Computer Science 3 - 2017

Programming Language Translation

Practical 6 for week beginning 21 August 2017 - 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. 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 - Checking on Tokens

Something like this seems to be required:

```
using Library;

COMPILER TokenTests $CN
/* Test scanner construction and token definitions - C# version
   The generated program will read a file of words, numbers, strings etc
   and report on what characters have been scanned to give a token,
   and what that token is (as a magic number).  Useful for experimenting
   when faced with the problem of defining awkward tokens!

   P.D. Terry, Rhodes University, 2017 */

  static void Display(Token token) {
  // Simply reflect the fields of token to the standard output
    IO.Write("Line ");
    IO.Write(token.line, 4);
    IO.Write(" Column");
    IO.Write(token.col, 4);
    IO.Write(":  Kind");
    IO.Write(token.kind, 3);
    IO.WriteLine("  Val |" + token.val.Trim() + "|");
  } // Display

CHARACTERS  /* You may like to introduce others */

  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  ULetter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  LLetter    = "abcdefghijklmnopqrstuvwxyz" .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  alphanum   = letter + digit .
  PstringCh  = ANY - "'" - control .
  CstringCh  = ANY - '"' - control - backslash .
  printable  = ANY - control .
  inComment  = ANY - '/' - '*'.

TOKENS      /* You may like to introduce others */

  ident      = letter { letter | digit } .
  integer    =   digit { digit }
               | digit { digit } CONTEXT ("..") .
  double     = digit { digit } "." digit { digit }  .
  Pstring    = "'" { PstringCh | "''" } "'" .
  Cstring    = '"' { CstringCh | backslash printable } '"' .
  Ccomment   = "/*" { inComment | '/' | '*' { '*' } inComment } '*' { '*' } '/'.
  stuNum     = digit digit ULetter digit digit digit digit .

// You cannot have both of the following, of course,  and GenEmail is too restricted really

//  RUEmail    = "G" digit digit ULetter digit digit digit digit "@campus.ru.ac.za" .
    GenEMail   =     alphanum { alphanum | ( "_" | "-" | "." ) alphanum }
                   '@' alphanum { alphanum | ( "_" | "-") alphanum } { "." alphanum { alphanum } } .

  IGNORE control

  PRODUCTIONS

     TokenTests
     = { (    ANY              // Matches any token other than . .. the key word ANY and EOF
            | "."
            | ".."
            | "ANY"
         )                     (. Display(token); .)
       } EOF                   (. Display(token); .)
       .

  END TokenTests.
```

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

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

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

Task 4 - Who are our most highly qualified staff?

This was intended to be quite simple. A point many people missed was that first names had to be replaced by initials - or they ended up getting too many, too few, or in the wrong order..

```
using Library;
using System.Text;

COMPILER Staff1 $CN
/* Describe a list of academic staff in a department, and extract those with PhD degrees
   P.D. Terry, Rhodes University, 2017 */

  static StringBuilder sb;
  static string lastName;
  static bool hasPhD;

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

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


IGNORE control

PRODUCTIONS
  Staff1    = { Person } EOF .

  Person                                (. sb = new StringBuilder();
                                           hasPhD = false; .)
  = [ Title ] FullName { "," Degree }
    SYNC "."                            (. if (hasPhD)
                                             IO.WriteLine("Dr "  + sb.ToString() + lastName); .) .

  FullName  = NameLast {                (. sb.Append(lastName[0]);
                                           sb.Append('.'); .)
              NameLast } .

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

  Title
  =   "Professor" | "Prof" | "Dr"       (. hasPhD = true; .)
    | "Mr" | "Mrs" | "Ms" | "Mx"
    | "Miss" .

  Degree
  = (  "BA" | "BSc" | "BCom" | "BMus"
     | "BSocSci"    | "BSc(Hons)"
     | "BCom(Hons)" | "MA" | "MSc"
     | "PhD"                            (. hasPhD = true; .)
    )
    [ university ] .

END Staff1.
```

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

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

```
  Person                              (. sb = new StringBuilder();
                                         hasPhD = false; .)
  = [ Title ] FullName { "," Degree }
  SYNC "."                            (. if (hasPhD)
                                           IO.WriteLine("Dr "  + sb.ToString()); .) .

  FullName  = NameLast {              (. sb.Append(token.val[0]);
                                         sb.Append('.'); .)
              NameLast }              (. sb.Append(" " + token.val); .) .

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

Here is a solution using parameterised productions. That is probably a preferred way; using global variables is all very well for small silly exercises, but you have to learn the parameter passing some time. Note that the *hasPhd* parameter has to be passed by reference. The *StringBuilder* parameter does not need a *ref* mark - `sb` is an object, and so when it is used as a parameter it is the pointer to the object (the address, the reference) which is passed by value. Notice, too, exactly where the actions are injected into the grammar.

