EXPECTATIONS OF A HIGH-LEVEL PROGRAMMING FACILITY: SOME EXAMPLES *

by

David A. Smith

Technical Report #121

Department of Information and Computer Science
University of California, Irvine

April 3, 1978

*This work was supported by the National Science Foundation under grant #MCS75-13875A01.

Contents

Note: In this document a number of liberties are taken with the type font in order to represent mathematical symbols and other special characters.

{ }	denote braces: { }
!	denotes vertical bar: |
in, notin	denote set membership: $\in, \notin$
or, not	boolean/set operations: $\cup, \sim$
gt, ge, lt, le	comparison operators: $>, \geq, <, \leq$
up, down	up/down arrow: $\uparrow, \downarrow$
lambda	lambda: λ
all	"for all": $\forall$

Expectations of a High-Level Programming Facility: Some Examples

Introduction

In order to make substantial progress in the area of automatic software development, it is helpful to limber up one's imagination in order to see where an automated programming assistant may be helpful. This paper takes such an approach by considering a number of programming problems and showing how solutions to these problems can be given, suppressing a comparatively large amount of detail, and assuming a a relatively large amount of knowledge and initiative in the programming assistant. Of the many considerations in automated programming assistance, the greatest emphasis in this paper is on enhancing the expressive power of the human programmer. While practical considerations of implementation must be addressed sooner or later, steps must first be taken both to enrich and to unify the medium of program expression. It is the purpose here to propose language features and philosophy so that more programs which are easy to think about will also be easy to express.

Although no attempt is made to give a formal language definition at this point, the features proposed here can be attributed to a language which we shall call MPD, Mathematical Programming Domain. It may well be that the programmer will have more than one channel or kind of communication with the programming assistant, MPD being used for the high-level description of programs, while other kinds of direction will also be needed regarding new semantic domains, or specific program implementations.

The Example Programs

Four example problems were selected which illustrate the need for expressive power in a programming language. They can each be understood with ease by a reasonably competent programmer and should therefore find easy expression in a suitable high-level language. The problem definitions below are in English, and are intended for human consumption only. The problems themselves are stated somewhat incompletely, perhaps as the problem might first be stated to the human programmer. Having proposed language features (described below), we shall see how each of these problems can be expressed without resort to extensive detail.

Problem A: Cross-Reference Program. The input to this program is a text file divided into lines. The program prints this file with the lines numbered. The program identifies the set of "identifiers" or "words" which occur in the text and prints a summary, in alphabetical order, showing for each identifier the line numbers on which it occurs in the text.

Problem B: Minimal-Cost Spanning Tree. Given an undirected graph where a nonnegative cost function is defined on the edges of the graph, and given that the graph is connected, find a subgraph which spans all the vertices of the original graph such that the sum of the costs of its edges is minimal.

Problem C: Dijkstra's Shortest Path Algorithm. Given a directed graph where a nonnegative cost function is defined on the edges of the graph, and also given a distinguished node V0, find for each node X whether there is a path from V0 to X, and if so find the path with the minimum cost.

Problem D: Change Making Problem. Given a repertoire of coin values

(e.g., pennies = 1¢, nickles = 5¢, etc.) and given a desired amount A, count all the unique ways of making change for A using combinations of any number of each kind of coin.

Philosophy of the Programming Assistant

To an experienced programmer each of the above problems is fairly clear in its meaning, although the "words" of problem A are not well defined. In spite of some incidental ambiguities and the absence of any precise definitions, the basic intent of each problem is understood. This is because education or experience has given the programmer familiarity with the concepts of "identifiers", sorting, following paths in graphs, generating permutations and combinations, and so on. By analogy, we would expect a "well educated" programming assistant to be conversant in these and other common terms so that program steps involving their use do not require excessive elaboration. A major provision (and a major problem) in building a high-level programming assistant will be to permit regular and orderly augmentation of the "education" of the assistant.

