November 2008

The Advantages of Data Flow Diagrams for Beginning Programming

K. S. R. Anjaneyulu
National Centre for Software Technology

John R. Anderson
Carnegie Mellon University

Published In
.

The Advantages of Data Flow Diagrams for Beginning Programming[1]

K. S. R. Anjaneyulu

National Centre for Software Technology; Gulmohar Cross Road, No. 9;
Juhu, Bombay 400 049; India.

John R. Anderson

Department of Psychology, Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract

In this paper we describe DRLP (Dataflow Representation Language for Programming) a language that we have designed. DRLP is a concise and easy to use representation for novices learning programming. At the moment, DRLP is a visual isomorph of LISP. However we believe that some of ideas in DRLP could be extended to cover other languages. We describe an experiment that was conducted to determine whether DRLP has any effect on novices' programming performance. The results suggest that DRLP has advantages in that it avoids some of the obstacles of the LISP programming language but that many of the conceptual difficulties that students have are unaffected by the choice of a linear or graphical representation.

1 Introduction

In this paper we are concerned with analyzing a graphical, data-flow (Davis & Keller, 1982) representation for programming. To permit our investigation we have developed a language called DRLP (Dataflow Representation Language for Programming). DRLP is a visual isomorph of LISP and that is the way it will be described in this paper. However we believe that DRLP could be extended to cover other languages.

Figure 1 shows a DRLP representation of a program that converts a Fahrenheit temperature to centigrade. A Fahrenheit temperature is entered in the open box. It and the constant 32 flow into the subtraction triangle. The output of this and the constant 1.8 flow into the division triangle and the output flows into the output box at the bottom. In DRLP this processing is animated and one can observe the data-flow through the graph.

The major purpose of this paper is to evaluate the strengths and weaknesses of a data-flow representation relative to a linear programming language. We have chosen to compare it to LISP because LISP is a functional language and can be fairly directly mapped onto a data-flow representation. Thus, the essential difference becomes one of a linear versus graphical representation.

---

[1]This research was supported by grant MDR 89-54745 from the National Science Foundation to the second author. The first author was supported by a United Nations Development Project Fellowship. Address for correspondence: Prof. John R. Anderson, Department of Psychology, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

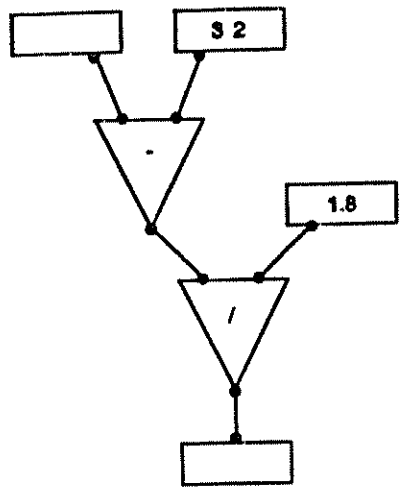

Figure 1: An example of a data-flow graph that converts Fahrenheit into Centigrade.

One complaint that is made against programming in LISP is that it is necessary to specify first the last computation which has to be performed. For instance, consider the LISP code for the Fahrenheit to centigrade computation.

(/ (- temp 32) 1.8)

The division is specified first and then the subtraction. We have observed of students working with the LISP tutor (Anderson & Reiser, 1985; Corbett & Anderson, 1990) that they constantly have difficulty with operator precedence. In a data-flow representation, one can choose to specify either the subtraction first or the division first.

2 Related Work

Before getting into a detailed examination of DRLP, we give a brief description of two related efforts. Wight, Feurzeig and Richards (1988) describe a system called Pluribus, which is a visual programming environment that can be used for teaching programming and designing parallel algorithms. In the paper they claim that Pluribus is a universal programming language and describe how it can be used as an environment to teach Logo. There are a several similarities between the language they propose and the one we have proposed. The language uses a function-machine metaphor that views mathematical functions as machines that communicate through data-flow and control flow. Machines in the language are represented as icons. Machines communicate data to each other by 'pipes'. The sequence of execution of machines is determined by 'wires' which connect machines. If two machines are not connected by 'wires', the two machines can execute in parallel. A structure consisting of several machines that function as a unit can be subsumed under a single icon. Unless otherwise constrained by inhibitory connections, machines fire when their inputs are available.

