CHARACTERISTICS OF A FUNCTIONAL PROGRAMMING LANGUAGE

by

Chr. Gram* and E. I. Organick

UUCS-80-103
JULY 1980

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
U. S. A.

ABSTRACT

A programming language kernel is presented where an algorithm is a function defined through a functional expression. The only data structure introduced is an object that may be an atom or a sequence of objects. A number of functional forms are defined, with a notation close to ordinary mathematical notation, and their usage is demonstrated through several examples. The language allows a high degree of parallelism in an underlying interpreting machine.

CONTENTS:

This work was supported in part by the Danish Natural Science Research Council and by a Fellowship Grant from Burroughs Corporation.

---

* On leave from Dept. of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark.

(McCarthy 62) J.McCarthy, P.W.Abrahams, D.J.Edwards, T.P.Hart, and M.J.Levin, "LISP 1.5 Programmer's Manual." The M.I.T. Press, Cambridge, MA (1962).

Doc.no UUCS-80-103
Date: 1980-07-01
Author: Chr. Gram
E.I. Organick

CHARACTERISTICS

of

A FUNCTIONAL PROGRAMMING LANGUAGE

1.0 INTRODUCTION.

This report is a preliminary presentation of a Functional Programming Language. It presents the kernel of the language, sufficiently powerful to allow the user to express any sequential algorithm, but it does not define input or output facilities, nor does it indicate the linking to a user's environment such as a file system or a library.

The work is heavily inspired by R.S.Barton and a number of the semantic concepts and notations used are his or emanated through discussions with him and his collaborator, B.J.Clark. Another source has been the paper by J.Backus on functional programming [Backus 78], and some of his notation is also followed. The important contributions of J.McCarthy, K.Iverson, and P.J.Landin on most of the work in the area of applicative programming are also acknowledged.

Backus distinguishes between Functional Programming (FP) systems and Formal Functional Programming (FFP) systems. An FP system is a 'closed' applicative system consisting of a set of primitive functions, a fixed set of functional forms, and a set of basic definitions; its expressive power is determined through the choice of functional forms (i.e., combining rules). In an FFP system new functional forms may be created by use of the so-called metacomposition rule and an Apply-function; this is a very strong facility, but a Pandora's box, that in essence yields unlimited expressive power. We believe that this is not needed, and one purpose of this report is to demonstrate that with a carefully chosen set of primitives and functional forms, an FP system is sufficiently 'rich'. It allows you to develop and design algorithms in a well-structured way and it encourages top-down design, as the examples will show.

2.0 BASICS.

Any algorithm will be written as a functional expression, applying to one object (the argument) and producing one object (the result). When applied to an input object - the given data - it produces an output object - the desired result. Since any function and expression in the language maps one object into one other object, the objects must be able to 'carry' data structures. This is obtained through the following definition of objects : An object is one of

a) an atom, denoting one of the primitive objects under consideration. They comprise at least the logical values {true,false} and (a suitable subset of) N, the integers; depending on the applications wanted, the atoms may also include real and/or complex numbers, character sets, and other sets.
A special atom denoted $ is the nil atom or the 'no value' atom (for all kinds of atoms).

b) 'undefined'. Any function applied to the undefined object yields the result 'undefined'. (The role of 'undefined' is explained more detailed in an accompanying paper [Gram, Organick c].)

c) a finite, ordered sequence <x1,x2,...,xn> whose elements are objects. A sequence may be the empty sequence, denoted <>, and it may contain 'undefined' as well as other sequences among its elements.

Clearly the set of atoms determine the set of objects under consideration. The recursive definition b) allows an object to be a sequence of sequences of ..., thereby allowing representation of any finite data structure.

In [Backus 78] the nil atom $ and the empty sequence <> are considered one and the same object. The distinction between these was suggested to us by Paul Black [Black 80]. It is adopted here because it clarifies the contrasting roles of atoms and sequences, and because some of the basic function definitions can be made slightly more general.

For the time being we consider a function to be applied just once, to one argument object producing one result object. This is not in conflict with an implementation model where every function is repeatedly applied to a stream of input objects, until the stream is exhausted. But it simplifies the description in the following to consider one application at a time.

The syntactical form of an algorithm is

```
<algorithm>    ::=  <functional expression> ! <function def>

<function def> ::=

  <fct.name> { (<parameter list>) } = <functional expression>
  { where
         <function def> ,
         . . .
         <function def>    }

<fct.name>        ::= <identifier>
<parameter list> ::= <param> { , <parameter list> }
<param>           ::= <identifier>
```

where anything enclosed in {} is optional. The possible forms of <functional expression> will be defined below. As is seen, a <function def> may contain definitions of subfunctions, thus allowing algorithms to have a hierarchical structure. The frame containing the definition of a function and subfunctions is called a definition-tree (or a d-tree) because definitions exhibit a tree-like structure, as explained later.

A function definition may be prefixed by a <well-formed-condition>, which is a logical predicate expression, in which case, the function is only defined when this expression evaluates to true when applied to the input object.

In the definition of a functional form or a primitive function a choice must be made as to when it is defined and when it should yield 'undefined'. We have chosen to leave as few cases as possible undefined, i.e., to maximize the domains of functions and functional forms. This makes the language easier to use (more 'user-friendly'), as long as the syntactically correct programs form a reasonable algebraic system where transformation rules may be set up, allowing equivalence proofs and manual or automatic transformation of a program into more 'convenient' or more 'efficient' forms.

Within the same abstract syntax and semantics one may choose different concrete syntactic representations of algorithms. In mathematics there is a tradition to choose terse notations, with one letter names and little or no 'syntactical sugar', whereas the tendency in data processing is to use a more verbose, 'natural English' style notations, together with 'long', mnemotechnic identifiers. It is not clear what is more readable and teachable in general, and the question has to be given careful consideration before finally deciding on a specific language representation and teaching style. In this report we have chosen to use a semi-verbose notation when introducing the concepts, but a number of examples are shown both in that notation and in much more terse, redundancy free style (the two styles being semantically equivalent), to give the reader the possibility to judge for himself.

Also, one may take issue with a number of smaller design choices we have made in this report. They reflect our preferred style of programming at this time but could easily be changed without changing the basic spirit of the language. B.Barton and B.Clark stress the importance of building on pure mathematical ideas and models and not leaning on constructs that are inherited from present programming languages. Yet, to be able to write down explicit example algorithms we have violated these principles to some extent, and more specifically we deviate from the Barton/Clark notation on the following points:

1. We number the elements of a sequence 1,2,... instead of 0,1,...

2. We use [ ] for construction and ( ) for general grouping/delimiting instead of using ( ) in all cases.

3. We use an explicit symbol & for composition instead of letting it be implicit in the juxtaposition of functions.

4. We present two syntax styles which we call verbose and terse styles, respectively, considering them equivalent and equally suitable. In Barton/Clark notation only the terse style is used, this style being more closely related to conventional mathematical notation and more directly manipulable by functional algebra.

5. Our definitions of the primitive functions (head, tail, ...) are more lenient in some special cases (like evaluation of head of an atom).

6. Our suggestion for the binding priorities of the functions and operators is slightly different from the Barton/Clark model.

A preliminary example is given here to show how a very simple function definition appears. The algorithm to solve the linear equation

```
        a x  +  b  =  0
```

may be given as

```
LINEQ(a, b) = if  a = 0  then  [false, $] ;
                               [true , -b/a]
```

where the meaning is: The function LINEQ takes as its argument a sequence of two real numbers the first of which is denoted a, the second b. The algorithm depends on a; if a=0 the result of applying the function is a sequence with two elements <false,$>, and otherwise the result is a 2-element sequence containing true and the value of the solution. In the more terse notation the same function definition would be

```
lineq(a,b)  = ( (F,$) , (T,-b/a) )
                                 a /= 0
```

where the logical predicate subscript expression selects the first or the second pair depending on whether the predicate is false or true.

Most of the functional forms and the primitives are described using the notation

<fct.name or form> : <input object> --> <output object>

meaning: When the function is applied to the <input object> it gives <output object> as result. We write explicitly all the cases of arguments for which the function is well defined as well as some of the 'undefined' cases. In all other cases the output object is the 'undefined' object. Some of the definitions are more 'lenient' than those found in [Backus 78]. Fewer cases are left undefined because it is considered an advantage that the domain of each function is as large as possible.

The use of parameters and subfunctions in function definitions is introduced in section 4.1 and discussed more thoroughly in section 7.2.

3.0 BASIC FUNCTIONAL FORMS.

The most important part of the language is the set of rules for how functions may be combined to form new functions. These combination rules are called Functional Forms and they determine the expressive power of the language; the primitive functions are the building blocks but the Functional Forms define 'directions' and 'dimensions' of the space in which new functions may be built.

A functional form is an expression containing some function names (and in some cases object names) which are parameters of the expression. It denotes a new function class where the parameters may be replaced by any functional expressions (or objects) to select an instance from the new function class.

We distinguish between basic and derived functional forms in the sense that the derived forms can be defined in terms of the basic forms. Hence the derived forms do not - strictly speaking - add to the expressive power of the language but they are believed to express rather fundamental and often needed operations, and thereby the proper choice may have great importance in the programming activity as well as for the programming style to be used. In most cases it is shown how a derived form may be defined in terms of the basic forms (and earlier introduced derived forms). There is a certain amount of arbitrariness in the choice of which functional forms are considered as basic and which as derived; you may turn 'upside down' the definitions of some of the derived forms and get definitions of some of the basic forms instead. The choice presented here should be considered a first approximation, and further study may lead to a different classification of the functional forms.

The distinction between basic and derived forms is purely logical-mathematical and bears no significance regarding the implementation. In an actual machine one may choose to implement some or all the functional forms as built into the underlying interpreter structure.

