Iterative Constructs in the Visual Data Flow Language

Mikhail Auguston, Alfredo Delgado
Department of Computer Science, New Mexico State University
Las Cruces, NM 88003-0001, USA
e-mail: {mikau, adelgado}@cs.nmsu.edu

Abstract

We suggest a solution for iterative processing in data flow diagrams based on the notion of a conditional data flow switch, and a specialized iterative construct based on pattern matching for vectors and matrices. Both of these constructs can be seamlessly incorporated into a data flow visual programming language. We demonstrate how these constructs may be used to reveal the spatial/temporal dualism of data streams.

1 Introduction

The design of the V visual data flow language [4] is an experiment with a visual representation of dependencies between data and processes. Data-flow diagrams are most commonly used to represent those dependencies in visual programming languages, for instance, in LabVIEW [5], and Prograph [8].

Iterative control constructs in data flow programming languages have always been a challenging part of language design. In this paper we consider such data structures as sequences (vectors), and 2-dimensional matrices. The aim of our work is to demonstrate how iterative control constructs typically used for sequence and matrix traversal could be visualized and adapted into a data-flow paradigm. The motivation behinds this is to achieve a clear and straightforward correspondence between the program source code (the data flow diagram in this case) and the order of program execution.

The use of a pattern matching mechanism for vectors and matrices eliminates the need for explicit index variables and makes the position of adjacent items within an aggregate visible and easy to understand. It is especially useful in the case of two-dimensional objects (matrices). Patterns also introduce temporary names for the items.

Pictograms and text can be combined together. Simple expressions could be better rendered as a plain text.

In many cases an iterative algorithm description for vectors and matrices is preferable to recursive descriptions. The V language allows combinations of iteration and recursion.

Two-dimensional notation makes it possible to introduce such useful control structure as iteration synchronization. This provides a way to describe operations that involve multiple vectors or matrices.

This paper presents some of the basic V language constructs and a collection of examples demonstrating how these language features fit together.

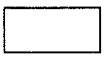
2 Data Flow Diagram Notation

A program in V is rendered as a two-dimensional data flow diagram that visualizes the dependencies between data and processes. The diagram defines the order of function calls and the data dependencies between function calls.

The data flow diagram supports the possibility of parallel execution of threads within the diagram. This approach to Visual Programming Language design has became quite common in recent years, see e.g. [5], [10], [13].

The data flow paradigm is closely related to the functional programming paradigm [6]. Diagrams may be nested, and actually are similar to the notion of procedures in common programming languages. Diagram calls may be recursive.

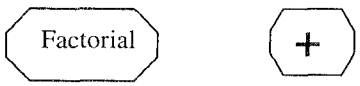
The following pictograms are used in the rest of this paper.



data item box, denotes a value (scalar or aggregate)

An operation (user-defined or built-in) is represented by an octagonal box with a name or description of

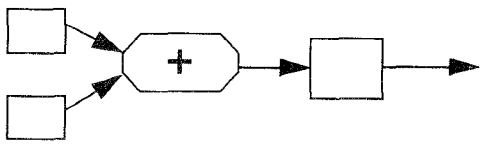
an operation inside:



Semantics. An operation box has input and output ports to connect the operation with its input and output data. An operation box fires when and only when the following conditions are satisfied:

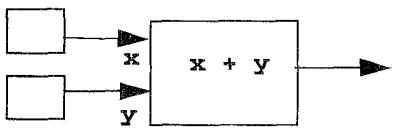
- all input values are delivered to the corresponding input ports;
- all output values produced at the previous execution cycle are consumed by input ports of the connected nodes downstream;

It is convenient to draw data flow in the diagram from left to right. Here is an example of a diagram that adds two numbers:



There is never more than one data item in the channel between the sender's output port and a receiver's input port.

The previous example can be presented in the abbreviated form by providing names at the input ports.

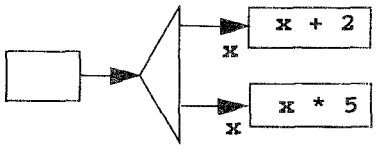


Input values denoted by **x** and **y** are added using predefined operation '+'. Box expression represents the result of an expression evaluation.

Semantics. Names defined at the input ports of a data box are visible only within this box.