As an aid in conceptualizing the data structures in a program it was occasionally useful to write an expression which "typically" represented the form of the item, but which did not require the precision of a formal declaration. For example, to show that an item x is a sequence of ordered pairs one can write "x: ((a1,a2), (b1,b2), ...)". These may be termed "expository declarations" and are distinguished from formal declarations since they do not have sufficient precision to serve as formal definitions. Still, they are desirable since they are easier to write. In the present examples all data declarations are written in this manner, although there may be equally useful notations which are completely rigorous.

It should be clear that when we attempt to limit the information in a

channel of communication (i.e., from the programmer to the assistant) that some capability must be lost. To a certain extent we can recover lost precision of expression by relying on mutually understood ellipses in frequently occurring cases. The philosophy of MPD is to go further, however, and to expect the programmer to write programs which are compact in expression and correct in a literal rendering of the algorithms, but which may be inefficient if implemented in the most obvious or direct translation from MPD statements. Transforming inefficient programs into efficient programs is an important area of research, and in general the problem is a very difficult one. It is hoped that once a "basic" version of MPD has been defined that a class of first-level transformations can be defined which are in fact a repertoire of implementations which are keyed to certain subsets of operations which have been used, or to certain patterns of reference. It may be said that the algorithms described in the examples of this paper are somewhat "optimized" versions of very abstract programs which would be horribly inefficient. It is not the object here to place an impossible re-programming burden on the transformation and optimization phase of the programming assistant but rather to increase the expressive power available to the programmer and to provide a certain level of "coverage" for him as he takes expressive shortcuts.

In general not too much consideration has been given to possibilities for syntactic ambiguities in MPD. In the process of language definition it is always possible to require keyword-type lexemes here and there to keep things straight, but while these make the job of the parser easier, they place "unnatural" constraints on the programmer and hence are an impediment to expression. It is expected that the programmer will not get into trouble very often, and when he does he can be informed of the

of the ambiguity and be asked to clarify it by suitable parenthesization or other rewriting. It is the intent of the design of MPD always to allow ellipsis of some operators when the meaning is "clear". The goal here is to make the interpretation of MPD expressions as flexible as possible. It is also intended that explicit, fully qualified forms of expression be available when necessary. MPD should also do as much type checking as possible within the declarations available in order to help avoid obvious errors. It should also be possible to check different references to the same data for consistency with respect to data types.

Features of the Language, MPD

We assume the availability of the primitive data types, operations, and statements shown in Table 1. Note that the scoping of nested statements in MPD is determined by textual indentation. The language elements in this table are not intended to be definitive, but merely to provide a vocabulary for the remaining discussion in this paper. The remainder of this section describes the basic data structuring techniques of MPD sets and sequences, after which some meta-operators are introduced. The higher-level concepts of relations and graphs are then presented using sets and sequences. There remains more work to be done to unify the ideas of input/output, co-routines, and formal grammars.

A set is an unordered collection of values. A set with a fixed number of elements is an unordered n-tuple. Membership in sets is determined by value. If unique objects are to be created and moved around where their presence in one place or another is a matter of concern, then some attribute must be attached to each item and assigned a unique value as the item is created. For graphs a particular scalar value can be designated as the "identity" of a given node. The elements of a set need not all be of the

Table 1.
Basic Data Types, Operations, and Statements

Data types

real: +,-,*,/,min,max,!x! (absolute value)
integer: +,-,*,/,mod,min,max,!x! (absolute value)
boolean: &,or,not
character

Comparison operators (yield boolean values)

=,≠,gt,ge,lt,le

Statements

```
v := expr

while expr do
   ...

if expr then ...
   ...
else ...
   ...
```

Abbreviations

v :+ expr	means	v := v + expr
v :- expr	means	v := v - expr
v :or expr	means	v := v or expr
v :& expr	means	v := v & expr
v :not	means	v := not v

of the same type, although this is probably the typical case. Table 2 shows the operations that can be used with sets.

