PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

DOPLs : A NEW TYPE OF PROGRAMMING LANGUAGE

Graham Lee

Department of Computer Science
University of Western Australia

ABSTRACT

The importance of *operand description* in programming is emphasised, and programming languages are classified into *Description-Oriented Programming Languages* (DOPLs) and *Identifier-Oriented Programming Languages* (IOPLs) according to their *operand-description* facilities. Several examples are used to illustrate DOPLs, and the advantages, in terms of the level of transparency in programs, of using DOPLs over IOPLs.

1. DOPLs and IOPLs

Programming languages can be classified according to their facilities for describing which *operands* are to be used in an operation. There are two main classes:

* Languages which have a large variety of *operand-description* facilities. These will be called *Description-Oriented Programming Languages* (DOPLs) [Lee, 1978].
* Languages whose only operand-description facilities are identifiers and names. These will be called *Identifier-Oriented Programming Languages*[1] (IOPLs).

Examples of IOPLs range from very primitive languages such as a von Neumann machine code, through the simpler high-level languages such as Fortran, to much more sophisticated languages such as Pascal and Algol 68.

An example of a language with a large variety of operand-description facilities is English. In fact, one of the main differences between English and existing programming languages lies in its use of, for example, adjectives, participles, adverbs, nouns, pronouns and names when describing operands. These operand-description facilities account for much of the expressive power of English, and it therefore seems worthwhile to incorporate similar facilities in an algorithmic language. The design of a DOPL can be influenced by the operand-description facilities of English, as far as is commensurate with a formal, unambiguous programming language.

The advantage of using a DOPL, as opposed to an IOPL, is that more transparent, though possibly less efficient, programs can be written. The level of operand-description facilities available in a language greatly influences the structure of, and amount of detail in, programs. The operand-description facilities available in a DOPL enable algorithms to be specified without using variables, data structures, control

---

[1] This represents a change of terminology from Lee [1978]

structures with nested statements, or input statements. On the other hand, because identifiers and names can only refer to one operand at a time, all the above features are required in IOPLs mainly to support the computation of names for individual operands. IOPL programs are oriented towards specifying a detailed, controlled series of operations on *individually named operands*, whereas DOPL programs are oriented towards direct descriptions of the *whole sequence of operands* to be used in an operation. The latter is more transparent than the former. In IOPL programs, there is a conceptual gap between the explicit information given - the detailed sequence of operations on individually named operands - and the actual information required to understand the algorithm - information on the whole group of operands involved. IOPL programs cannot fill this gap, which must be bridged for each individual reading of a program. DOPL programs, on the other hand, give the latter information explicitly.

The operand-description facilities of the DOPL discussed here can be used to describe the sequence of all the operands to be used in an operation, the data for a program, the required results of an operation, to define new description facilities, and to define data structures.

Although existing languages vary in their operand-description facilities, and although there are examples of languages with operand-description facilities other than identifiers and names (see, for example, Astrahan and Chamberlain [1975], Barron [1977], Burger et al [1975], Chamberlain and Boyce [1974], Feldman and Rovner [1969], Findler [1969], Hebditch [1973], Housel and Shu [1976], Martin [1976], Potts [1970]), and although there have been suggestions for language extensions which are actually concerned with operand-description facilities (Herriot [1977], Nylin and Harvill [1976]), no existing programming language seems to have the breadth and type of operand-description facility envisaged here.

In subsequent sections, several examples are used to introduce a DOPL and to compare it to Pascal. The syntax and semantics of DOPLs are discussed in section 6. To facilitate discussion prior to this section, the following brief definitions are given. A DOPL program contains a sequence of *requests*, and is executed by using each of these requests in turn. Requests may specify operations, or define data, results or new operand-description facilities. An operational request contains *operators* and *operand descriptions*. These descriptions specify the whole sequences of operands to be used in the operation, and the request is executed by applying the operators to each of these operands in turn. In an operand description, each word is a *descriptor*, and nouns, pronouns, adjectives and identifiers are among the kinds of descriptor used. In the DOPL examples, all operators (and all operator-like terms) are in upper case, and all descriptors are in lower case. User-introduced operators and descriptors are in script.

2. THE SIEVE OF ERATOSTHENES

Consider first the following DOPL request for generating all the prime integers less than or equal to a given data integer:

PRINT each *prime* integer <= the data integer

It consists of the operator PRINT followed by an operand description which describes the sequence of operands to be used in the PRINT operation. The operand description is built from several descriptors, of which each, integer, <=, the, data, are primitive, and *prime* is user-defined.

