2-1-1982

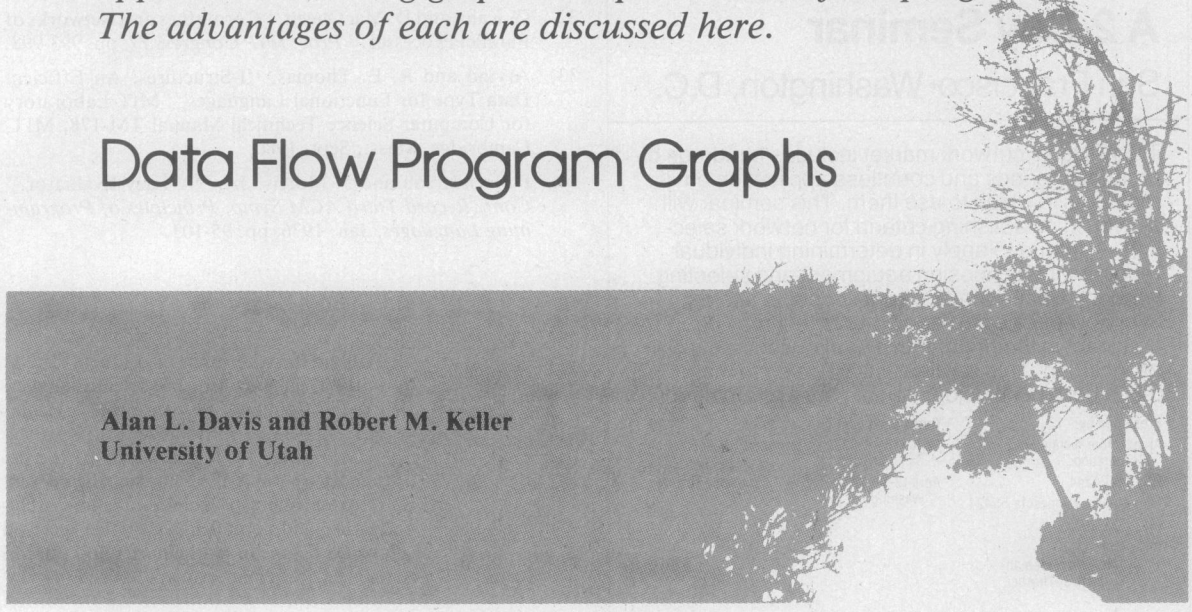
Data Flow Program Graphs

Alan L. Davis
University of Utah

Robert M. Keller
Harvey Mudd College

Token models and structure models are two basic approaches to using graphs to represent data flow programs. The advantages of each are discussed here.

Data Flow Program Graphs



Alan L. Davis and Robert M. Keller
University of Utah

Data flow languages form a subclass of the languages which are based primarily upon function application (i.e., applicative languages). By *data flow language* we mean any applicative language based entirely upon the notion of data flowing from one function entity to another or any language that directly supports such flowing. This flow concept gives data flow languages the advantage of allowing program definitions to be represented exclusively by graphs. Graphical representations and their applications are the subject of this article.

Applicative languages provide the benefits of extreme modularity, in that the function of each of several subprograms that execute concurrently can be understood *in vacuo*. Therefore, the programmer need not assimilate a great deal of information about the environment of the subprogram in order to understand it. In these languages, there is no way to express constructs that produce global side-effects. This decoupling of the meaning of individual subprograms also makes possible a similar decoupling of their execution. Thus, when represented graphically, subprograms that look independent can be executed independently and, therefore, concurrently.

By contrast, concurrent programs written in more conventional assignment-based languages cannot always be understood *in vacuo,* since it is often necessary to understand complex sequences of interactions between a subprogram and its environment in order to understand the meaning of the subprogram itself. This is not to say that data flow subprograms cannot interact with their environments in specialized ways, but that it is possible to define a subprogram's meaning without appealing to those interactions.

There are many reasons for describing data flow languages in graphical representations, including the following:

(1) Data flow languages sequence program actions by a simple *data availability firing rule:* When a node's arguments are available, it is said to be firable. The function associated with a firable node can be fired, i.e., applied to is arguments, which are thereby absorbed. After firing, the node's results are sent to other functions, which need these results as their arguments.

A mental image of this behavior is suggested by representing the program as a directed graph in which each node represents a function and each (directed) arc a conceptual medium over which data items flow. Phantom nodes, drawn with dashed lines, indicate points at which the program communicates with its environment by either receiving data from it or sending data to it.

(2) Data flow programs are easily *composable* into larger programs. A phantom node representing output of one program can be spliced to a phantom node representing input to another. The phantom nodes can then be deleted, as shown in Figure 1.

There are two distinct differences between this type of composability and the splicing of two flowcharts. First, spliced data flow graphs represent *all* information needed at the interface. With flowcharts, the connectivity among variables is not represented by splicing. Second, flowchart splicing represents a one-time passing of control from one component to the next. With data flow graphs, splicing can indicate information that crosses from one component to the next; this motion is distributed over the entire lifetime of the computation.

0018-9162/82/0200-0026$00.75 © 1982 IEEE

(3) Data flow programs avoid prescribing the specific execution order inherent to assignment-based programs. Instead, they prescribe only essential data *dependencies.* A dependency is defined as the dependence of the data at an output arc of a node on the data at the input arcs of the node. (For some functions, the dependency might be only apparent.) The lack of a path from one arc to another indicates that data flowing on those arcs can be produced independently. Hence, the functions producing those data can be executed concurrently. Thus, graphs can be used to present an intuitive view of the potential concurrency in the execution of the program.

(4) Graphs can be used to attribute a formal meaning to a program. This meaning can take the form of an *operational* definition or a *functional* one. The former defines a permissible sequence of operations that take place when the program is executed. The latter describes a single function represented by the program and is independent of any particular execution model.

This article explores the utility of graphical representations for data flow programs, including the possibility and advantages of dispensing entirely with the text and viewing the graph itself as the program. This suggests a programming style in which the user deals with graphs as the primary representation in programming, editing, and execution. In this context, human engineering rather than concurrent execution becomes the motivation for investigating data flow program graphs.

The following section elaborates on the meaning of graphs as functional programs. The discussion focuses on the two prevailing models for data flow representation: the token model and the structure model. The terms classify data flow languages that developed along different lines, each with certain implementation subtleties.

Token models

The term *token* is a shortening of *token-stream,* which more accurately describes the behavior of these models. Data is always viewed as flowing on arcs from one node to another in a stream of discrete tokens. Tokens are considered carriers or instantiations of data objects. Each object is representable by a finite encoding.

When a node is labeled with a scalar function, such as + or *, it is understood that the function is repeated as tokens arrive at its inputs. Each repetition produces a token at its output. For example, suppose that we use a token model to interpret the graph in Figure 2. This graph defines a repeated computation of the polynomial function of X: $X^2-2*X+3$ for a sequence of values of X. The fanout of an arc from a node, such as the phantom node X, denotes the conceptual replication of tokens leaving that node. A node marked with a constant value is assumed to regenerate that value as often as it is needed by nodes to which it is input.

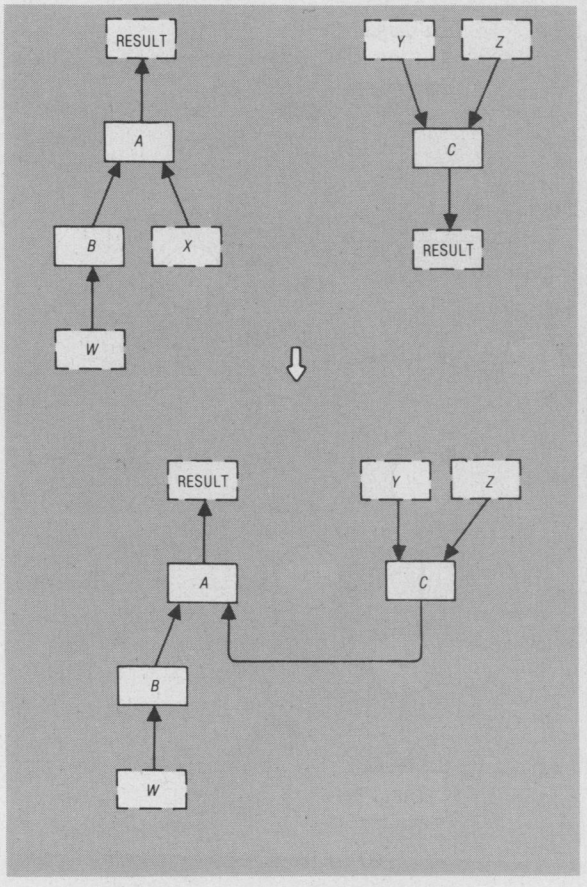


Figure 1. Splicing of two data flow graphs.

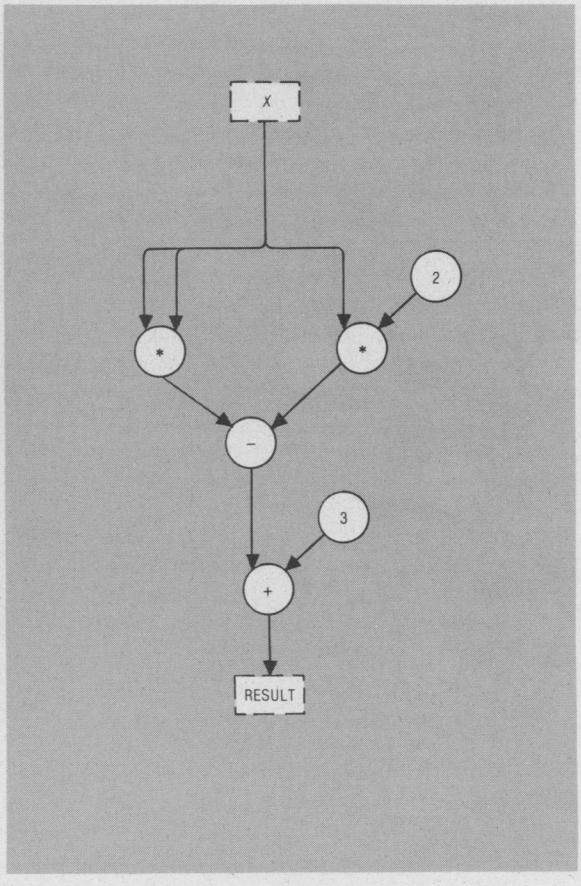


Figure 2. Data flow graph for $X^2-2*X+3$.