A sequence is an ordered collection of values. A sequence with a fixed number of elements is an n-tuple. The elements of a sequence need not all be of the same type, although this will happen frequently. Simple Pascal records are nonhomogeneous n-tuples. A homogeneous n-tuple is like an array, but a more general concept of arrays will be presented below. Table 3 shows the operations that can be used with sequences.

It seems convenient to make no distinction between a single item and a sequence which is a 1-tuple. This makes definitions pertaining to relations (below) easier to state, and makes parentheses unambiguous when they are used for arithmetic grouping in expressions. A singleton set, however, is distinguished from the single item by itself.

We have seen that the meta-operator ":" is used with monadic and dyadic operators to denote various abbreviations for the assignment statement. (The assignment statement itself ":=" is an exception.) These can be generalized as follows:

V :(dyadic op) expr	means V := V (dyadic op) expr
V :(monadic op)	means V := (monadic op) V

(There is some syntactic ambiguity in the precedence of the "(dyadic op)" and "(monadic op)" in the right hand statements, but this is a detail.)

Another meta-operator is necessary to indicate that an element operator is to be "distributed" inside a compound data structure. This is the character "'" and has a monadic as well as dyadic use:

A '(dyadic op) B	means that A and B are isomorphic structures and that the result of this expression is a similar structure formed by applying the (dyadic op) to corresponding pairs of elements.
'(monadic op) S	denotes a value isomorphic to S, but with (monadic op) applied to each element of S.

Table 2.
Notation for Sets

declarations:	S:{t_i} or S:{t1,t2, ... ,tn}	
predicates:	S1=S2, S1≠S2, S1 lt S2, S1 le S2, S1 gt S2, S1 ge S2 x in S, x notin S	
constant:	∅ (the null set)	
expressions:	S1 or S2, S1 & S2, S1-S2 {x1,x2, ... ,xn} = the set formed from the enumerated elements !S! = the number of elements in S {all t} = the set of all values of type t	
statements:	V := S all x in S do	(the indicated statements, containing "x" as a free variable, are executed once for each element x in S)
expression abbreviations:	S + x	means S or {x}
	S -- x	means S - {x}
statement abbreviations:	V :or S	means V := V or S
	V :& S	means V := V & S
	V :- S	means V := V - S
	V :+ x	means V := V + x
	V :-- x	means V := V -- x

S: a set expression
V: a set variable
x: an element expression
t: a type

Table 3.
Notation for Sequences

declarations:	$S:(t_1)$ or S:(t1,t2, ... ,tn)	
predicates:	S1=S2, S1≠S2	
constant:	∅ (the sequence with no elements)	
expressions:	S1 ! S2	= concatenation of S1 and S2
	!S!	= the length of S
	x1..x2	= the sequence of integer values x1..xn
	S(1), S(2), ...	= the elements of S
	(x1,x2, ... ,xn)	= the sequence of enumerated elements
	'message'	= a sequence of characters
statements:	V := S	
	all x in S do ...	(the indicated statements, containing
	...	x as a free variable, are executed
		once for each element x in S, in order)
expression abbreviations:	S + x	means S ! (x)
	-- S	means (S(2), ... ,S(!S!))
statement abbreviations:	V :! S	means V := V ! S
	V :+ x	means V := V + x
	V :--	means V := -- V

S: a sequence expression
V: a sequence variable
x: an element expression
t: a type

This meta-operator can also be used with "(" and "{" in various contexts. The distribution meta-operator makes explicit the process available in APL which distributes a scalar operator into a vector or matrix. This can be elided in MPD when the application of the operator to the structure as a whole is not defined. Explicit notation is necessary in certain cases to cause a structure operator to operate on lower level elements (which are structures themselves). For example, we can give an alternative definition for the "tail" operator on sequences, as follows:

--S = S'(2..!S!)

Distributing a dyadic operator into two sets is not defined, since there is no structure in the sets to say what elements should be paired.

