PROGRAPH: A FRIENDLY TOOL FOR MICROCOMPUTER SOFTWARE ENGINEERING

Tomasz Pietrzykowski
School of Computer Science
Acadia University, Wolfville, Nova Scotia BOP 1XO

Stanisław Matwin
Department of Computer Science
University of Ottawa, Ottawa, Ontario K1N 9B4

Tomasz Müldner
School of Computer Science
Acadia University, Wolfville, Nova Scotia BOP 1XO

ABSTRACT

The programming language PROGRAPH has been designed as a programmming tool for fifth generation computers. It is particularly suitable for data flow environment. The paper presents the basic concepts of PROGRAPH as well as a brief overview of a user-driven computer program for creating and editing of prographs. A number of examples is included.

This work has been supported by NSERC grant No. A5551.

PROGRAPH: UN OUTIL CORDIAL POUR LA CONCEPTION DE LOGICIEL DE MICROORDINATEUR.

RESUME

Le langage de programmation PROGRAPH fut conçu comme un outil de programmation pour la cinquième génération d'ordinateurs. Il est particulièrement bien adapté pour un environnement de transmission de données. On présente les concepts de base du PROGRAPH avec les grandes lignes d'un logiciel sous le contrôle de l'usager pour créer et éditer des "prographs". On illustre la présentation avec de nombreux exemples.

INTRODUCTION.

One of the recent developments in hardware is the concept of so called data flow machines. They involve a number of dedicated processors which may run in parallel. These advances in computer architecture require new software tools. Since the classical von Neumann's instruction counter does not exist in data flow environment, linear text does not seem any more to be the appropriate medium for program representation. The way humans interpret the text in a left-to-right fashion is compatible with von Neumann's word-at-a-time computational paradigm. If program graphs are used instead of program text, advantage can be taken of potential parallelism of such a representation.

Some explanation of basic data flow concepts seems to be in order. We shall illustrate them on the example of the well known sorting algorithm. The underlying idea of this algorithm is to split the list into two halves, sort each half (recursion), and then merge both sorted lists into the final result. One can notice that the algorithm itself does not require any ordering of the two instances of sublists sorting. As a matter of fact, it is completely irrelevant for the algorithm whether the sorting of the first sublist is followed by sorting of the second sublist, or the process is organized the other way around. However, in classical sequential programming languages one must make that decision. In the data flow model this type of irrelevant sequencing of operations does not concern the user. In data flow environment the notion of an operation "preceding" some other operation does not even exist: an operation can be executed as soon as its arguments are available. Therefore a number of operations may be executed in parallel. This may greatly improve program efficiency.

More detailed investigation indicates that in many cases a further improvement may be achieved by combining the right blend of data flow and demand flow rules. Demand flow means that, roughly speaking, an operation is executed as soon as its output (results) is required as input by some other operation. Innovative character of PROGRAPH is due, among others, to the fact that its underlying execution rules combine both data flow and demand flow principles.

PROGRAPH is also a functional language in the sense that it uses combining forms for creating programs. Several functional languages have been defined in the span of the last twenty years [Henderson], [Backus]. The main features of functional programming which designers of these new languages draw upon include absence of variables, inherent modularity achieved by the widespread use of functions, stron predefined operations and word-at-a-time style of programming. Although all these features seem to be highly attractive, in our opinion, practical programming in these languages tends to be a difficult and cumbersome task, mainly because of the notation involved. These problems seem to be at least alleviated if a graphical program representation is used. First step in that direction has been done by the group at the University of Utah [Davis, Drongowski], [Davis, Keller], [GPL]. Their language, GPL, follows strictly data flow rules. The implementation is divided into two parts: graphical front-end, running on DEC-10, and back-end, running on

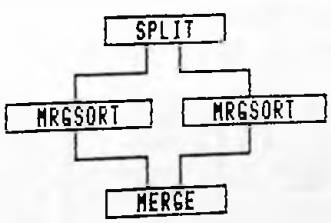
DDM1 - a data flow machine custom-built for the project [Davis].

Our language, PROGRAPH, is yet another step in the direction initiated by them. In PROGRAPH, the expressive power of GPL is enriched by a number of concepts: possibility of subroutine-like user definitions, recursion, high level control operations, synchronizing tools. Moreover, PROGRAPH provides the user with a new concept of "applicative updates", which, to our knowledge, is a novel feature. Applicative languages leave the state of computation (meant as a memory snapshot) unchanged, except for the function result, which is clearly a severe limitation in a number of applications.

BASIC CONCEPTS OF PROGRAPH.