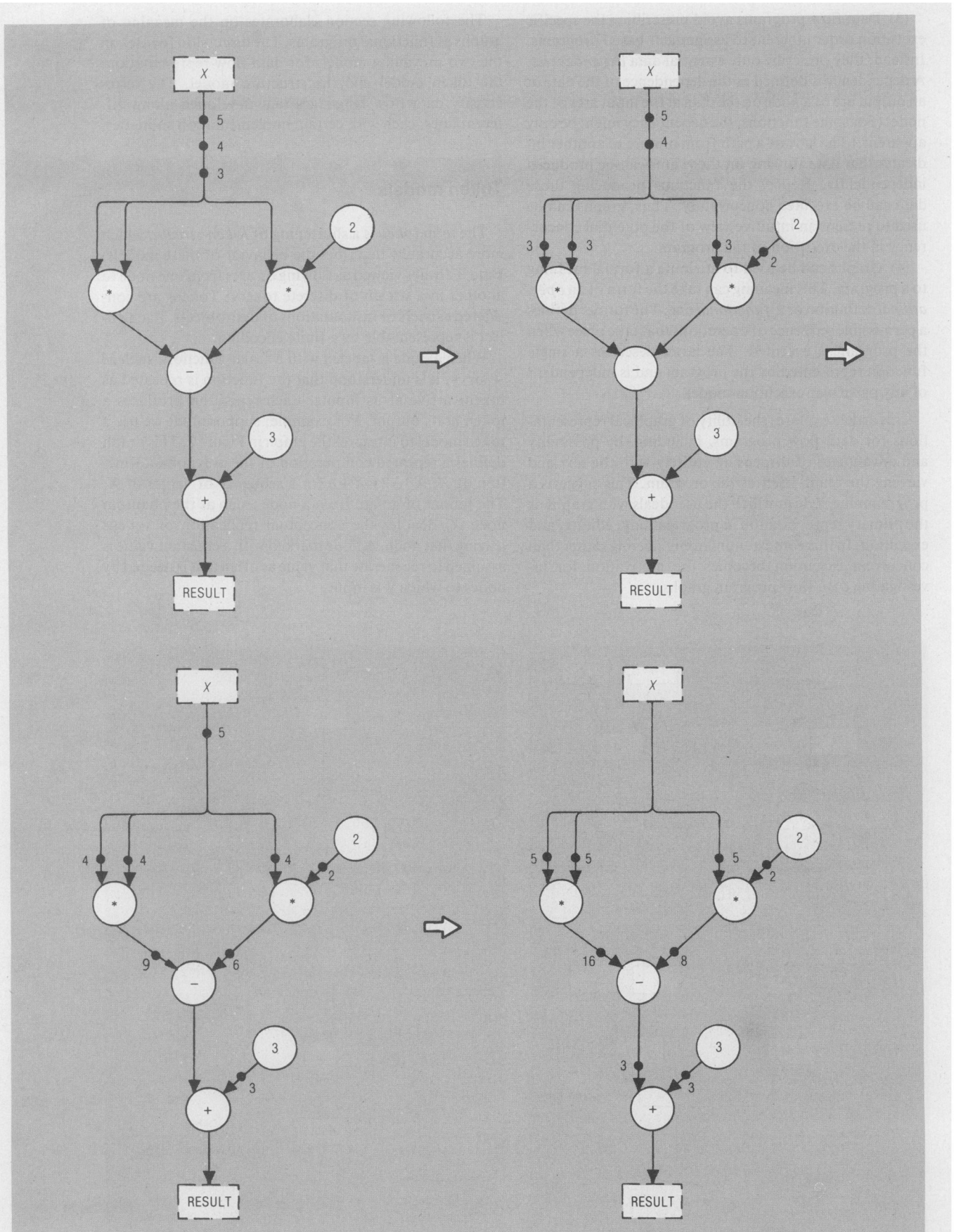


Figure 3. Pipelined graph computation.

The results of node operations correspond to the inputs in first-in-first-out order. The operation is usually, but not necessarily, performed on those inputs in the same order. Furthermore, each node corresponds to a function, and no fan-in of arcs is allowed—there is no opportunity for tokens to be interleaved arbitrarily at an arc. It follows, therefore, that data flow graphs ensure *determinate* execution. That is, the output of any program or subprogram for a given input is guaranteed to be well defined and independent of system timing. This has been clearly demonstrated for several data flow models.[1-5]

Determinacy also imples an absence of side-effects, which are apt to be present in conventional read and write operations. Such operations often require extra concurrency-reducing synchronization to prevent time-dependent errors. By contrast, the only errors possible in data flow programs are those due to an improper functional formulation of the solution to a problem; these are always reproducible, due to the feature of determinacy. A more thorough discussion of data flow errors is given elsewhere.[6]

In Figure 2, no arc connects the two $*$ operators. This implies that there is no data dependency between the two nodes; that is, the node functions can be computed concurrently. The lack of data dependency between the multiply operators in Figure 2 is sometimes called *horizontal,* or *spatial,* concurrency. This contrasts with *temporal* concurrency, or pipelining, which exists among computations corresponding to several generations of input tokens. A brief scenario of both types of concurrency is illustrated by the sequence of snapshots in Figure 3.

Conditional constructions in data flow programs achieve selective routing of data tokens among nodes. Boolean or index-valued tokens can be produced by a node that performs some decision function. Figure 4 shows the *selector* and the *distributor,* two nodes used in conditional constructs. In the case of a selector, a token is first absorbed from the horizontal input. The value of that token, either true or false, determines from which of the two vertical inputs the next token will be absorbed; any token on the other input remains there until selected. The firing of a selector is a two-phase process, since the value at the horizontal input must be known before the corresponding vertical input can be selected. In the case of the distributor, a token is absorbed from the vertical input and passed to one of the vertical ouputs. Again, the choice of output depends on the value of the token at the horizontal input.

Generalizations of the selector and distributor are easily devised:

- selection (or distribution) is based on a set of integer or other scalar values instead of on booleans, or
- the vertical arcs are replaced by bundles of arcs so that tokens pass through in parallel.

Iteration can be achieved through cyclic data flow graphs. The body of the iteration is initially activated by a token that arrives on the input of the graph. The body subgraph produces a new token, which is cycled back on a feedback path until a certain condition is satisfied. An example of an iterative graph is shown in Figure 5, which illustrates Newton's method to find the roots of a function. Node f could be replaced by the graph shown in Figure 2. A similar graph could replace node f' to compute the derivative $2*X-2$. An execution scenario for Figure 5 is as follows:

(1) The program is started by introducing a real-number token at the output of phantom node X.

(2) The selector is now firable, as a *true* token exists on its horizontal input in the initial state shown. When the selector fires, the token from X passes to the two $-$ operators and to the boxes that calculate $f(x)$ and $f'(x)$.

(3) Neither of the $-$ operations can fire yet, as their right-operand tokens are not available. Nodes $f(x)$ and $f'(x)$ can fire concurrently to produce their output values and absorb their input values. At this point, the $\div$, $-$, abs, and $<$ operations fire, in that order.

(4) The ouput of the $<$ node is a boolean value that indicates whether or not the new and old approximations have converged sufficiently. If they have, a *true* token is produced by $<$. This causes the feedback-in value to be distributed and the approximation to be passed through as the result of the iteration. At this point, a *true* token has been regenerated at the horizontal input to the selector, putting the graph in its original state. Thus, the program is reusable for a subsequent input token.

(5) If the $<$ node produces *false,* the feedback-in token is passed through the distributor and selector and becomes the next feedback-out token, to be used in the next iteration.

In the example in Figure 5, very little pipelining can take place because of the use of the selector function at the input. This selector requires each set of iterations to be com-

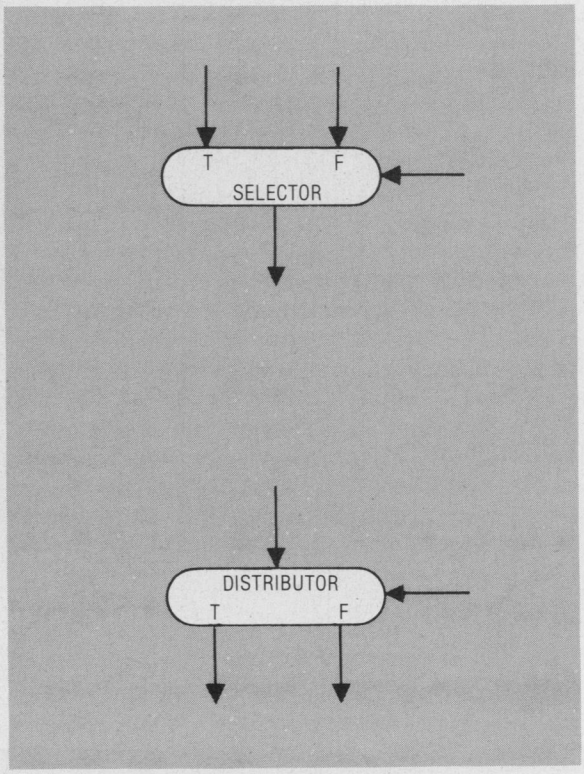


Figure 4. Selector and distributor functions.