The reduction meta-operator "/" is stolen directly from APL, and can operate on sets as sell as sequences. "(dyadic op)/ S" can be used whenever "(dyadic op)" is an associative operator which returns a value of the same type as its arguments. The meaning of this statement is the same as if all of the elements had been written on a line with "(dyadic op)" between consecutive elements in the list. For example,

max/ S = the maximum of all elements in S, and

+/ S = the sum of all elements in S.

A general control statement is proposed for MPD which implies exhaustive search, which (upon guided optimization, if necessary) may be reduced to more efficient code in special cases:

find x in S st P(x)

find x in S st min F(x) or min x in S: F(x)

find x in S st max F(x) or max x in S: F(x)

Each of these constructs can be used to designate a value (in an enclosing expression) or to assign a value to "x" for use by subsequent statements. The first statement finds an instance of x which is an element of S and

which satisfies some predicate, P(x). The value returned is $\emptyset$ if there is no such x. The second and third statements find a value of x such that some function of x, F(x), is minimized or maximized. If the result is ambiguous -- that is, there is more than one appropriate x in S -- then the set of values can be designated as follows:

find {x in S} st P(x) or find y={x in S} st P(x)

find {x in S} st min F(x), or find y={x in S} st min F(x)

find {x in S} st max F(x), or find y={x in S} st max F(x).

A relation is defined to be a set of n-tuples. The n-tuples must all be of the same length, and for each i, the i'th component of all n-tuples must be of the same data type. If for a given i there is at most one n-tuple with a given value in component i, then the relation is said to be functional on component i. In addition to the set notation for dealing with relations as sets of n-tuples, we introduce some additional notation which is familiar and handy for programmers.

R(x1, ... ,xn)	means the expression "(x1, ... ,xn) in R"
R(x)	requires that R be functional on component 1. The meaning is as follows: find y in R st y(1)=x if y=$\emptyset$ then the result = $\emptyset$ else the result = y'(2..!y!)
R(x) := expr	requires that R be functional on component 1. The meaning is as follows: find y in R st y(1)=x if y≠$\emptyset$ then R :-- y. R :+ x!expr
R{x}	This doesn't require functionality. The meaning is as follows: find z={y in R} st y(1)=x the result = z'(2..n)
R{x} := expr	This doesn't require functionality. The meaning is as follows (expr is a set of n-1'tuples): R :- x'!R{x} R :or x'!expr

R(up)	The sequence of all elements of R (n-tuples) with their first elements sorted in increasing order.
R(2up,down)	The sequence of elements of R (n-tuples) sorted in decreasing order on the second element. Within groups having the same second element, the first elements are sorted in increasing order.

The R(x) operations provide for generalized arrays or functions. Note that if R consists of ordered pairs, then R(x1) is a 1-tuple, (x2), which is just the single item x2. The component being "indexed on" can be more general than integers; using character strings, for example, one has an instant dictionary notation. These notations can be extended by using other components than the first. For example if R has two components and is functional on both of them, then the inverse of R(x) can be written R(,x). The R{x} operations can be used for operating on a directed graph -- for example, collecting all the sons of a node in a tree. It is convenient to use special notation when declaring a relation that is used as a function. For example, the following are equivalent:

R: {$(t1,t2)_i$} and R: t1 -> t2.

The facilities described above can be used to give MPD the mathematical definition of a graph. A directed graph is a pair (V,E) where V is a set of vertices (possibly whose only property is that they are distinguishable from each other) and E is a set of ordered pairs of elements of V. We can tag the vertices with information using functions, and we can do the same with edges. Notice that in implementation the identity of vertices can be established by pointers, or by integer values. Edges can be pairs of integers, or may not even be used explicitly. Relation notation can be used for traversing a graph. A path can be defined as a sequence of vertices, or as a sequence of edges.

The Examples, Rendered in Mud

This section discusses the four example programs, as they can be solved using the language features of MPD. They show that the concepts presented in the preceding sections are useful in defining the required data manipulations.

