

# Computer Science 3 - 2016

## Programming Language Translation

### Practical for Week 6, beginning 22 August 2016 - Solutions

A few groups seemed rather at a loss here and there. Full sources for the solutions can be found in the file PRAC6A.ZIP for those who wish to experiment.

#### Tasks 2 and 3 - Checking on Tokens

This is harder to get right than it at first appears. One has to be able to recognise and distinguish (assume spaces precede and follow the patterns below):

```
30      (integer 30)
0       (integer 0)
030    (integer 0 followed by integer 30)
30.    (integer 30 followed by period)
0.     (integer 0 followed by period)
9.     (integer 9 followed by period)
.0     (period followed by integer 0)
.9     (period followed by integer 9)
0.3    (double 0.3)
3.0    (double 3.0)
3.3    (double 3.3)
```

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, 2016 */

/* Some code to be added to the parser class */

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

CHARACTERS /* You may like to introduce others */

sp      = CHR(32) .
backslash = CHR(92) .
control  = CHR(0) .. CHR(31) .
letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
digit09  = "0123456789" .
digit19  = "123456789" .
stringCh = ANY - "'" - control - backslash .
charCh   = ANY - "'" - control - backslash .
printable = ANY - control .

TOKENS /* You may like to introduce others */

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

double  = ( digit19 { digit09 } "." | "0." ) digit09 { digit09 }
           [ "E" [ "+" | "-" ] digit09 { digit09 } ] .
```

```

ident      = letter { {"_"} (letter | digit09) } .
string     = '"' { stringCh | backslash printable } '"' .
char       = "'" ( charCh | backslash printable ) "'" .

IGNORE control

PRODUCTIONS

TokenTests
= { ( ANY
    | "."
    | ".."
    | "ANY"
  )
  } EOF          (. Display(token); .)
                (. Display(token); .)
.

END TokenTests.

```

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

```

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

```

#### Task 4 - The Parva 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).

All the references to identifiers in the Parva grammar are channelled through the `Ident` non-terminal, so all we have to do is add the actions here to add the identifier to the table, suitably tagged according to how it has been encountered,

This should suggest attributing the `Ident` production with a Boolean value parameter. The details follow of the other productions that refer to it. Note the careful way in which we attribute the `ForStatement` production, which has the option of incorporating a highly local control identifier if the "type" is made explicit..

```

COMPILER Parva $CN
/* Parva level 1 grammar - Coco/R for C# (EBNF)
   P.D. Terry, Rhodes University, 2016
   Cross reference generator */

static final bool
  declared = true,
  used     = false;

PRODUCTIONS
  Parva                                (. Table.ClearTable(); .)
  = "void" Ident<declared>
    "(" ")" Block                      (. Table.PrintTable(); .)

  OneConst
  = Ident<declared>
    "=" Constant .

  OneVar
  = Ident<declared> [ "=" Expression ] .

  Designator
  = Ident<used> [ "[" Expression "]" ] .

```

```

ForStatement          ( . bool how = used; .)
= "for" "("
  [ [ BasicType      ( . how = used; .)
    ] Ident<how>
    "=" Expression
    ] WEAK ";"
    [ Condition ] WEAK ";"
    [ Step ]
  "]" Statement .

Step
= Ident<used>
  ( "++" | "--" | ( "=" | CompoundAssignOp ) Expression ) .

Ident<bool declared>
= identifier          ( . Table.AddRef(token.val, declared, token.line); .) .

```

Of course, we need to have a table handler. In this case we can just write very simple code like the following. Running the search loop from the bottom makes for a very simple `addRef` method. Note that this handler allows us to enter an identifier that has been used before being declared. While this may be "wrong", it prevents crashes of the cross-referencer itself.

```

// Handle cross reference table for Parva
// P.D. Terry, Rhodes University, 2016

using Library;
using System.Collections.Generic;

namespace Parva {

    class Entry {                // Cross reference table entries
        public string name;      // The identifier itself
        public List<int> refs;   // Line numbers where it appears
        public Entry(string name) {
            this.name = name;
            this.refs = new List<int>();
        }
    } // Entry

    class Table {
        static List<Entry> list = new List<Entry>();

        public static void ClearTable() {
            // clears cross-reference table
            list.Clear();
        } // ClearTable

        public static void AddRef(string name, bool declared, int lineRef) {
            // Enters name if not already there, and then adds another line reference (negative
            // if at a declaration point in the original source program)
            int i = 0;
            while (i < list.Count && !name.Equals(list[i].name)) i++;
            if (i >= list.Count) list.Add(new Entry(name));
            list[i].refs.Add(declared ? -lineRef : lineRef);
        } // AddRef

