THE PROGRAMMING LANGUAGE PARVA (Level 2)

P.D. Terry, Computer Science Department, Rhodes University

27 June 2003

1 Introduction

Parva is a minimal toy programming language that evolved from Clang (Terry, 1997). Its features are deliberately chosen as a mixture of those found in Pascal and C, so as to support introductory courses in compiler development.

This specification of level 2 of Parva (which supports multiple function declarations and global variables and constants) is modelled on the Modula-2 and Oberon specifications (Wirth (1985), Reiser and Wirth (1992)), but is not intended as a programmer's tutorial. It is intentionally kept concise. Its function is to serve as a reference for programmers, implementors, and manual writers. What remains unsaid is mostly left so intentionally, either because it is derivable from stated rules of the language, or because it would require a general commitment in the definition when a general commitment appears unwise.

2 Syntax

A language is an infinite set of sentences, namely the sentences well formed according to its syntax. Each sentence is a finite sequence of symbols from a finite vocabulary. The vocabulary of Parva consists of identifiers, numbers, strings, operators, delimiters, and comments. These are called lexical symbols and are composed of sequences of characters. (Note the distinction between symbols and characters.)

To describe the syntax the variant of extended Backus-Naur formalism called Cocol/R is used. This is described in full detail elsewhere (Terry, 2005). Brackets [and] denote optionality of the enclosed sentential form, and braces { and } denote its repetition (possibly 0 times). Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language vocabulary (terminal symbols) are denoted by strings enclosed in quote marks (these include words written in lower-case letters, so-called reserved key words).

3 Vocabulary and representation

The representation of symbols in terms of characters is defined using the ASCII set. Symbols are identifiers, numbers, string literals, character literals, operators, delimiters, and comments.

The following lexical rules must be observed. Blanks and line breaks may appear between symbols but must not occur within symbols (except that line breaks are allowed in comments, and blanks are allowed within string and character literals). They are ignored unless they are essential for separating two consecutive symbols. Capital and lower-case letters are considered as being distinct.

```
CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"
IGNORE CHR(9) .. CHR(13)
```

Comments may be inserted between any two symbols in a program. They are arbitrary character sequences either opened by the bracket `/*` and closed by `*/`, or opened by the bracket `//` and closed at the end of that line. Comments do not affect the meaning of a program.

```
TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .
```

Identifiers are sequences of letters and digits. The first character must be a letter.

Examples:

```
x   scan   Parva   Get_Symbol   firstLetter
```

Numbers are (unsigned) integers. Integers are sequences of digits, and have their usual decimal interpretation. (The implementation uses 32-bit arithmetic.)

String literals are sequences of zero or more characters or escape sequences enclosed in quote marks ("). The number of characters in a string is called the length of the string. Strings can be used only in fairly limited contexts. A string literal may not extend over a line break in the source text.

Character literals are denoted by a single graphic character or a single escape sequence between single quote marks ('). Character literals denote the integer value of the corresponding ASCII character.

Within a string or character literal the following escape sequences denote non-graphical characters

```
\b         Backspace        (CHR(8))
\t         Horizontal tab   (CHR(9))
\n         Line feed        (CHR(10))
\f         Form feed        (CHR(12))
\r         Carriage return  (CHR(13))
\"         Quotation mark   (CHR(34))
\'         Apostrophe       (CHR(39))
\x         (where x is not b, t, n, r or f) denotes x itself
```

(Within a string \n denotes the sequence Carriage return, Line feed, on Microsoft systems.)

Examples:

```
"Parva will become Magna"    "He said\"food!\" and fed a line\n"
```

Operators and delimiters are the special characters, character pairs, or reserved keywords listed below. The reserved words cannot be used in the role of identifiers. Those in parentheses are not currently keywords, but are reserved for use in later extensions .

```
||   (    ==   ,    bool       (default)  halt      return
&&   )    !=   ;    (break)    (do)       if        (switch)
*    {    >    ++   (case)     (else)     int       true
/    }    >=   --   (char)     false      new       void
%    [    <    =    const      (for)      null      while
+    ]    <=   !    (continue) (goto)     read      write
-    []
```

4 Programs

A program is a collection of declarations of functions, constants and variables (whose values are said to constitute the program state). Sequences of statements within the functions alter the program state.

```
Parva = { Declaration } .
```

5 Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration. Declarations also serve to specify certain permanent properties of an entity, such as its type, and whether it is a constant, a variable, a parameter, a function, or an array.

