# UC Irvine
## ICS Technical Reports

**Title**
Programming in a viable data flow language

**Permalink**
https://escholarship.org/uc/item/0md35184

**Authors**
Arvind
Gostelow, Kim P.
Plouffe, Wil

**Publication Date**
1976

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Programming in a
Viable Data Flow Language

by

Arvind
Kim P. Gostelow
Wil Plouffe

89

August 1976
REVISED

Technical Report #89

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

This research is supported by NSF Grant MCS 76-12460
The UCI Dataflow Architecture Project.

## ABSTRACT

Current solid state technology suggests that future computers will be highly asynchronous machines comprising small intercommunicating processors, each processor contributing its effort to some part of the ongoing computation. The functional basis of such a machine demands a totally different foundation than that of current machines and languages. Data flow has been suggested as an alternative approach to vonNe mann type machines and associated sequential languages. A major cricism of data flow in the past has been the lack of a suitable higher-level programming language for coding programs. We feel that the sketch of a higher-level language presented here not only answers the criticism but also brings an asynchronous control structure into programming languages, as well as a strong theoretical basis for such properties as modularity and verifiability of programs. The paper contains a description of the Irvine Data Flow language ID. Three complete examples of programs in ID are included in section 3. For sake of reference, a BNF grammar for ID has been included in Appendix A.

# Programming in a Viable Data Flow Language

## 1. Why another language?

Future computers will be highly asynchronous machines comprising thousands of small intercommunicating processors, each processor contributing its effort to some part of the ongoing computation. The functional basis of such a machine demands a totally different foundation [GIMT74] than that of current machines and programming languages. The central problem is removing the inhibitions imposed by "classical von Neumann architecture", its emphasis on centralized control, and the non-functional un-modular programming languages it invites.

In place of current principles, we, along with others [Dennis73 a, GIMT74], feel that data flow can provide the needed foundation for these future machines. By data flow, we mean a language in which

(1) an operation proceeds when and only when all operands needed for that operation become available.

and (2) operations, at whatever level they might exist, are purely functional and produce no side-effects.

Based upon a theoretically sound low-level data flow language proposed by Dennis [Dennis73 a], we have developed a new machine-level interpreter capable of effectively exchanging blocks of processors for slices of time

[AG75,AG76]. This new interpreter operates by "unraveling" or "unfolding" programs (consider, for example, a loop) and asynchronously executing not only distinct statements, but also distinct initiations of the same statement. The unraveling interpreter does this in a straightforward and mechanical way with no preliminary analysis.

For example, the usual $\Theta(n^3)$ time complexity of matrix multiply on a standard machine is $\Theta(n)$ under the unraveling interpreter, and the usual program for Hoare's quicksort under this new interpreter executes in $\Theta(n)$ with a worst case time of $\Theta(n^2)$. The computations are short in time, but require more space, thus potentially utilizing the large numbers of processors which new technology can provide. We are reducing the length of the time critical path without changing the total computational flux.

Nevertheless, a major criticism of data flow in the past has been the lack of a suitable higher-level programming language for coding data flow programs. (Recently, Weng [Weng75] has proposed a programming language for data flow; his work, however, concentrated largely on other aspects of data flow.) We feel that the sketch of a higher-level data flow language as presented here not only answers the criticism, but also brings an asynchronous control structure into programming languages, as well as a strong theoretical basis for such properties as modularity and verifiability of programs. It is these aspects of data flow which are fundamental and which have been captured in the Irvine Data Flow language ID.

In the following, Section 2 presents much of the language ID by example program segments, while Section 3 gives three complete example procedures. Section 4 then reviews how asynchrony, modularity, and verifiability are reflected in ID.

## 2. The language ID

The Irvine Data Flow language ID is a textual and high-level version of Dennis' Data Flow (DDF) language [Dennis73a]. While a complete BNF syntax for ID appears in Appendix A, the language will be largely explained by the use of program segments, and only the most important syntactic entities are refered to in the discussion. The version of ID that appears here does not address all issues which must be resolved if ID were to become a programming facility. Instead, in our opinion, we have concentrated on the more important issues in the design of a data flow language. While such constructs as blocks, loops, and conditionals are discussed at some length, issues regarding typing of variables, selection of primitive operators, and input-output are not raised. In addition, more advanced topics such as the introduction of abstract data types have not yet been addressed. The new semantics, rather than the syntax of the language, has been our main focus. One of the design goals is to be able to translate ID into DDF which is known to be side-effect free and has easily understood semantics. Significant progress toward this goal has been made.

### 2.1 Programs and expressions

A program in ID is defined to be an expression; execution of a program means to evaluate the associated expression. Conceptually an expression is a box with labelled inputs and ordered outputs. The simplest expression may be a constant, a variable, a procedure call, or a selection of a value from a structure. Other expressions can be formed by the usual technique of joining two expressions by a binary operator or by forming a list of expressions. An expression list (Figure 1) is two or more expressions separated by a comma, where the comma imposes

the necessary ordering.