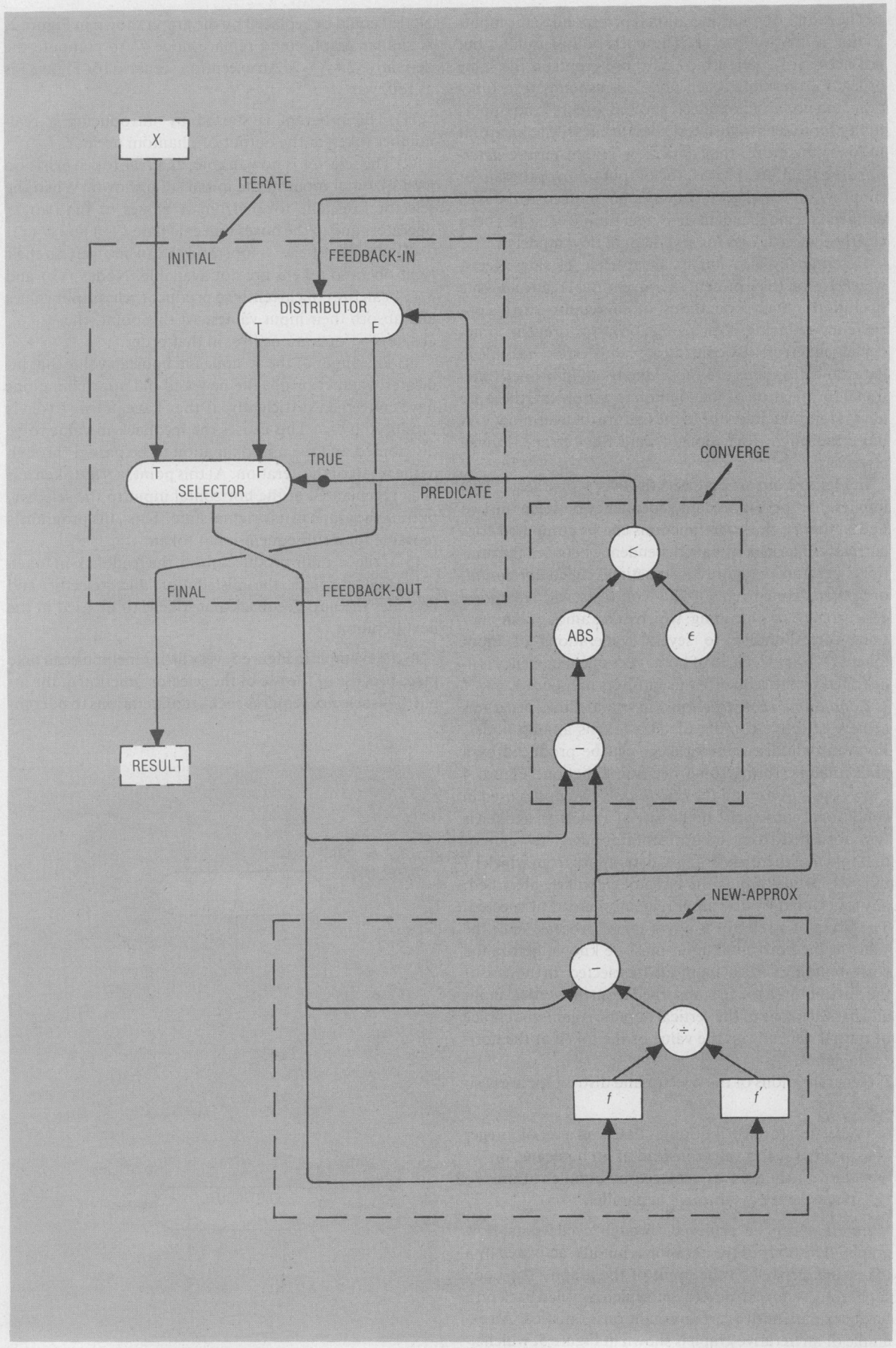


Figure 5. A graph for Newton's method.

plete (i.e., the horizontal input to be true) before the next token can be absorbed into the graph. However, the horizontal input is repeatedly false during any given set of iterations. Thus, an alternative formulation is necessary for concurrent processing of several input tokens.

Program structuring

It is cumbersome to deal with graphical programs consisting of single very large graphs. Just as subroutines and procedures are used to structure conventional programs, macrofunctions can be used to structure graphical ones. This idea appeals to intuition: A macrofunction is defined by specifying a name and associating it with a graph, called the *consequent* of that name. A node in a program graph labeled with that name is, in effect, replaced by its consequent; the arcs are spliced together in place of the phantom nodes. In the diagrams in this article, the orientation of the arcs in the consequent is assumed to match that of the node the consequent replaces. This replacement is called *macroexpansion,* in analogy to the similar concept used in conventional languages. It is valid to view data flow languages as performing macroexpansion during execution rather than during compilation. A macroexpansion is shown in Figure 6.

Macrofunctions often aid in understanding and developing graphical programs. For example, one might wish to encapsulate the iterate subgraph in Figure 5 into a node type called Iterate. *Atomic* functions are those that are not macrofunctions. In some systems, a node such as Iterate could be either atomic or a macro available from a library.

Recursion is easy to visualize in data flow graphs. As mentioned above, a node labeled with a macrofunction can be thought of as replaced by its consequent. This rule can be adopted for recursively specified functions, such as those in which a series of macroexpansions from a node labeled *G* can result in a subgraph containing a node labeled *G*.

For example, Figure 7 shows a recursive specification of the example given in Figure 6. Unlike its predecessor, this version can be understood without appealing to the definition of the complicated Iterate subgraph. Although a graph with recursive macrofunction can, in concept, be expanded to an infinite graph, either distributors or the underlying implementation must ensure that expansion takes place incrementally as needed. Since copies of

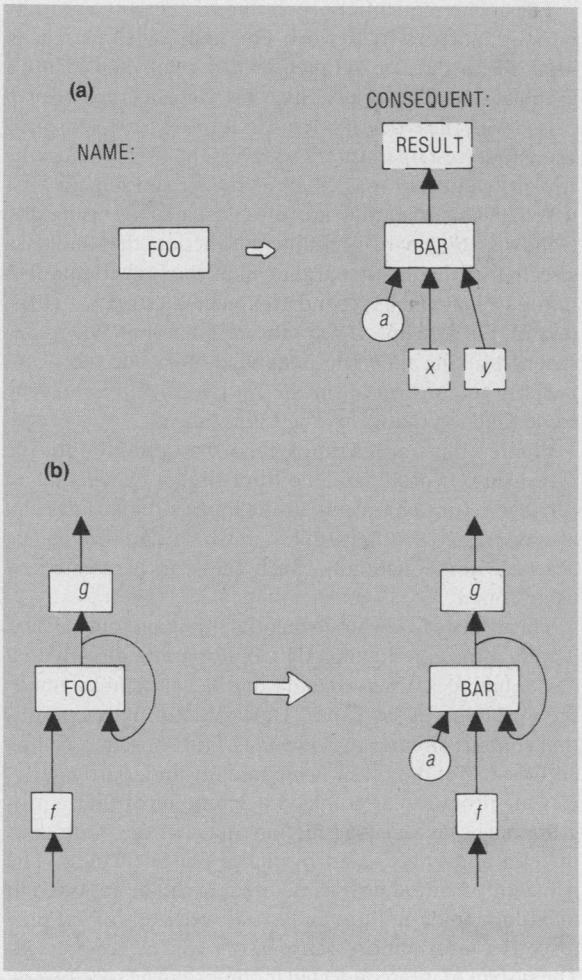


Figure 6. Example of graphical macroexpansion: (a) definition; (b) expansion.

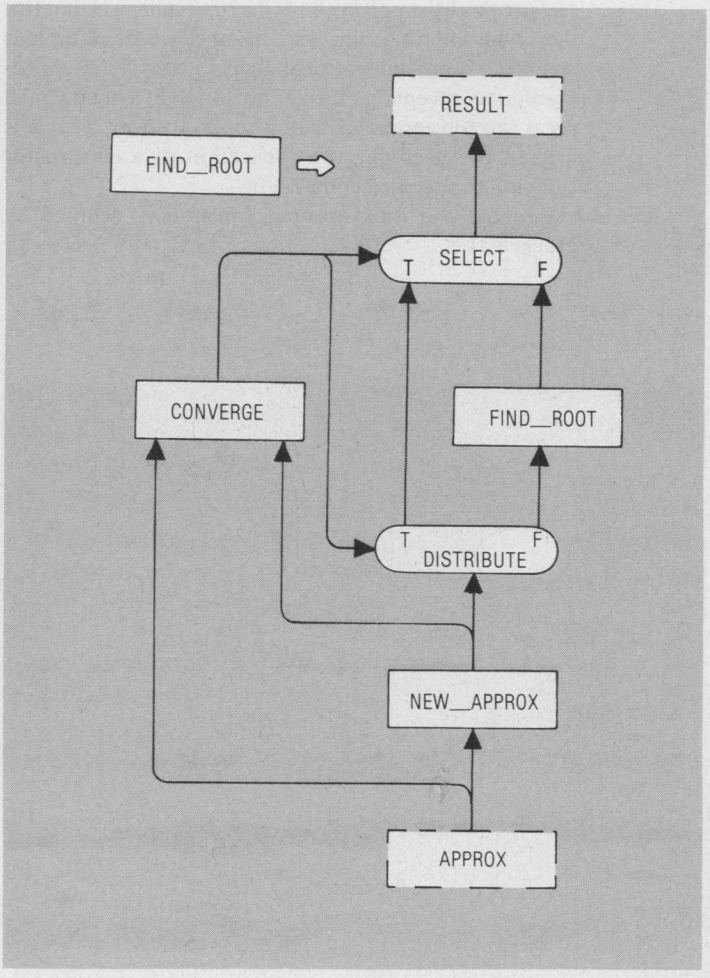


Figure 7. Recursive Newton graph.

find_root can be instantiated for different generations of input tokens, this graph permits pipelined concurrency in contrast to Figure 6.

Data structuring

Graph programs, like conventional programs, are amenable to incorporation of various data structuring features. Examples thus far have had numbers and boolean values as tokens. But succinct expression of solutions to complicated problems requires structuring operations that build tokens containing complex data objects from more primitive tokens. Data structuring also provides a way of exploiting concurrency; operations that deal with large structures, such as adding two vectors, can often perform many subfunctions concurrently. Such concurrency is frequently of a much higher degree than that resulting from a lack of visible data dependencies among nodes.