The identifier is then used to refer to the associated entity. This is possible only in those parts of a program that are within the scope of the declaration. No identifier may denote more than one entity within a given scope.

We distinguish between identifiers declared outside of functions and those declared within functions. Identifiers declared outside of functions (including the identifiers of the functions themselves) are said to possess *global* scope, which extends textually from the point of declaration to the end of the program.

Identifiers declared within functions (including identifiers that denote formal parameters of those functions) have a scope that extends textually from the point of the declaration to the end of the block (see 13.1) in which the declaration has been made, and to which the entity denoted by the identifier is said to be *local*.

```
Declaration = ConstDeclarations | VarDeclarations | FunctionDeclaration .
```

6 Type

In general, a type determines the set of values which variables, constants and parameters declared to be of that type may assume, and the operators that are applicable to such variables, constants and parameters.

```
Type      = BasicType [ "[]" ] .
BasicType = "int" | "bool" .
```

(The so-called `void` type has a special meaning in the context of function declarations and is not used in other contexts.)

Level 2 of Parva supports only two basic data types, representing the set of Boolean values {`false, true`} and the set of integer values {`min ... max`}, where `min` and `max` are implementation defined (the implementation uses 32-bit arithmetic {-2147483648 … 2147483647}).

A type that incorporates the `[]` token is said to be an *array reference type*. Arrays are constructed as components of expressions by using the keyword `new` (see section 11.4).

Two entities are said to be *type-compatible* (for the purposes of assignment or comparison) if they are of the same type. In general the type of a value defined by an *Expression* (see section 11) is computed from its operators and the types of its operands; the type of a variable, constant or parameter is determined from the type used in its declaration. However, the value denoted by `null` is a value of all array reference types.

7 Constant declarations

A constant declaration permanently associates an identifier with a constant value.

```
ConstDeclarations = "const" OneConst { "," OneConst } ";" .
OneConst          = identifier "=" Constant .
Constant          = number | charLit | "true" | "false" | "null" .
```

The type of the identifier is determined from the context of the declaration. `true` and `false` are the Boolean constants. Numbers and character literals are deemed to define constants of integer type.

Examples:

```
const
  max = 100;
  YES = true;
  CapitalA = 'A';
```

8 Function declarations

A function declaration serves to define operations that may be performed on global constants and variables, on the values of its local constants and variables, and on values passed to the function as arguments.

```
FunctionDeclaration = ("void" | Type ) identifier "(" FormalParameters ")"  Block .
FormalParameters    = [ OneParam { "," OneParam } ] .
OneParam            = Type identifier .
Block               = "{" { Statement } "}" .
Type                = BasicType [ "[]" ] .
BasicType           = "int" | "bool" .
```

A restriction is imposed in the current implementation which requires each identifier in a program to be "declared" before it is "used". In many cases this is easily achieved by constituting a program as a set of declarations arranged in an order that meets this requirement. In some situations - typically those in which mutually recursive methods need to invoke one another - this will be found to be impossible. Solutions to this restriction are left as an exercise.

A `void` function is activated by a function call that is a form of *Statement* (see section 13.3). A function of any other type is activated by a function call that forms a constituent part of an *Expression* (see section 11), and yields a result that is an operand of that expression.

The *Block* uniquely associated with each function incorporates a collection of declarations of constants and local variables (whose values are said to constitute the program state), and a sequence of other statements whose purpose is to alter the program state by manipulating the local constants and variables, formal parameters and global constants and variables that are in scope.

The *FormalParameters* associated with a function provide one mechanism by which data may be transmitted from one function to another (see sections 11.3 and 13.3).

One function is uniquely designated as the `main` function. This is the last to be declares, but the first to be executed when the program as a whole is executed. It must be of type `void` and has no formal parameters; that is, it is declared with the header `void main()`.