An integer is an item in the 2-way infinite sequence of negative and positive whole numbers, and the descriptor each in the above operand description specifies all of those integers satisfying the conditions specified by the adjective *prime* and the relation

<= the data integer

Thus the operand description specifies a sequence of prime integers up to a given data integer, and the PRINT request is executed by PRINT-ing each one of these in turn.

A DOPL program for generating primes using the above request is shown in program 2.1. It consists of three requests.

```
program    prime-number generation:
DATA IS    an integer.
ADJECTIVE  prime
AS IN      prime integer
IS         integer >1
SUCH THAT  (the prime integer)
           mod
           (any integer > 1
                 and <= square root (the prime integer))
           <> 0.
PRINT      each prime integer <= the data integer
end.
```

Program 2.1 A DOPL program for generating prime numbers

The first one defines the program's data to be an integer, which can subsequently be referred to as *the data integer*. The second one defines the adjective *prime*. The line

ADJECTIVE *prime*

specifies that a new adjectival descriptor is being defined. The line

AS IN *prime* integer

specifies that this descriptor must be used with other descriptors which specify an integer. The line

IS integer > 1

says that a *prime* integer is an integer (> 1) subject to the condition following SUCH THAT, which specifies that a *prime* integer is one which is not divisible by any other integers >1.

Given the usual definition of a prime, and given that a non prime is divisible by an integer <= its square root, this program must be correct. It is evident from the operand descriptions used that the printed results consist of all the primes up to the given data integer.

Consider now the Sieve of Eratosthenes. The essential feature of this prime-number-generation algorithm is the *removal* of multiples of integers from a sequence initially containing all the integers between 2 and a given data integer. First the multiples of 2 are removed, then the multiples of 3, then the multiples of 5 (4 having been removed because it is a multiple of 2), and so on. At each stage, the multiples of the next non-removed integer (which must be a prime - the fact that it has not been removed means that it cannot be a multiple of any integer less than it) are removed. When all multiples have been removed, the non-removed integers constitute the primes between 2 and the given data integer.

This process can be specified in a DOPL by the request:

```
REMOVE each multiple <= the data integer
            of each non remove-ed integer
                    between 2 and the data integer
```

This consists of the user-introduced operator *REMOVE*, followed by an operand description which is built from several descriptors, of which each, <=, the, data, integer, of, non, between, 2, and, are primitive, and *multiple*, *remove*-ed are not.

The operand description specifies a sequence of operands consisting of each *multiple* (<= the data integer) of each of the integers described by the *nested operand description* (the one following of):

```
each non remove-ed integer
     between 2 and the data integer
```

The request is executed by applying the *REMOVE* operator to each of these operands.

Although *REMOVE* is a non-primitive operator, it is not necessary to give a procedure specifying how to remove integers! This is because of the use of the adjective *remove*-ed, which specifies a condition on integers which becomes true when they are used as operands of *REMOVE*. Initially, no integers have been so used, and therefore the condition

non *remove*-ed

is true of all integers to begin with.

The description:

each integer between 2 and the integer data

specifies the sequence of integers: 2, 3, 4, ..., the data integer, and causes each one of these to be generated in turn so that the condition

non *remove*-ed

can be checked. Thus the first integer specified by the nested operand description is 2, and the first operands specified by the entire operand description of the request are therefore:

```
each multiple <= the data integer
     of 2
```

and so the multiples of 2 are *REMOVE*-ed. After this, the condition *remove*-ed is true of the multiples of 2.

The nested operand description now specifies the next non *remove*-ed integer, which is 3, and so

```
     each multiple <= the data integer
          of 3
```

is *REMOVE*-ed. This process continues until there are no further non *remove*-ed integers.

After executing the *REMOVE* request, the prime numbers can be printed using the request:

```
     PRINT each non remove-ed integer
                between 2 and the data integer
```

A complete DOPL program for the Sieve process is shown in program 2.2.

```
program   Sieve of Eratosthenes:
DATA IS   an integer.
NOUN      multiple
AS IN     multiple of an integer
IS        (the integer)*(any integer >1).
REMOVE    each multiple <= the data integer
               of each non remove-ed integer
                       between 2 and the data integer.
PRINT     each non remove-ed integer
               between 2 and the data integer
end.
```

Program 2.2 A DOPL version of the Sieve of Eratosthenes

The program consists of four requests. The first describes the data, the second defines the noun *multiple*, the third is the *REMOVE* request, and the fourth prints the primes.