        public static void PrintTable() {
            // Prints out all references in the table
            for (int i = 0; i < list.Count; i++) {
                Entry e = list[i];
                IO.Write(e.name, -16);                // left justify in 16 spaces
                for (int j = 0; j < e.refs.Count; j++)
                    IO.Write(e.refs[j], 4);          // right justify in 4 spaces
                IO.WriteLine();
            }
        } // PrintTable
    } // class Table
} // namespace

```

The `PrintTable` method above suffers from a possible disadvantage in that multiple occurrences of an identifier on one line, as in

```
i = i + (i - list[i] * i);
```

create unwanted duplicate entries. These can 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 a suggestion.

```
public static void PrintTable() {
    // Prints out all references in the table (eliminate duplicates line numbers)
    for (int i = 0; i < list.Count; i++) {
        Entry e = list[i];
        IO.Write(e.name, -16);           // left justify in 16 spaces
        int last = 0;                  // impossible line number
        for (int j = 0; j < e.refs.Count; j++) {
            int line = e.refs[j];
            if (line != last) {
                IO.Write(line, 4);     // right justify in 4 spaces
                last = line;          // remember we have printed this line number
            }
        }
        IO.WriteLine();
    }
} // PrintTable
```

Several solutions revealed that people either had not thought that far or were confused about the point of the `declared` argument to the `AddRef` method. In particular, they might not have realized that an identifier may legally be declared several times, in different scopes, as in the following example:

```
void main () {
    int i = 1;
    while (i < 10) {
        int k = 1;           // first declaration
        while (k < i) {
            write(k); k = k - 1;
        }
        i = i + 1;
    }
    bool k = false;        // second declaration
    write(k);
} // main
```

## Tasks 5 - 8 - The Extended Calculator

Tasks 5 - 8 provided more of a challenge. There were some excellent submissions, but most showed that not nearly enough effort had been expended on the exercise. Pity!

It is possible to solve this problem by embedding the table handling into the start of the ATG file, and the solution kit has versions showing this. However, it may be worth introducing a proper (even if simple) table handler.

It makes for enhanced readability to assign names to a set of magic numbers useful for mapping the types. Besides the obvious `intType` and `boolType`, it is convenient to introduce a fictitious `noType` that can be used to denote the resulting type when an expression is badly formed.

The `Entry` class defines the nodes needed in the symbol table, which need to record the name, type, and value for each variable.

We introduce a wrapper class `Node` to allow the various "nodes" in an expression to pass back both the value and type of an operand. Though similar, this class is not the same as the `Entry` class, because it does not require a name field. Note how the default constructor assigns a value of 1 - this will prevent problems if you try to divide by an undefined variable. Clearly one could arrange define these classes in other ways as well.

```
// Handle symbol/value table for integer/boolean calculator

using Library;
using System.Collections.Generic;

namespace Calc {
    public class Types {
        public const int // enumerations
            noType = 0,
            intType = 1,
            boolType = 2;
    } // Types
```

```

public class Entry {
    public string name;
    public int type = Types.noType;           // other fields could be added
    public int value = 1;

    public Entry(string name, int type, int value) { // constructor
        this.name = name;
        this.type = type;
        this.value = value;
    }
} // Entry

public class Node {
    public int type = Types.noType;
    public int value = 1;

    public Node() { // constructor
        this.type = Types.noType;
        this.value = 1; // fudge for safety
    }

    public Node(int type, int value) { // constructor
        this.type = type;
        this.value = value;
    }
} // Node

public class Table {
    static List<Entry> varTable = new List<Entry>();

    public static Entry Find(string name) {
        // Returns entry with search key name in list, or null if not found
        int i = varTable.Count - 1; // index of last entry
        while (i >= 0) {
            if (name.Equals(varTable[i].name)) return varTable[i];
            i--; // will reach -1 if no match
        }
        return null; // undeclared
    } // Find

    public static void Add(Entry e) {
        varTable.Add(e);
    } // Add

} // Table
} // namespace

```

Here is a complete grammar itself, heavily commented to make various points that may have been missed (these comments are not in the solution sources). Note that all error handling uses the `SemError` interface - do not simply write to the standard output stream directly.