We introduce four basic functional forms:

3.1 Composition.

```
F & G :  x --> F :  (G :  x)
```

where F and G are any functional expressions while x is an arbitrary object. The result is 'undefined' if G:x yields 'undefined'. This form expresses composition of functions as used in mathematics, and it is read left to right ("F is composed with G"). However evaluation is from right to left. That is, first apply G to the argument and then apply F to the result of that. Figure 1 is a graphical flow-graph (process chart) illustration of composition, demonstrating the sequencing nature of composition.

Fig. 1 Functional composition: F & G

Composition is associative, i.e.

F & (G & H) = (F & G) & H

but not commutative (in general, F&G is different from G&F).

In the terse notation we may choose to express composition of functions just by juxtaposition of their names, ommitting the & character when its presence can be inferred from the context. Composition may also be written as application of a parametered function, and in section 7.2 we sometimes shall write

```
F & G         as    F(G)
F & [G, H]    as    F(G, H)  .
```

3.2 Construction.

[F1, F2, ..., Fn] : x --> <F1:x, F2:x, ..., Fn:x>

where F1,F2,...,Fn are n arbitrary functional expressions (n>=1) and x any object. This form is used to build new objects from 'parts and pieces' or to change the structure of an object. Note that if any of the functional expressions F1,F2,...,Fn yields 'undefined' when applied to the argument, the constructed sequence contains undefined element(s). Thus

[head, tail] : a --> < a , 'undefined' > (a an atom)

because tail:a is undefined. Figure 2 is a graphical illustration of construction demonstrating the concurrent, parallel nature of construction as each of the functions apply to the same argument.

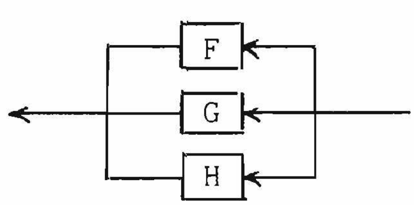


Fig. 2. Construction: [F, G, H]

Examples of the usage of construction are:

To find the length of a sequence and keep it together with the sequence we construct a new 2-element sequence:

```
[id, len] : <x1,x2,...,xn>  -->  < <x1,x2,...,xn> , n > .
```

To delete x1 and update the length accordingly, we may apply the construction (where $\underline{1}$ denotes the constant function with value 1)

$$[\text{tail}\ \&\ \text{id}_1\ ,\ \text{id}_2 - \underline{1}]$$

to the above result yielding the new result

```
< <x2,...,xn> , n-1 > .
```

The entire operation could also be expressed at once with one functional expression as a composition of the two constructions

$$[\text{tail}\ \&\ \text{id}_1\ ,\ \text{id}_2 - \underline{1}]\ \&\ [\text{id},\ \text{len}]\ .$$

The construction [head,tail] imposes a structure on a sequence: Applied to a sequence <x1,x2,...,xn> it gives as result the 2-element sequence

```
< x1 , <x2,...,xn> > .
```

To create a more 'symmetric' splitting into a sequence consisting of <x1> and <x2,...,xn> (which in a certain sense is the inverse of concatenation) we must construct

```
[ [head] , tail ] .
```

In the terse notation we use only one set of parentheses and hence construction will be written as (F1,F2,...,Fn).

3.3 Condition (or Functional Selection).

The syntactical form of a condition is

$$\underline{\text{if}}\ \ p\ \ \underline{\text{then}}\ \ F\ ;\ G$$

where F, G are arbitrary functional expressions while p must be a functional expression that evaluates to true or false when applied to the argument x. The result of applying the condition to an argument is

$$\underline{\text{if}}\ \ p\ \ \underline{\text{then}}\ \ F\ ;\ G\ :\ x\ \longrightarrow \begin{cases} F:x & \text{if}\ \ p:x=\text{true}, \\ G:x & \text{if}\ \ p:x = \text{false} \end{cases}$$

The value is undefined if p:x is undefined, or if the actually applied branch (either F or G) yields 'undefined' when applied to x.

The semicolon is chosen instead of 'else' because conditional expressions often are nested, as in the following example:

```
if  p  then  F ; (if q then G ; (if r then H ; J))
```

and using 'else' would make this look very clumsy. Parentheses may be omitted when no ambiguity arises. Thus the above nested structure may also be written as:

```
( if  p  then  F ;
  if  q  then  G ;
  if  r  then  H ; J )
```

and represents a 4-way branch (a 'case'-expression) where the branch taken depends on p,q, and r; the last function J represents the 'else' case and is used if p, q, and r all give false when applied to the argument. Similarly, the expression

```
if  p  then ( if  q  then  F ; G ) ; H
```

is equivalent to

```
if  p  then  if  q  then  F ; G  ; H .
```

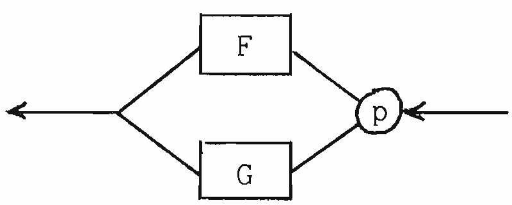


Fig. 3. Condition: if p then F ; G

Figure 3 shows a graphical representation of the conditional form with two branches, the upper branch representing the operative flow path when p applied to the argument is true.

In the terse notation the condition if p then F ;G is written

$$(G , F)_p$$

This may be considered a special case of the more general functional selection

$$(F1 , F2 , \ldots , Fn)_K$$

where K is a functional expression such that K:x is an integer (or even a sequence of several integers) defining which of the functional expressions F1,F2,...,Fn is (are) selected. This will be discussed further in a following section.

3.4 Constant.

A constant function with value y is denoted $\underline{y}$ where y may be any object (including \$ and <>). When this function is applied to an object the result is y:

$\underline{y}$: x --> y for any object x not undefined
$\underline{y}$: 'undefined' --> 'undefined'

Constant functions are used to introduce constants and initial values into objects. Thus, for instance,

$(id_1 + \underline{2}) : \langle x1, x2, ..., xn \rangle$

which is the infix form of $+(id_1, \underline{2}) : \langle x1, x2, ..., xn \rangle$, means:

apply id_1 to the object to get x1 (it must be a number)
apply $\underline{2}$ to the object to get the value 2
apply $\underline{+}$ to the sequence of previous two results
to get the atom whose value is x1+2.

In functional expressions every name symbol denotes a function and never an object or a value. Hence no ambiguity arises from writing the constant function $\underline{y}$ as just y, and we shall - in most cases - use the latter notation. This means that when an object name or a constant value appears in a functional expression, it denotes the corresponding constant function, but in an object it denotes 'itself'. Therefore, in the above example we will allow the notation

$id_1 + 2$, meaning $id_1 + \underline{2}$.

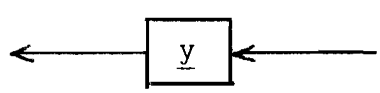


Fig. 4. Constant function: $\underline{y}$

4.0 BASIC PRIMITIVE FUNCTIONS.

In order to manipulate objects we need a set of 'primitives', functions that perform the fundamental types of mappings needed. These primitive functions are then used to build more elaborate functions.

A primitive function belongs to one of the following types:

- selector-function, that takes a sequence as argument and delivers a part of it . The result is an object consisting of one or some of the elements in the input sequence and may be an atom or a new sequence.
- constructor-function, that maps a sequence on to a sequence of the same elements in different order or with a different structure, such as concatenating or merging subsequences.
- operator-function, that performs any other mapping, such as an arithmetic or logical operation. Many of the operators are dyadic and take as argument a sequence of two atoms of some type and outputs an atom of the same type. There are of course also relational operators, for which the result is always an atom of type logical.

The set of primitive functions must not be considered as the ultimate answer to the question, which primitive functions should be included in an implementation. Rather it is a 'minimal' working set which may be expanded in different 'directions' depending on the application wanted.

4.1 Selector Primitives.

Some selector primitives map a sequence into one of its elements, others map a sequence into a sequence consisting of some of the original elements.

select first element:

```
head : <x1,x2, ...,xn>  -->  x1
head : <x1>             -->  x1
head : <>               -->  'undefined'
head : a                -->  a    (a any atom).
```

If the element x1 is itself a sequence, the result is this sequence, and the repeated application of head, written as:

head & head

applied to the argument will deliver the first element of x1 as the result.

<u>select</u> <u>i-th</u> <u>element</u>:

id_i : <x1,...,xi,...xn> --> xi

id_i : <> --> 'undefined'

id_1 : a --> a (a an atom)

where i is an integer constant the value of which satisfies $1 \leq i \leq$ input sequence length. Hence

head $\equiv id_1$

If the argument x is a sequence of sequences, a repeated application of selectors such as

id_j & id_i

(read from right to left like functional composition in mathematics) selects the j-th element from the i-th element xi . We shall also use the notation

$id_{i,j} \equiv id_j$ & id_i

$id_{i,j,k} \equiv id_k$ & id_j & id_i

. . .

analogous to the use of indices in usual matrix and tensor notation. How to select a 'variable' element or subsequences of several elements from a sequence is discussed in the section "Derived Functional Forms".

In a parameter list of a function definition (see below) we will use arbitrary (mnemotechnic) identifiers to denote id_1, id_2, etc. Thus the function definition

newfct(a, b, last) =
<functional expression with a, b, and last>

is equivalent to the definition

newfct = <functional expression with all occurrences of a, b, and last replaced by id_1 , id_2 , id_3 > .

<u>select</u> <u>tail</u>:

tail : <x1,x2, ...,xn> --> <x2, ...,xn>
tail : <x1> --> <>
tail : <> --> 'undefined'
tail : a --> 'undefined' (a an atom)

The result of tail is always a sequence, not an atom, and hence

tail:<x1,x2> --> <x2> while id_2 :<x1,x2> --> x2

and if x2 is itself a sequence <y1,...,yk>, the result of tail:<x1,x2> is <<y1,...,yk>>. tail should never be applied to an atom, as it results in 'undefined'.

identity:

id : x --> x (for any x)

This primitive is used (and needed) when a new object is constructed by adding data to an existing object: The functional construction

[id , FF] : x

where FF is some function, creates the result object <x, FF:x>.