A group of people at the Cognitive Science Lab at Princeton University are working on the GIL project (Reiser, Kimberg, Lovett, & Ranney, 1988). The objective here is to develop a graphical interface to an Intelligent Tutoring System for LISP. The graphical representation

used in GIL is similar to DRLP. Initially the student is given the input and the output of the function to be defined along with a description of the function. The student has to select intermediate nodes and connect them appropriately in the graph (either top-down or bottom-up). Whenever the student creates an intermediate node he/she has to specify both the input and the output to the node. This approach was adopted to enable the system to give tutoring on individual steps. This process is continued until the graph connects the inputs and the goal. The student then translates the graphical representation into a program in LISP. The tutor monitors the translation process and provides appropriate feedback. Reiser, Beekelaur, Tyle and Merrill (1991) showed that GIL has advantages over traditional LISP.

3 The DRLP Language

In DRLP, programs are represented as graphs that look similar to data-flow graphs. To create a program, a user has to create nodes for the different inputs, outputs, functions and predicates in the program and connect them appropriately. The way in which the nodes are connected indicate the direction in which control flows. The important feature of the language, however, is that branches also carry data from one node to another.

We will first give a brief description of the types of nodes in DRLP. The actual functions and predicates will not be described, as they are roughly the same as the corresponding LISP functions and predicates. There are a few differences however, that we will point out later. DRLP currently has 5 types of nodes. They are input nodes, output nodes, function nodes, predicate nodes, and enable nodes. Each one of these will be briefly described below.

Input Node Input nodes can either be constant input nodes or variable input nodes. Variable input nodes correspond to the inputs of a program. Constant input nodes correspond to constants used in a program. For example if a program is supposed to compute the perimeter of a square, the variable input node would correspond to the side of the square and the constant input node would correspond to the number 4.

Output Node Output nodes can either be variable output nodes or constant output nodes A variable output node is used when you want the program to output a value that is a function of the inputs to the program. A constant output node is used when you want the program to output a predefined constant, if control flows to that node. In a graph there may be multiple output nodes corresponding to the different paths in the graph. However for any given set of input values (in a well-defined graph), control should flow only to one output node.

Function Node Function nodes take zero or more inputs, apply a function on the inputs, and output the value of the function.

Predicate Node Predicate nodes take zero or more inputs and check whether the predicate is true for the inputs. They have two output branches. Control is transferred along the left-hand side branch if the predicate is true and along the right-hand side branch if the predicate is false. The data that flows out of a predicate node (along either the 'true branch' or the 'false branch') is the first input to the predicate.

Enable Node The enable node acts like a buffer. It has one or more data inputs and one control input. It holds the inputs until the control input is active. When the control input is active it allows the inputs to flow out of the node.

Most DRLP functions and predicates are the visual analogs of their LISP counterparts. However there are some differences: (a) DRLP predicates do not return a Boolean value like LISP predicates. The predicates just check whether a test is true or not and transfer control appropriately. The output of a predicate node is always its first input. In LISP, predicates normally return a value that is either *nil* or not nil. (b) In DRLP, the logical operators *not*, *and*, and *or* are treated only as functions and not as predicates. In LISP *not*, *and*, and *or* can be used either as functions or predicates.

As a contrast to Figure 1, Figure 2 presents a relatively complex example illustrating use of the enable node as part of a conditional structure. However, in most cases we only need predicate nodes and not enable nodes to achieve conditionality. The DRLP graph in Figure 2 implements the solution to the following problem: "Define the function ADDLIST. It takes two inputs, an item and a list. If the item is in the list, it outputs the list unaltered. If the item is not in the list, it outputs the list with the item in the first position."

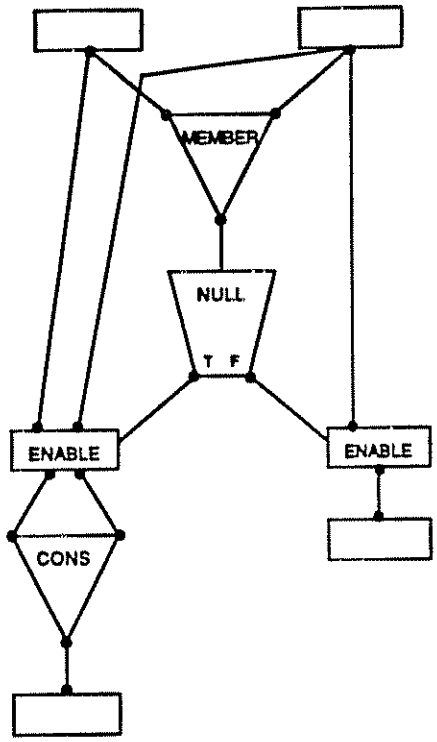

Figure 2: A data-flow graph that adds an element to a list if it is not already there.

