Computer Science 3 - 2015

Programming Language Translation

Practical for Week 6, beginning 5 October 2015 - solutions

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

```
using Library;

COMPILER Calc  $CN
/* Simple four function calculator with 26 memory cells - extended
   P.D. Terry, Rhodes University, 2015 */

  class MemCell {
    public double value = 0.0;
    public bool 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[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 = Convert.ToDouble(token.val);
                                        } catch (Exception) {
                                          factVal = 0; SemError("number out of range");
                                        } .)
    | Variable                        (. int index = token.val[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 (the solution here will even work if there is only one argument - max(5.0) is 5.0 of course!)

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

Task 4 - Internet, anyone?

This was intended to be quite simple. Here is one solution, passing parameters:

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

using Library;
using System.Collections.Generic;

COMPILER Check5 $CN
// Check lists of IP numbers and computer names for duplicates and errors
// P.D. Terry, Rhodes University, 2015
// This version uses parameters
//
// Typical data
// 146.231.122.131   bungee.ru.ac.za       #comments appear like this
//                   humfac.ru.ac.za       #alternative name
// 146.231.122.75    bungee.ru.ac.za       #invalid - name already used
// 146.231.122.1123  pdt1.ict.ru.ac.za     #invalid IP address
// 146.231.122.156   pdt2.ict.ru.ac.za
// 146.231.122.156   pdt3.ict.ru.ac.za     # non-unique IP address

IGNORECASE

CHARACTERS
  digit   = "0123456789" .
  letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  number = digit { digit } .
  name   = letter { letter | digit | "." | "-" } .

COMMENTS FROM "#" TO eol

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

```
PRODUCTIONS
  Check5                        (. List<int>     IPList   = new List<int> ();
                                   List<string>  NameList = new List<string>(); .)
  =
  { Entry<IPList, NameList> }
  EOF                           (. if (Successful()) IO.WriteLine("all correct"); .)
  .

  Entry<. List<int> IPL, List<string> NL .>
  = IPNumber<IPL> Name<NL> { Name<NL> } .

  Name<. List<string> NL .>
  =
  name                          (. string s = token.val;
                                   if (NL.Contains(s)) SemError("duplicate computer name");
                                   else NL.Add(s); .)
  .

  IPNumber<. List<int> IPL .>
  =                             (. int n, m; .)
  Number<out n>
  "." Number<out m>             (. n = 256 * n + m; .)
  "." Number<out m>             (. n = 256 * n + m; .)
  "." Number<out m>             (. n = 256 * n + m;
                                   if (IPL.Contains(n)) SemError("duplicate IP number");
                                   else IPL.Add(n); .)
  .

  Number<out int n>
  =  number                     (. try {
                                     n = Convert.ToInt32(token.val);
                                   } catch (Exception) {
                                     n = 256;
                                   }
                                   if (n > 255) SemError("number too large"); .)
  .

END Check5.
```

This system is haschsimple enough that the two lists can be defined as static, accessible to all the methods in the parser class. Here is a solution that does that. **Warning:** Be careful about trying to make everything global and static. It doesn't usually work properly for highly recursive grammars - which this one isn't!

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

using Library;
using System.Collections.Generic;

COMPILER Check6 $CN
// Check lists of IP numbers and computer names for duplicates and errors
// P.D. Terry, Rhodes University, 2015
// This version uses static Lists, but no parameters
//
// Typical data
// 146.231.122.131   bungee.ru.ac.za       #comments appear like this
//                   humfac.ru.ac.za       #alternative name
// 146.231.122.75    bungee.ru.ac.za       #invalid - name already used
// 146.231.122.1123  pdt1.ict.ru.ac.za     #invalid IP address
// 146.231.122.156   pdt2.ict.ru.ac.za
// 146.231.122.156   pdt3.ict.ru.ac.za     # non-unique IP address

static List<int>     IPList   = new List<int> ();
static List<string>  NameList = new List<string> ();

IGNORECASE

CHARACTERS
  digit   = "0123456789" .
  letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  eol     = CHR(10) .

TOKENS
  number = digit { digit } .
  name   = letter { letter | digit | "." | "-" } .

COMMENTS FROM "#" TO eol

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

```
PRODUCTIONS
  Check6
  =
  { Entry }
  EOF                           (. if (Successful()) IO.WriteLine("all correct"); .)
  .