4.2 Constructor Primitives.

A constructor primitive maps a sequence on to a new sequence with the same elements in a different order and/or with a different structure.

Concatenate sequences:

concat : <<x1,...,xp> , <y1,...,yk>> --> <x1,..,xp,y1,..yk>
concat : < <x1,...,xp> , <> > --> <x1,...,xp>
concat : <<x1,...,xp> , a > --> <x1,...,xp,a>
concat : < a , b > --> < a , b >
(a and b are atoms or 'undefined')

(plus the obvious 'symmetric' definitions obtained by interchanging first and second part of the argument). concat is a dyadic operator, i.e., the argument must be a sequence of length 2, and in cases where no ambiguity arises we shall also use the infix notation for this function

x concat y ≡ concat : <x , y>

The concatenation function is a generalization of several of the primitive functions introduced in [Backus 78] except for some special cases. In an accompanying paper [Gram, Organick 80c] is shown how Backus' append, reverse, and rotate functions may be defined in terms of the concat function.

Concatenation is used to construct lists and to string together objects into 'sets' without deepening the hierarchical structure. It is also a tool to get rid of superfluous sub-structure ('extra parentheses') when creating new objects by functional construction, as demonstrated in some of the examples.

Delete nil elements:

```
compress : <x1, ..., xn>  -->  the sequence containing all
                               non-nil elements of the
                               argument (in the same order
                               as they appear in the
                               argument)
compress : a              -->  a   (a any atom)
compress : <$,$,...$>     -->  <>
```

In certain algorithms (e.g., some binary tree operations) it is useful to work with an object containing 'extra' nil elements in certain positions. The compress function may then be used to get rid of the dummy elements at a later stage.

4.3 Operator Primitives.

The operator primitives are mostly monadic and dyadic functions delivering a number valued or a logical valued atom as the result. A basic set might be (where x,y are number valued atoms, n,m are integer valued atoms, and z,v are logical valued atoms, and none of them are $, the nil atom):

Arithmetic operators:

```
+    : <x, y>  -->  x + y
-    : <x, y>  -->  x - y
*    : <x, y>  -->  x * y
/    : <x, y>  -->  x / y
**   : <x, y>  -->  x ** y  (exponentiation)
div  : <n, m>  -->  n div m (integer division)
rem  : <n, m>  -->  n rem m (integer remainder)
abs  : x            -->  absolute value of x
len  : <a1,...,ak>  -->  k  (number of elements in sequence)
len  : <>           -->  0
len  : a            -->  'undefined'  (a  any atom)
```

Logical operators:

```
and : <z, v>  -->  z and v
or  : <z, v>  -->  z or  v
not : z       -->  negation of z
```

Comparison operators (a, b are atoms, not $, belonging to a type with an ordering, and x,y are arbitrary objects (not 'undefined')):

```
<   : <a, b>  -->  a<b    and analogous for  <= , >= , >
=   : <x, y>  -->  x=y  and analogous for  /= (not equal).
atom   : x    -->  true if x is an atom, otherwise false
number : y    -->  true if y is a number valued atom,
                   otherwise false.
```

For the arithmetic operators we shall not specify whether the domain is the integer, the real, the complex numbers, or some other set of numbers. That depends on the application areas and different implementations may implement different number domains. But the usual laws of arithmetic must hold (with good approximation).

For all the dyadic operators - i.e., all the above except length, not, atom, and number - we shall also allow and use the infix notation and parentheses as in ordinary mathematical notation. That means that we shall write arithmetic and logical expressions in standard mathematical form when no ambiguity arises. (+ and - may here also appear as unary operators meaning 0+... and 0-... as usual.) Note that in an expression with several operators with the same priority (see section 7.1) the order of evaluation is right to left (as in the indexed reduction form, section 6.4). Thus, e.g., a - b - c means a - (b - c).

5.0 DERIVED PRIMITIVE FUNCTIONS.

A number of other functions may be defined depending on the type of applications wanted. E.g., in numerical calculations a number of standard mathematical functions may be implemented as primitives, or they may be defined through <function definitions>. The square root of a real number may serve as an example here:

```
sqrt(a)  =  END & ITERATION & START

where

   START          = [ a , 1 ] ,

   ITERATION(a,x) =
             while abs((a-x*x)/a) > 10**(-8) do
                               [ a , (x + a/x) / 2 ] ,

   END = id
           2
```

with the interpretation: The sqrt function takes as argument a single number denoted a, and the evaluation consists of the three steps START, ITERATION, and END in this order (functional composition, right to left). The first step START constructs the 2-element sequence <a,1>. The second step is an iteration that takes as input a sequence <a,x_n> and produces the result <a,x_{n+1}> with the next approximant x_{n+1} (using the Newton iteration scheme); the iteration continues until the relative error is <= 10**(-8). The original value a must be 'carried through' (i.e., made repeatedly available during) the iteration because it is used in every iteration step, but in the final step END it is deleted by selecting the resulting x as the only output.

6.0 DERIVED FUNCTIONAL FORMS.

In this section we introduce functional forms for iteration and arbitrary selection from a construction, as well as a class of indexed reduction forms. The latter is a generalization of the sigma-summation used in everyday mathematics.

6.1 Dynamic Iteration.

Repeated application (composition) may be written as

    while p do F

where p and F are functional expressions. The semantics of this form is repeated application of F

    F & F & ... & F

where the number of iterations is determined by p: As long as p applied to the current argument is true, F is applied to the current argument, yielding the next argument.

The dynamic iteration while p do F is equivalent to the recursive function definition

    WHILE ≡ if p then WHILE & F ; id

and thus derivable from the basic forms composition and condition. It is defined when the functional expressions p and F satisfy the conditions: (i) The result object of F must be compatible with (i.e., exhibit a similar structure as) the input object because it is used as argument for the following iteration of F. (ii) p must evaluate to true or false when applied to an argument of F or a result for F.

An example of the use of the while-form is found in the square root function defined earlier.

If wanted, it would be easy to introduce a form 'repeat F until q' form with semantics as the similar construct in Pascal, and it can be defined recursively through

    REPEAT ≡ (if q then id ; REPEAT) & F .

The reason for not introducing it here is simply that it is not used in the examples shown.

In the terse notation we shall write the dynamic iteration as

$$F^{p*}$$

where the star indicates repetition of F and repeated application of p until p:x=false (the star is chosen because of a certain similarity with the Kleene star).

6.2 Fixed Iteration.

A fixed number of repetitions (n) of a functional expression F (a 'for'-loop) is - both in terse and in verbose notation - written as:

$$F^{N}$$

where N and F are functional expressions, with the semantics: First N is applied to the argument and must evaluate to a non-negative integer n; then F is applied n times (composition as above). If n=0 the iteration is the identity function, in close analogy with ordinary algebra where $x^0=1$, x being any variable. Here also the result object of F must have the same form as the input object. As an example, if <x1,x2,x3,...,xn> is a sequence of length≥3, then

$$tail^{2}$$

gives as result the sequence <x3,...,xn>.

The fixed iteration may be defined in terms of the dynamic iteration and is thus also expressible in terms of the basic functional forms:
F^N is equivalent to the functional expression:

<u>if</u> N < 0 <u>then</u> 'undefined' ;

$(id_1$ & (<u>while</u> $id_2 > 0$ <u>do</u> [F & id_1 , id_2 - 1]) & [id , N])

Note the semantic difference between the two iteration forms: In dynamic iteration the 'conditional' p is repeatedly applied to the new argument, while in fixed iteration N is applied only once to the original argument.

6.3 General Selection.

So as to extract very general substructures from a composite object, we introduce a functional form that allows selection of a much more flexible nature than do the primitive selector functions. The notation is introduced in an informal way, after which we offer a more precise description, where the functional form is defined in a step-wise manner, beginning with the simplest case and gradually increasing the complexity.

Informally, let A be a construction of functions A = [F1,F2,...,Fn] and let I be a function which when applied to the argument yields an integer i. Then

A_I selects the function F_i

to be applied to the argument.
If K is a construction [K1,K2,...,Kp] which yields a sequence of integers (k1,k2,...,kp), then

A_K selects the functions $[F_{k1}, \ldots, F_{kp}]$

to be applied (as a construction) to the argument.
Now, let A be a construction of constructions from which we want to select one or more functions. This is accomplished by double indexing, written as

$A_{I,J}$ meaning: select the J-th function from the I-th construction of A,

and similarly for triple indexing, etc.
To extract more general substructures from '2-dimensional' constructions we extend the multiple index notation to indices that are themselves constructions:

$A_{[I1,I2],J}$ means $[A_{I1,J}, A_{I2,J}]$

$A_{I,[J1,J2]}$ means $[A_{I,J1}, A_{I,J2}]$

and finally, if both indices are constructions:

$A_{[I1,I2],[J1,J2]}$ means $[[A_{I1,J1}, A_{I1,J2}], [A_{I2,J1}, A_{I2,J2}]]$

Thus $A_{I,J}$ is understood to mean the set of A-functions selected by all pairs of I-s and J-s, with a 'matrix-structure' similar to that of A. The notation is like the indexing of vectors and matrices as used in mathematics, and it may indeed also be used here to select elements from sequences: If the argument x is a sequence <x1,x2,...,xn> and A is a mnemonic for the identity function, then

$A_I : x$ means select one or more elements from x

and if x is a matrix (a sequence of rows each of which is a sequence):

$$x = \langle \langle x11,\ldots,x1n\rangle , \langle x21,\ldots,x2n\rangle , \ldots , \langle xm1,\ldots xmn\rangle \rangle$$

then A :x similarly selects one or more elements from the matrix.

The more formal definition of General Selection is done below in 8 steps. Let A, I, J, and K denote functional expressions, and let x denote an object such that:

A : x is a sequence, say of length lx.
I : x is an integer i, $1 \le i \le lx$.
K : x is a sequence of integers in the interval $1 \le k \le lx$.

We first define selection of one element:

(1) $A_I : x \;\text{-->}$ the i-th element of A:x.
If A:x is an atom and I:x=1 ,
the result is A:x .