9 Formal Parameters

Formal parameters that are a component part of a *FunctionDeclaration* are specified in a (possibly empty) list of identifiers that denote actual parameters that are specified only when a function is called. The correspondence between formal and actual parameters is established only when the call takes place.

```
FormalParameters = [ OneParam { "," OneParam } ] .
OneParam         = Type identifier .
Type             = BasicType [ "[]" ] .
BasicType        = "int" | "bool" .
```

The type of a formal parameter is specified in its declaration; this type must be compatible with the type of the actual parameter used in the *FunctionCall*. There must be as many arguments in the *ArgList* of the *FunctionCall* as there are parameters in the *FormalParameters* of the *FunctionDeclaration*. In particular, a function declared without parameters has an empty parameter list, and must be invoked by a *FunctionCall* whose actual *ArgList* is empty.

Examples:

```
int larger (int a, int b) {
// Returns larger of the two arguments
  if (a > b) return a;
  return b;
}

int sum (int[] list, int n) {
// Returns sum of n elements - list[0] ... list[n-1]
  int total = 0;
  int i = 0;
  while (i < n) { total = total + list[i]; i = i + 1; }
  return total;
}
```

10 Variable declarations

Variables are those data items whose values may be changed by execution of the program. Variable declarations serve to introduce variables and to associate them with identifiers that must be unique within their given scope. They also serve to associate a fixed data type with each variable so introduced.

```
VarDeclarations = Type OneVar { "," OneVar } ";" .
OneVar          = identifier [ "=" Expression ] .
Type            = BasicType [ "[]" ] .
BasicType       = "int" | "bool" .
```

Each variable identifier denotes either a simple scalar variable of a specified *BasicType* (integer or Boolean), or a reference to an array structure of scalar elements that all have the *BasicType* specified in their declaration. Arrays are constructed as components of expressions by using the keyword `new` (see section 11.4).

Variables whose identifiers appear in the same list are all of the type specified in the declaration.

When the statement sequence that is defined by a *Block* is activated, all local variables are deemed to have initially undefined values, except for those variables that have been assigned the values of expressions within the variable declaration sequence where they were declared.

Similarly, variables with global scope are deemed to have initially undefined values, except for those variables that have been assigned the values of expressions within the global variable declaration sequence where they were declared.

If a variable is assigned the value of an *Expression* the type of the variable must be compatible with the type of the *Expression*.

Examples:

```
int i, j = 8;
bool startedDating, suitableSpouse;
int[] list = new int[12], list2 = null;
```

Variables used in future examples are assumed to have been declared as indicated above.

11 Expressions

Expressions are constructs denoting rules of computation whereby constants and current values of variables and parameters are combined to derive other values by the application of operators. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands where the normal rules of precedence are unsuitable on their own.

```
Expression   = AddExp [ RelOp AddExp ] .
AddExp       = [ "+" | "-" ] Term { AddOp Term } .
Term         = Factor { MulOp Factor } .
Factor       =   Designator | Constant | FunctionCall
               | ArrayValue | "!" Factor
               | "(" Expression ")".
ArrayValue   = "new" BasicType "[" Expression "]" .
Designator   = identifier [ "[" Expression "]" .
Constant     = number | charLit | "true" | "false" | "null" .
FunctionCall = identifier "(" ArgList ")" .
ArgList      = [ OneArg { "," OneArg } ] .
OneArg       = Expression .
AddOp        = "+" | "-" | "||" .
MulOp        = "*" | "/" | "%" | "&&" .
RelOp        = "==" | "!=" | "<" | "<=" | ">" | ">=" .
```

11.1 Operands

With the exception of numbers and character literals, many operands in expressions are denoted by designators. A *Designator* consists of an identifier referring to the constant, variable or parameter to be designated. This identifier may be followed by a selector if the identifier denotes an array.

```
Designator = identifier [ "[" Expression "]" ] .
```

If `A` designates an array, then `A[E]` denotes that element of `A` whose index is the current value of the expression `E`. `E` must be of integer or character type, and the value of `E` must lie within the range of possible values for the index of `A`. This range is from `0 ... ArraySize - 1`, as specified when `A` was created.

If the designated entity is a variable or an array element, then, when used as an operand, the designator refers to the variable's current value, and the type of the operand is the type of that variable or array element. An operand that is a literal constant denotes a value that is the value of that constant, and the type of the operand is the type of that literal constant.

An operand may also be designated by a function call (see section 11.3) or by the application of the `new` operator (see section 11.4).