We have created a Macintosh-based system for creating such graphs. The system allows the user to create nodes, connect the nodes, specify the values of input nodes, and execute the graph. When the graph is executed, the nodes flash in the order in which they are taken by the system (which is roughly in the same way that LISP evaluates its arguments). The values of the nodes will also be displayed in their 'windows'. After the graph is executed, the input and output values of any of the nodes may be examined, by selecting a menu item and clicking on the appropriate node. During execution the system also checks the inputs to each node to make sure that the data types of the inputs are compatible to the data types of the inputs expected by a function/predicate.

5. Conclusions

Our observations and analysis point to the following advantages of a data-flow language like DRLP.

1. By eliminating the need for explicit representation of variables and evaluation it eliminates one major difficulty faced by beginning programmers. Our empirical evaluation supported this.

2. The use of the graph structure permits easier evaluation of a program. Our results on number of iterations are consistent with this.

3. Subjects can write functions in the order of evaluation and so should not have difficulty with function composition. Almost invariably our subjects did create their programs in order of evaluation (top-down in the DRLP graphical representation).

This study did contain one major negative result with respect to the goal of using a data-flow language to teach beginning programming. This is the result in Table 2 that it seemed to have no impact in the conceptual difficulties associated with understanding what the various LISP functions do. Perhaps this should not be surprising as the representation differences concern overall program organization and not the components. We believe this negative result is an important conclusion because it shows a modularity between understanding the functions and understanding their overall organization.

References

1. J. R. Anderson, B. J. Reiser: The LISP Tutor. *Byte, 10,* 159-175, 1985.

2. J. R. Anderson, A. T. Corbett, B. J. Reiser: *Essential LISP.* Reading, MA: Addison-Wesley, 1987.

3. A. T. Corbett, J. R. Anderson: The effect of feedback control on learning to program with the Lisp Tutor. Twelfth Annual Conference of the Cognitive Science Society, 796-806. Cambridge, MA, 1990.

4. A. L. Davis, R. M. Keller: Data Flow Program Graphs. *Computer,* 26-41, 1982.

5. B. J. Reiser, R. Beekelaur, A. Tyle, D. Merrill: GIL: Scaffolding learning to program with reasoning-congruent representations. Proceedings of the 1991 International Conference on the Learning Sciences, 382-388, 1991.

6. B. J. Reiser, D. Y. Kimberg, M. C. Lovett, M. Ranney: Knowledge representation and explanation in GIL, an intelligent tutor for programming. In J. Larkin & R. Chabay, (Eds.), *Computer Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approach.* Hillsdale, NJ: Erlbaum, 111-149, 1992.

7. S. Wight, W. Feurzeig, J. Richards: Pluribus: A visual programming environment for education and research. Proceedings of the IEEE Workshop on Languages for Automation, 122-128, 1988.

Table 2 presents an analysis of the distribution of errors on the post-test. We categorized errors into those that reflected confusions about the LISP combiners (append, test, cons), confusion about the LISP extractors (car, cdr, reverse, last), confusion about other functions and predicates, other confusions, two language-specific categories, and an unclassifiable category. The DRLP-specific errors were unnecessary output nodes and misunderstanding of the enable node. The LISP-specific errors largely concerned function syntax, use of variables, and use of quotes. The LISP subjects made marginally more errors in the classifiable common categories. However, their real disadvantage was in the LISP-specific category. So, it seems that the major advantage of DRLP is that it eliminates the potential for certain errors.

Representing control flow in dataflow languages can be a tricky issue. We had tried to adopt a simple way of representing it in DRLP, using the predicate node. We observed from the experiment that DRLP subjects appeared to have no special difficulty dealing with conditionality. Our solution therefore seems to have been effective.

