  = 1,
      RHS  = 2,
      BOTH = 3;

    static int extraNames = 0;
    static ArrayList<Entry> list = new ArrayList<Entry>();
    static int maxLength = 0;

    public static String newName(String oldName, String ext) {
      extraNames++;
      return oldName + ext + extraNames;
    } // newName

    public static boolean alreadyDefined(int position) {
      // Returns true if production at known position has already been defined
      return list.get(position).left > 0;
    } // alreadyDefined

    public static int add(String name, int where) {
    // Search for, and possibly add a nonterminal "name" to the table, according as
    // to "where" in a production it was found, and return the position as the
    // index where the name was either added previously or added by this call
    // debug: IO.writeLine("add " + name + " " + where);
```

```
      int position = 0;
      while (position < list.size() && !name.equals(list.get(position).name)) position++;
      if (position >= list.size()) {
        list.add(new Entry(name));
        if (name.length() > maxLength) maxLength = name.length();
      }
      else if (alreadyDefined(position)) {
        list.get(position).left++;
        position = list.size();
        list.add(new Entry(name));
      }
      switch (where) {
        case LHS  :
          list.get(position).left++; break;
        case RHS  :
          list.get(position).right++; break;
        case BOTH :
          list.get(position).left++;
          list.get(position).right++; break;
      }
      return position;
    } // add

    public static void addText(String str, int i) {
    // add str to the list of strings for the rule in the table at position i
    // debug: IO.writeLnie("addtext " + str);
      list.get(i).text.add(str);
    } // addText

/*
    public static void listProductions() {
    // List all productions to standard output file - simple version
      for (int i = 0; i < list.size(); i++)
        if (list.get(i).text.size() > 0) {        // only list if the non-terminal was defined
          IO.write(list.get(i).text.get(0), -maxLength);
          for (int j = 1; j < list.get(i).text.size(); j++) {
            IO.write(list.get(i).text.get(j), 0); // width of 0 inserts a leading space
          }
          IO.writeLine(" .");
        }
      IO.writeLine();
    } // listProductions
*/

    public static void listProductions() {
    // List all productions to standard output file - space out better to avoid long lines
      for (int i = 0; i < list.size(); i++)
        if (list.get(i).text.size() > 0) {        // only list if the non-terminal was defined
          int paren = 0;
          IO.write(list.get(i).text.get(0), -maxLength);
          for (int j = 1; j < list.get(i).text.size(); j++) {
            String s = list.get(i).text.get(j);
            if (s.equals("(")) paren++;
            else if (s.equals(")")) paren--;
            else if (s.equals("|") && paren == 0) {
              IO.writeLine();
              IO.write(' ', maxLength + 1);       // neater output for alternatives
            }
            IO.write(s, 0);                       // width of 0 inserts a leading space
          }
          IO.writeLine(" .");
        }
      IO.writeLine();
    } // listProductions

    public static void testProductions() {
    // Check that all non terminals have appeared once on the left side of
    // a production, and at least once on the right side of a production (other than
    // the first, which is assumed to define the goal symbol of the grammar
      boolean OK = true; // optimistic
      int i;
      for (i = 0; i < list.size(); i++) {
        if (list.get(i).left == 0) {
          OK = false;
          IO.writeLine("(* " + list.get(i).name + " never defined *)");
        }
        else if (list.get(i).left > 1) {
          OK = false;
          IO.writeLine("(* " + list.get(i).name + " defined more than once *)");
        }
      }
```

```
    for (i = 1; i < list.size(); i++) {
      if (list.get(i).right == 0) {
        OK = false;
        IO.writeLine("(* " + list.get(i).name + " cannot be reached *)");
      }
    }
    if (!OK) IO.writeLine("\n(* Cannot be correct grammar *)");
  } // testProductions

} // Table
```

Another solution might be as follows. Here we have taken cognisance of the fact that there is nothing really wrong with productions like

```
Supper =  "pizza" .
Supper =  "lasagna" .
```

which are of course equivalent to

```
Supper = "pizza" | "lasagna" .
```

so that if a further rule is discovered for a nonterminal we can simply concatenate the alternatives together.

The table handler needs an addition (only the changes are shown here)

```
public static int positionLHS(String name) {
// Return the position in the table where name can be found on the LHS,
// or -1 if not yet entered found in that way
// debug: IO.writeLine("positionLHS " + name + " " + size());
  int position = 0;
  while (position < list.size() && !name.equals(list.get(position).name)) position++;
  return (position >= list.size() || list.get(position).left == 0) ? -1 : position;
} // positionLHS
```

The grammar also needs a few changes - note the use of the `Warning` method!

```
Production
= SYNC nonterminal                (. String name = token.val;
                                     int i = Table.positionLHS(name);
                                     if (i == -1) {
                                       i = Table.add(name, Table.LHS);
                                       Table.addText(name, i);
                                       Table.addText("= ", i);
                                     } else {
                                       Table.addText("|", i);
                                       Warning("More than one production found for " + name);
                                     } .)
  WEAK "=" Expression<name, i>
  SYNC "." .
```