  Entry
  = IPNumber Name { Name } .

  Name
  =
  name                          (. string s = token.val;
                                   if (NameList.Contains(s)) SemError("duplicate computer name");
                                   else NameList.Add(s); .)
  .

  IPNumber
  =                             (. int n, m; .)
  Number<out n>
  "." Number<out m>             (. n = 256 * n + m; .)
  "." Number<out m>             (. n = 256 * n + m; .)
  "." Number<out m>             (. n = 256 * n + m;
                                   if (IPList.Contains(n)) SemError("duplicate IP number");
                                   else IPList.Add(n); .)
  .

  Number<out int n>
  =  number                     (. try {
                                     n = Convert.ToInt32(token.val);
                                   } catch (Exception) {
                                     n = 256;
                                   }
                                   if (n > 255) SemError("number too large"); .)
  .

END Check6.
```

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

Tasks 5 - 7 - An assembler/cross referencer/pretty-printer for the PVM

Since these tasks each added one refinement at a time to a working system, it should suffice simply to produce a fully integrated solution. In summary, you were asked to:

- Add a few opcodes to the language;
- Deal with escape sequences in the strings used in PRNS operations;
- Add the ability to use names for variables, rather than ordinal numbers, and for the assembler to keep track of these in a table, in which would be recorded their offset addresses for use in LDL, LDA and STL operations;
- Print a table of cross references, giving a list of the line numbers in which reference had been made to each variable, and also a table of line numbers for the declaration and use of each of the labels used in BRN and BZE operations;
- Pretty-print the source code;

Each of these refinements is relatively easy to achieve, with the aid of the parsing environment that is built and maintained in the synbol table.

An attributed grammar follows. In some respects this is very close to the one presented in the text book in Chapter 11. The code generator is as in Chapter 11, and the version of the PVM is essentially one that corresponds to one suggested as a solution in Practical 2.

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

using Library;

COMPILER Assem $NC
/* Simple assembler/cross referencer/pretty printer for the PVM - C# version
   P.D. Terry, Rhodes University, 2015 */

  const bool
    known = true;
```

```
  /* By naming the fieldwidths we would easily be able to change the appearance of
     pretty-printed programs should we wish to do so. There seems to be an unfortunate
     reluctance among students to name "magic number".  Learn to do so! */

  const int                             // for pretty-printing
    LabelWidth    = 8,
    Opcode2Width  = 7,
    Address2Width = 15,
    Opcode1Width  = Opcode2Width + Address2Width;

  static int localVarSize = 0;          // will usually be updated by a DSP n instruction

  public static OutFile pretty;         // Pretty-printer and cross-reference output

  // This next method might better be located in the code generator.  Traditionally
  // it has been left in the ATG file, but that might change in future years
  //
  // Not that while sequences like \n \r and \t result in special mappings to lf, cr and tab
  // other sequences like \x \: and \9 usually simply map to x, ; and 9 .  Most students
  // don't seem to know this!

  // Kirk-Cohen, Linklater, Haschick and Boswell (2015) suggested using
  //
  //    System.Text.RegularExpression.Unescape(s)
  //
  // in place of the routine below.  This needs further investigation.  I suspect it may
  // trip up if RE meta-characters like [ ] { } | are found in the string, but it
  // would have worked on the sort of strings in their test cases, and their innovation
  // was welcomed.

  static string Unescape(string s) {
  /* Replaces escape sequences in  s  by their Unicode values (ASCII in our implementation) */
    StringBuilder buf = new StringBuilder();
    int i = 0;
    while (i < s.Length) {
      if (s[i] == '\\') { // escaped character detected
        switch (s[i+1]) {
          case '\\': buf.Append('\\');   break;
          case '\'': buf.Append('\'');   break;
          case '\"': buf.Append('\"');   break;
          case  'r': buf.Append('\r');   break;
          case  'n': buf.Append('\n');   break;
          case  't': buf.Append('\t');   break;
          case  'b': buf.Append('\b');   break;
          case  'f': buf.Append('\f');   break;
          default:   buf.Append(s[i+1]); break;
        } // switch
        i += 2;
      } // escaped
      else {                // normal character detected
        buf.Append(s[i]);
        i++;
      } // normal
    }
    return buf.ToString();
  } // Unescape

 /* IGNORECASE willl have the effect that all opcodes (but not identifiers/labels/variables)