Remark: Formally speaking, the functional expressions A and I are applied to the same argument, x, before the selection is performed. But in an efficient implementation it may be preferred to postpone application of A until the 'select-value' I:x is known. If A=id and I is a constant function, the definition coincides with the primitive selector function.

The new form may be defined in terms of the previously introduced functional forms:

A_I $\underline{=}$ SELECT & CUTOFF & APPLYINIT
<u>where</u> APPLYINIT $\underline{=}$ [A , I] , CUTOFF $\underline{=}$ <u>while</u> $id_2 > 0$ <u>do</u> [tail & id_1, $id_2 - 1$] , SELECT $\underline{=}$ head & id_1

Let A:x be the sequence $\langle a1,a2,\ldots,alx\rangle$, and let K:x be the sequence of integers $\langle k1,k2,\ldots,kp\rangle$, all between 1 and lx. We then define

(2) $A_K : x \;\text{-->}\; \langle a_{k1}, a_{k2}, \ldots, a_{kp} \rangle$

As a very special example, if A:x is an atom and all the k-s are equal to 1, then the form (2) constructs a sequence with p copies of the same atom. The definition may also be written (a little sloppy)

$$A_K : x \equiv [A_{k1}, A_{k2}, \ldots, A_{kp}] : x$$

Definition (2) easily generalizes to the case where K is a construction whose components yield integers when applied to the argument. Hence

(3) $$A_{[I,J,\ldots]} \equiv [A_I, A_J, \ldots]$$

where I:x, J:x,... each yields an integer or a sequence of integers such that the elements on the right hand side are defined through (1) and (2).

Note that with this definition we distinguish between

$$A_I \quad \text{and} \quad A_{[I]} \equiv [\, A_I \,]$$

the second expression being a construction with the first function as its only element.

Now let A:x be a sequence of sequences (a 'matrix'):

$$< <a11,\ldots,a1n>,<a21,\ldots,a2n>, \ . \ . \ . \ ,<am1,\ldots,amn> >$$

and let I:x=i and J:x=j. Then multiple indexing - selection of a matrix element - is defined as

(4) $$A_{I,J} : x \text{ --> } \begin{array}{l}\text{the j-th element of}\\ \text{the i-th element of } A:x\end{array}$$

Double indexing may be defined in terms of single indexing (using definition (1)) as follows:

$$A_{I,J} \equiv (id_1)_{id_2} \ \& \ [(id_1)_{id_2}, id_3] \ \& \ [\ A\ ,\ I\ ,\ J\ ]$$

or, a little sloppy, using parentheses:

$$A_{I,J} \equiv (\ A_I\ )_J$$

where it is understood that A, I, and J all must be applied to the argument x before selection takes place.

Definition (4) is used to select a single element from a matrix-structured object. Selection of a set of elements is done by a generalization of (4). If K:x is the sequence <k1,...,kp>, then

(5) $$A_{I,K} \equiv [A_{i,k1}, \ldots, A_{i,kp}]$$

$$A_{K,J} \equiv [A_{k1,j}, \ldots, A_{kp,j}]$$

If application of both index functions yield integer sequences, the selection rule is: Apply (5) as above, 'expanding' the index functions in order from left to right, and an index function yielding an integer sequence gives rise to a construction in the result. Thus, if L:x=<l1,...,lq>, then

(6) $$A_{K,L} \equiv [\, A_{k1,L} , A_{k2,L} , \ldots , A_{kp,L} \,]$$

$$\equiv [\, [A_{k1,l1} ,\ldots, A_{k1,lq}], \ldots ,[A_{kp,l1} ,\ldots, A_{kp,lq}] \,]$$

such that the index pair K,L implies forming all the individual integer pairs ki,lj (somewhat like a cross product) and use these as single element selectors. Note that by the ordering and sequence structuring used in (6), we preserve the matrix structure from the object A:x, and if, e.g., K and L yields all the indices of A:x,

K:x = <1,2,...,n> and L:x = <1,2,...,m>

then the functional form (6) is the identity function. If A:x is a 'multi-dimensional' object, selection may be done using a multiple index expression, e.g.,

(7) $$A_{K,I,L}$$

Constructions occurring among the indices are 'expanded' left to right, such that if I, K, and L are defined as above, the meaning of (7) is:

(7a) $$A_{K,I,L} \equiv A_{[k1,\ldots,kp] , I , L}$$

$$\equiv [A_{k1,I,L} , \ldots , A_{kp,I,L}]$$

$$\equiv [A_{k1,i,L} , \ldots , A_{kp,i,L}]$$

$$\equiv [A_{k1,i,[l1,\ldots,lq]} , \ldots , A_{kp,i,[l1,\ldots,lq]}]$$

$$\equiv [\, [A_{k1,i,l1} ,\ldots, A_{k1,i,lq}] , \ldots , [A_{kp,i,l1} , \ldots] \,]$$

Thus, in a sense, the comma in multiple indexing works as a right-associative cross product operator on indexsequences.

Selection of one or more rows from a matrix is now easily done by applying a form like

$$A_{I}$$

Selection of a column requires a construction like

$$A_{[1,2,\ldots,n],J} : x$$

where n is the number of rows in A:x. The sequence of all row index values may be constructed by concatenating the integers 1,2,...,n and this may be expressed as

$$\text{concat}_{i=1}^{len} (i)$$

using the indexed reduction form defined below. But since it is a useful construction in many applications, we introduce for this purpose a star index notation meaning 'all index values':

```
(8)    * : x  -->  <1,2,...,N>  where N is the number of
                                rows    if * is used as index 1,
                                columns -  * -   -   -    2, etc.
                                        in the object to which this
                                        subscript expression is
                                        applied.
       * : a  -->  'undefined'      (a an atom).
```

With this definition the following holds:

$$A_{*} \equiv A$$

$$A_{I,*} \equiv A_{I}$$

$$A_{*,J} :x \;\; \text{-->} \;\; \text{the column(s) selected by J:x}$$

(Strictly speaking, this definition holds only if x and A:x has the same structure - same number of rows etc. - but this will be the case in most applications.)

6.4 Indexed Reduction.

In mathematics, notations like

$$\sum_{i=1}^{n} A_i \qquad \text{and} \qquad \prod_{k=1}^{100} p(k)$$

are used as short-hands for repeated application of a dyadic, associative operator to a sequence of operands all of the same type.

A similar notation is introduced here, very much resembling the reduction operator in APL. Let OP be an operator, A(i) some functional expression depending on an undefined integer, 'dummy' variable i, and let I1, I2 be two 'index' functional expressions. Then the functional form which we shall call <u>indexed reduction</u> is written as below, with the meaning indicated by the right hand side:

$$(9)\quad \underset{i=I1}{\overset{I2}{\textcircled{OP}}} \; (A(i)):x \; \text{-->} \; A(i1):x \;\textcircled{OP}\; A(i1+1):x \;\textcircled{OP}\; \ldots \;\textcircled{OP}\; A(i2):x$$

More precisely, the entities occurring here must satisfy the conditions:

1. OP must be a dyadic function, the result of which is of the same type as its two operands (as, e.g., several of the arithmetic and logical operators, as well as the concatenation primitive).

2. I1 and I2 are functional expressions that evaluate to integers i1 and i2, $0 < i1 \leq i2$, when applied to the argument x.

3. A(i) is a functional expression, in which i denotes a constant function, such that A(i):x is defined for all i in the interval $i1 \leq i \leq i2$, and A(i):x must all be objects of OP-operand type.

Logically (but not necessarily so in a real implementation), the application of indexed reduction proceeds as follows:

1. Evaluate I1:x --> i1 and I2:x --> i2.

2. Evaluate A(i1):x-->x1, A(i1+1):x-->x2, ..., A(i2):x-->xp .

3. Evaluate the result as $x1 \;\textcircled{OP}\; x2 \;\textcircled{OP}\; \ldots \;\textcircled{OP}\; xp$ in right to left order.

In the most common applications of this functional form, the function OP is one of the operators: addition, multiplication, or concatenation, and we shall in some of the examples below use the notations

Σ for indexed reduction with +

Π - - - - *

$\mathbb{C}$ - - - - concat, equivalent to the construction of a sequence from its single elements.

When the reduction is to be applied for all members of a certain set (e.g., all elements in a sequence), a star notation is used:

$$(10) \quad \text{(OP)}^{*}_{i=1} \; (\; A(i)\;):x \;\text{-->}\; A(1):x \;\text{(OP)}\; A(2):x \;\text{(OP)}\; \ldots \;\text{(OP)}\; A(N):x$$

where N is the last integer in sequence for which A(i):x is defined and gives an object of OP-operand type. (N must be finite.)

Nested application of a reduction is often useful, especially in matrix manipulation. Since A(i) in the above definition may be any functional expression, it can be a reduction form itself. Hence an expression such as

$$(11) \quad \sum_{i=1}^{n} \left(\sum_{j=1}^{i} (A_i * A_{n-j}) \right) = \sum_{i=1}^{n} (\; B(i)\;)$$

can be interpreted according to the given rules:

1. In the outer form, I1=1 and I2=n. Hence we must evaluate

$$B(1):x = \sum_{j=1}^{1} (A_1 * A_{n-j}):x$$

$$B(2):x = \sum_{j=1}^{2} (A_2 * A_{n-j}):x$$

. . .

$$B(n):x = \sum_{j=1}^{n} (A_n * A_{n-j})):x$$

   and then add together all these values.

2. In each of the inner forms, I1=1 and I2 = some number i. Hence we must evaluate

$$(A_i * A_{n-1}):x$$

$$(A_i * A_{n-2}):x$$

. . .

$$(A_i * A_{n-i}):x$$

   and add together all these values to get B(i):x.