Tuples. The tuple is an important example of a complex token. A tuple is simply a grouping of objects into a single object. A tuple can flow as a single token, and the original objects can be recovered from it. The objects are ordered within the tuple so that recovery of one of the original objects occurs by supplying both the tuple and an ordinal index to an indexing function. In other words, a tuple is similar to a one-dimensional array in conventional programming, except that there is no notion of assignment in regard to the components of a tuple. Instead, a tuple is created by specifying the application of a constructor function to specified components.

Suppose the tuple-creating function is denoted by brackets. Let

$$t = [c_1, c_2, \ldots, c_n]$$

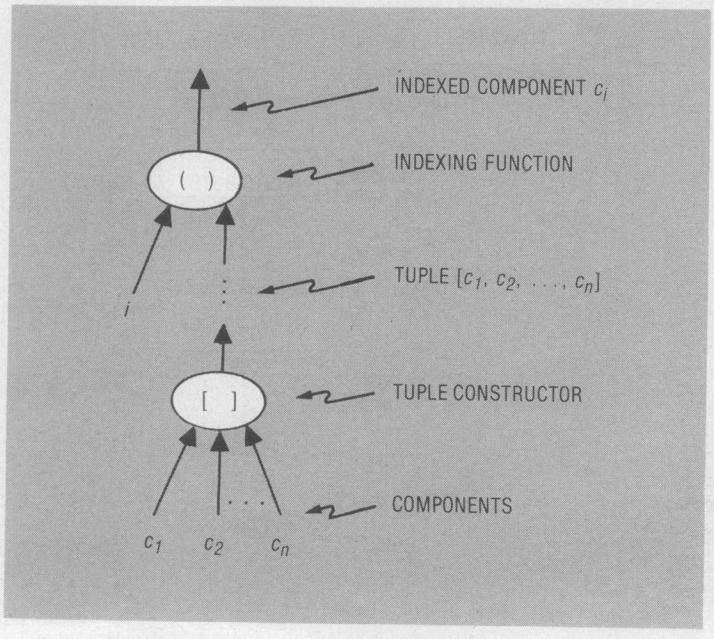


Figure 8. Tuple formation and component selection.

be the tuple with components $c_1, c_2, \ldots, c_n$. Parentheses denote the indexing function, so $t(i)$ will be c_i. This notation is often used for application of a function to its argument. Indeed, one might consider a tuple to be a function applicable to its range of indices.

We can now extend our basic data flow graphs to permit the tuple constructors and indexing function to be operators at a node, as shown in Figure 8. It is possible to view computations involving tuple tokens and operators as if the entire tuple flows from one node to another on an arc. However, the entire object need not flow in a particular implementation; more economical approaches are possible.

In addition to constructing tuples by using the [. . .] operator, other operations, such as *concatenation,* can be defined on tuples, that is

$$\text{conc}([c_1, c_2, \ldots, c_n], [d_1, d_2, \ldots, d_m]) = [c_1, c_2, \ldots, c_n, d_1, d_2, \ldots, d_m]$$

Conc can similarly be extended to more than two arguments.

The tuple concept leads to the construction of strings (tuples of characters), lists, etc. Tuples can have tuples as components to any number of levels of nesting.

Files. The sequential processing of files fits naturally into the data flow framework. One approach is to treat an input file as a stream of tokens, which is introduced into a graph at one of its inputs. Such a stream can then be processed with the types of operators introduced above or by using first/rest operators. *First* gives the first token in the stream, while *rest* passes all of the stream but the first token. Sequential files are often created by using *fby* (followed by), a two-argument function that builds a stream by using its first argument as the first component of the stream and its second argument—a stream—as the rest of the stream. Often, the rest has not been constructed at the time fby is applied. Instead, there is a *promise* to construct it in the future, as represented by some function that gives the rest as output.

Figure 9 illustrates a simple recursive definition for file processing. It produces an output stream by deleting all carriage-return characters in the input stream. Files can also be treated as single tuples, in which case they can be accessed nonsequentially. Such files can be created by using conc.

The utility of viewing sequential input and output as if it were a file coming directly from or going directly to a device has been observed and exploited in such systems as the pipe concept[7] of Unix.* Data-structuring operations that support streams provide one of the most compelling arguments for data flow programming and, particularly, viewing programs as graphs. Each module of such a program can be viewed as a function that operates on streams of data and is activated by the presence of data. The behavior of a module over its lifetime can be captured in one function definition, and low-level details of the protocol for information transmission can be suppressed. The system of interconnected modules can be specified by

*Unix is a trademark of Bell Laboratories.

a graph, which might even be constructed dynamically through use of macroexpansion. Contrast this with conventional assignment-based programs, which require explicitly set-up processes. Furthermore, communication of these processes is based on shared variables or explicit interchange of messages. Although characterizing the behavior of such a process for any single interaction requires only sequential program analysis techniques, it is generally difficult to succinctly characterize the long-term behavior.

Functions as values. Data objects that can represent functions and be applied to other objects by a primitive operator *apply* can enhance the power of data flow programming considerably. In one approach, a constant-producing node *N* can contain a graph, which is conceived as the value of a token flowing from *N*. The graph generally flows through conditionals, etc. When and if it enters the first input arc of an apply node, its phantom nodes serve a role similar to that of a macrofunction definition: Input/output phantom nodes are spliced to the input arcs of the apply node, and the graph effectively replaces the apply node. More generally, it is possible for arcs to enter the node *N* and be connected to the graph inside of it. These arcs are called *imports* to the graph. The same tokens flow on them, regardless of where the graph itself might flow. Since a graph-valued node can be present within the consequent of any macrofunction, many versions of the encapsulated graph can be generated, each customized by different import values. The concept forms a graphical equivalent to the notion of *closures*[8] or *funargs*.[9] A simple example is given in Figure 10. Further examples are in the literature.[5,10]

Data typing. The notion of data type is playing an increasingly important role in modern programming language design. Data flow graphs lend themselves to the support and exhibition of typing. It is quite natural to indicate, on each arc of a data flow graph, the type of data object that flows there. Hence, most developments concerned with typing are applicable in the domain of graphical programs.

Structure models

As we have seen, a token model views each node as processing a single stream of tokens over its entire lifetime. Each node produces tokens on output arcs in response to tokens absorbed on input arcs. Each operator is expressed in terms of what it does token-by-token. In structure models, a single data structure is constructed on each arc. The construction can, possibly, spread out over the lifetime of the graph. The structure might be interpretable as a stream of tokens, but it might be a tuple, a tree (formed, for example, by nested tuples), or just a scalar value. Within a structure model, each argument of an operator is one structure on which the operator operates to produce a new structure. A nontrivial structure model allows tuples or some equivalent structure to be conceptually infinite. That is, while a tuple always has a definite first component, it might not have a last component. If it does, it

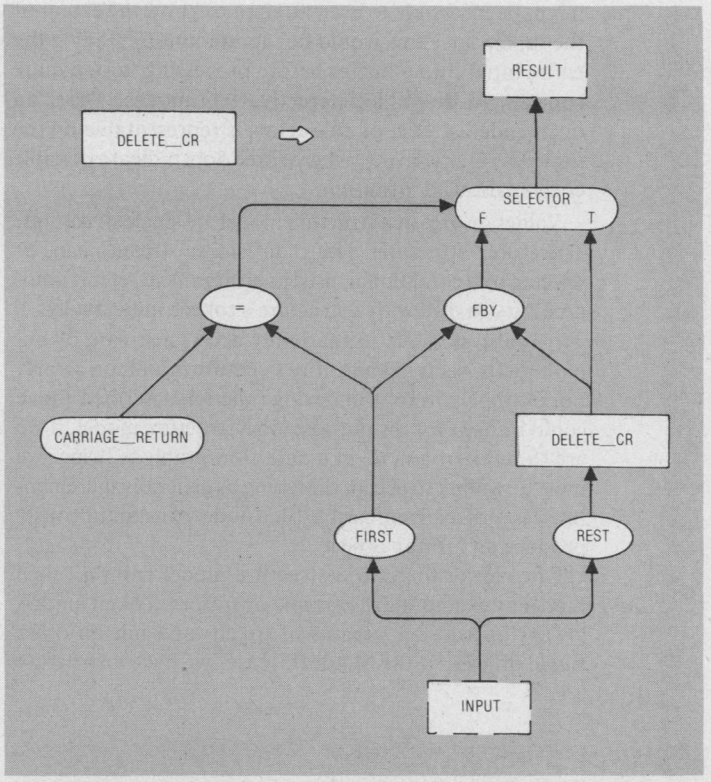


Figure 9. Graphical program for a file-processing problem.

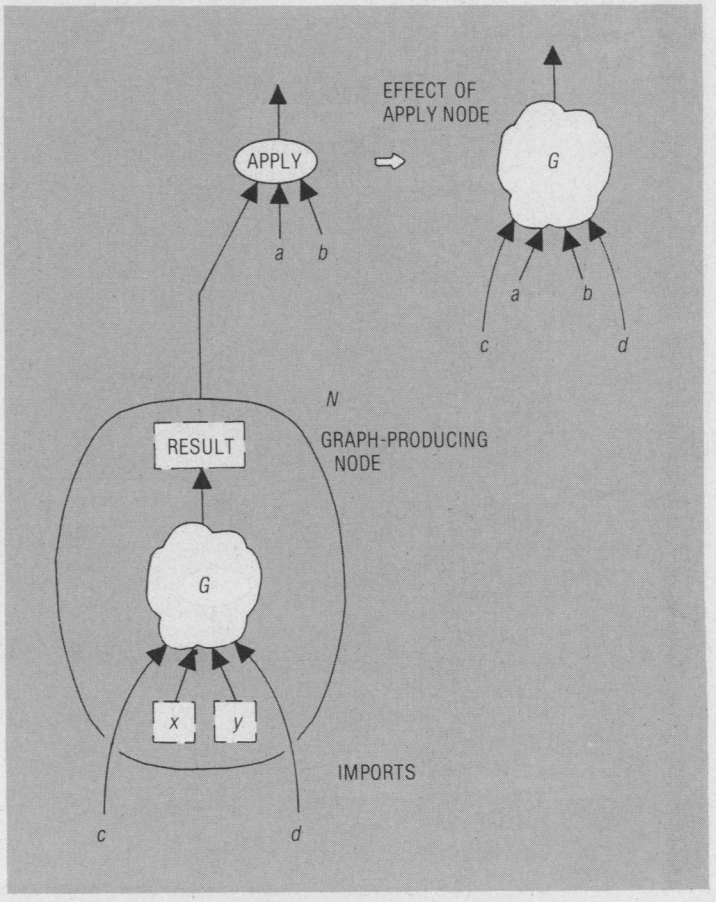


Figure 10. The apply function and use of imports.

might be infeasible to even attempt to know the extent of the tuple, since this would be tantamount to reading the entire input into a buffer before processing, a step quite contrary to desirable interactive I/O modes. Thus, an open-ended stream of characters is representable by the tuple $[c_1, c_2, c_3, \ldots]$, where three dots indicate possible continuation ad infinitum.