    will become case-insensitive, thus making the assembler easier to use. */

IGNORECASE

CHARACTERS
  lf         = CHR(10) .
  cr         = CHR(13) .
  backSlash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  printable  = ANY - control .
  stringCh   = printable - '"' - backSlash .

  /* Since we need to retain comments we cannot use the   COMMENTS FROM   directive.
     Since each statement ends at the end of a line we must be careful.  So this won't work:

          comment = ";" { printable } lf .

     Although this works, it gets a bit tricky:

          comment = ";" { printable } cr .
```

```
     as it relies on there being a cr/lf sequence in the source code - which is true for
     Wintel systems, but not for Unix.  The best solution is probably as shown below:   */

TOKENS
  identifier = letter { letter | digit } .
  number     = [ '+' | '-' ] digit { digit } .
  label      = letter { letter | digit } ":" .
  stringLit  = '"' { stringCh | backSlash printable } '"' .
  EOL        = lf .
  comment    = ";" { printable } .


IGNORE CHR(9) .. CHR(13) - lf

PRODUCTIONS

  /* Note the following syntax, which allows any number of empty lines to appear
     before ASSEM, between ASSEM and BEGIN and after the final full stop.  It
     will, however; not permit you to have comments attached to the ASSEM or BEGIN,
     or to have lines with nothing but comments preceding the BEGIN.  As an easy exercise
     you might consider what you would need to change to permit such comments. */

  Assem
  = { EOL                       (. pretty.WriteLine(); .)
    }
    "ASSEM"                     (. pretty.Write("ASSEM"); .)
    { EOL                       (. pretty.WriteLine(); .) }
    "BEGIN" EOL                 (. pretty.WriteLine("BEGIN"); .)
    { Statement }
    "END" "." { EOL }           (. LabelTable.CheckLabels();
                                   pretty.WriteLine("END.");
                                   pretty.WriteLine();
                                   LabelTable.ListReferences(pretty);
                                   VarTable.ListReferences(pretty);
                                   pretty.Close(); .) .

  /* Note the use of an "action" applied to an empty option, as mentioned in the
     handout.  There are other ways of doing this pretty-printing, of course.  You will
     will note that, in the way suggested here, much use is made of the "left justified"
     output routines in my library, which work very well in this sort of application. */

  Statement
  = (   Label                   (. pretty.Write(token.val, -LabelWidth); .)
      |                         (. pretty.Write(" ", -LabelWidth); .)
    )
    (   Instruction
      |                         (. pretty.Write(" ", -Opcode1Width); .)
    )
    [ comment                   (. pretty.Write(token.val); .)
    ] SYNC EOL                  (. pretty.WriteLine(); .) .

  Instruction
  = TwoWord | OneWord | Branch | WriteString .

  /* Adding the extra opcodes as key words is very straightforward! */

  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"
      | "LDL_0"   | "LDL_1"  | "LDL_2"  | "LDL_3"  | "STL_0"  | "STL_1"  | "STL_2"  | "STL_3"
      | "LDA_0"   | "LDA_1"  | "LDA_2"  | "LDA_3"  | "LDC_0"  | "LDC_1"  | "LDC_2"  | "LDC_3"
      | "LDC_M1"  | "STO"    | "SUB"    | "INC"    | "DEC"
    )                               (. CodeGen.OneWord(token.val);
                                       pretty.Write(token.val, -Opcode1Width); .) .

  /* Note that the DSP and LDC opcodes must still take an argument limited to a known
     constant.  I don't think many students realised this.  In a better assembler one would
     probably be able to "name" magic number constants with syntax like

           MAX   EQU   1234   ; define constant MAX
                 LDC   MAX    ; Push 1234 onto the evaluation stack

     but that will have to wait for another day/night/predawn session.

     Note that the method for retrieving the offset address of a variable is now
     passed the line number of the instruction where that label is mentioned.  There
     is no need to count the lines separately, as some people try to do - the token
     returned by the scanner has the fields    token.line  and  token.col   which can be
     used to identify the start of the token in the source text being assembled.
```

```
   James and Dixon (2015) worked in a check that no variable could be auto-generated
   an offset greater or equal to the argument to the DSP instruction normally found
   as the first in these PVM programs.  Rather than treat this as a hard error, the
   code below adapts their innovative idea to issue a warning (since a DSP opcode might
   legitimately be found in other places. */

TwoWord                         (. int value = 0;
                                   string name; .)
=   ( "DSP" | "LDC" )           (. string mnemonic = token.val;
                                   pretty.Write(mnemonic, -Opcode2Width); .)