The parentheses in (11) may be a help for reading and understanding the expression, but they are not required in this case. No ambiguity arises if the parentheses are left out because of the rule of syntactic scanning left to right and evaluation right to left (see "Functional Definitions"). Hence, exactly the same result is obtained from the expression without parentheses:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} A_i * A_{n-j} \quad .$$

The indexed reduction form is derivable from the forms and primitives introduced earlier. If OP is addition, e.g., the form may be defined as follows (using the terse notation for dynamic iteration):

$$\sum_{i=I1}^{I2} (A_i) \equiv \text{RESULT \& SUMMATION \& INITSUM \& CONSTRUCTSEQ \& APPLYINIT}$$

where

APPLYINIT $\equiv$ [<>, id , I1 , I2] ,

CONSTRUCTSEQ $\equiv (id_1 \text{ concat } A_{id_4} , id_2, id_3, id_4 - 1)^{(id_3 \leq id_4)*}$,

INITSUM $\equiv$ [0 , id_1] ,

SUMMATION $\equiv (id_1 + \text{head \& } id_2, \text{tail \& } id_2)^{(id_2 /= <>)*}$,

RESULT $\equiv id_1$

7.0 FUNCTIONAL DEFINITIONS.

7.1 Functional Expressions.

Using the primitive functions and the functional forms as building elements, algorithms defining new functions are expressed by combining the elements in functional expressions. A functional expression is one of the following:

1. A primitive function.

2. A functional form.

3. A <fct.name>, i.e., the name of a function defined elsewhere in the current context (see scope rules as defined below). The function may be applied with or without parameters.

Since a functional form may contain functional expressions, the definition above is recursive and allows construction of arbitrarily complex functional expressions. Parentheses are used to express grouping when necessary, i.e., whenever the built-in priorities of the functions and operators don't suffice. The following list is a preliminary suggestion for the built-in binding priorities, from the highest to the lowest:

(highest) index selection F_G and condition F_p

iteration F^N and F^{p*}

composition F & G

dyadic operators (when written in infix form):

- **
- * / mod rem
- + -
- $<$ $\leq$ $=$ $\geq$ $>$ /=
- and
- or
- concat (lowest)

The construction form [...,...] groups like ordinary parentheses and has thus - in a sense - the highest priority. A similar rule holds for iteration and index expressions: any subscript or superscript is implicitly taken to be surrounded by parentheses and is evaluated per se, before being applied to the 'radicand' expression.

As an illustration of these rules, the expression

$$[\ D \text{ concat } E_q \ , \ A + B \ \& \ C_{I,J}^{P \text{ and } Q*} \ ]$$

is equivalent to the fully parenthesised expression

$$[\ (D \text{ concat } (E_q)) \ , \ (A + (B \ \& \ (C_{I,J})^{(P \text{ and } Q)*})) \ ]$$

[Note: In Barton/Clark notation, iteration (functional exponentiation) is considered to be more binding than selection. Thus,

(a) F^N_i means the ith component of F^N, as does

(b) ${F^N}_i$, whereas

(c) ${F_i}^N$ means the Nth iterate of F_i .]

If condition is written in the verbose form, the above rules imply that, say,

if p *then* F ; G & H $\underline{=}$ (*if* p *then* F ; G) & H

Whenever confusion may arise as to the extent of a conditional expression, parentheses should be used to bracket it.

As an example of the use of the functional expressions, consider the problem of finding the maximum element in a sequence of real numbers <a1,a2,...,an> . The definition-tree for the function MAX below gives as its result a 2-element sequence with the maximum element and its index in the form:

$$\langle \text{ max a , index of max a } \rangle \ .$$

```
MAX(A) ≡ RESULT & LINSEARCH & INITIALIZE
------------------------------------------------------------------
where

   INITIALIZE          ≡ [A, [A ,1], 2],
                               1
  LINSEARCH(A,max,I) ≡
                                                          N-1
           [ A , if  A > max  then [A ,I] ; max , I+1]
              I       1             I
  where

     N     ≡  length( A )

   RESULT              ≡ id
                           2
```

Here the first line defines the MAX function as a functional expression, being the composition of three functions defined in the next lines of the d-tree. The single parameter A is here just a mnemotechnic for id. The subfunction INITIALIZE is defined through its functional expression as a construction of three objects, of which the middle one itself is a construction. The parametered subfunction LINSEARCH is defined as a d-tree because it again has a subfunction N; LINSEARCH works on a 3-tuple and performs the linear search by performing N-1 constructions of the same form as made by INITIALIZE: A is kept unchanged, the index I is increased by 1 per iteration, and the middle element max is updated whenever a larger element is found. Finally, RESULT is defined by a very simple functional expression being just the selector id , delivering the latest <Ai,i> as the result.

7.2 Semantics Of D-trees And Parameters.

In this section we shall gain understanding of the syntax and semantics of algorithms expressed as tree-structured (hierarchic) function definitions. From the BNF syntax in Section 2, we see that, in keeping with conventional mathematical notation, a d-tree is a function consisting of a main function (definition), followed by a set of mutually independent subfunctions (definitions), each having, recursively, a similar structure.

Examples will be given in the terse notation; parentheses will be used both for bracketing parameter lists and for denoting constructions.

One should keep in mind three key rules:

1. The text of a definition is to be read (scanned) top-down (line-by-line), with each line read from left to right.

2. Functional expressions within definitions are to be understood (evaluated) primarily from right to left. If any expression extends beyond one line, then it is evaluated bottom-up (line-by-line).

3. The argument of a d-tree is the argument of its root function.

These three rules will help you to understand the use and scopes of parameters and subfunctions within a function definition, as defined below. Subfunctions (sub d-trees), which are introduced under the where mark, similar to usual mathematical notation, are applied to carry out application of the root function to its argument. In the sequel, we shall mainly/exclusively deal with main functions with parameters.

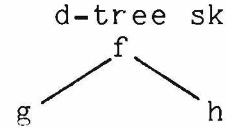
Preliminary concepts needed to understand d-tree semantics

Several examples will help make more precise the points just made.

Example 1

```
f(w,x,y) ≡ (g(w,x), h(w,y))
   where
         g(u,v) ≡ u + v,
         h(a,b) ≡ a * b
```

d-tree skeleton



This d-tree is applied to an argument being a sequence of three objects represented, respectively, by parameters w, x, and y. The functional expression for f consists of the construction of two mutually independent functions, g and h. Parameters of g and h, in the definitions under where, are matched, via the usual rules of positional correspondence, with their corresponding arguments in the application on the first line. Thus, for g,

the substitution is: $\left\{\begin{array}{l} w \text{ --> } u \\ x \text{ --> } v \end{array}\right\}$, and for h,

the substitution is: $\left\{\begin{array}{l} w \text{ --> } a \\ y \text{ --> } b \end{array}\right\}$.

Since g and h are each to be applied to argument structures dependent on the argument structure of f, the application of g and of h must be deferred until their respective arguments have been produced from that of f. In general, application of any subfunction that is defined with parameters takes place only after the argument structure of the main function is properly mapped to the desired argument structure for the subfunction.

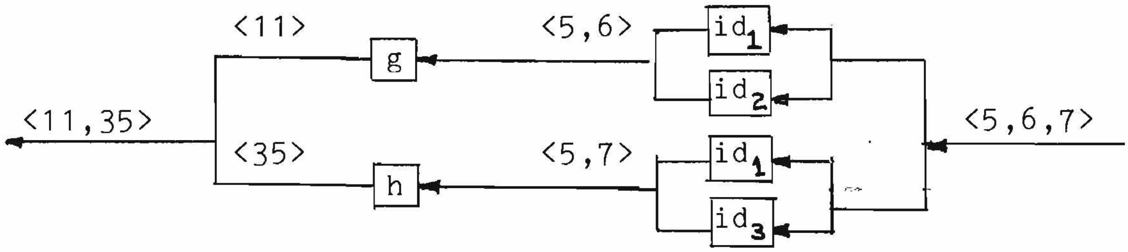
In the context of f being applied to its argument,

g(w,x) is a shorthand for g & (id_1, id_2)

and

h(w,y) is likewise a way to express h & (id_1, id_3).

Now suppose the argument of f is the sequence <5, 6, 7>; then, application of the constructions (w,x) and (w,y) to the argument of f yields <5,6> and <5,7>, respectively. The subfunctions g and h are then applied to these new arguments, eventually forming the result, <11,35>, as the process (or data flow) diagram below suggests.



In this diagram we have elected to suppress the details for

<---[g]<--- and <---[h]<---, which in this case may be simply

replaced by <---[+]<--- and <---[*]<---, respectively.

The right hand sides of the definitions of g and h were originally given as infix expressions. But, as said in section 4.3, we shall allow syntactical alternatives, such as

$\left\{\begin{array}{l} g(u,v) = +(u,v) \\ h(a,b) = *(a,b) \end{array}\right\}$ or, even more succinctly, $\left\{\begin{array}{l} g(u,v) = + \\ h(a,b) = * \end{array}\right\}$

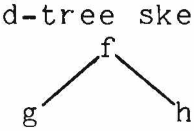
In our first example, we have illustrated the case where the main function refers to (invokes) subfunctions whose arguments are derived by functional composition from the main function. We might even say this is the usual relationship between a main function and its subfunctions. Such subfunctions must be evaluated (applied) each time they are referenced.

Another case arises where the argument of the main function, f, and that of a subfunction are the same, as in the next example.