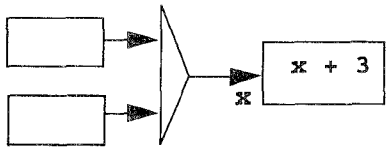
We'll use unary operations, e.g. the C-like decrement operation (--) as well.

The *fork* pictogram receives a single value and passes several copies of this value downstream.



Semantics. The fork node fires when it's input port receives a value and all output values from the previous cycle are consumed by input ports downstream. To certain degree the fork notion may be considered as an analog of a transition in Petri net [14].

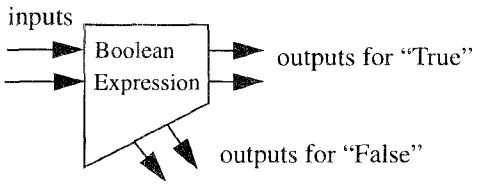
The *merge* node fires when at least one input port receives a value and passes it to the output port.



Semantics. It is assumed that the merge node preserves the temporal order of data items arriving at its input ports (the order is nondeterministic if two or more inputs arrive "at the same time"). The fairness property guarantees that each input value will be processed. This implies that the merge node maintains a queue no longer that the number of input ports. To certain degree the merge notion may be considered as an analog of a place in Petri net [14].

2.1 Conditional data flow switch

In order to be able to draw diagrams with branches and loops we need a conditional flow switch.



A conditional switch has several input ports (at the left side of the pictogram) and the same number of output ports on each of "True" and "False" sides. Some of output ports may be left unconnected.

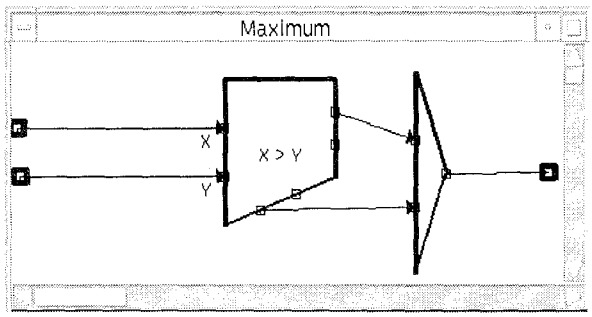
Semantics. The Boolean Expression is evaluated and the flow of input values is switched either to the "True" or to the "False" side. We believe that this notation is more consistent with respect to the principle of data flow comparing, e.g. to Show-and-Tell [12], or to Prograph [8] since it preserves the data flow uninterrupted and simply redirects it depending on the value of condition. This notation

is pretty similar to the distributor notion presented in [9], [11], although in our case the control data is not separated from the controlled one. This feature makes the conditional data flow switch a quite useful control construct for a general iterative constructs and as a synchronization node for parallel computation threads.

3 User-defined operations (diagrams)

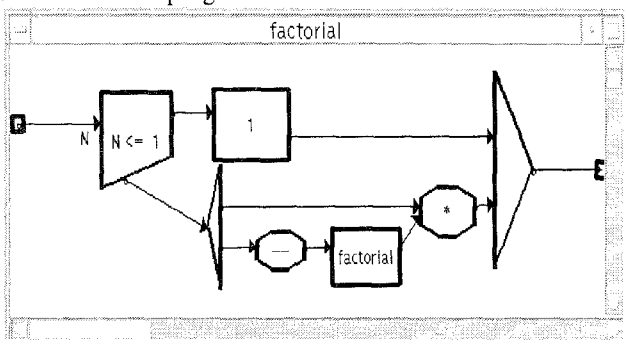
The diagram corresponds to the notion of procedure in common programming languages. We assume that input ports of the diagram are at the left side and output ports are at the right. An instance of a diagram is activated when all input values are delivered to its input ports and all output values produced during the previous call are consumed by the input ports of the connected nodes downstream. A diagram does not retain any data items from the previous calls on its data flow.

Example. The following diagram computes a maximum of two numbers.