Values on arcs in a structure model are single structures. Therefore, structures (e.g., an entire stream) can be selected by a conditional, used as an argument, etc. It is not necessary to deal with a structure's tokens individually.

It is also possible to randomly access a stream by exploiting the fact that an entire structure is built on an arc. For example, the tuple indexing function described above could be used for ths purpose. In a structure model, if we are to think of a node in a data flow graph as firing, we must be willing to accept this firing as generally incremental. It is physically infeasible for a node to instantaneously produce an infinite object.

One can distinguish a structure model from a token model by examining the atomic operators. Token models always operate on streams of tokens and not on other single objects; if this is not the case, we have a structure model. A structure model achieves the effects of stream processing by means of macrofunctions. In the case of Figure 2, this could be done in two ways: by defining a stream-procesing version of each of the arithmetic operators, or by creating a function (the graph shown) and applying it to each component of an input stream. Likewise, some of the functions (such as switching streams) of structure models can be emulated in some token models by macrofunctions.

We have alluded to the fact that the behavior of a data flow program over its lifetime can be captured as a single function. Perhaps the most important distinction between structure and token models is that in the former this behavior is expressible as a recursive function within the model, while the latter requires a more encompassing language to capture the behavior. For a token model, the long-term behavior is captured as a function on histories of token streams.[11] In a structure model, the notion of history degenerates since each arc carries exactly one object.

To maintain certain aspects of the history of a stream in a token model program, special provisions must be coded to save relevant components as they pass through a function. In structure models, for reasons cited above, it is easy to maintain the equivalent of the history of all or part of a token stream, as the entire stream is accessible as a single object.

Token model interpreters usually process tokens in sequence. This causes asynchrony and concurrency to be less than the maximum possible.[12] In structure models, there is no implied order for processing structure components. Thus, these componenets can be processed out of sequence without use of a more specialized interpreter.

Some functions are easily expressed in structure models but more difficult to express in token models. An example is the generalized indexing function, or gen_index, which operates on two finite or infinite tuples, stream $= [s_1, s_2, s_3, \ldots]$ and indices $= [i_1, i_2, i_3, \ldots]$, to produce $[s_{i_1}, s_{i_2}, s_{i_3}, \ldots]$. This function can be expressed by the graph in Figure 11 or by the textual expression below.

```
gen_index(stream, indices) =
        if indices = [ ]
                then [ ]
                else fby(stream(first(indices)),
                        gen_index(stream, rest(indices)))
```

This is obviously a recursive definition. The condition eventually becomes true when indices is finite. When the indices or tuples are not empty, the result is the tuple obtained by indexing the component of the stream that corresponds to the first index (using the parenthesis notation for indexing given above). This is followed by the result of the gen_index function on the remainder of the tuple.

This example illustrates a correspondence between a graphical and a textual notation. It is possible to design a language so that each program graph has an exact textual equivalent, and vice-versa.[13] This permits use of graphical programming concepts even when there is no graphical input device, as well as the use of hybrid representations, which help suppress uninteresting graphical detail.[14]

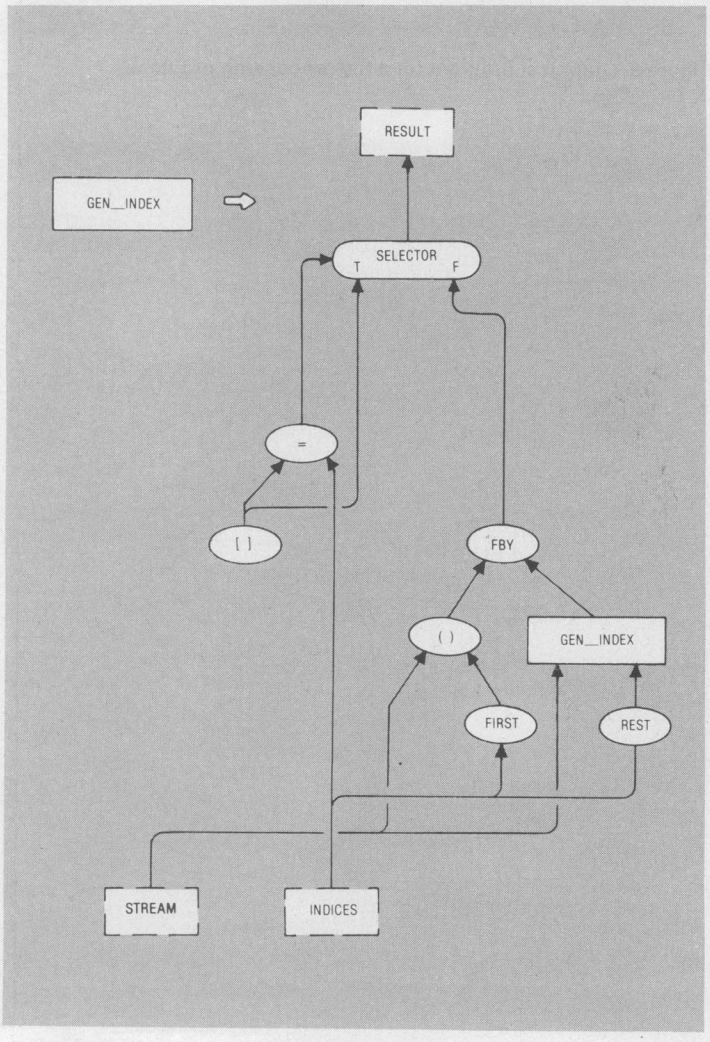


Figure 11. A generalized indexing function.

Structure models can perform the functions of token models. Is the opposite also true? A relaxation of our definition, to allow tokens to be infinite objects, would make it possible. This, however, seems contrary to the notion of a token as an object that can flow in a single step. Another means is to introduce *pointer,* or *reference,* tokens. The infinite objects then become homomorphic images of a network of tokens involving pointers. Although necessary to perform correctness proofs of a structure model,[15] the introduction of such objects into a programming language should be avoided where possible because they make the language less machine-independent. Introducing infinite objects into token models[16] is counter to the interests of conceptual economy. One could use, instead, a structure model, which does not necessarily require token streams yet allows representation of stream-like behavior by using the appropriate structures. Furthermore, a great many applications cannot exploit the repetitive stream-processing capability of a token model. For these applications, a token-model language amounts to overkill.

Why would one ever choose a token model (with only finite tokens, as defined herein) instead of a structure model, given the flexibility of the latter? The answer lies in the trade-off of execution efficiency vs. ease in programming. The token-by-token processing of token models often results in efficient storage-management. Since structure models use structures to emulate all stream-processing functions of token models (which are often macrofunctions involving recursion) token-by-token processing is difficult to detect. Therefore, structure models typically use fully general storage management, which recycles storage in a more costly manner than do the more specialized token models. Compiler optimization and special execution techniques that improve the efficiency of structure model execution without sacrificing generality are topics of active research.

Machine representation and execution of graphical programs

Program storage. It is possible to compile graphical programs into conventional machine languages. However, if the main goal is the execution of such programs, there are advantages in directly encoding the graphs themselves as the machine program for a specially constructed processor. Alternatively, such an encoding can be interpreted on a conventional processor as virtual machine code. The advantages of the special-processor and virtual-machine approaches include direct exploitation of the concurrency implicit in the graphical formulation and a clearer connection between a higher-level graphical language and its machine representation.

A survey of the numerous possible encodings of graph programs is beyond the present scope. The discussion below is a qualitative look at one version for a token model. The first task is to establish an encoding for a program graph, which consists of nodes labeled with function names and arcs connecting the nodes. The orientation of the arcs entering the nodes is, of course, relevant. We represent the entire graph as a set of contiguous memory locations, each corresponding to one node of the graph. Thus, relative addressing can be used to identify any particular node. Our use of relative addressing should not be interpreted as the use of a single memory module. Use of one address space to address a multitude of physical memories is a common way to avoid memory contention. In practice, addressing might take place on two levels: short addresses within the consequent of each macrofunction, and long addresses for the global interconnection of such functions.[17]

For simplicity, assume that each node has a single output arc. As long as nodes represent functions, nodes with more than one output arc can always be decomposed into one function for each arc, and thus represented as several nodes with fanout from the same input arcs (Figure 12). This allows association of a node with its only output arc, and vice-versa.

Having identified a location for each node, we can discuss the encoding of the relevant information for each node. Obviously, there must be a field in each location to indicate an encoding of that node's label, since the label determines the function to be executed. There is an ordered listing of each node's input arcs, but since each of these arcs is identified as the output of some node, we can use that node's location to represent the arc. Similarly, we include an ordered listing of the destinations of each node's output arc. Figure 13 illustrates the encoding of a graphical program as a contiguous set of locations, called a *code block*.

Data-driven execution. Let us now consider data-driven execution in the context of our representational model. For simplicity, assume that one token, at most, is present on any arc at a given time. This is an invariant property maintained by the execution model. To simplify further, we treat only node functions that are *strict,* i.e., require tokens on all arcs in order to fire. A slight modification is necessary for other cases, such as selector nodes.