Example 2

```
f(w,x,y) ≡ (g, g, h)
    where
      g ≡ w + x
      h ≡ w * y
```

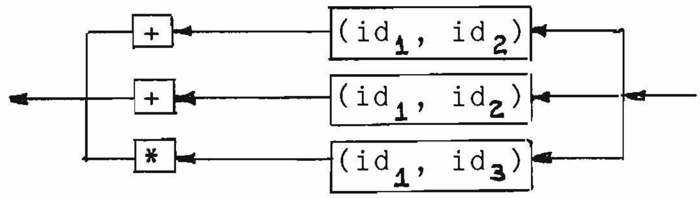
d-tree skeleton



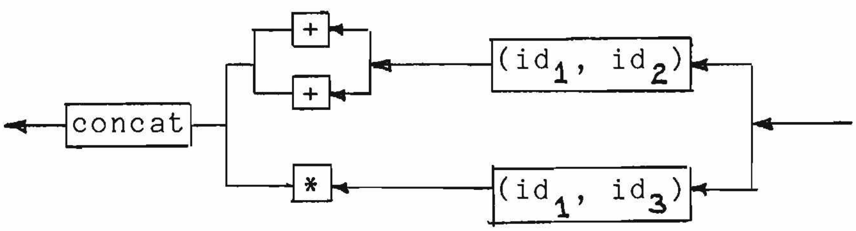
In an application of this d-tree to the argument, <5,6,7>, for instance, it is sufficient to evaluate the right hand sides of g and h only once, by evaluating g and h prior to evaluating the right hand side of f. Here, because g and h have no parameters, they depend directly on the argument of f. This is characteristic of what we shall denote as parameterless subfunctions.

The meaning of a d-tree is independent of the order in which its parameterles subfunctions (if any) are evaluated (applied to the d-tree's argument). Therefore, there is no loss of conceptual generality if, in some underlying implementation, it is convenient to evaluate each parameterless subfunction before the root function's right hand side is evaluated.

Of course, it is possible to draw a process diagram to suggest how an underlying implementation may evaluate f, such as:



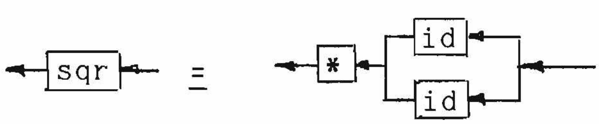
but it cannot be regarded as necessarily the best way to achieve the result. Other interpretations, mathematically equivalent, come to mind, such as:



Depending on the relative speeds of executing selections, constructions, and concatenations in the underlying machine, one alternative may be preferred to another.

A reference to a parameterless subfunction may not be followed by a list of arguments. On the other hand, a reference to a parametered function, k, ordinarily includes an argument list that conforms to the (formal) parameter list of k. Thus, examples 3, 5, 6, 8, and 9 are all mathematically equivalent. But, their interpretations in our frame of reference differ as follows:

Example 3 has the interpretation given in 4. Examples 5 and 6 have the identical interpretation given in 7. Here [f] and [g] are composed, respectively, with the "filters" [r] and [s] to transform p's argument to those of f and g. Examples 8 and 9 have the identical interpretation given in 10. Here, [f] and [g] are composed with the identical filters, [(r, s)], because in this case f and g each require arguments that happen to be identical copies of p's argument. (In this example set, we have assumed that sqr is a primitive squaring function, i.e.,

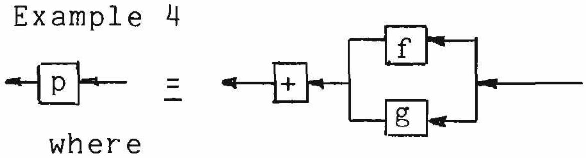


.)

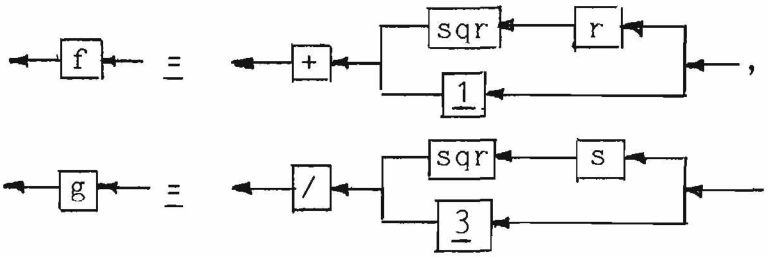
Example 3

```
p(r,s) ≡ f + g
  where
    f ≡ sqr(r) + 1,
    g ≡ sqr(s)/3
```

Example 4



where



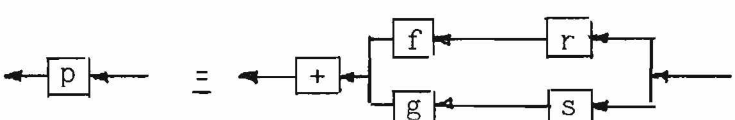
Example 5

```
p(r,s) ≡ f(r) + g(s)
  where
    f(a) ≡ sqr(a) + 1,
    g(c) ≡ sqr(c)/3
```

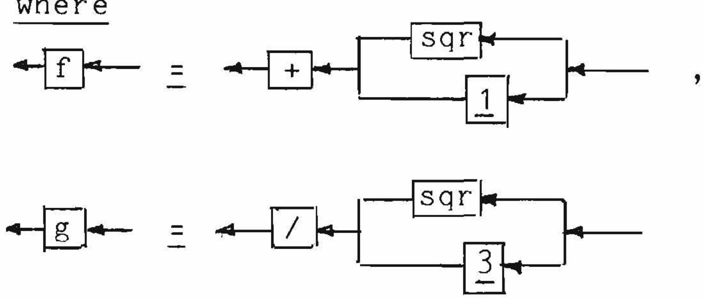
Example 6

```
p(r,s) ≡ f(r) + g(s)
  where
    f(r) ≡ sqr(r) + 1,
    g(s) ≡ sqr(s)/3
```

Example 7



where



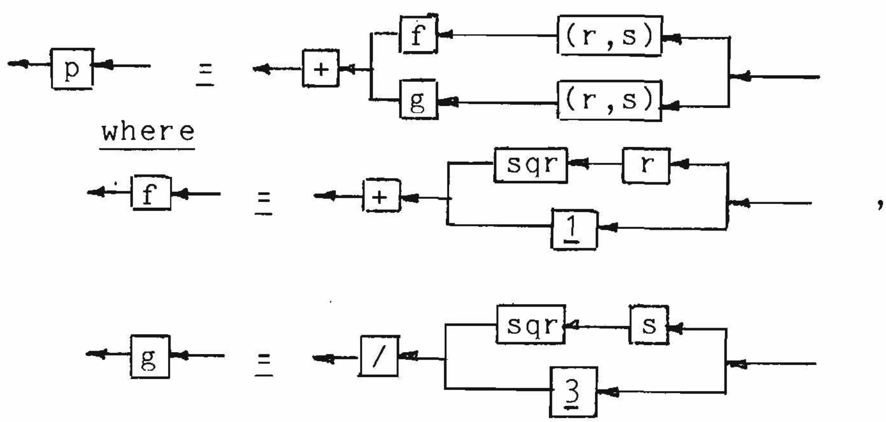
Example 8

```
p(r,s) ≡ f(r,s) + g(r,s)
  where
    f(a,b) ≡ sqr(a) + 1,
    g(c,d) ≡ sqr(d)/3
```

Example 9

```
p(r,s) ≡ f(r,s) + g(r,s)
  where
    f(r,s) ≡ sqr(r) + 1,
    g(r,s) ≡ sqr(s)/3
```

Example 10



When the actual argument of a parametered subfunction, g, is the result of a composition, the argument list for g is implicit, as seen in Example 11.

Example 11

```
P(u,v) ≡ A & B & C(u,v)
  where
    C(r,s) ≡ ( sqr(r) , cube(s) ) ,
    B(a,b) ≡ a + b ,
    A(x)   ≡ sqr(x)
```

Thus if P is applied to <3,4>, the implicit argument list for the reference to B will be <9,64>, because C(3,4) ≡ (sqr(3),cube(4)) ≡ <9,64>. In turn, the implicit argument list for A is 73, which is the result of applying B to <9,64>. Incidentally, in this case even the argument list for C may be omitted since it comprises the entire argument list for P. In other words, there would be no change in the meaning of P if it were defined as in Example 12.

Example 12

```
P(u,v) ≡ A & B & C
  where
    C(r,s) ≡ ( sqr(r), cube(s) ),
    B(a,b) ≡ a + b,
    A(x)   ≡ sqr(x)
```

Rules for Evaluation of a D-tree

We are now ready to offer an informal definition for the semantics of d-trees, including the use of recursive definitions.

(1) Application of a d-tree implies (is achieved by) application of its root function.

(2) A step preliminary to evaluation of the right hand side expression of the root function is the evaluation of each parameterless subfunction -- applied to the argument of the d-tree. This leads to the constraint that a parameterless subfunction, g, may not appear on the lefthand side of a composition, such as g & h. Execution of h would necessarily produce a new context for g which will in general differ from that of the containing d-tree.

Example 13a

```
P(u,v) ≡ g & h(u-1, v+1)
  where
    h(a,b) ≡ a * b ,
    g      ≡ u + v
```

illegal

Example 13b

```
P(u,v) ≡ g & (h(u-1,v+1),5)
  where
    h(a,b) ≡ a * b ,
    g      ≡ u + v
```

illegal

To see why 13a is illegal, note that computing P(3,4) could lead to an attempt to apply g to the argument, h(2,5) = 10. This leads to an attempt to apply g in the context, 10, which is not even conformable with (u,v) let alone equal to <3,4>, the required context. Another way to see the illegality of 13a (and also of 13b) is to notice the ambiguity involved. One would get a different result when evaluating P(3,4) if g is applied as the first step in the application of P, rather than in the last step. Thus, in Example 13b the two possible values for P(3,4) would be 7 and 15.

(The order in which parameterless subfunctions are evaluated is inconsequential, and they may be performed concurrently, if the underlying computing system is so organized.)

(3) Following (2) above, each referenced parametered subfunction is applied as required in the evaluation of the root function. The actual parameter list in a reference to a parametered subfunction may be suppressed (remain implicit, as was seen in Examples 11 and 12 above) when the argument is the result of a preceding function application (by composition) or when the argument of the subfunction is the argument of the its d-tree.

(4) To preserve the strict hierarchic intent of the d-tree, no subfunction may refer to a sibling subfunction. (It may only refer to its immediate parent of to its own direct offspring.)