The descriptor *multiple* is used as a noun (the syntax of operand descriptions is discussed in section 6) in the *REMOVE* request, and so its definition begins with NOUN. The line

```
     AS IN multiple of an integer
```

specifies that *multiple* is to be used with a nested operand description which specifies one or more integers. The line

```
     IS   (the integer)*(any integer >1)
```

defines a *multiple* to be a product of two integers. The descriptor the in the factor

```
     the integer
```

refers back to the previous mention of an integer, which is in

```
     multiple of an integer.
```

The descriptor each specifies each item of a sequence from the first onwards. The definition of *multiple* can be interpreted as a definition of a sequence of multiples by virtue of the factor

```
     any integer >1
```

in the expression following IS. Thus, in the *REMOVE*-request operand description

each *multiple* of <an integer>

the descriptor each specifies the sequence of multiples

(the integer)*2
(the integer)*3
(the integer)*4

and so on.

The DOPL program can be *judged* to be correct given the definition of a prime and given that every non prime is a multiple of some integer less than it.

The DOPL program can be contrasted with the Pascal version in program 2.3.

```
program  Eratosthenes(input, output);
const    n = ?;
var  sieve : array[2 .. n] of integer;
     data, i, m : integer;
begin
read(data);
for  i := 2 to data do sieve[i] := i;
sieve[data + 1] :=1 ;
i := 1;
repeat
     repeat i := i + 1 until sieve[i] > 0;
     m := 2*i;
     while m <= data do
          begin
          sieve[m] := 0;
          m := m + i
          end
until i > data;
for  i := 2 to data do
     if   sieve[i] > 0 then writeln(sieve[i])
end.
```

Program 2.3 A Pascal (IOPL) version of the Sieve of Eratosthenes

(Straight-forward representations of the sieve and of the removal operation are used in this example, in order to facilitate comparison of the two versions. Another, more efficient and more complex IOPL version, and its proof using invariants, can be found in Hoare [1972]. This IOPL version does not necessarily represent the way in which the DOPL version would be implemented.)

This Pascal version is more difficult to understand and prove correct than the DOPL version. Removal of multiples is done using the assignment

sieve[*m*] := *0*

but because this can only reference one operand at a time, it has to be placed inside two levels of nested loop, one to vary *m* so that all multiples are removed, and one to vary *i* so that all multiples of all primes are removed. Also, an extra loop is required to search for non-removed integers. The loops are used solely to compute the names

sieve[*m*]

of the removed multiples, and the array data structure, and the other variables, are used mainly to construct the above names.

In the IOPL version, the remove (assignment of *0* to a *sieve* component) operation is nested inside two levels of loop, and involves several variables. Before encountering this operation, the explicit loop statements, and other nested operations, have to be read. In fact, there is no syntactic clue to the fact that the assignment to a *sieve* component *is* the main operation. Rather, this has to be gathered from a complex combination of information given in several different places in the program. Once it is known that this is the main operation of the loops, the information on all the variables, which is distributed in different places in declarations, initialisations and updates, together with the explicit nested looping information, has to be gathered together and used to decide what the entire group of remove-ands and non remove-ands are. It is only this operand information which enables an understanding of the total process specified by the loops. In the DOPL version, on the other hand, the main operator *REMOVE* is placed first, and the sequence of all its operands is made explicit using one operand description. The detailed control information is implicit in the semantics of the descriptors used. Also, the DOPL version can define the data, and the terms *prime* and *multiple* (the adjective *prime* in program 2.2 is equivalent to non *remove*-ed). For these reasons the DOPL version is more transparent than the IOPL version.

3. <u>SORTING</u>

Consider first the problem of sorting a sequence of data integers:

```
DATA IS    several integer
```

This could be done using the request

```
PRINT      smallest non print-ed data integer
UNTIL      print-ed each data integer
```

However, sorting in a DOPL can be specified without using a particular algorithm, by specifying what the result of sorting should be:

```
to         SORT a sequence of integers:
           RESULT IS ascendingly-ordered permutation
                                         of the parameter sequence
end
```

This is an example of a *procedure*. It defines the user-introduced operator *SORT*, by specifying what the result of an operation such as

```
SORT       the data sequence
```

should be.

The descriptor *ascendingly-ordered* can be defined as follows:

```
ADJECTIVE       ascendingly-ordered
AS IN           ascendingly-ordered integral sequence
IS              integral sequence
SUCH THAT       each integer of the sequence
                is - <=
                next integer of the sequence
```

The descriptor *permutation* (which might actually be primitive in a DOPL) can be defined as

```
NOUN            permutation
AS IN           permutation of sequence
IS              sequence containing each item of
                          sequence
                          of-which permutation is-being-defined
SUCH THAT FOR   any item of the permutation
WE HAVE         number of item = such-that-for-and
                          of the permutation
                is - =
                number of item = such-that-for-and
                          of sequence
                             of-which permutation is-being-defined
```

This contains rather involved conditions in the descriptions after IS and WE HAVE. These specify that a permutation contains exactly the same items as the original sequence, but not necessarily in the same order.