      Number<out value>         (. CodeGen.TwoWord(mnemonic, value);
                                   pretty.Write(token.val, -Address2Width);
                                   if (mnemonic.Equals("DSP") && value > localVarSize)
                                     localVarSize = value; .)
  |
    ( "LDA" | "LDL" | "STL" )   (. string mnemonic = token.val;
                                   pretty.Write(mnemonic, -Opcode2Width); .)
    (   Number<out value>
      | Ident<out name>         (. value = VarTable.FindOffset(name, token.line); .)
    )                           (. CodeGen.TwoWord(mnemonic, value);
                                   pretty.Write(token.val, -Address2Width);
                                   if (value >= localVarSize)
                                     Warning("Offset " + value
                                             + " will exceed max specified in DSP "
                                             + localVarSize + " instruction"); .) .

Number<out int value>
= number                        (. try {
                                     value = Convert.ToInt32(token.val);
                                   } catch (Exception ) {
                                     value = 0; SemError("number too large");
                                   } .) .

/* In this system, strings appear only in PRNS statements.  The string returned by
   the scanner includes the delimiting quotes " at either end, which must be
   stripped off, and then any internal escape sequences like \n \\ and \x must be
   replaced by the byte value of the (usually control) character so designated.
   Note that the converted string is passed to the code generator, but the original
   string in token.val is passed to the pretty-printer. */

WriteString                     (. string str; .)
= "PRNS"                        (. string mnemonic = token.val;
                                   pretty.Write(mnemonic, -Opcode2Width); .)
  StringConst<out str>          (. CodeGen.WriteString(str);
                                   pretty.Write(token.val, -Address2Width); .) .

StringConst<out string str>
=  stringLit                    (. str = Unescape(token.val.Substring(1, token.val.Length - 2)); .) .

/* When a label field is encountered, all that is needed to help with the
   cross-reference of labels is to add the line number when a label is declared,
   (if this happens before it is used), or to add the reference (if it is used first,
   when the branches to it will be backpatched later).  By supplying the negative of the
   line number, the declarations will be highlighted later when the list is displayed. */

Label                           (. Label lab; .)
= label                         (. string name = token.val.Substring(0, token.val.Length - 1).ToLower();
                                   LabelEntry entry = LabelTable.Find(name);
                                   if (entry == null) {
                                     lab = new Label(known);
                                     LabelTable.Insert(new LabelEntry(name, lab, -token.line));
                                   }
                                   else if (entry.label.IsDefined())
                                     SemError("redefined label");
                                   else {
                                     entry.label.Here();
                                     entry.AddReference(-token.line);
                                   } .) .

/* When labels are used in the address fields of branch instructions, a new reference
   (with a positive line number) is made, if the label has not previously been seen,
   or an additional reference is added to the corresponding list (if the location of
   the label is already known). */

Branch                          (. int target;
                                   string name;
                                   Label lab; .)
= ( "BRN" | "BZE" )             (. string mnemonic = token.val;
                                   pretty.Write(mnemonic, -Opcode2Width);  .)
  (   Number<out target>        (. CodeGen.TwoWord(mnemonic, target); .)
    | Ident<out name>           (. LabelEntry entry = LabelTable.Find(name);
```

```
                                        if (entry == null) {
                                          lab = new Label(!known);
                                          LabelTable.Insert(new LabelEntry(name, lab, token.line));
                                        }
                                        else {
                                          lab = entry.label;
                                          entry.AddReference(token.line);
                                        }
                                        CodeGen.Branch(mnemonic, lab); .)
      )                                (. pretty.Write(token.val, -Address2Width); .) .

    Ident<out string name>
    = identifier                       (. name = token.val.ToLower(); .) .

END Assem.
```