11.2 Operators

The syntax of expressions distinguishes between four classes of operators with different precedences (binding strengths). The operator `!` has the highest precedence, followed by multiplication operators, addition operators, and then relational operators. Operators of the same precedence associate from left to right. Thus, for example, `x - y - z * w` stands for `(x - y) - (z * w)`.

The available operators are listed in the following tables.

11.2.1 Logical operators

symbol	result
`||`	logical disjunction (Boolean OR)
`&&`	logical conjunction (Boolean AND)
`!`	negation

These operators apply only when both operands are of the Boolean type, and yield a Boolean result.

It is desirable that the implementation should enforce *short-circuit semantics* such that

`p || q`	stands for "if `p` then `true`, else `q`"
`p && q`	stands for "if `p` then `q`, else `false`"
`! p`	stands for "not `p`"

However, this has been left as an exercise.

11.2.2 Arithmetic operators

symbol	result
`+`	sum
`-`	difference
`*`	product
`/`	quotient
`%`	modulus

These operators apply only to operands of integer type, and yield a result of integer type. When used as operators with a single operand, `-` denotes sign inversion and `+` denotes the identity operation.

The operator / produces a result that is truncated towards zero. Implementation of the operator % has been left as an exercise: it is required to produce a result so that the following relation holds for any dividend x and divisor y:

```
x  =  (x / y) * y  +  (x % y)
```

Examples

```
   x       y     x / y   x % y

  12       7       1       5
  12      -7      -1       5
 -12       7      -1      -5
 -12      -7       1      -5
```

11.2.3 Relational operators

symbol	relation
==	equal
!=	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal

These operators yield a result of Boolean type. The ordering operators <, <=, > and >= apply to operands of the integer type. The relational operators == and != also apply when both operands are of the Boolean type or of the same array reference type (when the comprison is between the values of the references, and *not* between the individual elements of the arrays referred to). The operands of a relational operator may be evaluated in any convenient order.

Examples of expressions:

```
1996                              (Integer)
i / 3                             (Integer)
!startedDating || suitableSpouse  (Boolean)
(i+j) * (i-j)                     (Integer)
(0 <= i) && (i < max)             (Boolean)
```

11.3 Function calls

A *FunctionCall* that serves to activate a non-void function may appear as an operand in an *Expression*, where it represents the value computed and returned by that particular activation of the designated function.

```
FunctionCall = identifier "(" ArgList ")" .
ArgList      = [ OneArg { "," OneArg } ] .
OneArg       = Expression .
```

The call may contain an *ArgList* of arguments or actual parameters which are substituted in place of the corresponding *FormalParameters* defined in the *FunctionDeclaration*. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The types of corresponding actual and formal parameters must be compatible, and there must be as many actual arguments in the *ArgList* as there are parameters in the *FormalParameters* of the function called.

Note: Parva does not support function overloading, whereby two or more functions may share a common name and return type but differ in the number of arguments that may be supplied.

There are two kinds of actual parameters. If the formal parameter is of a reference type, the actual parameter, while syntactically an *Expression*, must be a *Designator* denoting an entity of that type (or a formal parameter of that type in the function in which the *FunctionCall* is an operand). In the case of a value parameter, the actual parameter may be a more general *Expression*. Each expression is evaluated prior to the function activation, and the resulting value is assigned to the formal parameter, which then constitutes a local variable of the called function.

The use of a function identifier in an *Expression* within the *Body* associated with the definition of that function implies a recursive activation of that function.

11.4 Array instantiation

Operands in expressions may have values that are obtained when a new instance of an array is created.

```
ArrayValue = "new" BasicType "[" Expression "]" .
BasicType  = "int" | "bool" .
```

An array value allows for the creation of a new instance of an array, whose elements are all of the type specified by *BasicType*. The number of elements is specified by the value of the *Expression*, which must yield a positive value of integer type. The value of the operand is the reference to the newly created array. Typically this is assigned to a variable declared to be of a type specified as `BasicType[]`, whereafter the elements of the array may be accessed by use of appropriate *Designators*.

Examples:

```
int[] list = new int[100];
bool[] sieve = new bool[Max + 1];
```

12 Statements

Statements denote actions or the declaration and possible initialization of variables and constants. Apart from declarative statements, a distinction is drawn between elementary and structured action statements.

Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the void function call, the input statement, the output statement, the return statement, the halt statement and the empty statement.

Structured statements are composed of parts that are themselves statements. They are used to express sequencing, as well as conditional, selective, and repetitive execution.

A statement may assume the form of a *Block*, and may thus incorporate declarations of identifiers, whose scope is local to that block.

```
Statement =   Block | ConstDeclarations | VarDeclarations
            | Assignment | VoidFunctionCall | EmptyStatement
            | IfStatement | WhileStatement
            | ReadStatement | WriteStatement
            | ReturnStatement | HaltStatement
```

12.1 Blocks (statement sequences)

A *Block* statement may be used to group several statements and declarations into one indivisible unit. Block statements are frequently used as parts of structured statements.

```
Block = "{" { Statement } "}" .
```

Statement sequences within a *Block* statement denote the sequence of actions and declarations specified by the component statements, executed in the order specified by the sequence.

12.2 Assignments

An *Assignment* denotes the replacement of the value of a variable with a new value, and results in a change to the state of a program.

```
Assignment = Variable "=" Expression ";" .
Variable   = Designator .
```

If the *Expression* is present, the assignment serves to replace the current value of the designated *Variable* by the value specified by the *Expression*. The assignment operator is written as "=" and pronounced as "becomes". The types of the *Expression* and of the *Variable* must be compatible, and the *Variable* must designate a scalar variable, an array element or an array reference.

Examples:

```
i = 0;
suitableSpouse = age < 30;
list = new int[max];
list[i] = list[i] + 1;
```

12.3 Void function calls

A *VoidFunctionCall* serves to activate a `void` function.

```
VoidFunctionCall = identifier "(" ArgList ")" .
ArgList          = [ OneArg { "," OneArg } ] .
OneArg           = Expression .
```

The designated function must have been declared of type `void`. There is no concept, as in C# or Java, of invoking a non-void function and silently discarding the return value.

The call may contain an *ArgList* of arguments or actual parameters which are substituted in place of the corresponding *FormalParameters* defined in the *FunctionDeclaration*. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The types of corresponding actual and formal parameters must be compatible, and there must be as many actual arguments in the *ArgList* as there are parameters in the *FormalParameters* of the function called.

Note: Parva does not support function overloading, whereby two or more `void` functions may share a common name but differ in the number of arguments that may be supplied.

There are two kinds of actual parameters. If the formal parameter is of a reference type, the actual parameter, while syntactically an *Expression*, must be a *Designator* denoting an entity of that type (or a formal parameter of that type in the function in which the *VoidFunctionCall* is a statement). In the case of a value parameter, the actual parameter may be a more general *Expression*. Each expression is evaluated prior to the function activation, and the resulting value is assigned to the formal parameter, which then constitutes a local variable of the called function.

The use of a function identifier in a *VoidFunctionCall* within the *Body* associated with the definition of that `void` function implies a recursive activation of that function.

12.4 If statements

An *IfStatement* specifies the conditional execution of a guarded statement.

```
IfStatement = "if" "(" Condition ")" Statement .
Condition   = Expression .
```

The *Expression* (guard) must yield a Boolean result. If the guard evaluates to `true` the guarded *Statement* is executed.

(There is no `else` clause in level 2 of Parva. Implementation of this is intended as an exercise.)

Examples:

```
if (i > max) { write("limit exceeded\n"); return; }
if (j % 3 = 0) write("divisible by 3");
```

12.5 While statements

A *WhileStatement* is one form of specifying the repetition of an associated statement.

```
WhileStatement = "while" "(" Condition ")" Statement .
Condition      = Expression .
```

The *Expression* that is the *Condition* must yield a Boolean result. If this expression yields `true`, the associated *Statement* is executed. Evaluation of the condition is performed once at the start of each complete iteration, and the test and statement execution are repeated as long as the *Condition* yields `true`.

Example:

```
while (j > 0) { j := j / 2; i := i + 1; }
```

12.6 Input statements

```
ReadStatement = "read" "(" ReadElement { "," ReadElement } ")" ";" .
ReadElement   = string | Variable .
```

A *ReadStatement* specifies a list of variables that are to be assigned new values from an input source external to the program. Each *Variable* must designate a scalar variable, or an element of an array. A *string* appearing in a *ReadStatement* serves to act as a prompt to the user of the program.