(5) Application of a parametered subfunction implies (is achieved by) the construction of a new argument context (as specified by the formal parameter list) which is used in evaluating the parametered subfunction's right hand side and which temporarily hides the caller's argument context. Therefore, parameters that denote objects in antecedent contexts may not appear on the right hand sides of parametered subfunction definitions. (No free variables (globals) allowed.) Hence, the right hand side of a parametered subfunction definition may not include a reference to the context of the d-tree in

the form of a parameter of the root function (unless the root function parameter has been properly repeated as a parameter of the subfunction).

The following are, respectively, illegal and legal examples vis a vis the above constraint.

Example 14a

```
p(u,v) = h(u-1,v)
  where
     h(a,b) = a*b + v
```

illegal

Example 14b

```
p(u,v) = h(u-1,v)
  where
     h(a,v) = a*v +v
```

legal

Parametered functions must be applied with care when combined with composition, as illustrated in Example 15.

Example 15a

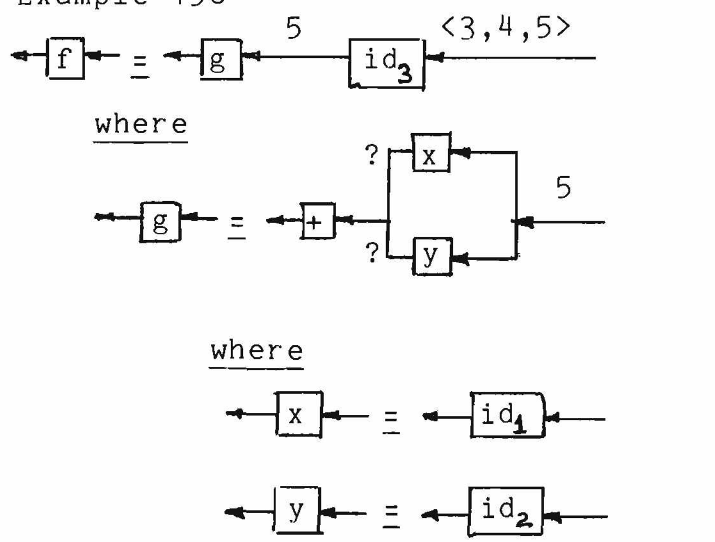
```
f(a,b,c) = g(a,b) & id3
 where
    g(x,y) =  x + y
```

or, alternatively,

Example 15b

```
f(a,b,c) = g  &  id3
  where
    g(x,y) = x + y
```

Example 15c



Composing g(a,b) with id (or g with id) leads to what may be regarded as an unexpected (or unintended) result. Application of id tranforms the original argument context of the d-tree to a new one that may be incompatible with the one required for application of g. The equivalent process diagram in Example 15c reveals the potential inconsistency of the definitions in 15a. For example, when the d-tree is applied to the triple, <3,4,5>, application of id produces 5, which is then supplied as the argument for the construction, (x,y). Recall that (x,y) is merely a shorthand for (id_1,id_2). Since (x,y) cannot be applied to 5, the computation must fail at this point. On the other hand, if the argument of the d-tree were <3, 4, <5, 6> >, the interpretation of the diagram in Example 15b would lead to the perfectly reasonable result, 11, which may or may not have been intended.

This example shows that parametered functions must be applied with care when combined with composition.

(6) Within the above framework, the following recursive definition structures are permitted:

(a) A main function or a subfunction may be recursively defined,

(b) Mutual recursion involving a main function and one (or more) of its parametered subfunctions is permitted, provided, of course, no subfunction refers directly to a sibling subfunction.

It is easy to see why a parameterless subfunction, g, may not be defined mutually recursive with its root function, f, for if so, g could be evaluated first, leading to a first actual application of f from within the tree, rather than from outside.

(7) The above semantics (1 through 6) are unchanged under the generalization that each subfuntion of a d-tree's root function may itself be the root function of a sub d-tree.

8.0 EXAMPLES.

8.1 Building A Search Tree.

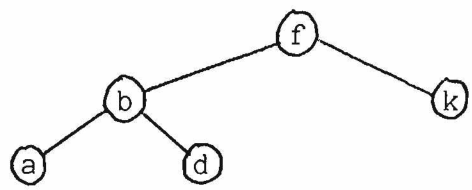
A binary tree may be represented as a 3-element sequence

< tree , left subtree , right subtree >

where the two last items again are binary trees in the shape of 3-element sequences. Hence all branch nodes in the tree have the above form, while a leaf is represented as

< node , $, $ >

Thus the binary tree



will be represented as

< f , <b, <a,$,$>, <d,$,$>> , <k,$,$> > .

The function, INSERT, defined below inserts a new element as a leaf in an ordered binary tree. The function takes as its argument an object of the form <Tree ,x > , where Tree is an ordered binary tree as above, and x is an element of the same kind as the first element of each triple in the tree. The result of applying the function is a new tree-object where x is inserted in a new leaf <x,$,$>.

INSERT is defined as a recursive function, and the idea behind the algorithm is to search down the tree until a $-node is found; then this node is replaced by the new leaf <x,$,$>. In the first version, INSERT is a parameterless function:

```
INSERT  ≡
   if Tree = $ then [ x  ,  $  ,  $ ] ;
   if ROOT > x then [ROOT, INSERT & [LEFT,x], RIGHT] ;
                    [ROOT, LEFT, INSERT & [RIGHT,x]]
------------------------------------------------------
where

   Tree  ≡  id_1 ,
   x     ≡  id_2 ;

   ROOT  ≡  id_1,1  ,  LEFT  ≡  id_1,2  ,  RIGHT  ≡  id_1,3
```

Instead of giving the structure of the argument indirectly through the subfunction definitions of Tree and x, this may be displayed more clearly by use of parameters:

```
INSERT( Tree , x )  =
   if Tree = $ then [ x   ,  $   ,  $ ] ;
   if ROOT > x then [ROOT, INSERT(LEFT,x), RIGHT] ;
                    [ROOT, LEFT, INSERT(RIGHT,x)]
-----------------------------------------------------------
where
   ROOT  =  Tree_1 ,  LEFT  =  Tree_2 ,  RIGHT  =  Tree_3
```

or, without using subfunctions at all:

```
INSERT( Tree , x ) =
   if Tree = $ then [ x   ,  $   ,  $ ] ;
   if Tree_1 > x then [Tree_1 , INSERT(Tree_2 ,x), Tree_3] ;

                    [Tree_1 , Tree_2 ,INSERT(Tree_3 , x)]
```

The three definitions are equivalent, and none of them check whether the new element x already exists in the tree. If this check is done, the result must convey the information about success or failure, hence we change the wanted result to be:

< the new tree , true >   or   < the old tree , false >

Reporting of the success or failure at the deepest level of the recursion may be done as follows (corresponding to the second version above):

```
INSERT( Tree , x )  =
   if Tree = $ then [ [x, $, $] , true ] ;
   if ROOT = x then [    Tree     , false] ;
   . . .
```

But the next part of the definition must be changed in order to 'carry back' the logical value through the recursion levels as the second element of the result. First, note that each of the two constructions in the conditional if ROOT>x ... no longer gives a tree object as result but a structure like:

< root , <tree, logical value> , tree > ,

and this must be rearranged into:

< <root, tree, tree> , logical value > .

At every level of recursion, the result of INSERT must have this structure. To perform these transformations, the subfunctions TRANSFORML and TRANSFORMR may be introduced, and we thus get the complete definition of INSERT as follows:

```
INSERT( Tree , x ) =
   if Tree = $ then [ [x, $, $] , true ] ;
   if ROOT = x then [    Tree    , false] ;
   if ROOT > x then
      TRANSFORML & [ROOT , INSERT(LEFT,x) , RIGHT] ;
      TRANSFORMR & [ROOT , LEFT , INSERT(RIGHT,x)]
-----------------------------------------------------------
where
   ROOT = Tree  ,  LEFT = Tree  ,  RIGHT = Tree  ,
              1               2                3

   TRANSFORML(Node, L, R) = [ [Node, L , R] , L   ] ,
                                      1        2

   TRANSFORMR(Node, L, R) = [ [Node, L, R ] , R   ]
                                         1     2
```

If sibling subfunctions, such as TRANSFORML, ROOT, LEFT, and RIGHT, might refer to each other, the definition of TRANSFORML could also be written as

$$\text{TRANSFORML} \equiv [\ [\text{ROOT},\ \text{LEFT}_1,\ \text{RIGHT}]\ ,\ \text{LEFT}_2\ ]$$

and correspondingly for TRANSFORMR.

8.2 Iterative Solution Of Linear Equations (Jacobi Method).

Let A be an m-row by n-column positive definite matrix of reals, and let B be an m-element sequence (vector) of reals. The Jacobi function (defined below) returns an m-element vector, X, which is an approximate solution of the equation

$$A * X = B.$$

The result returned is the kth iterate (k>0) of X, given the zeroth iterate, X0, and subject to the constraint that k may not exceed MAXITER, an upper bound on the allowed number of iterations. If convergence proceeeds as expected, the kth iterate of X will be the first iterate for which the Euclidean norm of $X_{k+1} - X_k$ is less than the given tolerance, TOL:

Jacobi(A,B,X0,TOL,MAXITER) $\underline{=}$ J(nextx(A,B,X0),X0,MAXITER,A,B,TOL)

<u>where</u>

J(XNEW,XOLD,MAXITER,A,B,TOL) $\underline{=}$

$$\left((nextx(A,B,XNEW),XNEW,MAXITER-1,A,B,TOL)^{pred(XNEW,XOLD)*} \right)_1$$

<u>where</u>

$$pred(XNEW,XOLD) \underline{=} \sum_{i=1}^{len(XNEW)} sqr(XNEW_i - XOLD_i) > TOL \text{ and } MAXITER > 0$$

$$nextx(A,B,X) \underline{=} \mathbb{C}_{i=1}^{M} (B_i - \sum_{j=1}^{i-1} A_{i,j} * X_j - \sum_{j=i+1}^{N} A_{i,j} * X_j)/A_{i,i}$$

<u>where</u>
M $\underline{=}$ len(A),
N $\underline{=}$ len(head(A))

$$nextx(A,B,X) \underline{=} \mathbb{C}_{i=1}^{M} (B_i - \sum_{j=1}^{i-1} A_{i,j} * X_j - \sum_{j=i+1}^{N} A_{i,j} * X_j)/A_{i,i}$$

<u>where</u>
M $\underline{=}$ len(A),
N $\underline{=}$ len(head(A))

(An alternative condition for convergence of the Jacobi method is that the matrix A exhibits diagonal dominance and that the system system of equations defined by A and B is irreducible.)