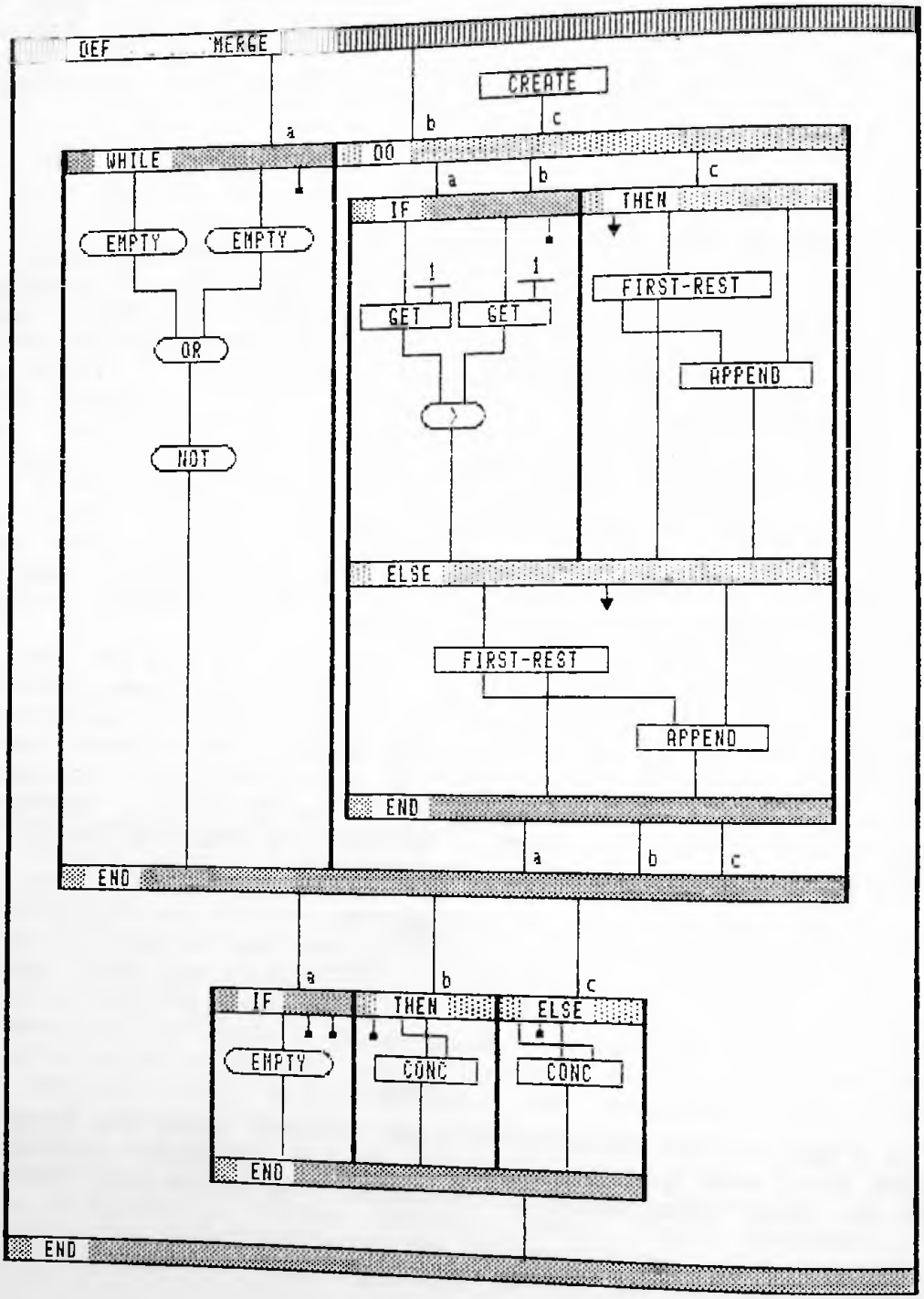
We shall introduce basic notions and ideas of PROGRAPH using as illustration the defintion of sort-by-merge algorithm in our language. Basic building blocks of PROGRAPH are boxes connected by wires. There are three classes of boxes: atomic (predefined) boxes, user-definition boxes and complex boxes. The atomic boxes are calls of predefined operations, as well as calls of user-defined operations. Complex boxes define operations introduced by the user or specify control flow in some typical situations (e.g. IF...THEN). The whole algorithm is expressed using the complex box IF...THEN...ELSE. Input of the complex box is also considered as input of the operations occuring inside IF, THEN, and ELSE, referred to as C, T1, T2, respectively. Output of the whole complex box is either the output of T1, or the output of T2, depending on the (logical) value of the output from C. The abbreviations IF...THEN is used when T2 represents an identity operation.