```

using Library;

COMPILER Calc SNC
// Integer/Boolean calculator
// P.D. Terry, Rhodes University, 2016

static int ToInt(bool b) {
    // return 0 or 1 according as b is false or true
    return b ? 1 : 0;
} // ToInt

static bool ToBool(int i) {
    // return false or true according as i is 0 or 1
    return i == 0 ? false : true;
} // ToBool

/* It makes for enhanced readability to assign names to the magic numbers useful for mapping the
operators. Rather use this idea than work with many tedious string comparisons! */

```

```

const int // enumerations
  opnop = 1,  opadd = 2,  opsub = 3,  opor = 4,
  opmul = 5,  opdvd = 6,  opmod = 7,  opand = 8,
  opeql = 9,  opneq = 10, oplss = 11, opgeq = 12,
  opgtr = 13, opleq = 14;

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

TOKENS
  number = digit { digit } .
  identifier = letter { letter | digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calc
  = { Print | Assignment } "quit" .

  Print
  = "print"
  OneExpr { WEAK "," OneExpr }
  SYNC ";" ( IO.WriteLine(); ) .

/* OneExpr is introduced as a simple way of cutting down on actions that might otherwise clutter up
the Print production. It becomes responsible for evaluating a single expression that is to be
printed rather than stored. Note that some care is taken to ensure that Boolean values are
printed in a user friendly form. */

  OneExpr
  =
  Expression<out rootNode> ( . switch(rootNode.type) {
    case Types.intType:
      IO.Write(rootNode.value); break;
    case Types.boolType:
      IO.Write(ToBool(rootNode.value)); break;
    default:
      IO.Write(" bad expression"); break;
  } .) .

/* Note that in the Assignment production we do not meddle with the list of variables until the
expression has been evaluated. In particular we do not want to enter the incomplete entry into
the table after parsing the Variable to find its name. In this way we avoid treating an initial
definition of, say ABC = ABC + 1, as being correct when it is not (as the ABC on the right side
has no defined value yet). Some people missed this. */

  Assignment
  = Variable<out name>
  "=" Expression<out rootNode> ( . Entry e = Table.Find(name);
    if (e == null) // first definition
      Table.Add(new Entry(name, rootNode.type, rootNode.value));
    else { // updating a variable
      e.type = rootNode.type;
      e.value = rootNode.value;
    } .)

  SYNC ";" .

/* Note the form of the type checking - several people omitted it or had erroneous variations on the
lines of if (value.type != Entry.boolType && val2.type != Entry.boolType) ... Note how the ToInt
and ToBool methods are used to provide "type casting" of a sort between integer and Boolean values.
Notice how the return type is set to the "best guess" (even in the case of an error) - this can have
beneficial effects later on. Alternatively one could make more use of the noType idea */

  Expression<out Node node>
  = AndExp<out node>
  { "||" AndExp<out node2> ( . if (node.type != Types.boolType || node2.type != Types.boolType)
    SemError("boolean operands required");
    node.type = Types.boolType;
    node.value =.ToInt(ToBool(node.value) || ToBool(node2.value)); .)
  } .

  AndExp<out Node node>
  = EqLExp<out node>
  { "&&" EqLExp<out node2> ( . if (node.type != Types.boolType || node2.type != Types.boolType)
    SemError("boolean operands required");
    node.type = Types.boolType;
    node.value =.ToInt(ToBool(node.value) && ToBool(node2.value)); .)
  } .

```

/\* But the EqlExp production needs further comment. Note that the presence of an EqlOp between two integer operands yields a type "change" to Boolean - some people seem not to have taken advantage of this. Note also that the == and != operators have meaning only between two operands of exactly the same type. \*/

```

EqlExp<out Node node>      (. Node node2;
                           int op; .)

= RelExp<out node>
  { EqlOp<out op>
    RelExp<out node2>      (. if (node.type != node2.type)
                           SemError("incomparable operands ");
                           node.type = Types.boolType;
                           switch(op) {
                             case opeql:
                               node.value = ToInt(node.value == node2.value); break;
                             case opneq:
                               node.value = ToInt(node.value != node2.value); break;
                             default:
                               break;
                           } .)
  } .

```

/\* The RelExp production needs further comment. Note that the presence of an RelOp between two integer operands yields a type "change" to Boolean - some people did not appreciate this. The < <= > and >= operators have meaning only between two operands of the integer type. Note that the production only allows for one or two AddExp terms - we use [ ] and not { }. Do you see why? \*/

```

RelExp<out Node node>      (. Node node2;
                           int op; .)