Example:

```
read ("Supply Age", age, "\nSupply Mass", mass);
```

12.7 Output statements

```
WriteStatement = "write" "(" WriteElement { "," WriteElement } ";" .
WriteElement   = string | Expression .
```

A *WriteStatement* specifies a list of strings and expressions whose values are to be computed and then transferred to an output sink external to the program.

Example:

```
write("The result is ", i + j , "\n");
```

12.8 Return statements

A *ReturnStatement* causes immediate termination of execution of the function in which it appears, returning control to the invoking environment.

```
ReturnStatement = "return" [ Expression ] ";" .
```

In the case of `void` functions the *Expression* must be absent. In all other functions it must be present, and defines the value to be returned as the result of invoking that function. The type of this *Expression* must be compatible with that of the function as specified in the *FunctionDeclaration*. A function may incorporate several *ReturnStatements*, although, of course, only one can be executed in any particular activation of the function. In `void` functions an implicit *ReturnStatement* occurs at the end of the function *Block*.

12.9 Halt statements

A *HaltStatement* causes immediate termination of the program execution, returning control to the invoking environment.

```
HaltStatement = "halt" ";" .
```

12.10 Empty statements

An *EmptyStatement* causes no change to the program state.

```
EmptyStatement =  ";" .
```

The *EmptyStatement* is included in order to relax punctuation rules in statement sequences.

13 Complete example

```
// Find all solutions to the N Queens problem - how do we place N queens on
// an N*N chessboard so that none of them is threatened by any other one.
// Code closely based on that by Niklaus Wirth in "Algorithms + Data
// Structures = Programs" (Prentice-Hall, 1976; page 145), but generalised
// to allow N Queens, rather than only 8.
// Count solutions and iterate the process a requested number of times
// Modifications by Pat Terry, Rhodes University, 2003

  int solutions;

  void DisplaySolution(int[] x, int n) {
  // Display one solution to the N Queens problem
    int i = 1;
    while (i <= n) {
      write(x[i]);
      i = i + 1;
    }
    write("\n");
  }

  void Place(int i, int n, bool[] a, bool[] b, bool[] c, int[] x) {
  // Place the i-th queen on the board of size n * n
    int j = 1;
    while (j <= n)  {
      if (a[j] && b[i+j] && c[i-j+n])  {
        x[i] = j;
        a[j] = false; b[i+j] = false; c[i-j+n] = false;
        if (i < n) Place(i+1, n, a, b, c, x);
        if (i >= n) {
          solutions = solutions + 1; DisplaySolution(x, n);
        }
        a[j] = true; b[i+j] = true; c[i-j+n] = true;
      }
      j = j + 1;
    }
  }

  void main() {
    int n, iterations;
    read("Board size? ", n);
    read("Iterations? ", iterations);
    bool[] a = new bool[n + 1];
    bool[] b = new bool[2 * n + 1];
    bool[] c = new bool[2 * n + 1];
    int[] x = new int[n + 1];
    int count = 0;
    while (count < iterations) {
      solutions = 0;
      int i = 1;
      while (i <= n) {
        a[i] = true; i = i + 1;
      }
      i = 1;
      while (i <= 2 * n) {
        b[i] = true; c[i] = true; i = i + 1;
      }
      Place(1, n, a, b, c, x);
      count = count + 1;
    }
    write("Board size ", n);
    write(" Solutions ", solutions);
    write(" Iterations ", iterations);
  }
```

14 Missing statements

There are no `do`, `goto`, `for`, `break`, `switch` or `continue` statements in this release of the language. Their implementation is left as a series of exercises!

15 Bibliography

Engel, J. (1999) *Programming for the Java Virtual Machine*, Addison-Wesley, Reading MA.

Mössenböck, H. (2004) `http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/`

Reiser, M. and Wirth, N. (1992) *Programming in Oberon*, Addison-Wesley, Wokingham, England.

Terry, P.D. (1997) *Compilers and Compiler Generators: an Introduction With C++*, International Thomson, London.

Terry, P.D. (2005) *Compiling with C# and Java*, Pearson, London.

Wirth, N. (1985) *Programming in Modula-2* (3rd edn), Springer, Berlin.