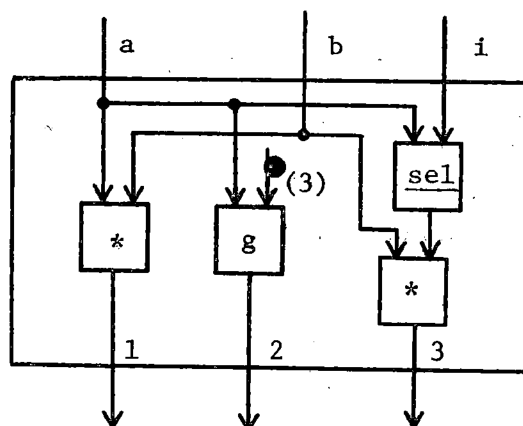


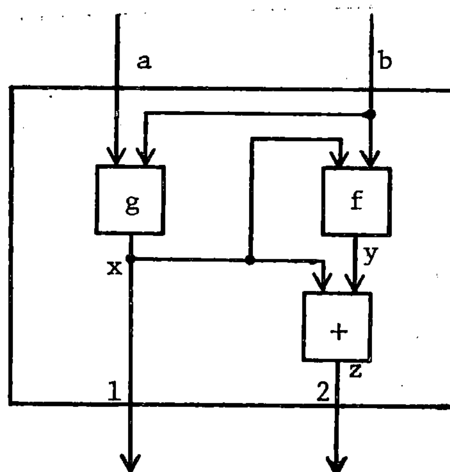

Figure 1

An expression list

The ordering of expressions in an expression list has no effect on the evaluation process. Evaluation of expressions is asynchronous, dependent solely on the availability of the operands.

When writing an expression list, it is sometimes convenient to be able to identify certain partial results. An assignment statement in ID represents the process of assigning names to the outputs of an expression. However, names must be chosen carefully so that no two partial results have the same name (the so-called "single-assignment" rule). ID, like DDF or pure LISP [McCarthy60], is essentially a language without variables, but when a name is assigned to a partial result it is usual to call that name a "variable". Since an assignment statement only names the output of an expression, it in itself is not an expression. The basic entity that represents an expression formed by using assignment statements is a block. A block contains assignment statements separated by semicolons, and a list of expressions to be returned as

the value of that block. Figure 2 gives an example of a block.




(x ← g(a,b); y ← f(x,b); z ← x+y return x,z)

Figure 2

An example of a block

Since variables represent partial results, it is always possible to write a program without them. For example, the block expression of Figure 2 can also be written as

(x ← g(a,b); y ← f(x,b) return x,x+y)

or as

(x ← g(a,b) return x,x+f(x,b))

or simply as a list of expressions

(g(a,b),g(a,b)+f(g(a,b),b))

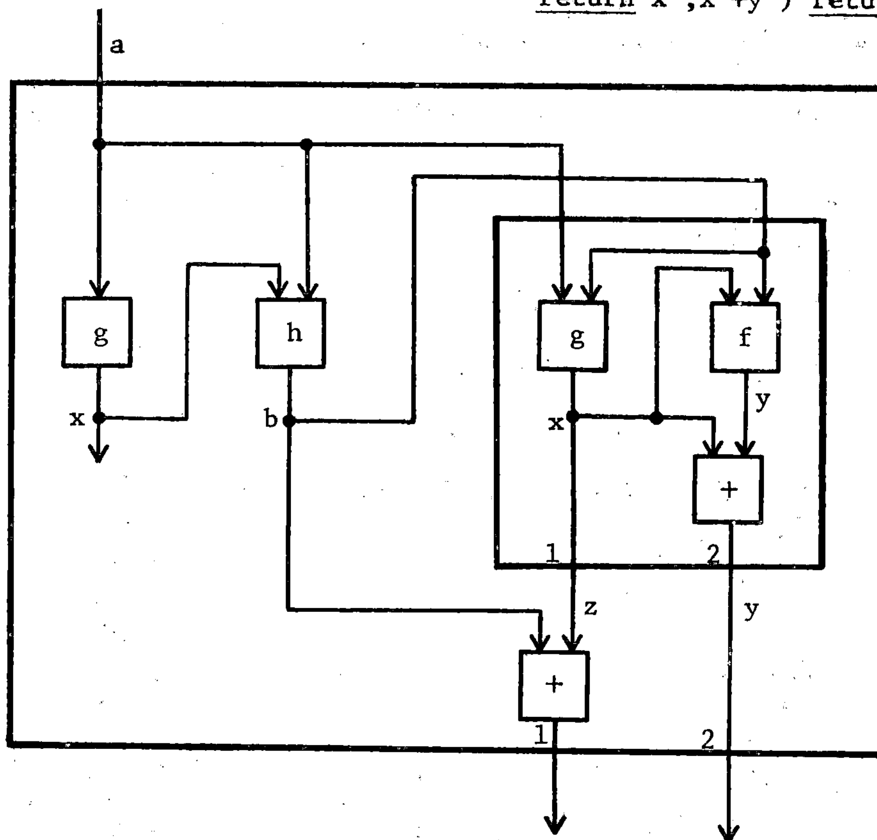
However, this last expression represents substantially more computation than the original block expression because g(a,b) would be evaluated three times as opposed to once.