In addition to the code block that specifies a data flow graph (discussed above) we provide a second *data block*

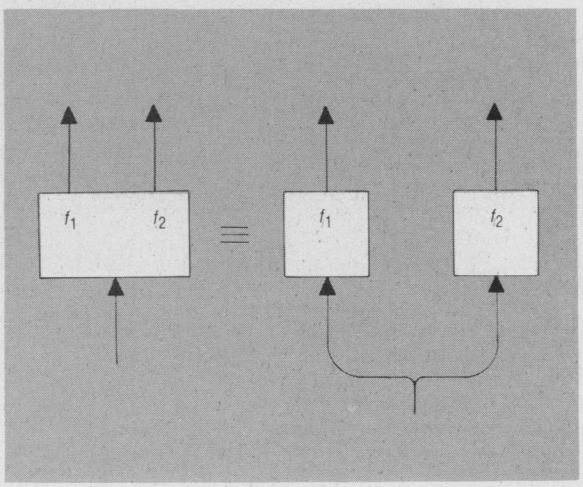
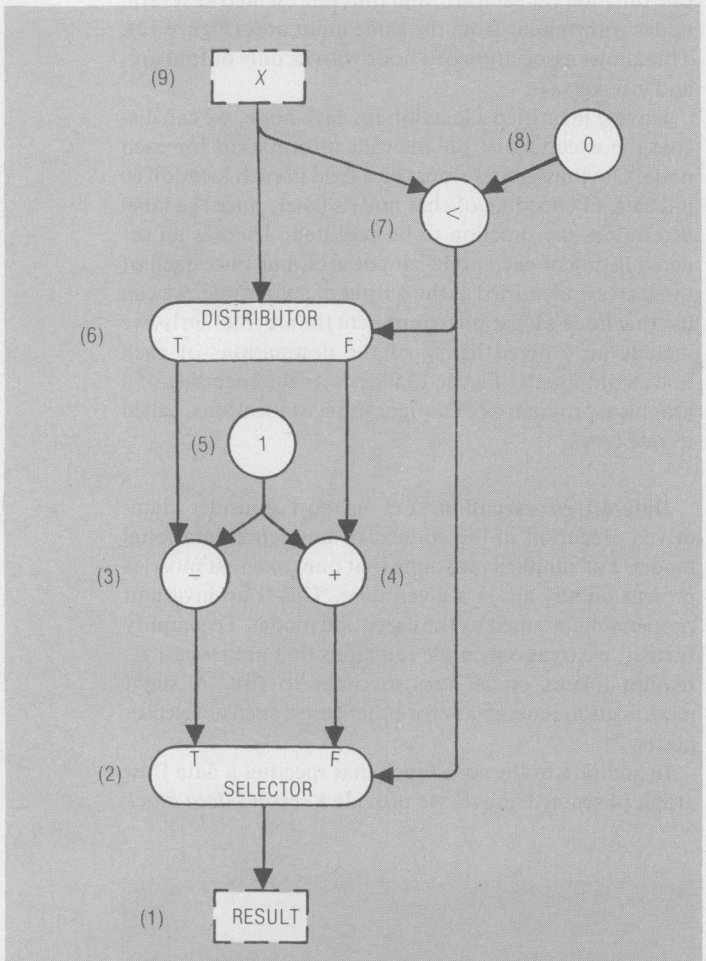


Figure 12. Replacing multiple-output-arc with single-output-arc nodes by means of fanout.

of contiguous locations. It contains the data tokens that are to flow on the arcs of the graph. These locations exactly parallel those representing the graph itself. That is, if location *i* in the graph encoding represents a particular node (and, by convention, its output arc), then location *i* in the data block represents the token value on that arc.

Initially, an arc is empty. To indicate this, its corresponding location is marked with a special bit pattern. It is convenient, but not absolutely essential, to include with this bit pattern a *shortage count* that indicates how many remaining input arcs must get tokens before the corresponding node can fire. As an arc gets a token, the shortage counts of nodes to which that arc is input are each decreased by one. This act is called *notification,* as if one node notifies another that data is ready. When the shortage count is zero, the node is firable. Shortage counts are initialized from values stored in the code block.



LOCATION	FUNCTION	ARGUMENTS	RESULTS
1	RESULT	2	
2	SELECTOR	7,3,4	1
3	−	6,5	2
4	+	5,6	2
5	1		3,4
6	DISTRIBUTOR	7,9	3,4
7	<	9,8	2,6
8	0		7
9	INPUT		

Figure 13. Internal encoding of a data flow graph.

Suppose that a node has become firable, as indicated by its zero shortage count. The processor can then compute the function specified for that node. It does so by fetching the values in the node's input arcs (as indicated by the encoding in the code block location) and then storing the result value of the function in the corresponding data location. The nodes needing the stored value are then notified. The shortage count of one or more of these nodes might be decreased to zero, indicating that the node is firable. The process then repeats.

Any number of firable nodes can be processed concurrently. For any system state, the set of nodes that need attention (e.g., are firable) can be recorded on a *task list* of their addresses. This list need not be centralized; it can be distributed over many physical processing units.

To start things off, we need only put data in locations corresponding to the phantom input nodes of the graph. Then we add to the task list the nodes to which the input nodes are connected. The firing of nodes continues in a chain reaction until no firable nodes are left. By this time, all results have been produced. That is, the output values are either resident on selected output locations or have been moved to some output device.

We have not discussed the *recycling* of data locations. Most token model implementations suggest that some form of reset signal be used to return a location to its initialized state. An alternative approach, in which data blocks are "thrown away," is presented by Keller, Lindstrom, and Patil.[17]

Representation of complex tokens. The use of complex data objects, such as tuples and functions, as tokens that can conceptually flow on arcs has been described above. It is possible, in the case of finite objects, to send packets consisting of the complete objects.[18] This, however, presents difficulties in storage management, as the size of an object might be unknown before it arrives. Another difficulty is that this approach can involve much unnecessary copying, as an object can be sent to a number of nodes, each of which selects only a small portion for its use. It might be more efficient to introduce, at the implementation level, pointers to take the place of objects that exceed a certain size.

For example, a tuple might be constructed of objects, one or more of which is itself a tuple. In this case, we want to build the outer tuple by using pointers to the inner tuples, rather than by copying the inner tuples themselves. Of course, this type of representation is essential in dealing with conceptually infinite objects; in such a case, the outermost tuple could never be completely constructed. The same is true for recursively defined function objects. Thus, while the conceptualization of program graphs supports the flow of arbitrarily complex objects on arcs, pointers might be required to efficiently implement this conceptual flow. Techniques from Lisp and its variants are especially relevant.[9,19,20]

Demand-driven execution. The mode of execution described in the previous section can be termed *data-driven* because it involves the following (possibly overlapping) phases of the execution of a function node within its environment (the rest of the graph to which the node is connected):

(1) A node receives data from its environment via its input arcs.
(2) A node sends data to its environment via its output arc.

An alternative is the *demand-driven* evaluation mode. It has a more extensive set of phases, which can also be overlapping:

(1) A node's environment requests data from it at its output arc.
(2) A node requests data from its environment at its input arcs, if necessary.
(3) The environment sends data to a node via its input arcs, if requested.
(4) A node sends data to its environment via its output arc.

This suggests that a data-driven execution is like a demand-driven execution in which all data has already been requested.

Suppose that sufficient input data has been made available at the input arc in a demand-driven execution situation. Nothing would happen until a demand is made at the output arc. Although we need not implement it as such, we can think of this demand as being represented by a *demand token* that flows against the direction of the arcs. When a demand token enters a node at an output arc, it might cause the generation of demand tokens at selected input arcs of that node. As this flow of demands continues, data tokens are produced that satisfy the demand. At that point, computation takes place much as it does in the data-driven case. Demands and data can flow concurrently in different parts of the graph.

In demand-driven execution of a graph, a node becomes firable when its shortage count becomes zero *and* it has been demanded. An extra bit in the data location can be used to indicate whether or not the corresponding datum has been demanded. If destination addresses are set dynamically, the presence of at least one of them can indicate demand. Initially, certain nodes (usually those connected to output arcs of the graph) are marked as demanded. The processor attempts those nodes that are so marked and have shortage counts of zero.

The advantages of the demand-driven approach include the elimination of distributor nodes that, on the basis of test outcomes, prevent certain nodes from firing. This advantage accrues because only needed data values are ever demanded. Thus, demand-driven execution does not require the distributor shown in Figure 7 and can be simplified as shown in Figure 14.

Although token models can have either data- or demand-driven execution models, *structure* models seem to require a demand-driven one. Otherwise, the data-driven elaboration of infinite structures tends to usurp system resources unnecessarily. The prime disadvantage of demand-driven execution is the extra delay required to propagate demand. In part, this is balanced by the lack of distributor functions present in the data-driven approach. It is as if the selector and distributor operations are folded into a single selector in the demand-driven approach. It is also possible to optimize the demand-driven execution model to statically propagate demand at compile time and recover some of the efficiency of data-driven execution.

A certain minimum overhead is required in both demand- and data-driven execution but appears in different forms. When a computing system is integrated into an asynchronous external environment, an appropriate regulatory protocol must exist at any interface. It is impossible to have the environment dump arbitrarily large quantities of data into the system without having this process punctuated by handshaking signals from the system to the environment. Similarly, we don't normally wish to have the system dump large quantities of data into the environment without regulation. For example, when observing output on a CRT terminal, scrolling should stop when the screen is full and proceed at the viewer's command. Thus, every implementation, data-driven or demand-driven, must have a means of controlling the flow of data by sending signals in a direction that opposes the flow. This requirement exists within the system as well, in the form of controlling the flow of data from one subsystem to another. Propagation of demand before the flow

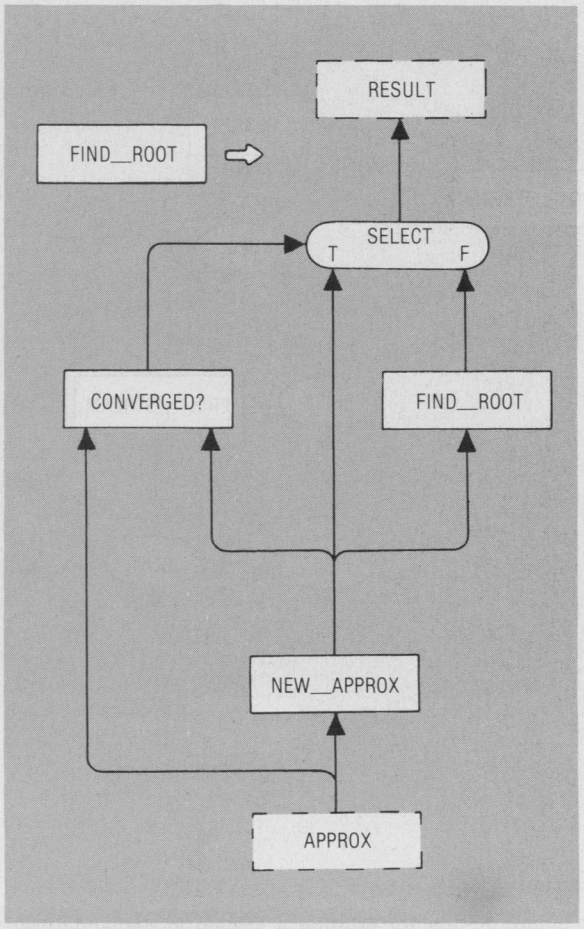


Figure 14. Demand-driven conditional graph.

of data is often balanced by reset signals, which occur after the flow of data and indicate that the location can be reused. These signals present overhead comparable to the flow of demands.