```
PRODUCTIONS
  Staff3    = { Person } EOF .

  Person                              (. StringBuilder sb = new StringBuilder();
                                         bool hasPhD = false; .)
  = [ Title<ref hasPhD>
    ] FullName<sb>
      { "," Degree<ref hasPhD> }
    SYNC "."                          (. if (hasPhD)
                                           IO.WriteLine("Dr "  + sb.ToString()); .) .

  FullName<StringBuilder sb>
  = NameLast<sb> {                    (. sb.Append(token.val[0]);
                                         sb.Append('.'); .)
    NameLast<sb> }                    (. sb.Append(" " + token.val); .) .

  NameLast<StringBuilder sb>
   = { initial                        (. sb.Append(token.val); .)
     } name .

  Title<ref bool hasPhD>
  =   "Professor" | "Prof" | "Dr"     (. hasPhD = true; .)
    | "Mr" | "Mrs" | "Ms" | "Mx" | "Miss" .

  Degree<ref bool hasPhD>
  = (  "BA" | "BSc" | "BCom" | "BMus" | "BEd" | "BSocSci"
     | "BSc(Hons)" | "BCom(Hons)" | "MA" | "MSc"
     | "PhD"                           (. hasPhD = true; .)
    )
    [ university ] .

END Staff3.
```

There was a tendency to write this sort of code:

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

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

    ....

    Person                            (. sb = new StringBuilder();
                                         hasPhD = false; .)
    = [ Title ] FullName { "," Degree }
      SYNC "."                        (. if (hasPhD)
                                           IO.WriteLine("Dr "  + sb.ToString() + lastName);
**                                       sb = new StringBuilder();     // re-initialise .) .
```

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

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

Task 5 - 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:

```
using Library;

COMPILER Trains1 $CN
/* Grammar for railway trains with simple safety regulations (C# version)
   P.D. Terry, Rhodes University, 2017 */

  const int  // type of train
    passenger = 0,
    freight   = 1,
    mixed     = 2,
    empty     = 3;
  static int type;
  static bool 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:

```
using Library;

COMPILER Trains2 $CN
/* Grammar for railway trains with simple safety regulations (C# version)
   P.D. Terry, Rhodes University, 2017 */

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

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!

This was the scheme that was suggested to the demonstrators, the assumption being made that the use of the (parentheses) would still be permissible, and leads to a grammar with productions that look like the ones below.

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!)

```
PRODUCTIONS
  EBNF1
  = { 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); .)   /* don't forget this one! */
    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); .)
    | "("                           (. Table.AddText("(", i); .)
        Expression<s, i>
      ")"                           (. Table.AddText(")", 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); .)
    | "{"                           (. 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 EBNF1.
```

As several groups noticed, if one can make use of the (parentheses) meta brackets, then it is not necessary to invent a new name when dealing with an [option], leading to alternative, simpler, code like

```
    | "["                           (. Table.AddText("(", i); .)
        Expression<s, i>
      "]"                           (. Table.AddText("| )", i); .)
```

The table handler to match this might be coded as below. 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. (This was covered in a tut!)

As the second thing that one might not think of - one must be careful not to concatenate non-terminals accidentally as the components of a production are output. For example, a production like

```
S = One Two Three .
```

must not end up like

```
S = OneTwoThree .
```

As the third 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, 2017 (C# version)

using Library;
using System;
```

```
using System.Collections.Generic;
using System.Text;

namespace EBNF3 {

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

  class Table {

    public const int
      LHS  = 1,
      RHS  = 2,
      BOTH = 3;

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

    public static string NewName(string oldName, string ext) {
      extraNames++;
      return oldName + ext + extraNames;
    } // Table.NewName