Since variables are assigned values only once, the order of assignments is irrelevant. The meaning of a block remains unaltered even if statements separated by a semicolon are interchanged. The reader can verify this for the example in Figure 2. This lack of ordering captures the basically

asynchronous nature of data flow.

For purposes of modularity we introduce scoping rules whereby a variable that is assigned a value within a block is not visible outside that block. Hence, no side-effects can occur in outer blocks due to assignments to names within inner blocks. In the example of Figure 2, variables x,y, and z are internal to the block while the variables a and b must come from outside that block. This is because a and b have not been assigned any value within the block, while x,y, and z have been so assigned. Consider the example of Figure 3, where the block from Figure 2 has been enclosed in another block. The names x and y that occur in the inner block represent different partial results than the ones represented by x and y in the outer block. The block expression from Figure 3 is reproduced below with changed variable names to clarify the meaning of the block expression.

$$(x \leftarrow g(a);\ b \leftarrow h(x,a);\ z,y \leftarrow (x' \leftarrow g(a,b);\ y' \leftarrow f(x',b)\ \underline{\text{return}}\ x',x'+y')\ \underline{\text{return}}\ b+z,y)$$




$$(x \leftarrow g(a);\ b \leftarrow h(x,a);\ z,y \leftarrow (x \leftarrow g(a,b);\ y \leftarrow f(x,b)\ \underline{\text{return}}\ x,x+y)\ \underline{\text{return}}\ b+z,y)$$

Figure 3

A block expression and scoping rules

The evaluation of this block will proceed whenever a value for a becomes available, and the inner block will execute only after both a and b become available. Even though the same name can appear at different lexical levels without violating the single-assignment rule, a name can never appear twice on the left hand side of assignment statements in the same block. For example, (x ← y; x ← g(a) return x) will not return y, nor g(a), nor either of them nondeterministically. It is an invalid expression.

Hence, a block (actually any expression) can be inserted anywhere in another expression without causing any kind of side-effects in the outer expression. It must be remembered, however, that due to scoping rules, evaluation of a block does depend upon the context.

## 2.2 Loop expressions

Like most conventional languages, looping constructs are essential for writing interesting programs in ID. All looping constructs in ID are expressions consisting of four parts: an initial assignment part, a predicate to decide conditions for further iteration, a list of assignment statements, and a list of expressions to be returned as the value of the loop. Consider the example in Figure 4. It represents a program to compute the smallest i such that the sum of all the integers from 1 to i exceeds some number s.

```
1    (initial i ← 1;
2             sum ← 1
3    while sum≤s do
4          i ← i+1;
5          sum ← sum+i
6    return i)
```

Figure 4

A loop expression to find the smallest i such that $\sum_{j=1}^{i} j > s$

Since an initial value for i and sum is needed to start the loop, both of these are specified in the <u>initial</u> part of the loop. The statements in the loop also show that i and sum are updated at each iteration. We had claimed while discussing blocks that two assignments can be interchanged without altering the meaning of the program. This raises the issue of whether the old value of i or the updated value of i is to be used in

sum ← sum + i          (line 5 of the loop expression in Figure 4).

If the meaning of the program is to remain unchanged by reordering statements 4 and 5, then the i on the right hand side of both assignment statements must carry the same value. Hence i on the right hand side can only represent the present or the un-updated value. Figure 5 gives an equivalent pseudo-Algol program.

```
1      i ← 1; sum ← 1;
2      while sum ≤ s do
3      begin
4        i' ← i+1;
5        sum' ← sum+i;
6        i ← i';
7        sum ← sum'
8      end;
```

Figure 5

A pseudo-Algol program representing the loop expression of Figure 4

An alert reader is certainly going to complain about the violation of the single-assignment rule. Even though the semantics of a loop expression may be clear, it would appear that multiple values are being assigned to both variables i and sum. A clearer and more correct picture of what is taking place in a loop is quite different. One should think of the new

variables (i.e. partial results) i and sum as being created at each iteration. Rather than using different names for variables generated at each iteration, we can treat i and sum as generic names representing all partial results (i.e. i+1 and sum+i). This picture is more correct because in ID (as well as in DDF) i and sum will not represent two memory cells whose values are updated at each iteration. Instead, each iteration generates a new token with the new value which can be used in the next iteration according to specific rules.