The operand description after IS has the form

    sequence containing <description of items to be contained>

In the description of the items to be contained, the nested description

    sequence of-which *permutation* is-being-defined

specifies the sequence in

    AS IN *permutation* of sequence

and the

    of-which ... is-being-defined

reverses the descriptor of in

    permutation of sequence

This could be shortened using an identifier:

    AS IN *permutation* of sequence called *x*

after which, throughout this request, the sequence of which *permutation* is being defined can be referred to as *x*. It seems better not to use the identifier.

The description after WE HAVE is a Boolean expression which has the structure

```
number of <description of an item of the permutation>
is-equal-to
number of <description of an item of
              the sequence of which permutation is being defined>
```

The noun such-that-for-and refers to the item described after SUCH THAT FOR. An identifier, for example *y*, could be used in place of this noun, if the description after SUCH THAT FOR is modified:

any item called *y*
of the *permutation*
The use of the primitive noun such-that-for-and is to be preferred. With this noun, it is rather more obvious which item is being referred to than with a user-introduced identifier such as *y*, which could have been declared anywhere in the request (or in the whole program).

An *ascendingly-ordered permutation* of a sequence can be produced by generating sequences in lexicographic order and checking all the conditions given in the definition, and then checking for *ascendingly-ordered*-ness. This would be impossibly inefficient for long parameter sequences. Even so, the *SORT* procedure is a formal specification of sorting.

Consider now program 3.1, which is a procedure for sorting a sequence of integers by partitioning it into three groups.

```
to  PARTITION SORT a sequence of integers:
    CHOOSE    any parameter integer.
    RESULT IS
         result of partition-sort-ing
              each parameter integer < the choose-and,
         each parameter integer = the choose-and,
         result of partition-sort-ing
              each parameter integer > the choose-and
end
```

Program 3.1 A DOPL procedure for sorting by partitioning

The procedure contains two requests. The first chooses one of the integers of the parameter sequence. This is subsequently referred to as the *choose*-and.

The RESULT IS request specifies a partition of the parameter sequence into three groups, which contain those integers less than the chosen integer, those equal to it, and those greater than it respectively. The commas in the operand description of this request can be read as "followed by", and the descriptor followed-by could be used in their place. In the description
result of *partition-sort*-ing
each parameter integer < the *choose*-and
the descriptor *partition-sort*-ing implies a recursive application of the operator *PARTITION SORT* to the sequence of parameter integers less than the chosen integer. There is no need to explicitly specify what the result is for a null sequence, because the following rule can be adopted in a DOPL:

the result of performing any operation on the null sequence is the null sequence (unless otherwise specified).

Because of the operand descriptions used, it is evident that this procedure *recursively partitions* the parameter sequence:

DEFINITION:

A *recursively partitioned* sequence is either a null sequence, or a sequence comprising a left partition, followed by a middle partition consisting of several equal items, followed by a right partition, such that

(a) each item of the left partition is < the middle items,
(b) each item of the right partition is > the middle items,
(c) the left and right partitions are recursively partitioned sequences.

It is intuitively obvious that a recursively partitioned sequence is ascendingly ordered. This can be proved as follows:

PROPOSITION:

A recursively partitioned sequence is ascendingly ordered.

PROOF by reductio. Suppose not, and consider the shortest sequence which is recursively partitioned but not ascendingly ordered. This sequence must have at least two adjacent items which are out of order. These cannot both be in the same partition, otherwise a shorter, recursively partitioned but non-ascendingly ordered sequence would exist. Also, if one of these items is in the left partition, the other cannot be in the middle partition because of the stated property of the left partition. Similarly, if one of these items is in the right partition, the other cannot be in the middle. This leads to a contradiction, and so the result is proved.

* * *

From this proposition, program 3.1 can be *judged* to be a correct sorting procedure. A DOPL program to sort a sequence of data integers can use the request

PRINT result of *partition-sort*-ing the data sequence

This will print the data integers in ascending order.

Another DOPL procedure for sorting by partitioning, this time into two groups called the *left-partition* and the *right-partition*, is shown in program 3.2. This procedure can be judged to be correct, given the definitions of the procedure *PARTITION* and the adjective *partitioned* below, by appealing to a proposition which is similar to the one above for program 3.1.

```
to       PARTITION SORT a sequence consisting-of 1 integer:
         RESULT IS the integer
end
to       PARTITION SORT a sequence consisting-of more-than 1 integer:
         PARTITION it.
         RESULT IS
              result of partition-sort-ing the
                  (left-partition, right-partition)
                   of the partition-result
end
```

Program 3.2 Another DOPL partition-sorting procedure

In this program, two specifications of *PARTITION SORT* are given, one for parameter sequences which consist of only one integer, and one for other parameter sequences. In a DOPL, operand descriptions can be used to specify the formal and actual parameters of a procedure. When a procedure is called, a case analysis on the actual parameters is performed to match them up to an appropriate procedure specification.