This diagram may be called as:

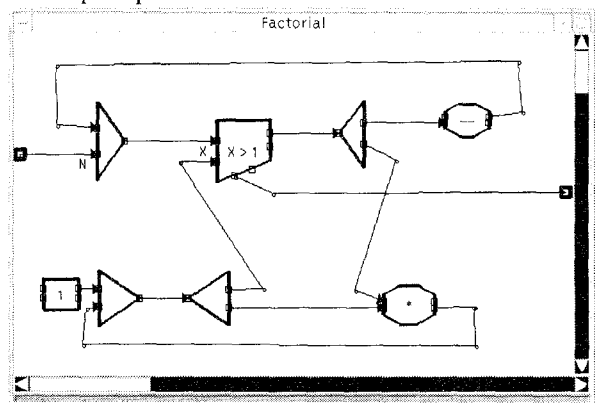
Example. This is the most common recursive version of the Factorial program.



4 Iterative construct

Conditional data flow switch may be used to control loops in the data flow diagrams.

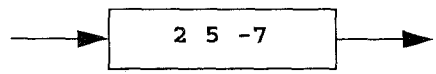
Example. A diagram that computes a factorial with an attempt to parallelize some threads.



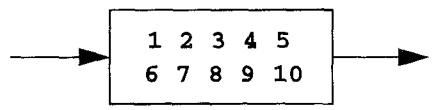
The constant 1 value in the left lower corner is injected in the flow only once, at the beginning of the diagram execution.

5 Vectors and Matrices

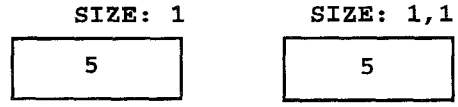
The value of a data box may represent a vector or a matrix. For example the vector [2, 5, -7] may be rendered as:



A matrix is rendered as a two-dimensional object:



To distinguish scalars, vectors, and matrices in the case when it might cause a confusion, explicit indication of aggregate size can be used. The following data boxes denote a vector, and a matrix, correspondingly.

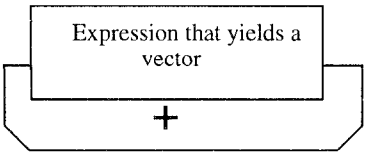


6 Regular Computations

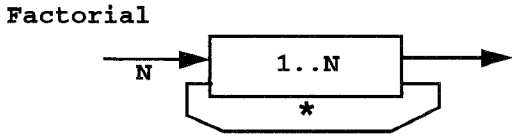
Regular computations, such as applying operations +, *, MIN, MAX, Boolean AND, Boolean OR to the whole vector or matrix, or along some dimension within a matrix

are visually represented as follows.

Sum of vector elements.



Example. Yet another way to describe factorial function in our formalism is as follows:



The `1..N` expression yields a sequence of integers from 1 to N.

7 Iterative patterns

A pattern provides temporary names for the values associated with the current item. The iterative pattern also defines the order of iteration performed on a vector or matrix.

a pattern that matches a single data item (scalar or aggregate)

a pattern that matches a group of values (e.g., a sequence within a vector, a row or a column within a matrix, or a submatrix within a matrix)

A name followed by ':' and placed inside a pattern box denotes the value or the set of values within this box.

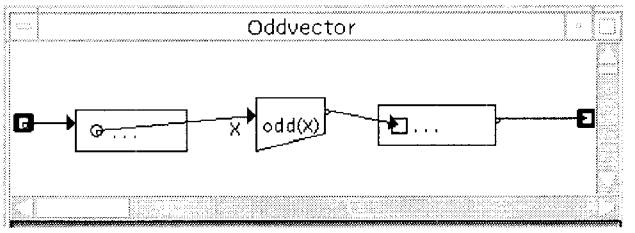
The *iterative pattern* defines an iteration over a vector or over a row or a column within a matrix.

This pattern defines a vector traversal from left to right (from the elements with the smallest index values towards the largest index values). The current iterative item is given a temporary name x, which is visible downstream of the diagram in operation and data nodes immediately connected to the pattern node. Ellipses used in the iterative patterns make the iteration description more visible by providing the direction of the iteration. The use of ellipses also solves the problem of nested iterations, e.g. within a matrix.

8 Synchronization

An iterative pattern in one data aggregate box and a data box in an another data aggregate box connected in the diagram are *synchronized*. This means that items in each of the aggregates are visited in the same order. The value of the second synchronized aggregate is constructed from the value of the first aggregate.