Like a block, or a list of expressions, a loop expression may begin execution any time after it receives all required inputs. In Figure 4, the only required input is s. The assignment statements in the initial part look syntactically like all other assignment statements, however, their meaning is quite different. The names appearing on the left side of initial assignments have meaning only inside the loop expression, while names appearing on the right side of initial assignments must come from outside the loop expression. If x← f(x) appears in the initial assignment part of a loop expression, it is completely valid and means to take the value of a variable named x from outside, and use function f applied to that value as the initial value for a variable named x inside the loop expression. Figure 6 gives an example of a loop expression nested in a block.

```
1   (x ← g(a);
2    y ← a*(initial x ← f(x)
3           for i from 1 to n do
4                y ← x+q;
5                x ← f(x)+y
6           return x)
7    return y)
```

Figure 6

A loop expression nested in a block

The meaning of this is clear if we replace the x in line 1 and the x inside f(x) in line 2 by x'. The DDF equivalent of this program is given in Figure 7. Note that q is a loop constant and both x' and q are externals to the loop expressions. For the outermost block, however, a and q are the external inputs. (Implementation considerations might imply distinct forms of handling loop constants.)

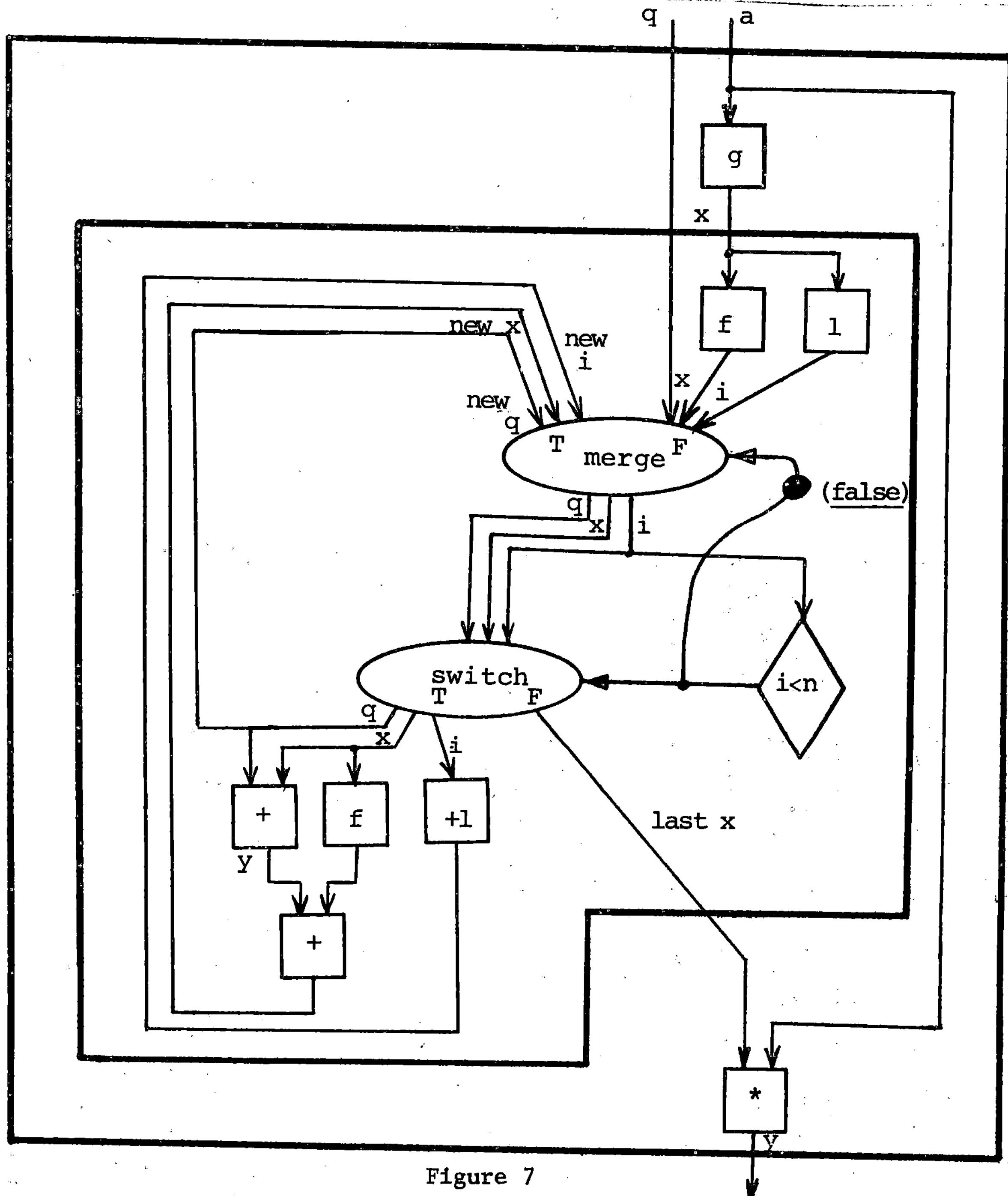



**Figure 7**

A DDF program equivalent to the expression of Figure 6

## 2.3 Conditionals

ID allows both conditional expressions and conditional statements. Both if-then-else and case type conditions are permitted. Here we will only discuss the if-then-else constructs. The basic restriction in if-expressions is that the else clause must be present. Consider the following assignment statement.

x,y ← (if p(a) then f(a),g(a) else f(b),g(b))