[Syntactical note:
To make long iterative forms, such as the one for J above, more easy to read, we are free to drop from the iterated expression, parameters which do not change under repeated composition and which would otherwise appear at the right end of that form. Thus we may rewrite the definition of J as:

J(XNEW,XOLD,MAXITER,A,B,TOL) $\underline{=}$

$$\left((nextx(A,B,XNEW),XNEW,MAXITER-1)^{pred(XNEW,XOLD)*} \right)_1$$

End syntactical note.]

The application of functions which also check the conformability of input arguments A and B, and which possibly also make more substantive checks, such as to determine if A is positive definite, may be preferred. For example, Pre-Jacobi, defined below, checks the dimensionalities of A and B for conformability, and if conformable, applies Jacobi, supplying a zero vector as the starting vector, X0, and the number 100 as the value for MAX-ITER in the application of Jacobi. Pre-Jacobi returns a two-tuple of the form:
(false, $) or (true, <result of Jacobi>).

```
Pre-Jacobi(A,B,TOL) ≡
                                         len(A)
    ((false, $), (true, Jacobi(A,B,  C         0,TOL,100))
                                         i=0                len(A)=len(B)
```

The possibility for exploiting parallel execution when calculating the components of an iterate X_{k+1} is expressed in the definition of nextx. The indexed concatenation means that the components of X_{k+1} are formed by a construction, and elements in a construction may be evaluated in parallel (if the underlying machine has processing elements that may be used for this purpose.)

In a related method (Gauss-Seidel), which has the same sufficient conditions for convergence of the iteration, elements of X_{k+1} are computed in sequence so that each newly calculated component of X_{k+1} immediately enters into the calculation of the next X_{k+1} component. This method converges faster and may hence be more attractive than Jacobi when it is known that the potential for parallelism cannot be exploited.

8.3 Binary Search

Let A be an ordered vector of numbers - say, increasing - <a1,a2,...,an> , and let 'key' denote the number whose place (index) in the vector is wanted. The argument to the search function is

< A , key > = < <a1,a2,...,an> , key >

and the result should be

< A , i , true > if ai = key,
< A , 0 , false > if the search fails.

The method used is to construct a pair of indices <low,high> such that A(low) ≤ key ≤ A(high) and continue 'halving the gap' until low=high. Thus the first step is a construction where the initial index pair is created:

(1)
```
BINSEARCH(A, Key) = . . . & FIRST
where
    FIRST ≡ [ A , [1, len(A)] , Key ] ,
    . . .
```

Next step is an iteration performed on this FIRST construction, consisting of A, an interval, and Key. The quantities A and Key are kept unchanged and the interval (initially [1,len(A)]) is repeatedly halved: In each cycle the midpoint M is found and the left or the right half selected in accordance with the test Key ≤ A(M) :

(2)
```
ITERATE(A, (LOW, HIGH), Key) ≡
                                                     (LOW<HIGH)*
  [A, if Key≤A_M then [LOW,M] ; [M+1,HIGH], Key]
-------------------------------------------------------------
where

    M  ≡  (LOW + HIGH) div 2
```

This iteration is guaranteed to terminate with LOW=HIGH because the interval under consideration becomes strictly shorter for each iteration step: As long as the difference between HIGH and LOW is 2 or more, M satisfies the strict inequalities LOW < M <HIGH; when HIGH=LOW+1 the next M becomes M=LOW and thus the next interval, either [LOW,M] or [M+1,HIGH], has the length 0, and that will cause the iteration to stop.

The iteration delivers a result of the form

< A , <index,index> , key >

and it now remains to test for success or failure and select the result object:

(3)
```
RESULT(A, (I1,I2), Key) ≡

        if A_I1 = Key then [ A , I1 , true] ;
                           [ A , 0 , false]
```

Thus the binary search algorithm is assembled by putting (1), (2), and (3) together:

(4)
```
BINSEARCH(A, Key) ≡ RESULT & ITERATE & FIRST
-------------------------------------------------------------
where

   FIRST ≡ [ A , [1, len(A)] , Key ] ,

   ITERATE(A, (LOW,HIGH), Key) ≡ { as in (2) above } ,

   RESULT (A, (I1 , I2 ), Key) ≡ { as in (3) above }
```

8.4 Linear Regression.

Let X = <x1,x2,...,xn> , Y = <y1,y2,...,yn> be two vectors which provide corresponding pairs of data (say, measurements) xi,yi. We want to define a function performing linear regression on these data, calculating the standard statistical quantities

slope and intercept of regression line	A, B
standard deviation	STDDEV
correlation coefficient	CORR
F-ratio	F

The formulae for these quantities may be found in any statistical handbook, and a complete Fortran program (1 page long) can be seen in R.L.Nolan: "Fortran IV Computing and Applications", section 15.1.

The LINREGR function takes the sequence <X,Y> as input and returns a sequence of the above 5 quantities:

```
LINREGR(X,Y) ≡ COMPUT_F    & COMPUT_CORR & COMPUT_STDDEV &
               COMPUT_A_B & COMPUT_D    & REDUCE_DATA   &
               COMPUT_N
where

 COMPUT_N ≡ [X,Y,len(X)],                 -- append N to <X,Y>

 REDUCE_DATA(X,Y,N) ≡ [SUMX, SUMY, SUMX2, SUMY2, SUMXY, N],
  where
```

$$SUMX \equiv \sum_{i=1}^{N} X_i, \quad SUMY \equiv \sum_{i=1}^{N} Y_i, \quad SUMX2 \equiv \sum_{i=1}^{N} (X_i * X_i),$$

$$SUMY2 \equiv \sum_{i=1}^{N} (Y_i * Y_i), \quad SUMXY \equiv \sum_{i=1}^{N} (X_i * Y_i),$$

```
                                          -- form basic 6-tuple of
                                          -- intermediate values

 COMPUT_D(SUMX, SUMY, SUMX2, SUMY2, SUMXY, N)  ≡
         [SUMX, SUMY, SUMX2, SUMY2, SUMXY, N,  D],
 where
  D ≡ N*SUMX,2 - SUMX*SUMX,
                                          -- append D to tuple

 COMPUT_A_B(SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D)  ≡
           [SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D,  A, B]
 where
  A ≡ (SUMX2*SUMY - SUMX*SUMXY)/D,
  B ≡ (N*SUMXY - SUMX*SUMY)/D,
                                          -- append A and B to tuple

 COMPUT_STDDEV(SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D, A, B)  ≡
              [SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D, A, B, STDDEV],
 where
  STDDEV ≡ sqrt(((SUMY2 - A*SUMY) - B*SUMXY)/(N-1)),
                                          -- append STDDEV to tuple

 COMPUT_CORR(SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D, A, B, STDDEV) ≡
            [SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D, A, B, STDDEV,
              CORR],
 where
  CORR ≡ B*B*D/(N*SUMY2 - SUMY - SUMY),
                                          -- append CORR to tuple

 COMPUT_F(SUMX, SUMY, SUMX2, SUMY2, SUMXY, N, D, A, B, STDDEV, CORR)
                               ≡ [A, B, STDDEV, CORR, F],
 where
  F ≡ B*(SUMXY - SUMX*SUMY/N)/CORR
                                          -- form final 5-tuple
```

In order for this function to work properly it must be applied to an argument consisting of two vectors, of the same length, of real numbers. This condition may be expressed as a well-formed condition on the argument (X,Y):

wf-condition:

$$len(X) = len(Y) \text{ and } \bigwedge_{i=1}^{N} (number(X_i) \text{ and } number(Y_i))$$

If wanted this condition could also be incorporated into the function definition itself, making the right-hand side a conditional expression yielding the result undefined if the condition is not fulfilled.

8.5 Numerical Integration

In a general applicable integration algorithm for (approximate) calculation of

$$\int_a^b F(x)\, dx$$

the user must be given the possibility to supply his own algorithm for the calculation of function values F(x) , and the integration algorithm must supply the 'skeleton' of the numerical integration.

Let us illustrate this by giving an algorithm for the trapezoidal integration scheme (with N sub-intervals)

$$\int_a^b F(x)\, dx \cong (F(a) + 2 * \sum_{i=1}^{N-1} F(xi) + F(b)) * (b-a)/2/n$$

$$INTEGRAL(a , b , N) \equiv (F(a) + \sum_{i=1}^{N-1} F(a + i*dx) + F(b)) * dx/2$$

where

$$dx \equiv (b - a)/N ,$$

$$F(x) \equiv \boxed{\text{user-supplied}}$$

The user must 'plug in' a functional expression that, when given a number valued object x, computes the corresponding function value F(x) .

If the functional language is implemented in an environment with 'subroutine'libraries, a special notation - a 'naming facility' - should be introduced to allow linking a pre-coded algorithm for F(x) to the INTEGRAL algorithm.

9.0 REFERENCES.

[Backus 78] J.Backus: "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs". CACM 21,8 (Aug. 1978), 613-641.

[Black 80] P.E.Black: "An Alternative to the '$' Object in Functional Programming Language", private commun., May 1980.

[Gram, Organick 80c] Chr. Gram, E.I.Organick: "Algebra of the Easy Functional Programming Language", Tech. Report ???????, Department of Computer Science, Univ. of Utah, Salt Lake City (July 1980).

[Iverson 62] K.E.Iverson: "A Programming Language". J. Wiley, New York (1962).

[Landin 64] P.J.Landin: "The Mechanical Evaluation of Expressions". Comp. J. 6,4 (1964), 308-320.