Example. Select all odd numbers from a vector. This diagram is an analog of the list comprehension operation in functional languages like Miranda [6].



This construct is similar to the partition multiplex with a list annotation in Prograph [8] which applies a predicate to each element of the list, and assembles the results into a new lists. It should be mentioned that Prograph's

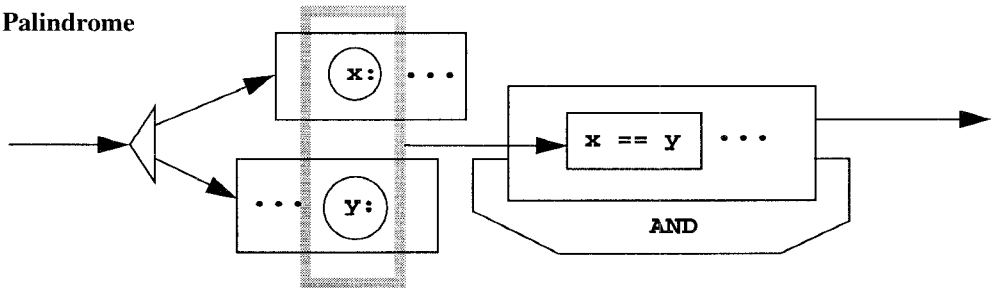
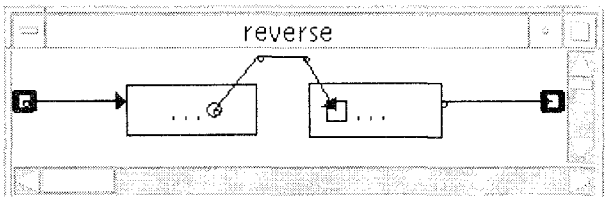


Figure 1. A diagram for Palindrome predicate

construct does not provide any elaborated pattern matching. As the following examples demonstrate the iterative pattern in V provides much more flexibility.

Example. To reverse a vector.



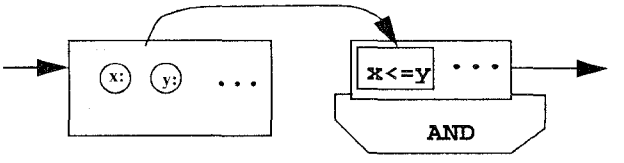
In this example the iteration over the first vector proceeds from the end towards the beginning of vector, and the iteration over the second vector proceeds from the beginning towards the end.

The pattern may involve more than one aggregate in the diagram. In this case the iteration over multiple aggregates may be synchronized.

Example. A diagram checking whether a vector is a palindrome is presented on the Figure 1. The shaded frame denotes the virtual group of objects which are synchronized during the iteration. The number of iterations is determined by the number of items in the shortest aggregate.

The iterated pattern may denote a group of objects.

Example. To check whether a vector is sorted in an ascending order the following iterative pattern may be applied. The iterated pattern matches two adjacent vector elements. The first iteration matches first and second vector elements, the second iteration matches second and third vector elements, and so on.



Each iteration adds a Boolean value to the resulting vector. The resulting Boolean vector is collapsed to a single Boolean value by the aggregate AND operation.

9 Object Manipulation

Pattern boxes that match a group of values (a sequence within a vector, a row or a column within a matrix, or a submatrix within a matrix) may be used for creating new objects as shown in the example of Quicksort algorithm in Figure 2.

In the partition diagram the iteration over the input vector starts with the second element as suggested by the patterns. The data item denoted by the **e** pattern matches the first vector element, and is in the scope of each iteration, and can be referred to by the conditional switch box.