The meaning of this expression is obvious. However, all of the following statements are semantically invalid and will be detected at compile time.

```
x,y ← (if p(a) then f(a),g(a))                 }
x,y ← (if p(a) then f(a) else f(b),g(b))       }  invalid
  x ← (if p(a) then f(a),g(a) else f(b))       }
```

Weng has discussed the semantics of conditional statements in his thesis [Weng75] and has pointed out that a variable assigned on one side of an if must be assigned on both sides of the if. That is

(if p(a) then x ← f(a) else y ← g(b))

is an invalid statement. We agree, but have introduced some default rules for conditional statements that have proven very useful in writing loop expressions. The default rule essentially states that if a variable is being assigned on only one side of an if, then it remains unchanged on the other side of the if.

```
1     (initial x←0; y←0
2      while  x < c do
3          (if x = y then x←f(x) else y←g(y))
4     return x,y)
```

Figure 8
Default values in an if statement

In ID the meaning of line 3 is

(<u>if</u> x ≠ y <u>then</u> x ← f(x); y ← y <u>else</u> x ← x; y ← g(y)).

It is obvious that the default rule will not be meaningful unless the variables being assigned to (i.e. x and y) have previous values.

## 2.4 Procedure calls

The meaning of a procedure call is absolutely straightforward and is in strict accordance with the rules of DDF. A procedure is an expression with an input-output specification, and the arguments are always passed by value. If the procedure can be compiled, it is guaranteed to be well-behaved [Dennis73a] and thus exhibits no side-effects (i.e. memory). Two invocations of a procedure can never interfere with one another, and since arguments are passed to a procedure only after they have been evaluated, there is no ambiguity about the context in which arguments are evaluated. Unlike Algol-60, the scoping of names does not extend inside the procedure body; that is, all the outside information used by a procedure must be explicitly passed as input arguments.

## 3. Data structures and programming examples

Up to this point, all examples have used variables or constants with simple values, e.g. boolean, integer, etc. The base language DDF is defined so that it is capable of dealing with structured values (these structures being a generalization of the list structures used in pure LISP [McCarthy60]). A structured value, or structure, is represented by a tree with selectors for tracing a path from the root node to any other node. There are exactly two operations allowed on structures, append and select, and one distinguished value Λ (the empty structure). The append operator--append (structure, selector, value)--is used to add a new subtree to a node, resulting in a <u>new</u> structure (the original is not modified) with the specified value and selector replacing the original subtree (if it exists).

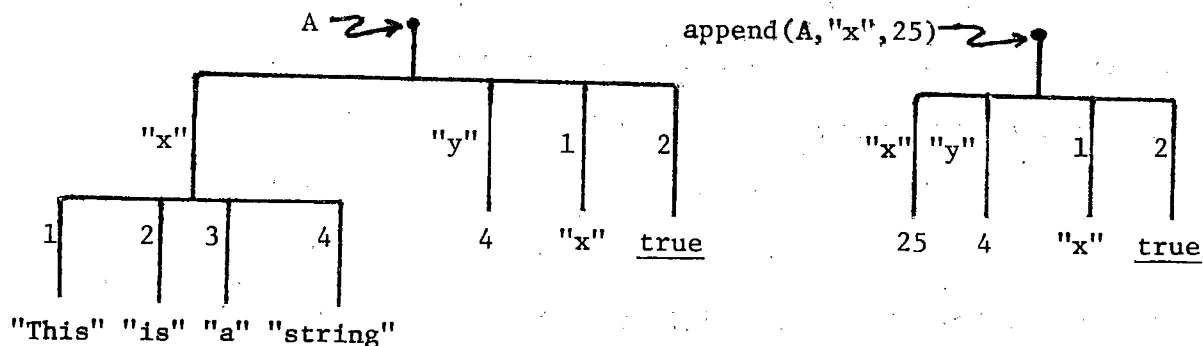



Figure 9
An append operation

The select operator--select (structure, selector)--is used to retrieve a subtree from a node.

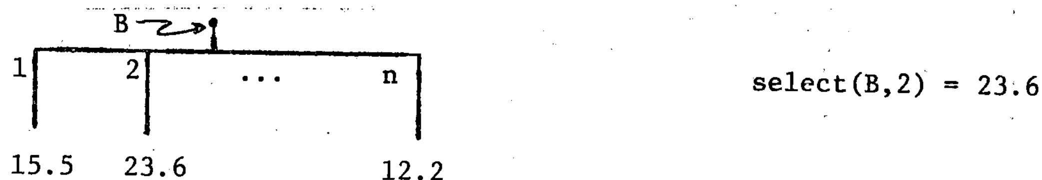



Figure 10
A select operation

The actions of these two operations may be summarized by

```
select(append(A,s,v),x) ≡ if s=x then v
                              else select(A,x)
```

and

```
select(Λ,x) ≡ Λ.
```

The syntax used to invoke these operations is of the form

```
structure + [selector]value
```

which corresponds to

```
append(structure, selector, value)
```

and

```
structure[selector]
```

which corresponds to

```
select(structure, selector).
```

An additional syntax is available only within loops:

```
structure[selector] ← value
```

which is equivalent to

```
structure ← structure + [selector]value
```

and which corresponds to

```
structure ← append(structure, selector, value).
```

Thus, a program which works with arrays in Algol may appear very similar when written in ID.