The pronoun it in

PARTITION it

is used to refer back to the previous operand description, which in this case is the parameter sequence. The *PARTITION* request is thus equivalent to

PARTITION the parameter sequence

Various kinds of pronoun can be included in a DOPL to make operand descriptions shorter, and, if used appropriately, to make them more transparent.

The operand description of the second RESULT IS request is *factored*, so as to shorten it, using the pair of nouns
(*left-partition*, *right-partition*)
It is interpreted by applying
result of *partition-sort*-ing the
and
of the *partition*-result
to both nouns in the pair. The comma in the pair specifies the concatenation of the resulting sequences. The parentheses are used for grouping only.

The effect of the procedure *PARTITION* can be specified as follows:

```
to       PARTITION a sequence consisting-of more-than 1 integer:
         RESULT IS a partitioned permutation of it
end
```

where the adjective *partitioned* is defined as

ADJECTIVE	*partitioned*
AS IN	*partitioned* integral sequence
IS	sequence comprising non null sequence said-to-be the *left-partition*, non null sequence said-to-be the *right-partition*
SUCH THAT	each integer of the *left-partition* is - <= each integer of the *right-partition*

There may be many *partitioned permutations* of a given sequence, and for any one of these there may be many possible *left-partitions*. The description

the *left- partition* of the *partition*-result

refers to whichever *left-partition* results from whichever method is used to check for *partitioned*-ness.

Although the obvious interpretation of the above procedure would involve generating permutations of the parameter sequence, there are other methods of producing a *partitioned permutation* of a sequence. For example, the partitioning process involved in Quicksort (Hoare, 1961, 1962; Foley and Hoare 1971) an IOPL version of which is shown in program 3.3, will produce a *partitioned permutation*.

```
procedure Quicksort(var A : integerarray;
                        m, n : integer);
{To sort the components of A between the m'th and n'th}
var  r, i, j : integer;
begin if m < n then
     begin {partition A between m'th and n'th components}
     r := A[(m+n) div 2]; i := m; j := n;
     while i <= j do
          begin while A[i] < r do i := i+1;
               while r < A [j] do j := j-1;
               if i <= j then begin
                              A[i] :=: A[j];
                              i := i+1;  j := j-1
                              end
          end;
     Quicksort (A,m,j);
     Quicksort (A,i,n)
     end
end;
```

Program 3.3 An IOPL Quicksorting procedure (from Alagic and Arbib [1978])

A specification of partitioning which is a little closer to that used in Quicksort is:

```
to        PARTITION a sequence consisting-of more-than 1 integer:
          CHOOSE    a parameter integer.
          RESULT IS a partitioned permutation
                                 of the parameter sequence
          SUCH THAT each integer of the left-partition
                    is - <= the choose-and
                and each integer of the right-partition
                    is - >= the choose-and
end
```

One of the main reasons for interest in Quicksort is that it is a very efficient sorting algorithm. Obviously, the DOPL procedures in programs 3.1 and 3.2, which are related to Quicksort in a certain sense, are far less efficient than program 3.3. However, it is less obvious that Quicksort actually sorts. In the last section of the paper, a combined DOPL/IOPL programming system is proposed. In such a system, it would be possible to express an algorithm in its gross, essential terms using a DOPL, and to transform this to an efficient IOPL version. The advantage of such a system, over an IOPL-only one, would be that, with a DOPL version which could be judged to be correct, if correctness-preserving transformations are used, the final optimised IOPL version would be known to be correct. At each stage of the transformation, proof of correctness would have a higher level, correct version to appeal to.

4. AN INTERPRETER FOR A SIMPLE IOPL

The following is an interpreter for a simple IOPL whose programs are sequences of assignment, read, write, while, if, case and compound statements. Only simple integer variables are used, and the only operator is +.