The table handler I hacked together for this solution (before issuing a bonus challenge to improve on it) was as follows - note that there are, effectively two table handlers, one for labels and one for variables (but with a number of close similarities). The changes from the system supplied in the tool kit are highlighted in the margin.

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

    // Handle label and variable table for simple PVM assembler
    // with cross reference generator
    // P.D. Terry, Rhodes University, 2015

    using Library;
    using System;
    using System.Collections.Generic;

    namespace Assem {

      class LabelEntry {

        public string name;
        public Label label;
**      public List<int> refs = null;

        public LabelEntry(string name, Label label, int lineNumber) {
          this.name  = name;
          this.label = label;
**        this.refs  = new List<int>();
**        this.refs.Add(lineNumber);
        }

**      public void AddReference(int lineNumber) {
**        this.refs.Add(lineNumber);
**      } // AddReference

      } // end LabelEntry

    // ---------------------------------------------------------------------------

      class LabelTable {
      // Simple linear list based table of labels.  Labels aer entered in the order of their
      // appearance and forward references are handled by the Label class

        private static List<LabelEntry> list = new List<LabelEntry>();

        public static void Insert(LabelEntry entry) {
        // Inserts entry into label table
          list.Add(entry);
        } // Insert

        public static LabelEntry Find(string name) {
        // Searches table for label entry matching name.  If found then returns entry.
        // If not found, returns null
          int i = 0;
          while (i < list.Count && !name.Equals(list[i].name)) i++;
          if (i >= list.Count) return null; else return list[i];
        } // find

        public static void CheckLabels() {
        // Checks that all labels have been defined (no forward references outstanding)
          for (int i = 0; i < list.Count; i++) {
            if (!list[i].label.IsDefined())
              Parser.SemError("undefined label - " + list[i].name);
          }
        } // CheckLabels
```

```
        public static void ListReferences(OutFile output) {
        // Cross reference list of all labels used on output file
        // Note the use of the Write(value, width) routines from the Library
**        output.WriteLine();
**        output.WriteLine("Labels:");
**        output.WriteLine();
**        if (list.Count == 0) output.WriteLine("none");
**        else
**          foreach (LabelEntry entry in list) {
**            output.Write(entry.name, -10);
**            if (!entry.label.IsDefined())
**              output.Write(" (undefined) ");
**            else
**              output.Write(" (defined)   ");
**            foreach (int line in entry.refs) output.Write(line, 5);
**            output.WriteLine();
**          }
        } // ListReferences

      } // end LabelTable

      class VariableEntry {

        public string name;
        public int offset;
**      public List<int> refs = null;

        public VariableEntry(string name, int offset, int lineNumber) {
          this.name   = name;
          this.offset = offset;
**        this.refs   = new List<int>();
**        this.refs.Add(lineNumber);
        }

**      public void AddReference(int lineNumber) {
**        this.refs.Add(lineNumber);
**      } // AddReference

      } // end VariableEntry

      class VarTable {
      // Simple linear list based table of variables, with automatic assignment of their offsets
      // in a PVM stack frame, in the order of their appearance

        private static List<VariableEntry> list = new List<VariableEntry>();
        private static int varOffset = 0;

        public static int FindOffset(string name, int lineNumber) {
        // Searches table for variable entry matching name.  If found then returns the known offset.
        // If not found, makes an entry and updates the master offset
**        int i = 0;
**        while (i < list.Count && !name.Equals(list[i].name)) i++;
**        if (i >= list.Count) {
**          list.Add(new VariableEntry(name, varOffset, lineNumber));
**          return varOffset++;
**        }
**        else {
**          list[i].AddReference(lineNumber);
**          return list[i].offset;
**        }
        } // FindOffset

        public static void ListReferences(OutFile output) {
        // Create a cross reference list of all variables on output file
        // Note the use of the Write(value, width) routines from the Library
**        output.WriteLine();
**        output.WriteLine("Variables:");
**        output.WriteLine();
**        if (list.Count == 0) output.WriteLine("none");
**        else
**          foreach (VariableEntry entry in list) {
**            output.Write(entry.name, -10);
**            output.Write("(offset  " + entry.offset, -10); output.Write(") ");
**            foreach (int line in entry.refs) output.Write(line, 5);
**            output.WriteLine();
**          }
        } // ListReferences

      } // end VarTable

    } // end namespace
```