Some examples of structure expressions and their semantics follow:

```
A["cat"] ≡ select(A,"cat")
A + ["columns"]n ≡ append(A,"columns",n)
A + [i] ≡ append(A,i,Λ)
A[i] ← v ≡ append(A,i,v)
A + [i,j]x ≡ A + [i](A[i] + [j]x)
           ≡ append(A,i,append(select(a,i),j,x))
A + [i]u + [k]v ≡ append(append(A,i,u),k,v)
```

The procedures which follow use fairly simple algorithms, but they illustrate the power of ID. The first procedure is Hoare's quicksort (Figure 11). Each algorithm described here has a pseudo-Algol equivalent given in Appendix B.

```
procedure Quicksort (a,n)
  (ℓ←n÷2;
   below,j,above←(initial below←Λ; j←0;
                          above←Λ; k←0
                  for i from 1 to n do
                     (if i≠ℓ
                        then (if a[i]≤a[ℓ]
                                 then below[j+1]←a[i];
                                      j←j+1
                                 else above[k+1]←a[i];
                                      k←k+1))
                  return   (if j>1 then Quicksort(below,j)
                                   else below), j,
                           (if k>1 then Quicksort(above,k)
                                   else above))
   return(initial sort←below+[j+1]a[ℓ]
        for i from j+2 to n do
            sort[i]←above[i-j-1]
        return sort))
```

Figure 11

ID version of Hoare's quicksort

The time complexity for quicksort has an average of $\mathcal{O}(n \log n)$ and a worst case behavior of $\mathcal{O}(n^2)$ for one processor. The ID counterpart, when compiled in DDF and executed under the unraveling interpreter, has an average behavior of $\mathcal{O}(n)$ and a worst case behavior of $\mathcal{O}(n^2)$, but requires an average of $\mathcal{O}(n)$ processors. The time complexity is decreased because of the possibility of executing the recursive calls in parallel.

The second algorithm (Figure 12) is matrix multiplication

$$(c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}).$$

```
procedure multiply (a,b,ℓ,m,n)
    (initial c←Λ
     for i from 1 to ℓ do
         c[i]←(initial d←Λ
               for j from 1 to n do
                   d[j]←(initial s←0
                         for k from 1 to m do
                             s←s+a[i,k]*b[k,j]
                         return s)
               return d)
     return c)
```

Figure 12
ID version of matrix multiply

The ID program of Figure 12 executes in $\Theta(\ell+m+n)$ time utilizing $\Theta(\ell mn)$ processors. (It is possible for the unraveling interpreter to process the DDF translation so that it executes in $\Theta(\ell+m+n)$ time but requiring only $\Theta(\ell n)$ processors.) The unraveling interpreter will try to execute all of the multiplications and $\ell n$ of the additions in parallel, thus reducing the usual time complexity from $\Theta(\ell mn)$ to $\Theta(\ell+m+n)$.

The final example is a subroutine used to solve the knight's tour problem [Wirth76] and is given in Figure 13.

```
procedure try (x,y,i,h,a,b,n,s)
! (x,y) represents the current position of the Knight,
   i represents the ith move of the knight!
 (initial h←h; k←1
  repeat u←x+a[k] ; v←y+b[k];
        (if in(u,s) and in(v,s)
         then (if h[u,v]=0
               then (if i<n^2
                     then q,h←try(u,v,i+1,h+[u,v]i,a,b,n,s)
                     else  q←true;h[u,v]←i)
               else q←false)
         else q←false);
         k←k+1
  until q or k=8
  return q,h)
```

Figure 13

ID version of procedure
try of Wirth's knight's tour

4. ID and important issues in programming languages

Programming is problem solving, and thus the only true methodologies we have for problem solving,abstraction and decomposition into subproblems, find themselves intimately involved in the activity of programming. It is primarily through aiding the "intellectual management" of a problem that abstraction and decomposition play a role, and facilitating the use of these methodologies should be a primary goal in the design of any programming language. This is certainly true when we recognize, in the words of Wirth, "...the strong and undeniable influence that our language exerts on our ways of thinking, and in fact defines and delimits the abstract space in which we can formulate-give form to-our thoughts." [Wirth74]

The notion of modularity plays a strong role by localizing the effects that a unit of computation have and by encouraging structure in programming. Procedures in ID have all the properties of modules, such as the ability to be compiled independently, evaluation which is context independent (except through the parameters), and no side effects. We are considering the following construct where any expression in ID can be made into an unnamed module:

$$(\underline{\text{initial}}\ x_1 \leftarrow y_1; x_2 \leftarrow y_2; \ldots x_n \leftarrow y_n\ \underline{\text{module}}\ \text{expression})$$

where $x_i$ is a variable external to the expression and $y_i$ is the value to be used for evaluation. The semantics of the initial part above are identical to the semantics of the initial part of a loop expression (section 2.2). In effect the $x_i$ serve as the formal parameters and the $y_i$ the calling parameters.