The interpreter does not need to specify input or parsing of the source program. It is not necessary to use data structures to store the source statements or variable values.

```
program         interpreter:
NOUN            identifier
IS              sequence <> 'while' or 'do' or 'if' or 'then'
                           or 'else' or 'case' or 'of'
                           or 'begin' or 'end' or 'read'
                           or 'write'
                comprising several alphabetic character .
NOUN            value
AS IN           value of integer
IS              the integer .
NOUN            value
AS IN           value of identifier
IS              last value assign-ed-to the identifier .
NOUN            term
IS              identifier or non-negative integer.
NOTE            the value of a term is well defined.
NOUN            expression
IS              several term separated-by '+'.
NOUN            value
AS IN           value of expression
IS              sum of value of each term of the expression.
NOUN            relational-operator
IS              '<' or '<=' or '>' or '>=' or '=' or '<>'.
NOUN            Boolean-expression
IS              expression, relational-operator, expression.
ADJECTIVE       true
AS IN           true Boolean-expression
IN CASE         Boolean-expression contains '<'
IS              Boolean-expression
                     containing
                          first expression having value <
                               value of second expression of the
                                          Boolean-expression

{and similar cases for the other relational operators}.

NOUN            assignment-statement
IS              identifier, ':=', expression.
NOUN            statement
IS              assignment-statement  or
                while-statement       or
                if-statement          or
                case-statement        or
                compound-statement    or
                read-statement        or
                write-statement.
NOUN            while-statement
IS              'while', Boolean-expression, 'do',statement.
NOUN            else-part
IS              'else', statement.
NOUN            if-statement
IS              'if', Boolean-expression,
                     'then', statement
                      optionally followed-by else-part.
```

```
NOUN           case-specification
IS             several distinct integer separated-by ',',
               ':', statement.
NOUN           case-statement
IS             'case', expression, 'of',
               several case-specification
                         not containing integer
                             contained-in any preceding
                                     case-specification
                                     of the case-statement
                         and separated-by ';',
               'end'.
NOUN           compound-statement
IS             'begin',
               several statement separated-by ';',
               'end'.
NOUN           read-statement
IS             'read', '(', several identifier separated-by',',')'.
NOUN           write-statement
IS             'write', '(', several identifier separated-by ',',')'.
NOUN           IOPL-program
IS             compound-statement.
DATA IS        IOPL-program, several integer.
EXECUTE        the IOPL-program
to EXECUTE     a compound-statement:
   EXECUTE     each statement
end
to EXECUTE     an assignment-statement:
   ASSIGN      value of expression
   TO          identifier
end
to EXECUTE     a while-statement containing true Boolean-expression:
   EXECUTE     statement of the while-statement.
   EXECUTE     the while-statement
end
to EXECUTE     a while-statement containing non true Boolean-expression:
   DO NOTHING
end
to EXECUTE     an if-statement containing  true Boolean-expression:
   EXECUTE     statement after 'then'
end
to EXECUTE     an if-statement containing non true Boolean-expression:
   EXECUTE     statement of else-part
end
to EXECUTE     a case-statement:
   EXECUTE     statement
                    of case-specification
                         containing integer = value of expression
end
to EXECUTE     a read-statement:
   TO          each identifier
   ASSIGN      first non assign-ed data integer
end
to EXECUTE     a write-statement:
   PRINT       value of each identifier
end
end.
```

5. EULERIAN CIRCUITS IN GRAPHS

An Eulerian Circuit in a graph is a sequence of arcs such that

(a) each arc of the graph is in the Circuit exactly once,
(b) consecutive arcs in the Circuit end at and begin at the same node,
(c) the last arc in the Circuit ends at the same node at which the first one begins.

Walking around an Eulerian Circuit would involve traversing each arc once, and passing through each node one or more times. Obviously, a graph having an Eulerian Circuit (and no trivial nodes) must be connected.

Given the descriptors *node*, *graph* and *connected-to*, an *Eulerian-Circuit* of a *graph* can be defined in a DOPL (actually as a sequence of nodes, pairs of which represent the arcs) as in program 5.1. It is assumed that there is at most one arc between any two nodes, and that no node is connected to itself. Rather than use the identifiers *a* and *b* in the description after WE HAVE, the descriptions

first such-that-for-and
second such-that-for-and

could be used. Naturally, in a language with many operand-description facilities, a choice can be made in each case whether to use a defined descriptor such as an identifier, or a primitive descriptor, such as the nouns above. It seems simpler in this case to use the identifiers.

```
NOUN                  Eulerian-Circuit
AS IN                 Eulerian-Circuit of graph
IS                    sequence of node of the graph
SUCH THAT             each node of the sequence
                      is-connected-to
                      next node of the sequence
AND SUCH THAT         last node of the sequence
                      is-connected-to
                      first node of the sequence
AND SUCH THAT FOR     any node called a
                          of the graph
                      and any node called b and connected-to a
                              of the graph
WE HAVE               either b is adjacent-to a
                                in the sequence
                      or    b is the last node of
                                the sequence
                            and a is the first node
                                of the sequence
                      or    b is the first node of
                                the sequence
                            and a is the last node
                                of the sequence
AND SUCH THAT         number of node
                            connected-to any node of the graph
                      is-equal-to 2*number of occurrences
                                of the node in the sequence
```

Program 5.1 Definition of an Eulerian Circuit of a graph

The Eulerian Circuits of a given graph can be generated:

CHOOSE any *node* of the *graph*.
PRINT each *Eulerian-Circuit* beginning-with the *choose*-and of the *graph*

Program 5.2 defines a *graph* as it might be presented for input punched on cards:

DATA IS *graph* punched-on cards

The descriptors *node* and *connected-to* are also defined in program 5.2. The descriptor said-to-be precedes a defining occurrence of a new descriptor. A relator is a type of descriptor which can be used in relations.

```
NOUN        node
IS          several alphabetic character.
NOUN        connections
IS          several distinct node separated-by ','.
NOUN        node-information
IS          node said-to-be connected-to
                 each node of following connections
                 and not = any node of following connections,
            ':', connections, ';'.
NOUN        graph
IS          several node-information
                      not containing node
                            = node of
                                  any preceding node-information
SUCH THAT   relator connected-to is symmetric
```

Program 5.2 <u>Definition of a graph</u>

From this definition, the description
node of *graph*
means
node of *node-information* of *graph*
and can be so interpreted by an implementation. The semantics of operand-description interpretation can be such as to allow the use of short descriptions which can be automatically extended according to the defined structure of sequences.

The Eulerian Circuits of a data graph can be printed using the above *CHOOSE* and PRINT requests. A copy of the *graph* itself can be printed as follows

PRINT the data *graph*

The question of whether or not a given connected graph has an Eulerian Circuit can be resolved without actually generating such a Circuit, by using the following theorem:

THEOREM (Euler)

A connected multi-graph has an Eulerian Circuit if and only if each node is connected to an even number of other nodes.

PROOF

Only if: An Eulerian Circuit, for each visit to a node, must enter and leave the node on different arcs.
If: Proceed by induction on the size of the graph.

The result is true for a graph with one arc and one node. Suppose it to be true for a connected graph with up to n arcs, and consider a graph with n+1 arcs. Choose any node of the graph, and any two nodes connected to the chosen one. Remove a connection from these two nodes to the chosen one, and insert a connection between the two nodes which bypasses the chosen one. This will result in a graph with either one or two components, but with one fewer arc. By the induction hypothesis, there is an Eulerian Circuit for each of these components. An Eulerian Circuit for the original graph can be made from these by replacing the inserted arc by the two removed ones, and then concatenating the two Circuits.

* * *

Assuming the data graph to be connected (an adjective *connected*, to be applied to graphs, can be defined in terms of the existence of *paths* between any two nodes - a path is a sequence of arcs with certain properties, and can be defined in a similar way to an Eulerian Circuit, which is a path with special properties), the following request can be used to decide whether a data graph has an Eulerian Circuit:

```
IF the data graph does-not-contain
        node connected-to an odd number of node
PRINT "This graph has an Eulerian Circuit"
```

This must be correct because of the above theorem.

6. DOPL SYNTAX AND SEMANTICS

A DOPL program is a sequence of requests separated by '.', and possibly followed by procedure definitions:

```
NOUN    DOPL-program
IS      'program', name, ':',
        several request separated-by '.'
        optionally followed-by several procedure,'end', '.'.
```

The program is executed by using each request in turn:

```
to      EXECUTE a DOPL-program:
        EXECUTE each request
end
```

A request is several *requestor/operand-description* pairs, where a *requestor* is an *operator*, a *preposition* or a term such as NOUN, ADJECTIVE, IS, AS IN, SUCH THAT, UNTIL:

NOUN *request*
IS several (*requestor*, *operand-description*)

An operational request is executed by applying the operators to all the operands of all the operand descriptions. For example, for a unary operator:

<u>to</u> *EXECUTE* *request* comprising (*operator*, *operand-description*):
APPLY the *operator*
TO each *operand* of the *operand-description*
<u>end</u>

APPLY would be defined for each primitive and user-defined operator (in the latter case, by executing the requests of the appropriate procedure definition), but not for user-introduced, non user-defined operators such as *REMOVE* (section 2) or *ASSIGN* (section 4). The semantics of these would be specified in terms of the associated descriptors. For example, the semantics of *remove*-ed, as in

remove-ed <description of an *operand*>

is

apply-ed *REMOVE* *to* the *operand*

and the semantics of *assign*-ed as in

assign-ed <description of an *operand*>
to <description of another *operand*>

is

apply-ed (*ASSIGN*, *TO*)
to (the *operand*, the other *operand*)

The basic structure of an operand description is

NOUN *operand-description*
IS several *adjective-type-descriptor*,
reference, *post-description*
optionally followed-by
('of', *nested-operand-description*)

where a *reference* is a description of an actual object, and may be a noun, a pronoun or an identifier. The adjectives either specify the generation of all the objects specified by the reference, or possibly, together with the post-description (an example of which is "<= the data integer" from section 2), specify the required properties of objects. In addition to the above structure, operand descriptions can be combined using descriptors such as either, or, and, (,) and others.