Table 4 shows an implementation of the Cross-Reference Program. Line 1 is a prototype statement showing how the program "xref" is to be called. This statement shows that "xref" is used as a function of one argument, "input" (a sequence of character strings), and yields a value, "output" (also a sequence of character strings). Line 2 in its literal interpretation takes the entire input sequence and forms a sequence of ordered pairs, assigning a line number to each input line. Line 3 causes the program to sequence through these numbered lines. Line 4 causes each line to be printed with its line number. Line 5 uses a function, "translate" (undefined here), which takes the line and produces a sequence of tokens (themselves character strings). It is intended that provision be made in MPD for defining such lexical mapping using regular expressions or perhaps, more generally, with formal grammars. Line 6 builds the dictionary; each dictionary entry is a sequence of line numbers. Line 7 causes the dictionary to be rendered as a sequence of ordered pairs, sorted on the tokens in increasing order. The remaining lines of the program print each token along with the list of references to that token. Provision is made for the list to occupy more than one output line. It would seem from the comparative awkwardness of the code that generates the output that something could be done to express the same function more economically -- perhaps also along the lines of regular expressions or a formal grammar.

Table 5 shows an implementation of the Minimal-Cost Spanning Tree program. Line 1 is a prototype statement declaring the output value T

Table 4.
Cross-Reference Program

```
1    define output: ( (char_i)_j )  :=  xref( input: ( (char_i)_j ))

2    temp := '( (1..!input!), input)
3    all (linenumber,line) in temp do
4       output :+ char 5(linenumber) ! '⅋' ! line
5       all token in translate(line) do
6          dictionary(token) :+ linenumber

7    all (id,linelist) in dictionary(up) do
8       line := id
9       all n in linelist do
10         if !line!+6 gt 132 then
11            output :+ line;  line := '⅋⅋⅋⅋⅋'
12         line :! '⅋' ! char 5(n)
13      output :+ line
```

Table 5.
Minimal-Cost Spanning Tree Program

```
1    define T:{ {v1,v2}_i }  :=  minspan( V:{v_i}, E:{ {v1,v2}_i }, C: E → real)

2    T := ∅
3    P := '{V}

4    temp := C(,up)
5    all ( {v1,v2}, c) in temp do
6       find p1 in P st v1 in p1
7       find p2 in P st v2 in p2
8       if p1 ≠ p2 then
9          T :+ {v1,v2}
10         P := P -- p1 -- p2 + (p1 or p2)
```

(a set of edges) and the input values V (a set of nodes), E (a set of edges), and C (a function from E into the real numbers). Line 2 initializes T to the empty set and line 3 initializes P (a partition of V) to the finest partition of V, all singletons. Line 4 sorts the edges on increasing cost. Line 5 sequences through these edges in increasing order, and lines 6 and 7 find the partitions containing the endpoints of each edge. If these partitions are distinct, then they are coalesced and the edge under consideration is added to T. Although the MPD formulation of this program suppresses a lot of significant detail from the point of view of execution efficiency -- in particular the "find" operations on lines 6 and 7 -- it should be noted that this program is a complete formal solution to the problem.

Table 6 shows an implementation of Dijkstra's Shortest Path Algorithm. This program assumes the availability of a real value "infinity", ∞, which when added to any finite real number yields infinity; comparisons will always show "infinity" to be greater than any finite real number. Lines 1-3 define the function "cost" of a path in the graph. Note in line 2 the use of in-line lambda notation to define a function to be distributed across the sequence y. This causes an undefined edge in the path to make the path cost infinite. Lines 6 and 7 initialize the data structure D which throughout the algorithm contains the shortest known path from V0 to each vertex. The set N, initially empty from line 5, identifies the set of vertices whose paths in D are known to be minimal. Line 9 finds the "closest" vertex not in N, and the path to it which is recorded in D is taken to be minimal. Hence this vertex is added to N in line 10 and all of D is updated to reflect possible shortcuts through this vertex. When N has grown to include all of V (line 8), the algorithm terminates. This implementation of Dijkstra's shortest path algorithm is comparable in complexity to the Algol/English description found in Aho, Hopcroft, and