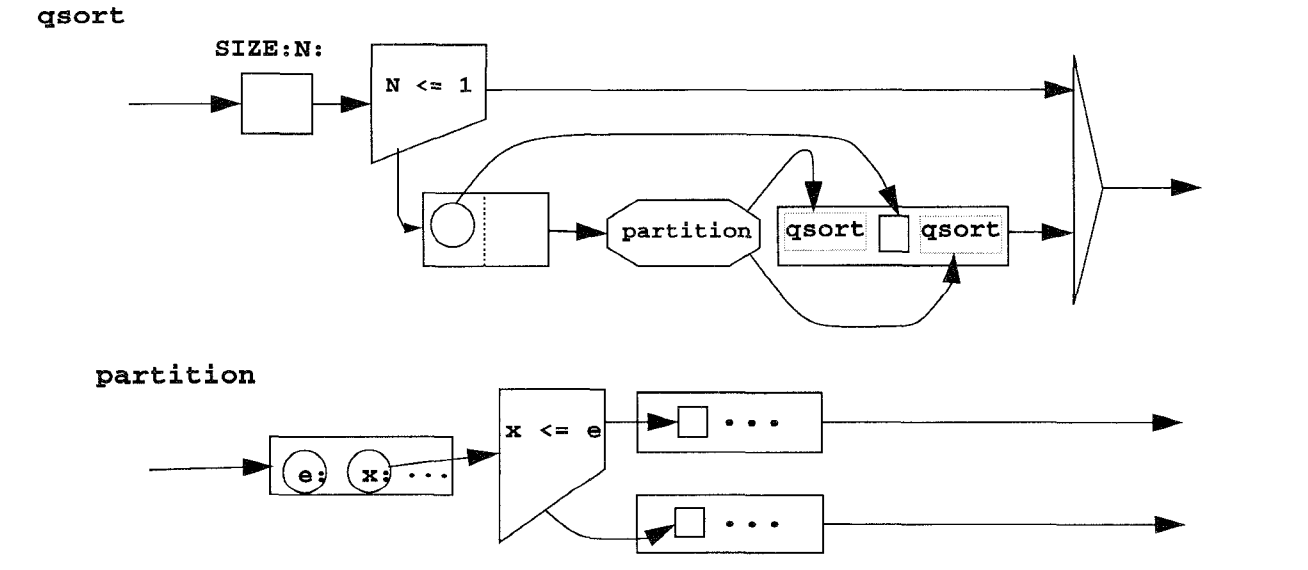
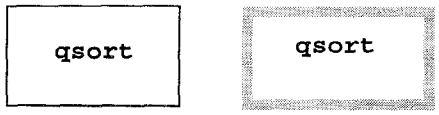


Figure 2. Quicksort algorithm

The solid data box denotes an object, e.g. the result of application of sort function, although the shaded data box denotes the sequence of elements within the vector. This sequence may be manipulated as a whole, e.g. may be incorporated into the resulting list. The first pictogram



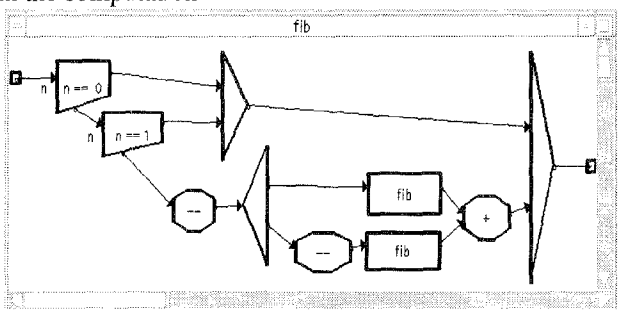
here stands for a result of qsort procedure, which is a vector; the second stands for the group of elements (a sequence) that constitute this vector.

As appears from this example the append operation in V has a simple visual representation.

Example. Figure 3 presents a diagram to calculate matrix determinant. This example has evolved from an example in [17]. Iterative pattern describes the synchronized iteration over the matrix first row. Patterns of submatrices S1 and S2 provide an easy and transparent way to describe matrix concatenation. Pattern `I` associated with the pattern `e` provides a name for the corresponding column index value.

10 Temporal streams vs. spatial sequences

The recursive version of a program computing Nth Fibbonacci number may be rendered as the following diagram. It clearly indicates presence of two parallel threads in the computation



Fibbonacci problem provides a good opportunity to discuss temporally-dependent iterations, where the outcome of a cycle depends on one or more of the previous cycles [2]. Typical solutions to this problem in data flow languages assume some means to refer to the values produced on the previous iteration of the loop. LUCID [16] considers variable names as placeholders for the temporal streams of values and provides powerful means to manipulate streams. Val [1] and Id [3] data flow languages provide