The sequence of all the operands of an operand description used in an operational request is the sequence comprising each referenced object (with the properties stated in adjectives and post-descriptions) of each object specified by the nested operand description.

7. PROPOSAL FOR A DOPL-BASED SYSTEM

A language containing a spectrum of DOPL and IOPL features would make an ideal programming system. Initially, programs could be written using the DOPL, possibly in a highly non-procedural fashion, as for example with *SORT* in section 3. Provided these were not too disproportionately inefficient (as with sorting 100 integers using a strict interpretation of *SORT*), they could be executed and used whilst a programmer and/or the implementation were refining the DOPL version to a more efficient IOPL one.

In the case of a well-defined, self-contained problem, such as sorting or the generation of primes or circuits in graphs, the DOPL version of an algorithm could be *judged* to be correct by appealing to what might be called the *factual basis of the algorithm*, this being the collection of proven properties of the objects involved in the algorithm. For example, for problems involving primes, the *factual basis* might include the definition of what is meant by a prime and propositions about the existence of factors of non primes. For problems involving circuits in graphs, the *factual basis* might include the theorem in section 5. In the case of more complex problems, such as large data-processing applications or the design of a new programming language, the DOPL version might be developed and agreed to by a committee of users and analysts, as the correct initial specification for a required system. In either case, an efficient implementation of the DOPL version could then be obtained using various automatic or manual correctness-preserving transformations.

The design of a DOPL presents a host of challenging problems. Many of these remain to be resolved. Nevertheless, the notion of *operand description*, and the incorporation of a variety of description facilities in a programming language, seem to hold the promise of a superior, general-purpose language for the future.

REFERENCES

ALAGIC, S., ARBIB, M.A., (1978): "The Design of Well-Structured and Correct Programs", Springer-Verlag, New York.

ASTRAHAN, M.M., CHAMBERLAIN, D.D., (1975): "Implementation of a Structured English Query Language", Comm. ACM, Vol. 18, No. 10, pp 580-588.

BARRON, D.W., (1977): "An Introduction to the Study of Programming Languages", CUP, Cambridge.

BURGER, J.F., LEAL, A., SHOSHANI, A., (1975) "A Semantic-Based Natural-Language Interface for Data Management Systems", Proceedings of International Conference on Systems Sciences, Hawaii, pp 218-220.

CHAMBERLAIN, D.D., BOYCE, R.F., (1974): "SEQUEL: A structured English query language", Proceedings of ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Michigan, pp 249-264.

FELDMAN, J.A., ROVNER, P.D., (1969): "An Algol-Based Associative Language", Comm. ACM, Vol. 12, No. 8, pp 439-449.

FINDLER, N.V., (1969): "Design Features of and Programming Experience with an Associative Memory, Parallel Processing Language, AMPPL-11", Proceedings of Fourth Australian Computer Conference, Adelaide, pp 321-325.

FOLEY, M., HOARE, C.A.R., (1971): "Proof of a recursive program: Quicksort", Computer Journal, Vol. 14, No. 4, pp 391-395.

HEBDITCH, D.L., (1973) :"Terminal languages for data base access", Data Base Management, Infotech State of the Art Report 15, pp 521-541.

HERRIOT, R.G., (1977): "Towards the Ideal Programming Language", SIGPLAN Notices, Vol. 12, No. 3, pp 56-62.

HOARE, C.A.R., (1961): "Algorithm 63, Partition", "Algorithm 64, Quicksort", Comm. ACM, Vol. 4, No. 7, p 321.

HOARE, C.A.R., (1962) : "Quicksort", Computer Journal, Vol. 5, No. 1, pp 10-15.

HOARE, C.A.R., (1972): "Proof of a structured program: The Sieve of Eratosthenes", Computer Journal, Vol. 15, No. 4, pp 321-325.

HOUSEL, B.C., SHU, N.C., (1976): "A High-level Data Manipulation Language for Hierarchical Data Structures", Proceedings of a Conference on DATA: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, No. 2, pp 155-168.

LEE, G., (1978): "Some design features of a Description Oriented Programming Language", Proceedings of the Eighth Australian Computer Conference, Canberra, pp 938-946.

MARTIN, J., (1976): "Principles of Data-Base Management", Prentice-Hall, Englewood Cliffs, N.J.

NYLIN, Jr., W.C., HARVILL, J.B. (1976): "Multiple Tense Computer Programming", SIGPLAN Notices, Vol. 11, No. 12, pp 74-93.

POTTS, G.W., (1970): "Natural language inquiry to an open-ended data library", Proceedings of the SJCC, Atlantic City, N.J., pp 333-342.