= AddExp<out node>
  [ RelOp<out op>
    AddExp<out node2>      (. if (node.type != Types.intType || node2.type != Types.intType)
                           SemError("integer operands required");
                           node.type = Types.boolType;
                           switch(op) {
                             case oplss:
                               node.value = ToInt(node.value < node2.value); break;
                             case opleq:
                               node.value = ToInt(node.value <= node2.value); break;
                             case opgtr:
                               node.value = ToInt(node.value > node2.value); break;
                             case opgeq:
                               node.value = ToInt(node.value >= node2.value); break;
                             default:
                               break;
                           } .)
  ] .

```

/\* The AddExp parser is easy and needs no further comment \*/

```

AddExp<out Node node>      (. Node node2;
                           int op; .)

= MultExp<out node>
  { AddOp<out op>
    MultExp<out node2>      (. if (node.type != Types.intType || node2.type != Types.intType)
                           SemError("integer operands required");
                           node.type = Types.intType;
                           switch(op) {
                             case opadd:
                               node.value = node.value + node2.value; break;
                             case opsub:
                               node.value = node.value - node2.value; break;
                             default:
                               break;
                           } .)
  } .

```

/\* The MultExp parser needs to check for division by zero for both the / and the % operator - most people checked only the obvious one involving / \*/

```

MultExp<out Node node>      (. Node node2;
                           int op; .)

= UnaryExp<out node>
  { MulOp<out op>
    UnaryExp<out node2>      (. if (node.type != Types.intType || node2.type != Types.intType)
                           SemError("integer operands required");
                           node.type = Types.intType;
                           switch(op) {
                             case opmul:
                               node.value = node.value * node2.value; break;

```

```

        case opdvd:
            if (node2.value == 0) SemError("division by zero");
            else node.value = node.value / node2.value; break;
        case opmod:
            if (node2.value == 0) SemError("division by zero");
            else node.value = node.value % node2.value; break;
        default:
            break;
    } .)
} .

/* The unary operators need careful type checking. Note that a unary + does not require any arithmetic */

UnaryExp<out Node node>      (. node = new Node(); .)
= Factor<out node>
  | "+" UnaryExp<out node>  (. if (node.type != Types.intType)
                             SemError("integer operand required");
                             node.type = Types.intType; .)
  | "-" UnaryExp<out node>  (. if (node.type != Types.intType)
                             SemError("integer operand required");
                             node.type = Types.intType;
                             node.value = -node.value; .)
  | "!" UnaryExp<out node>  (. if (node.type != Types.boolType)
                             SemError("boolean operand required");
                             node.type = Types.intType;
                             node.value =.ToInt(!ToBool(node.value)); .) .

/* The Factor parser is the one ultimately responsible for starting to pass value and type information
"up the tree". Note that we regard an undeclared identifier at this stage to be an error - but we
will still have assigned a value and type to the return argument to make sure that something
"sensible" propagates up from here (using the default Node constructor). Several people added to
the table if an undeclared identifier was encountered, but this may not be a wise thing to do. */

Factor<out Node node>      (. string name; int n; node = new Node(); .)
= Variable<out name>      (. Entry e = Table.Find(name);
                           if (e == null) SemError("undeclared variable");
                           else node = new Node(e.type, e.value); .)
                           // ++++++ why do we need a new node here +++++
  | Number<out n>          (. node = new Node(Types.intType, n); .)
                           (. node = new Node(Types.boolType, 1); .)
  | "true"                 (. node = new Node(Types.boolType, 1); .)
  | "false"                (. node = new Node(Types.boolType, 0); .)
  | (" Expression<out node> ") .

/* The production for Variable does nothing more than extract the name for us. Note the exception
handling in Number - some submissions omitted this. Can the exception ever arise (think about it) */

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

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

/* It is convenient for readability to define the operators by their own non-terminals and for them
simply to return an appropriate value from an enumeration. Note the use of a default opnop to mop
up errors that might otherwise creep in from undefined values */

MulOp<out int op>         (. op = opnop; .)
= "*"                     (. op = opmul; .)
  | "%"                   (. op = opmod; .)
  | "/"                   (. op = opdvd; .) .

AddOp<out int op>         (. op = opnop; .)
= "+"                     (. op = opadd; .)
  | "-"                   (. op = opsub; .) .

RelOp<out int op>         (. op = opnop; .)
= "<"                    (. op = oplss; .)
  | "<="                 (. op = opleq; .)
  | ">"                  (. op = opgtr; .)
  | ">="                 (. op = opgeq; .) .

EqOp<out int op>         (. op = opnop; .)
= "=="                   (. op = opeq; .)
  | "!="                  (. op = opneq; .) .

END Calc.

```