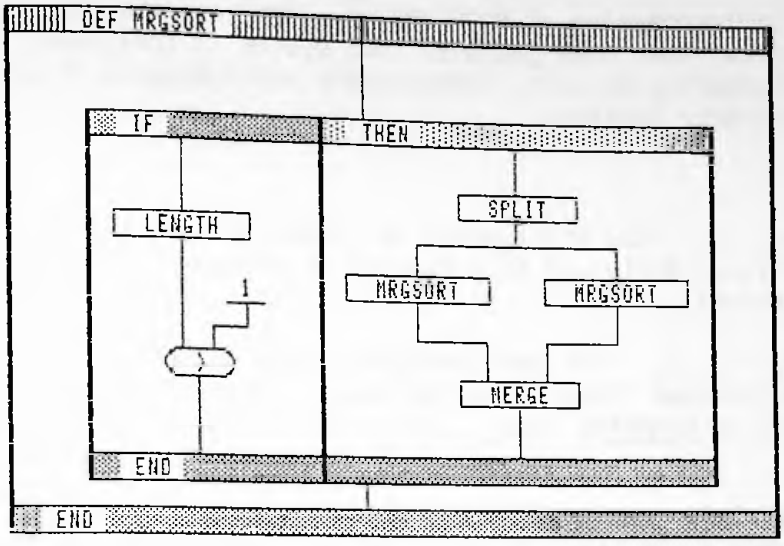
LENGTH is a predefined atomic operation on lists with the obvious meaning. Although our language is typeless, predefined operations with logical results, e.g. ">", are enclosed in oval boxes for readability. The constant "1" occuring as the second argument of ">" above has a particular interpretation when it comes to data flow: it is understood that it permanently feeds its output with a constant result. Therefore, ">" may be fired immediately after it receives the result of LENGTH. The following picture represents T1 of IF...THEN box:



SPLIT is a user-defined operation with two outputs, being the alves of its input list. When splitting is completed, two independent intances of MRGSORT are fired simultaneously. The reader may notice that PROGRAPH

allows for recursive definitions. Merging of sorted half-lists can start only after both of them are sorted. This is graphically reflected in the fact that both instances of MRGSORT direct their outputs as inputs to MERGE. To complete our example we show first the definition of MERGE in PROGRAPH, followed by the definition of MRGSORT.





INPUT PROTOCOL OF PROGRAPH.

In this section we shall outline how does one "prograph" (as opposed to program) in our system. The user draws his prographs (rather than programs) on the screen and the drawing process is driven by instructions entered by the user. There is a three-level hierarchy of instructions:

Box level:	Quit	Define	Show	Print
New box level:	Box	Connect	Next	End
Wire level:	Wire	Hang	Drop	

At the "box" level the user may define a new prograph, print a prograph, display it on the screen, or delete a prograph definition from the system. At the "new box" level the user may call another prograph definition, build complex boxes, and connect boxes by wires. At the "wire" level the user may create the wires and connect two boxes, hang the wires (it reflects unchanged data and is represented by an arrow terminating the wire), and drop the wires (this reflects data no longer needed - wire ends with a square).

Let us illustrate this protocol by giving an initial sequence of comands to create an IF...THEN box. Initially the system prompts the user for the level, the user responds with D to indicate that he wants to enter the definition mode. Now the system prompts the user for the particular type of object he wants to create (box or wire). If, as in this case, a box is to be created, the user will have to hit B and then enter the name of the box. For a name of a standard box, like IF, a frame of the box is drawn on the screen. By similar commands the user may fill in the boxes and connect them by wires.

CONCLUSIONS.

A trial implementation of PROGRAPH on a PERQ computer has now been completed. Unlike the Utah project, our system is contained to one computer. It integretes editing, interpreting and debugging functions as parts of one software package.

BIBLIOGRAPHY.

[Backus] Backus, J. "Can Programming be Liberated from the van Neumann Style? A Functional Style and it's Algebra of Programs". Comm ACM 21.8, Aug. 1978, pp 613-641.

[Davis] Davis, A. L. "The Architecture of System Method of DDM1: A Recursively Structured Data Driven Machine", Proceedings of the 5th Annual Symposium on Computer Architecture, Palo Alto, Calif.: ACM, 1978, pp. 210 - 215.

[Davis, Drongowski] Davis, A. L., Drongowski, P. J. "Data Flow Computers: A tutorial and survey" Tech. Report, Utah University UUCS-80-109.
[Davis, Keller] Davis, A. L., Keller, R. M. "Data Flow Program Graphs", Computer, Feb. 1982, pp. 26-41.

[GPL] "GPL Programming Manual". Reasearch Report, Comp. Sci. Dept., University of Utah, 1981.

[Henderson] Henderson, P. Functional Programming: Applications and Implementation. Prentice Hall 1980.