    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.Count && !name.Equals(list[position].name)) position++;
      if (position >= list.Count) {
        list.Add(new Entry(name));
        if (name.Length > maxLength) maxLength = name.Length;
      }
      switch (where) {
        case LHS  :
          if (list[position].left > 0) { // enter a second time
            position = list.Count;
            list.Add(new Entry(name));
            list[position].left++;       // this will force an error message later
          }
          list[position].left++;
          break;
        case RHS  :
          list[position].right++; break;
        case BOTH :
          list[position].left++;
          list[position].right++; break;
      }
      return position;
    } // Table.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.WriteLine("addtext " + str);
      list[i].text.Add(str);
    } // Table.AddText

    public static void ListProductions() {
    // List all productions to standard output file - simple version
      for (int i = 0; i < list.Count; i++)
        if (list[i].text.Count> 0) {              // only list if the non-terminal was defined
          IO.Write(list[i].text[0], -maxLength);
          for (int j = 1; j < list[i].text.Count; j++) {
            IO.Write(list[i].text[j], 0);         // width of 0 inserts a leading space
          }
          IO.WriteLine(" .");
        }
      IO.WriteLine();
    } // Table.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
      bool OK = true; // optimistic
      int i;
      for (i = 0; i < list.Count; i++) {
        if (list[i].left == 0) {
          OK = false;
          IO.WriteLine("(* " + list[i].name + " never defined *)");
        }
        else if (list[i].left > 1) {
          OK = false;
          IO.WriteLine("(* " + list[i].name + " defined more than once *)");
        }
      }
      for (i = 1; i < list.Count; i++) {
        if (list[i].right == 0) {
          OK = false;
          IO.WriteLine("(* " + list[i].name + " cannot be reached *)");
        }
      }
      if (!OK) IO.WriteLine("\n(* Cannot be correct grammar *)");
    } // Table.TestProductions

  } // Table

} //namespace
```

We can improves still further on the output by arranging that optional right hand sides for a production appear on subsequent lines. Rather than:

```
VehicleP5 =  "scooter" | "bicycle" .
Parents   =  "Dad" "Mom" | "Mom" "Dad" | "Dad" | "Mom" .
```

we might prefer

```
VehicleP5 =  "scooter"
           | "bicycle" .

Parents   =  "Dad" "Mom"
           | "Mom" "Dad"
           | "Dad"
           | "Mom" .
```

This can be done as follows:

```
    public static void ListProductions() {
    // List all productions to standard output file
      for (int i = 0; i < list.Count; i++)
        if (list[i].text.Count> 0) {              // only list if the non-terminal was defined
          int paren = 0;
          IO.Write(list[i].text[0], -maxLength);
          for (int j = 1; j < list[i].text.Count; j++) {
            string s = list[i].text[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();
    } // Table.ListProductions
```

Yet another improvement might be as follows. There is nothing really wrong with productions like

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

which are of course equivalent to

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

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

The grammar 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 "." .
```

The table handler needs an addition

```
public static int PositionLHS(string name) {
// Return the position in the table where name can be found on the LHS,
// or -1 if production for that name has not yet been defined
// debug: IO.WriteLine("positionLHS " + name + " " + list.Count);
  int position = 0;
  while (position < list.Count && !name.Equals(list[position].name)) position++;
  return (position >= list.Count || list[position].left == 0) ? -1 : position;
} // PositionLHS
```

Finally, suppose we wish to eliminate all meta-brackets. After what has gone before, the following should make for easy reading:

```
Factor<string s, int i>             (. string name1, name2;
                                       int j, k; .)
=   nonterminal                     (. name1 = token.val;
                                       j = Table.Add(name1, Table.RHS);
                                       Table.AddText(name1, i); .)

  | terminal                        (. name1 = token.val;
                                       Table.AddText(name1, i); .)

  | "("                             (. name1 = Table.NewName(s, "P");
                                       Table.AddText(name1, i);
                                       j = Table.Add(name1, Table.BOTH);
                                       Table.AddText(name1, j);
                                       Table.AddText("= ", j); .)
      Expression<s, j>
    ")"

  | "["                             (. name1 = Table.NewName(s, "O");
                                       Table.AddText(name1, i);
                                       j = Table.Add(name1, Table.BOTH);
                                       Table.AddText(name1, j);
                                       Table.AddText("= ", j); .)
      Expression<s, j>
    "]"                             (. Table.AddText("|", j); .)

  | "{"                             (. name1 = Table.NewName(s, "S");
                                       name2 = Table.NewName(s, "P");
                                       Table.AddText(name1, i);

                                       j = Table.Add(name1, Table.BOTH);
                                       Table.AddText(name1, j);
                                       Table.AddText("= ", j);
                                       Table.AddText(name2, j);
                                       Table.AddText(name1, j);
                                       Table.AddText("|", j);

                                       k = Table.Add(name2, Table.BOTH);
                                       Table.AddText(name2, k);
                                       Table.AddText("= ", k); .)
      Expression<s, k>
    "}" .
```