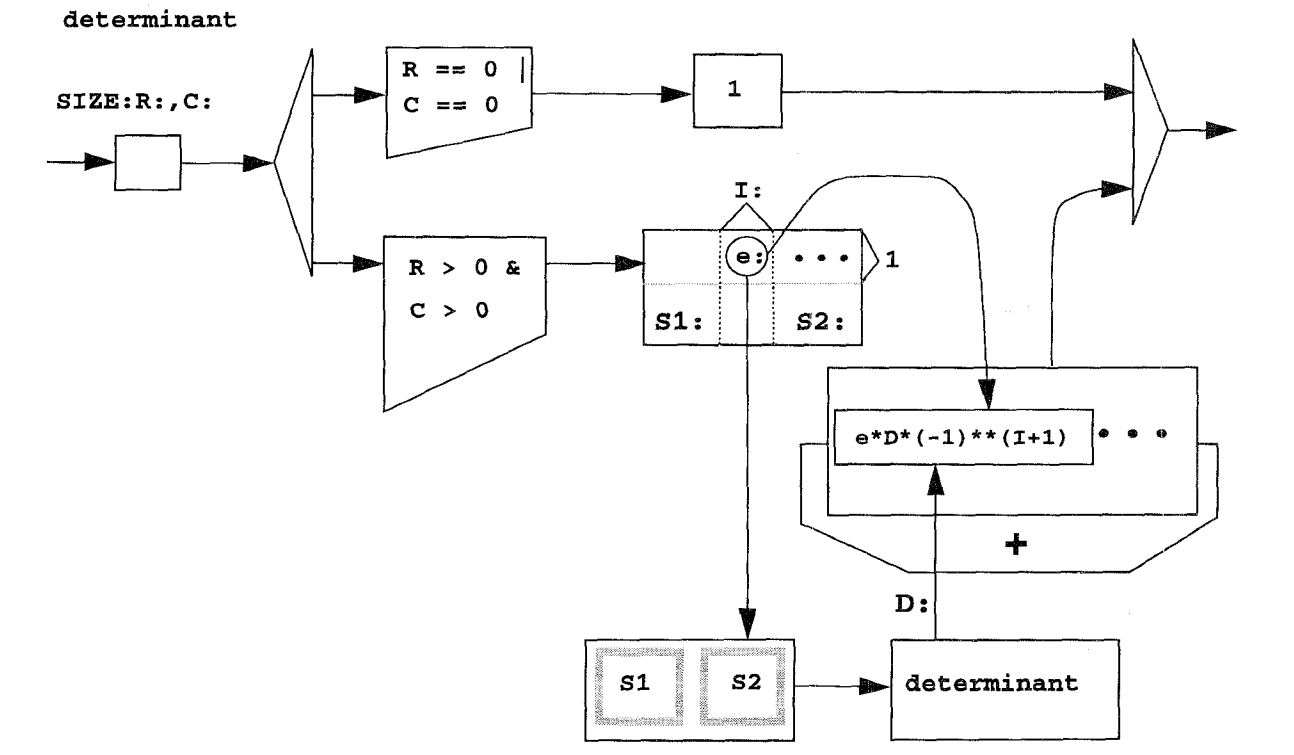


Figure 3. A diagram for matrix determinant computation

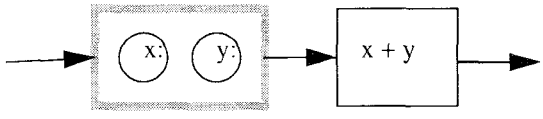
a form of multiple assignment statements inside special iteration construct that allows to refer to the values of previous iteration. Sisal [15] parallel functional language provides special keyword **old** to refer to the previous value of a loop variable.

Yet another way to incorporate iteration in the data flow framework is to adopt some form of imperative loop construct. In LabVIEW [5] boxes for FOR-loop and WHILE-loop represent an unfolding sequence of data flow diagrams where outputs of one diagram are coupled with inputs of the next. Similar iteration constructs are in Show-and-Tell [12]. In Forms/2 [2] a different approach is taken, in order to avoid cluttering, diagram cells (representing the data items) are not connected by arrows and previous iteration values are referred to by using a name and a relative iteration index, e.g. `Fibonacci [-1]:T`.

