Computer Science 3 - 2008

Programming Language Translation

Practical for Week 24, beginning 6 October 2008 - 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.

Many people had not done as requested and provided specimen output, which at least would have given some indication of how well their systems worked.

Tasks 3 and 4 - The Boolean calculator.

Most people had little difficulty with this. Most used two "parallel" arrays, one of the actual values for the variables, and one to mark those that had not yet been assigned values.

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.

```
import Library.*;
import java.util.*;

COMPILER Bool $CN
/* Boolean expression calculator Java version
   P.D. Terry, Rhodes University, 2008 */

  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" | "||" | "+" .
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 - Playing with 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 Trains $CN
/* Grammar for railway trains with simple safety regulations
   P.D. Terry, Rhodes University, 2008 */

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

  public static OutFile output;

IGNORECASE

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

IGNORE CHR(0) .. CHR(31)

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

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

END Trains.
```

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
   P.D. Terry, Rhodes University, 2008 */

  static boolean
    hasFreight, safe;
  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;

  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
  = "guard"                              (. 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); .)
      } "brake"                          (. output.write(" " + token.val);
                                            if (hasFreight) type = mixed;
                                            else type = passenger; .)
  .

  Truck
  = (   (   "coal" | "cold"
          | "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 6 - An assembler pretty printer

This is really rather easy once you get the general idea. A few points are worth making. Firstly, several people attempted to get the desired spacing by arranging for the output to contain "tabs" at suitable points. These have the disadvantage that tab setting differ from one output medium or library to another, and what might look pretty in some situations won't look pretty in others. Better far to use the "fixed width" output routines in the `Library`. Secondly, it would have been necessary to augment the grammar to handle the extended opcode set that includes such operations as `LDC_0`. Thirdly (and rather subtly), comments extended from the semicolon up to and including the `CR`. The convention on Wintel systems is that ends of lines are marked by a `CR LF` sequence, rather then `LF` only. Few people had realised that this called for special treatment (and doubtless did not notice, as most editors will handle `LF` or `CR LF` as almost equivalent.

```
import Library.*;

COMPILER PVMAsm $CN
/* Grammar for subset of PVM assembler language
   Pretty printer
   P.D. Terry, Rhodes University, 2005 */
```

```
  public static OutFile output;
  static int count = 0;

CHARACTERS
  control     = CHR(0) .. CHR(31) .
  Printable   = ANY - control .
  InString    = Printable - '"' .
  Digits      = "0123456789" .
  LF          = CHR(10) .
  CR          = CHR(13) .

TOKENS
  Number      = [ "-" ] Digits { Digits } .
  String      = '"' { InString } '"' .
  EOL         = LF .
  Comment     = ";" { Printable } CR .

IGNORE control - LF

PRODUCTIONS
  PVMAsm
  = { Statement } EOF .

  Statement
  = [ Number ]
    (                                   (. output.write(count, -5); .)
        Instruction
      |                                 (. output.write(" ", -27); .)
    )
    [ Comment                           (. output.write(token.val.substring(0, token.val.length() - 1)); .)
    ] SYNC EOL                          (. output.writeLine(); .)
  .

/*  Alternative that puts the count in place even for lines with no instruction
  Statement
  = [ Number ]                          (. output.write(count, -5); .)
    (   Instruction
      |                                 (. output.write(" ", -22); .)
    )
    [ Comment                           (. output.write(token.val.Substring(0, token.val.length() - 1)); .)
    ] SYNC EOL                          (. output.writeLine(); .)
  .
*/

  Instruction
  = TwoWord | OneWord | PrintString .

  TwoWord
  = (   "LDA" | "LDC" | "DSP"
      | "LDL" | "STL"
      | "BRN" | "BZE"
    )                                   (. output.write(token.val, -7); .)
    Number                              (. output.write(token.val, -15);
                                           count += 2; .)
  .

  OneWord
  = (   "ADD"   | "AND"   | "ANEW"   | "CEQ"
      | "CGE"   | "CGT"   | "CLE"    | "CLT"
      | "CNE"   | "DIV"   | "HALT"   | "INPB"
      | "INPI"  | "LDV"   | "LDXA"   | "MUL"
      | "NEG"   | "NOP"   | "NOT"    | "OR"
      | "PRNB"  | "PRNI"  | "PRNL"   | "REM"
      | "STO"   | "SUB"
    )                                   (. output.write(token.val, -22);
                                           count++; .)
  .

  PrintString
  = "PRNS"                              (. output.write(token.val, -7); .)
    String                              (. output.write(token.val, -15);
                                           count += 2; .)
  .

END PVMAsm.
```

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, 2005 */

  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. Here is one that builds upon the suggestion in the prac sheet:

```
// Handle cross reference table for EBNF productions
// P.D. Terry, Rhodes University, 2008

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>();
  }

  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));
  }

  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());
    }
  }

} // 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 undeclares 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());
    }
  }
```

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.