The above was largely concerned with one half of the programming problem-composition of programs. Verification remains the other half. With ID and its emphasis on modularity and locality of effect, correctness

should be easier to determine than with other programming languages. In the terminology of the axiomatic approach, an absence of side-effects implies invariants are more easily determined. In the functional approach, we have utilized fixpoints[Kahn74, Manna74] to discover relations between different data flow interpreters and to verify that two particular interpreters produce the same results, albeit in very different ways [AG76a]. The ease with which the theory was applied to data flow and yielded results leads us to believe that the principles evidence by data flow and ID can only aid correctness work.

C.A. Petri* has remarked that asynchrony has moved over the years in stages from early curiosity, to a nice idea for speeding up computation, then to a good thing if it were possible, and now to a fundamental concept for operating systems, and that in the future asynchrony will be recognized as fundamental for preserving the structure of computation. Certainly the history is clear. The prediction is also certain to become reality when one recognizes sequentiality and synchrony for the unnatural and arbitrary constraints they impose. For example, it is now universally accepted that an operating system is considerably easier to comprehend and to write when viewed as a system of asynchronous interacting processes.

So why have asynchronous languages not been developed? Consider Wirth's comment again--language influences our way of thinking. Most language designers would agree with his remark, but would not recognize the constraints placed on their thinking by the classical von Neumann machine. Machine architects have not volunteered a real asynchronous machine for the language designers, and language designers have never demanded one.

---

* Keynote address at First International Conference on Petri Nets and Related Methods, MIT, July 1-3, 1975.

## 5. Acknowledgements

We wish to thank very much, Valerie Isaac, Marion Kaufman, and Shirley Rasmussen for their help in preparing this report under less than optimal conditions.

## 6. References


[AG75] Arvind and K.P. Gostelow, A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture, TR 72, Department of Information and Computer Science, University of California, Irvine, November 1975.

[AG76] Arvind and K.P. Gostelow, A Computer Capable of Exchanging Processing Elements for Time, TR 77, Department of Information and Computer Science, University of California, Irvine, January 1976.

[AG76a] Arvind and K.P. Gostelow, The Relationship Between the Semantics of a Data Flow Language According to Two Different Interpretive Schemes, Department of Information and Computer Science, University of California, Irvine, August 1976.

[Dennis73] Dennis, J. "Modularity" in Advanced Course on Software Engineering - Lecture Notes in Economics and Mathematical Systems - Vol 81, Springer-Verlag, pp. 128-182.

[Dennis73a] Dennis, J.B., First Version of a Data Flow Procedure Language, MAC TM 61 (originally published as Computation Structures Group Memo 93, Nov 1973), Project MAC, MIT, May 1975.

[GIMT74] Glushkov, V.M., M.B. Ignatyev, V.A. Myasnikov and V.A. Torgashev, Recursive Machines and Computing Technology, Information Processing 74, North-Holland Publishing Company, Stockholm, Aug 1974, (pp 65-70).

[Kahn74] Kahn, G.,"The Semantics of a Simple Language for Parallel Programming",Proceedings IFIP Congress Conference, 1974.

[Kosinski73] Kosinski, P.R., A Data Flow Language for Operating Systems Programming, Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices Vol. 8, No. 9, September 1973. (pp 89-94)

[Manna74] Manna, Z.,Mathematical Theory of Computation, McGraw-Hill, 1974.

[McCarthy60] McCarthy, J.,"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", Comm. ACM, April 1960. (pp 184-195).

[Weng75] Weng, Kung-Song, Stream-Oriented Computation in Recursive Data Flow Schemas M.S. Thesis (MAC Technical Memorandum 68), Department of Electrical Engineering and Computer Science, MIT, October 1975.

[Wirth74] Wirth, N.,"On the Composition of Well-Structured Programs", ACM Computing Surveys (Special Issue:Programming) Vol 6, No. 4, Dec 1974. (pp 247-260)

[Wirth76] Wirth, N.,Algorithms + Data Structures = Programs, Prentice-Hall, 1976. (pp 140)

# Appendix A

The following is a simple BNF grammar for ID.