Input-output interfacing. We can interface the fby function to an output device so that individual components of an infinite stream can be constructed within a graph program and directed to that device. Interfacing these types of functions to external devices is much simpler than it might appear. All we need are primitives to input/output single atomic components; recursion at the graph program level does the rest. For example, Figure 15 shows the expression of a pseudofunction that sequentially prints the components of a stream ad infinitum, given a primitive pseudofunction *print* that prints one stream component. Assuming demand-driven mode, print causes its argument to be printed whenever its result is demanded. The function *seq* simply demands its arguments in sequence, the second being demanded only after the first has yielded its result. Of course, tests for end-of-stream atoms can be added to functions such as print_stream. Such I/O functions have been successfully implemented[13]

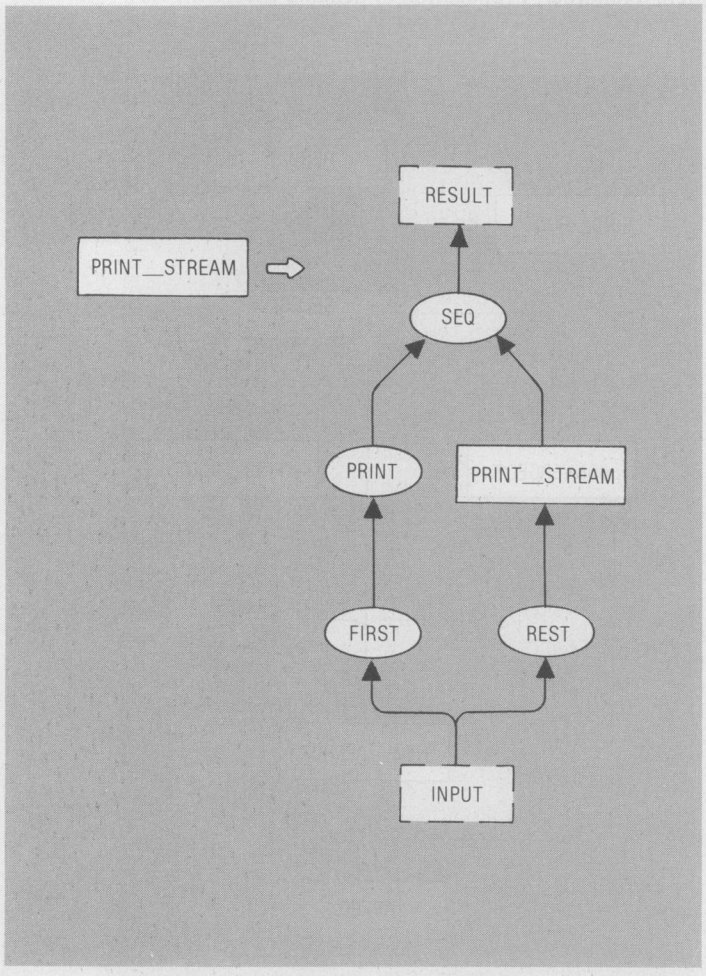


Figure 15. Pseudofunction for printing a stream in demand-driven mode.

Historical background

Many of the concepts presented in graphical representations have appeared in earlier work. Dynamo was an early language with a graphical representation.[21] Although used only to perform synchronous simulation, it is a true data flow language, since the ordering of its statement executions is governed by data dependencies rather than syntax. The integration of Dynamo-like functions into a general-purpose data flow language is discussed by Keller and Lindstrom.[22]

The literature of engineering sciences, particularly electrical engineering and control theory, describes many uses of graphical models for function-based systems. Zadeh, for example, discusses determinacy for general systems.[23] In the related area of digital signal processing, digital filters are often represented graphically.[24] However, most literature in that area presents algorithmic results by translating them to Fortran programs rather than employing a data flow language that can directly represent signal-processing structures. This is one area where graphical data flow programming has much to contribute.

The use of modules that communicate via streams as a structuring device appeared in Conway's definition of a *coroutine:* ". . . an autonomous program which communicates with adjacent modules as if they were input or output subroutines."[25] Conway's paper also included the observation that such coroutines could execute simultaneously on parallel processors. Since then, the coroutine notion seems to have become rather more implementation-oriented. Kahn and MacQueen argue for a return to the elegance of the original definition,[26] which is consistent with the type of programming advocated here.

Brown prophesies the use of applicative languages for the exploitation of parallel processing capability.[27] Patil discusses parallel evaluation in a graphical lambda calulus model.[3] Many others have published important related references on applicative languages.[8,9,28-34] Other aspects of structure models have also been reported.[4,5,19,35-38] Many modern methods for presenting semantics of most any language rely on the presentation of functional expressions for the primitives of the language.[39,40]

Fitzwater and Schweppe offered an early suggestion for data-driven computation.[41] After Karp and Miller's[1] discussion of determinancy in the data flow model, many graphical token models for data-flow have appeared.[2,6,42-47]

Although the firing-rule notion for data flow graphs is thought by some to derive from Petri nets,[48,49] this appears to be a matter of transferred terminology rather than historical dependence. In the Petri-net model, tokens are valueless. Therefore it does not directly represent data, as do data flow models. The basic Petri-net model includes ways to introduce indeterminate behavior; these are not present in basic data flow models.

Finally, an important topic is the incorporation of indeterminate operators into graphical and functional models. This must be considered if such models are to be able to represent operating systems and related programs in which tokens can be merged according to order of arrival from an external environment. Some preliminary discussions of semantic problems, syntax, etc., have appeared.[50-54]

Future developments

Researchers attempting to make data flow program graphs the basis for a practical programming language face several major problems. Some of these problems involve human engineering and the need to create a reasonably priced terminal system to support graphical programming.

Physical implementation decisions. To date, most data flow program graph models have been used as either *just models* or *intermediate languages* for data flow programs. Two exceptions are FGL[14] (based on a structure model) and GPL[55] (based on a token model), in which the models are used as high-level languages. These allow the programmer to draw the program and execute its graphical form.

Impediments to direct use of graphs as data flow programming languages include the expense of graphical terminals, the added cost of graphical software, and the increased workload imposed on execution resources due to interactive display and editing. The evolution of personal computers packaged with powerful microprocessors, bit-mapped displays, disk storage, etc., is beginning to change this equipment-based deficiency. Ongoing experiments indicate that graphical resolution of 4K × 4K pixels or greater might be desirable. Ironically, this is due mainly to the display of the text (comments, node names, etc.) that accompanies a program module of reasonable size. When adequate hardware is available, it will only be a matter of time before researchers develop the additional tools to make graphical programming practicable.

Graphical representations are widely used in non-programming disciplines because they provide a more intuitive view of system structure. It seems reasonable to investigate the possibility that such representations could do the same for programming. We suggest that tools for debugging, statistical monitoring, and resource management be coupled with data flow program graphs to allow a programmer to produce programs more productively than is currently possible.

The use of graphical tools for software development seems to be gaining momentum.[56-59] When similar tools are used in the context of a data flow language, an additional advantage accrues: the graphs have a well defined functional meaning, rather than just the ability to represent procedure nesting, loop nesting, calling sequences, etc. This meaning is a specification of the system under development.

The programmer education problem. For graphical data flow methods to succeed in a practical sense, they must have an acceptable link to the 30 prior years of software and hardware development, which present a legacy

of considerable inertia. The design of a clean interface between functional programs and existing assignment-based programs (e.g., data-base systems and operating systems) is one aspect of the problem. Several solutions are being pursued.

Professional programmers with years of experience in writing Fortran code have become very good at writing Fortran-like solutions to problems. The change to Algol, Cobol, Pascal, etc., is not a large conceptual step, in that the structural styles of these languages are not radically different from Fortran's. However, data flow languages require and support very different styles. Programmers trained only in conventional languages might be unwilling to try problem-solving techniques based on graphical or even functional program structures. Therefore, the potential gains of such techniques must be made apparent to programming management. ■

Acknowledgments

We wish to thank Arvind, Tilak Agerwala, Paul Drongowski, Chu-Shik Jhon, Gary Lindstrom, and Elliott Organick for numerous comments that helped improve the article. We also thank Kathy Burgi for drafting the figures.

This material is based upon work supported by a grant from the Burroughs Corporation and by National Science Foundation grant MCS 81-06177.