All these solutions are based on some form of temporal references to the previous iteration values. These constructs don't provide clear and visible correspondence between the program text (a two-dimensional diagram in the case of visual language) and the order of program execution. In the following sections we introduce some pictograms that visualize the notion of temporal value stream which, as we have seen, is the central issue for the iteration problem.

10.1 Buffers in data flow diagram

Quite often we need simultaneous access to several adjacent elements on the input stream. We introduce a special pictogram to denote a buffer that can hold several consecutive items from the data flow. This pictogram is similar to the pattern pictogram used within vectors and matrices to denote a group of elements, but this time it is placed outside of any data box and represents a virtual group of data items. Suppose that we need to sum up each pair of two adjacent elements from the data flow and generate a stream of corresponding sums. It could be achieved by the following diagram.



The pictogram that denotes a group of two adjacent items represents a buffer that holds two adjacent elements of the input stream and provides the box downstream with access to these two elements. In this example a stream of adjacent element sums is generated on the output. The buffer becomes available for the downstream box only when it accumulates two elements. After the downstream box has consumed the contents of the buffer, the buffer is ready to accept the next input item and shifts by one position.

The pictogram introduced above can be considered as a visualization for the concept of temporal stream and provides a clear visual construct for accessing two adjacent elements of the input stream consistent with the semantics of the V language. This construct can be used in the case of a buffer of arbitrary limited length.

Now we are in position to give yet another version of Fibonacci diagram in Figure 4. This diagram generates an infinite stream of values 2, 3, 5, 8, 13, ...

Similar iterative situations are quite common, so it makes sense to introduce yet another abbreviation. Following SequenceL [7] we'll call it a *generative construct.* Generation of first N Fibonacci numbers could be rendered

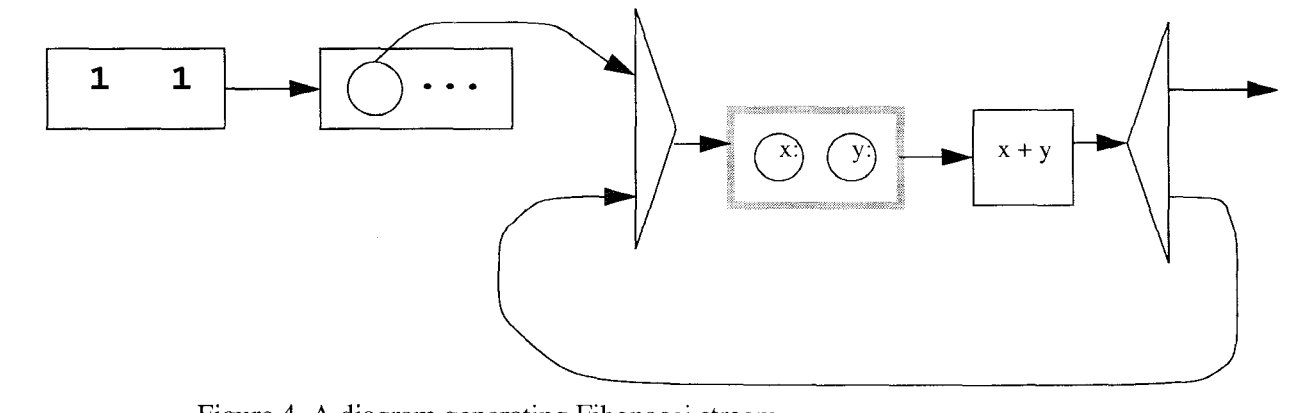
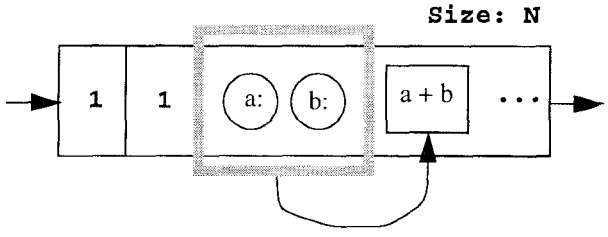


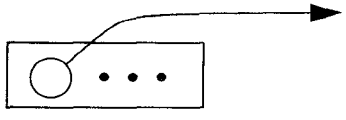
Figure 4. A diagram generating Fibonacci stream

by the following pictogram.