1. Program

```
1.1  <program> ::= <expression> □
```

2. Expressions

```
2.1  <expression> ::= <simple expression>
                | <unary operator><simple expression>
                | <simple expression><binary operator><simple expression>
                | <structure expression>

2.2  <simple expression> ::= (<expression list>)
                | <block expression>
                | <loop expression>
                | <if expression>
                | <case expression>
                | <procedure call>
                | <variable>
                | <constant>
                | Λ
                | <simple expression>[<selector>]

2.3  <expression list> ::= <expression>
                | <expression list>,<expression>

2.4  <structure expression> ::= [<append selector>]<simple expression>
                | <simple expression>+[<append selector>]
                | <simple expression>+[<append selector>]<simple expression>
                | <structure expression>+[<append selector>]
                | <structure expression>+[<append selector>]
                                                   <structure expression>

2.5  <block expression> ::= (<statement list><return clause>)

2.6  <loop expression> ::= (initial <statement list><loop construct>
                                                        <return clause>)

2.7  <if expression> ::= (if <Boolean expression> then <expression list>
                                                    else <expression list>)

2.8  <case expression> ::= (case <case list-expression>
                                else <expression list>)

2.9  <case list-expression> ::= :<Boolean expression>: <expression list>
                | <case list-expression> :<Boolean expression>:
                                                   <expression list>
```

```
2.10  <procedure call> ::= <procedure name>(<expression list>)
```

3. Statements

```
3.1  <statement> ::= <assignment statement>
                  | <procedure declaration>

3.2  <statement list> ::= <statement>
                  | <statement list>; <statement>

3.3  <assignment statement> ::= <variable list> ← <expression list>
                  | <if statement>
                  | <case statement>
                  | <null>

3.4  <assignment statement list> ::= <assignment statement>
                  | <assignment statement list>; <assignment statement>

3.5  <variable list> ::= <variable>
                  | <variable>[<append selector>]
                  | <variable list>, <variable>
                  | <variable list>, <variable>[<append selector>]

3.6  <if statement> ::= (if <Boolean expression>
                           then <assignment statement list>
                           else <assignment statement list>)
                  | (if <Boolean expression> then <assignment statement list>)

3.7  <case statement> ::= (case <case list-statement>
                               else <assignment statement list>)

3.8  <case list-statement> ::= :<Boolean expression>:
                                      <assignment statement list>
                  | <case list-statement> :<Boolean expression>:
                                            <assignment statement list>
```

4. Miscellaneous Important Productions

```
4.1  <variable> ::= <identifier>

4.2  <procedure name> ::= <identifier>

4.3  <return clause> ::= return <expression list>

4.4  <selector> ::= <expression>
                  | <selector>,<expression>

4.5  <append selector> ::= <expression>
                  | <append selector>,<expression>
                  | <append selector>][<expression>
```

## 5. Loops

```
5.1  <loop construct> ::= while <Boolean expression> do
                                    <assignment statement list>
              | repeat <assignment statement list> until
                              <Boolean expression>
              | for <variable> from <expression> <for construct> do
                              <assignment statement list>
              | for <variable> in <expression list> do
                              <assignment statement list>

5.2  <for construct> ::= to <expression> step <expression>
                              while <Boolean expression>
              | to <expression> step <expression>
              | to <expression> while <expression>
              | to <expression>
              | step <expression> while <Boolean expression>
              | while <Boolean expression>
```

## 6. Procedures

```
6.1  <procedure declaration> ::= (<procedure declaration>)
              | procedure <procedure name> (<formal parameters>)
                         <expression>

6.2  <formal parameters> ::= <variable>
              | <formal parameters>,<variable>
```

## Appendix B

The following are pseudo-Algol translations of the ID procedures presented in Section 3.

```
procedure Quicksort (a,n,sort)
    array a[1:n],above[1:n],below[1:n],sort[1:n],temp[1:n];
    integer i,j,k,m,n; value a,n;
    begin
        j:=k:=0;
        m:=n÷2;
        for i:=1 step 1 until n do
            if i≠m
                then begin
                    if a[i]≤a[m]
                        then begin j:=j+1; below[j]:=a[i] end
                        else begin k:=k+1; above[k]:=a[i] end
                end
        if j>1 then Quicksort(below,j,sort);
        if k>1 then Quicksort(above,k,temp);
        sort[j+1]:=a[m];
        for i:=j+2 step 1 until n do
            sort[i]:=temp[i-j-1];
    end
```

Pseudo-Algol for quicksort

```
procedure multiply (a,b,c,ℓ,m,n)
    array a[1:ℓ,1:m],b[1:m,1:n],c[1:ℓ,1:n];
    integer i,j,k,ℓ,m,n; real s; value a,b,c,ℓ,m,n;
    for i:=1 step 1 until ℓ do
        for j:=1 step 1 until n do
            begin
                s:=0;
                for k:=1 step 1 until m do
                    s:=s+a[i,k]*b[k,j];
                c[i,j]:=s;
            end
```

Pseudo- Algol for matrix multiply

```
procedure try (i,x,y,q)
    boolean q,q1; integer i,k,u,v,x,y; value i,x,y;
    begin
        k:=0;
        repeat
            k:=k+1; q1:=false;
            u:=x+a[k]; v:=y+b[k];
            if in(u,s) and in(v,s)
                then if h[u,v]=0
                    then begin
                        h[u,v]=i;
                        if i<nsq
                            then begin
                                try(i+1,u,v,q1);
                                if not(q1) then h[u,v]=0
                            end
                            else q1:=true
                    end
        until q1 or k=8;
        q:=q1
    end
```

Pseudo-Algol for try