References

1. R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Applied Mathematics,* Vol. 14, No. 6, Nov. 1966, pp. 1390-1141.
2. D. A. Adams, *A Computational Model with Data Flow Sequencing,* Technical Report CS117, Computer Science Dept., Stanford University, Palo Alto, Calif., 1968.
3. S. Patil, *Parallel Evaluation of Lambda-Expressions,* MS thesis, MIT Dept. of EE, Jan. 1967.
4. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing 74,* IFIP, North-Holland, Amsterdam, 1974, pp. 471-475.
5. R. M. Keller, *Semantics and Applications of Function Graphs,* Technical Report UUCS-80-112, Computer Science Dept., University of Utah, Salt Lake City, Utah, 1980.
6. A. L. Davis, *Data-Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems,* Technical Report UUCS-78-108, Computer Science Dept., University of Utah, Salt Lake City, Utah, 1978.
7. D. M. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Comm. ACM,* Vol. 17, No. 7, July 1974, pp. 365-381.
8. P. J. Landin, "The Mechanical Evaluation of Expressions," *Computer J.,* Vol. 6, No. 4, Jan. 1964, pp. 308-320.
9. J. McCarthy et al., *Lisp 1.5 Programmers Manual,* MIT Press, Cambridge, Mass., 1965.
10. J. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Trans. Computers,* Vol. C-26, No. 2, Feb. 1977, pp. 138-146.
11. S. Patil, "Closure Properties of Interconnections of Determinate Systems," *Proc. Project MAC Conf. Concurrent Systems and Parallel Computation,* June 1970, pp. 107-116.
12. Arvind and K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of a Dataflow Language," in *Formal Description of Programming Concepts,* E. J. Neuhold, ed., North-Holland, Amsterdam, 1978, pp. 95-119.
13. R. M. Keller, B. Jayaraman, D. Rose, and G. Lindstrom, *FGL (Function Graph Language) Programmers' Guide,* AMPS Technical Memorandum No. 1, Computer Science Dept., University of Utah, Salt Lake City, Utah, 1980.
14. R. M. Keller and W-C. J. Yen, "A Graphical Approach to Software Development Using Function Graphs," *Digest of Papers Compcon Spring 81,* Feb. 1981, pp. 156-161.
15. R. M. Keller and G. Lindstrom, "Hierarchical Analysis of a Distributed Evaluator," *Proc. Int'l Conf. Parallel Processing,* Aug. 1980, pp. 299-310.
16. K-S. Weng, *An Abstract Implementation for a Generalized Data Flow Language,* PhD thesis, MIT, Cambridge, Mass., May 1979.
17. R. M. Keller, G. Lindstrom, and S. Patil, "A Loosely-Coupled Applicative Multi-Processing System," *AFIPS Conf. Proc.,* Vol. 40, 1979 NCC, June 1979, pp. 613-622.
18. A. L. Davis, "The Architecture and System Method of DDM-1: A Recursively-Structured Data Driven Machine," *Proc. Fifth Ann. Symp. Computer Architecture,* 1978.
19. D. P. Friedman and D. S. Wise, "CONS Should Not Evaluate Its Arguments," in *Automata, Languages, and Programming,* S. Michaelson and R. Milner, eds., Edinburgh University Press, Edinburgh, Scotland, 1976, pp. 257-284.
20. R. M. Keller, "Divide and CONCer: Data Structuring for Applicative Multiprocessing," *Proc. Lisp Conf.,* Aug. 1980, pp. 196-202.
21. J. W. Forrester, *Industrial Dynamics,* MIT Press, Cambridge, Mass., 1961.
22. R. M. Keller and G. Lindstrom, "Applications of Feedback in Functional Programming," *Proc. ACM Conf. Functional Languages and Computer Architecture,* Oct. 1981, pp. 123-130.
23. L. A. Zadeh and C. A. Desoer, *Linear System Theory,* McGraw-Hill, New York, 1963.
24. L. R. Rabiner and C. M. Rader, *Digital Signal Processing,* IEEE Press, New York, 1972.
25. M. E. Conway, "Design of a Separable Transition-Diagram Compiler," *Comm. ACM,* Vol. 6, No. 7, July 1963, pp. 396-408.
26. G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes," *Proc. IFIP Congress 77,* Aug. 1977, pp. 993-998.
27. G. Brown, "A New Concept in Programming," in *Management and the Computer of the Future,* M. Greenberger, ed., John Wiley & Sons, 1962.
28. A. Church, *The Calculi of Lambda-Conversion,* Princeton University Press, Princeton, N.J., 1941.
29. A. Evans, Jr., "PAL—A Language Designed for Teaching Programming Linguistics," *Proc. ACM Nat'l Conf.,* 1968, pp. 395-403.
30. C. P. Wadsworth, *Semantics and Pragmatics of the Lambda-Calculus,* PhD thesis, University of Oxford, Oxford, England, 1971.
31. E. A. Ashcroft and W. W. Wadge, "Lucid, A Nonprocedural Language with Iteration," *Comm. ACM,* Vol. 20, No. 7, July 1977, pp. 519-526.

32. M. O'Donnell, "Subtree Replacement Systems: A Unifying Theory for Recursive Equations, Lisp, Lucid, and Combinatory Logic," *Proc. Ninth Ann. Symp. Theory of Computing,* May 1977, pp. 295-305.

33. D. P. Friedman and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing," *IEEE Trans. Computers,* Vol. C-27, No. 4, Apr. 1978, pp. 289-296.

34. D. A. Turner, "A New Implementation Technique for Applicative Languages," *Software—Practice & Experience,* Vol. 9, No. 1, 1979, pp. 31-49.

35. W. H. Burge, *Recursive Programming Techniques,* Addison-Wesley, Reading, Mass., 1975.

36. P. Henderson and J. H. Morris, Jr., "A Lazy Evaluator," *Proc. Third ACM Conf. Principles Programming Languages,* 1976, pp. 95-103.

37. R. M. Keller, *Semantics of Parallel Program Graphs,* Technical Report UUCS-77-110, Computer Science, Dept., University of Utah, Salt Lake City, Utah, July, 1977.

38. P. Henderson, *Functional Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1980.

39. R. Milne and C. Strachey, *A Theory of Programming Language Semantics,* Champman and Hall, London, 1976.

40. J. Stoy, *The Scott-Strachey Approach to the Mathematical Semantics of Programming Languages,* MIT Press, Cambridge, Mass., 1977.

41. D. R. Fitzwater and E. J. Schweppe, "Consequent Procedures in Conventional Computers," *AFIPS Conf. Proc.,* Vol. 26, Part II, 1964 FJCC, pp. 465-476.

42. J. B. Dennis, "Programming Generality, Parallelism, and Computer Architecture," *Proc. IFIP Congress,* 1969, pp. 484-492.

43. J. D. Rodriguez, *A Graph Model for Parallel Computation,* Technical Report TR-64, Project MAC, MIT, Cambridge, Mass., 1969.

44. D. Seror, *DCPL: A Distributed Control Programming Language,* Technical Report UTEC-CSc-70-108, Computer Science Dept., University of Utah, Salt Lake City, Utah, Dec. 1970.

45. J. B. Dennis, J. B. Fosseen, and J. P. Linderman, "Dataflow Schemas," in *Theoretical Programming,* Springer-Verlag, Berlin, 1972, pp. 187-216.

46. K-S. Weng, *Stream-Oriented Computation in Recursive Data Flow Schemas,* Master's thesis, MIT, Cambridge, Mass., Oct. 1975.

47. Arvind, K. P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine,* Technical Report TR 114a, University of California, Irvine, Calif., Dec. 1980.

48. C. A. Petri, "Fundamentals of a Theory of Asnychronous Information Flow," *Information Processing 62,* IFIP, North-Holland, 1962, pp. 386-391.

49. J. L. Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, Englewood Cliffs, N.J., 1981.

50. Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors, and Dataflow," *Operating Systems Rev.,* Vol. 11, No. 5, Nov. 1977, pp. 159-169.

51. R. M. Keller, "Denotational Models for Parallel Programs with Indeterminate Opeators," in *Formal Discription of Programming Concepts,* E. J. Neuhold, ed., North-Holland, Amsterdam, 1978, pp. 337-366.

52. P. R. Kosinski, "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs," *Proc. Fifth Ann. ACM Symp. Principles Programming Languages,* Jan. 1978, pp. 214-221.

53. D. P. Friedman and D. S. Wise, "An Approach to Fair Applicative Multiprogramming," in *Semantics of Concurrent Computation,* G. Kahn, ed., Springer-Verlag, Berlin, 1979, pp. 203-225.

54. B. Jayaraman and R. M. Keller, "Resource Control in a Demand-Driven Data-Flow Model," *Proc. Int'l Conf. Parallel Processing,* 1980, pp. 118-127.

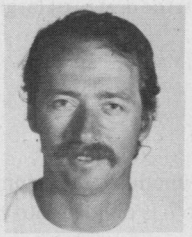
55. A. L. Davis and S. A. Lowder, "A Sample Management Application Program in a Graphical Data-Driven Programming Language," *Digest of Papers Compcon Spring 81,* Feb. 1981, pp. 162-167.

56. D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Trans. Software Eng.,* Vol. SE-6, No. 1, Jan. 1977, pp. 16-33.

57. V. Weinberg, *Structured Analysis,* Prentice-Hall, Englewood Cliffs, N.J., 1978.

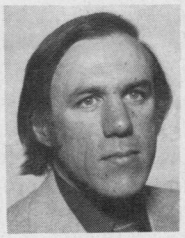
58. E. Yourdon and L. L. Constantine, *Structured Design,* Prentice-Hall, Englewood cliffs, N.J., 1979.

59. P. G. Hebalkar and S. N. Zilles, *Graphical Representations and Analysis of Information Systems Design,* IBM Research Report RJ 2465, Poughkeepsie, N.Y., Jan. 1979.



Alan L. Davis is an associate professor of computer science at the University of Utah. His current research interests include distributed architecture, graphically concurrent programming languages, parallel program schemata, device integration, asynchronous circuits, and self-timed systems. He has been a National Academy of Science exchange visitor and a visiting scholar in the Soviet Union, as well as a guest research fellow at the Gesellschaft fuer Matematik und Datenverarbeitung in West Germany.

Davis received a BS degree in electrical engineering from MIT in 1969 and a PhD in computer science from the University of Utah in 1972.



Robert M. Keller is a professor of computer science at the University of Utah. From 1970-1976 he was an assistant professor of electrical engineering at Princeton University. His primary interests are in asynchronous distributed systems, including their theory, implementation, verification, programming, and applications. Currently, these interests are manifest in the FGL/AMPS project, which entails research in construction of a usable general-purpose applicative language and in its support on a distributed multiprocessing system.

Keller received the MSEE from Washington University in St. Louis and the PhD from the University of California, Berkeley.