This construct defines an iteration of a pattern that matches two adjacent elements over a vector with initial first elements 1, 1. After each pattern matching a new element (a sum of the two elements at the current end of vector) is appended to the end of vector. The iteration terminates when the size of the vector reaches N.

The following diagram illustrates how to use an iterative pattern in order to convert a sequence of items into a stream.



11 Conclusions and future work

This paper presents visual constructs for iteration and for vector and matrix processing that emphasize the correspondence between the program text and the order of program execution and could be seamlessly incorporated into data flow visual language.

We are much obliged to Daniel Cooke, Juris Reinfelds, Joe Pfeiffer, and Karlis Kaugars for their kind support and suggestions during this work. The work on the V visual programming language has been supported during Summer 1996 by PACES (Pan American Center for Environmental and Earth Studies) and NASA. An early version of this paper (with additional chapters on multiset processing) has been presented at the 8th Israeli Conference on Computer Systems and Software Engineering in Herzliya, June 1997.

The prototype of the V language has been designed in JAVA by Alfredo Delgado and Shridhar Bidigalu. The first version of a graphical editor and a simple interpreter supports only integer data type and does not support advanced patterns. We continue to work on the language implementation and plan to implement the back end as a compiler to the Sisal [15] parallel functional language.

References

[1] W.Ackerman, "Data Flow Languages", *IEEE Computer*, February 1982, pp.15-25.

[2] A.Ambler, M.Burnett, "Visual Forms of Iteration that Preserve Single Assignment", *Journal of Visual Languages and Computing*, vol.1, 1990, pp.159-181.

[3] X.Arvind, K.P.Gostelow, W.Plouffe, "An asynchronous Programming language and Computing machine", Department of Information and Computer Science *Technical Report 114a*, University of California, Irvine, Dec. 1978.

[4] M.Auguston, The V experimental visual programming language, Computer Science Department, *Technical report NMSU-CSTR-9611*, New Mexico State University, October 1996.

[5] E. Baroth, C. Hartsough, "Visual Programming in the Real World", in *Visual Object-Oriented Programming, Concepts and Environments* (ed. M. Burnett, A. Goldberg, T. Lewis), Manning 1995, pp.21-42.

[6] R. Bird, P.Wadler, *Introduction to Functional programming*, Prentice Hall, NY, 1988.

[7] D. Cooke, "An introduction to SequenceL: a language to experiment with constructs for processing nonscalars", *Software: Practice & Experience*, vol. 26(11), pp.1205-1246, November 1996

[8] P.T. Cox, F.R. Gilles, T. Pietrzykowski, "Prograph", in *Visual Object-Oriented Programming, Concepts and Environments* (ed. M. Burnett, A. Goldberg, T. Lewis), Manning 1995, pp.45-66.

[9] A.L.Davis, R.M. Keller, "Data flow program graphs", *IEEE Computer*, vol.15, 1982, pp.175-182.

[10] Glaser H., Smedley T., "PSH - the next generation of command line interface", in *Proceedings of the 11th International Symposium on Visual Languages*, VL'95, IEEE Computer Society Press, 1995, pp. 29-36.

[11] D.Hills, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", *Journal of Visual Languages and Computing*, vol.3, 1992, pp.69-101.

[12] T.D.Kimura, J.W.Choi, J.M.Mack, "A visual language for keyboardless programming", *Technical report WUCS-86-6*, Department of Computer Science, Washington University, St. Louis, Missouri 63130, 1986.

[13] T. D. Kimura, "Object-Oriented Dataflow", in *Proceedings of the 11th International Symposium on Visual Languages*, VL'95, IEEE Computer Society Press, 1995, pp.180-186.

[14] J.Peterson, *Petri net theory and the modeling of systems*, Prentice-Hall, 1981.

[15] S.Skedzielewski, "Sisal", in *Parallel Functional Languages and Compilers*, (ed. B.Szymanski), Addison-Wesley, 1991, pp.105-158.

[16] W.Wadge, E.Ashcroft, *LUCID, the dataflow programming language*, Academic Press, 1985.

[17] R. Yeung, "MPL - A Graphical Programming Environment for Matrix Processing Based on Logic and Constraints", in *IEEE Proceedings of the Workshop on Visual Languages*, 1988, pp. 137-143.