Table 6.
Dijkstra's Shortest Path Algorithm

```
1    define x:real  :=  cost( y: ( (v1,v2)_i ) )
2       temp := lambda x (if x in E then c(x) else ∞ ) '(y)
3       x := +/ temp

4    define D:S→(E_i)  := dijk( V:{v_i}, E:{ (v1,v2)_i }, V0 in V, C:E→real)

5    N := ∅
6    all v in V--V0 do
7       if (V0,v) in E then D(v) := (V0,v)

8    while N ≠ V do
9       find w in V-N st min cost(D(w))
10      N :+ w
11      all v in V-N do
12         D(v) := min x in ( D(v), D(w)+(w,v)) : cost(x)
```

Ullman (p. 208). The three MPD lines defining "cost" do not appear in the Algol/English version, since this function is defined in the text description of the program. The MPD program has the added power of keeping track of the actual paths, and not just their costs, and has the advantage of being completely precise without the use of English.

Table 7 shows an implementation of the Change Making Problem. Lines 1 and 2 define a function which calculates the value of a sequence of coin values. The main program itself, "count", takes as input a sequence of coin denominations and a total desired amount. For output it produces a count of the number of ways of making change, and a list of coin combinations. The program calls a recursive procedure, "makechange", which builds a list of trial combinations. This procedure generates and tests all coin combinations whose total value is less than or equal to A. This illustrates quite well the convenience of MPD notation for building and sequence x in successive recursive calls, and also demonstrates the flexibility of building the output data structure "combinations" as a sequence which can then be handled in whatever way is desired.

Concluding Remarks

The programming examples considered here have demonstrated the usefulness of well-defined higher-level operators based on familiar notions of mathematics and programming. Programs written using these language concepts are brief, fairly easily understood, and are semantically precise. On the other hand, by suppressing programming details the implementation implied most literally by a MPD program may be quite inefficient and quite remote from the program that would be written by a programmer with more leisure in which to write. It is the intended thrust of this research to continue by investigating techniques for making MPD programs run efficiently, and for augmenting in a

Table 7.
Change Making Problem

```
1    define x: integer  :=  value( x: (integer_i))
2       x := +/ x'*denominations(1..!x!)

3    define nways: integer, combinations: ( (integer_i)_j)  :=
        count( denominations: (integer_i), A: integer)

4    nways := 0;  combinations := ∅
5    makechange(∅)

6    procedure makechange( x: (integer_i))
7       if !x! = !denominations! then
8          if value(x) = A then
9             combinations :+ x;  nways :+ 1
10         else "discard combination"
11      else x :+ 0
12         while value(x) le A do
13            makechange(x);  x(!x!) :+ 1
```

uniform way the knowledge domains and "bag of implementation tricks" of the programming assistant.

Some areas of interest also remain as refinements to the linguistic attributes of MPD considered in this paper. These are in the areas of input/output specification, the use of formal grammars, data declarations, and in the area of MPD disambiguation and more flexible language processing.

STUDENTS ASSIGNED TO SURGERY

To help us evaluate the teaching ability of the lectures in Surgery in a more objective and critical fashion, we would appreciate if you could take time to fill out this evaluation. These evaluations are an attempt to help the individual lecturers pinpoint their deficient areas and apply corrective procedures. Please write in the number which indicates the degree you feel descriptive of the instructor.

1 is very poor; 2-3 is average; 4 is above average and 5 is superior.

NAME OF LECTURER _____

Weight		1 through 5
(5)	Preparation of lecture	_____
(5)	Presentation of lecture	_____
(5)	Organization of lecture	_____
(5)	Relevance of material (to esoteric, to basic, practical, etc.)	_____
(5)	Involvement of group (questions, discussion)	_____
(3)	Use of blackboard or visual aids	_____
(3)	Reference to other sources, periodicals regarding subject	_____

Additional comments or suggestions....

Please return to: Surgery Administrative Office, Room 207, Bldg 53.