	DRLP Group (7 subjects)	LISP Group (5 subjects)
Combiner bugs	2.14	2.00
Extractor bugs	3.71	4.40
Other function and predicate bugs	.86	1.60
Other confusion	2.14	2.40
DRLP bugs	1.14	0.0
LISP bugs	0.0	4.40
Other (slips, not attempted, and unclassified)	1.43	2.2

Table 2: Bugs in Post-tests. Mean errors per subject.

Our observations of subjects using DRLP also indicate the unnaturalness of LISP coding order. DRLP is more flexible in terms of coding order than LISP. We noticed that the DRLP subjects almost always chose to write their programs in order of evaluation, as compared to the way the LISP subjects 'had' to write their programs. This cognitively natural order was top-down in our DRLP formalism, but Reiser et al. (1991) have observed the same result in their formalism where order of evaluations is bottom-up.

As a final point it should be noted that our results should be taken as a lower bounds on the advantage of a data-flow representation over LISP. DRLP is in first-draft and presumably would improve with further development. Also, the exercises were taken from an instructional sequence biased for a linear LISP.

After each chapter they were given a post-test of 100 marks. The questions in the post tests were similar in difficulty to the exercises they had done in the chapter. Each post-test consisted of 7 or 8 questions out of which 2 or 3 were questions where the subject had to comprehend what a program would do, given the definition of the function (or the graph for the function in the case of the DRLP group). There was no time limit for the tests, but the time taken by each subject was recorded. During the tests, the subjects were not allowed to use the system, but were allowed to look back at the chapters. After the tests were evaluated, they were shown to the subjects and the mistakes in the test explained.

4.2 Results and Discussion

Table 1 presents the time to go through the chapters. Overall there is no significant difference (t_{10}= .72) The DRLP group enjoyed a significant advantage in lesson 2 (t_{10} = 3.47; $p < .01$), while there is no significant differences for lessons 1 and 3 (t_{10} = .75 and t_{10} = .04 respectively). The advantage of lesson 2 reflects the fact that there is really little additional to learn for the DRLP group because there is no need to separately learn the abstraction involved with respect to function definition in a data-flow language.

	DRLP Group		LISP Group	
	Time	Marks	Time	Marks
Chapter 1	123.6	63.8	112.2	57.9
Chapter 2	92.9	70.7	153.6	60.1
Chapter 3	193.6	78.7	195.0	64.9
Total for All Chapters	410.0	213.1	460.8	182.9

Table 1: Mean Time in Minutes to go through Chapters and Post Test Performance

Subjects in the DRLP group averaged 1.65 iterations per problem while subjects in the LISP group averaged 2.79 iterations. An iteration is an attempt to create a program before evaluating it to see if it runs. The difference in number of iterations is significant (t_{10} = 2.53; $p < .05$). Note that this result, along with the previous result of no overall difference in time on lessons 1 and 3, implies DRLP subjects are taking longer per iteration. We assume this means that subjects are spending more time self-correcting their errors, whereas in LISP they rely on the LISP evaluator to find errors. This is consistent with our hypothesis that it would be easier to mentally evaluate data-flow representations.

There were no significant differences in time to complete the three post tests (mean for DRLP 85 min; for LISP 82.4 min). Table 1 also presents the mean number of marks obtained in the post test. The advantage for the DRLP group is not significant (t_{10} = .90); however, we will see differences among the groups when we break them down in to specific categories.

4 The Experiment

We conducted an experiment to compare students' performance using DRLP and LISP. We were hoping to get answers to some of the following questions:

(a) Will the use of DRLP have any effect on the performance time (time taken to do the exercises and the post-tests) and accuracy (performance in the post-tests)? (b) Does DRLP have any effect on the comprehension of programs? (c) What are the types of bugs that are independent of whether you use DRLP or LISP? What are the bugs that occur in either DRLP or LISP and not the other?

4.1 Method

There were two groups of subjects. One group used the DRLP system on the Macintosh and the other used a standard LISP system with a structured editor. The two groups will be referred to as the DRLP group and the LISP group. The subjects were undergraduate students who had little or no programming experience. They were paid for participating in the experiment and given a bonus based on the number of marks they obtained in the post-tests. The two groups of subjects were roughly at the same level of aptitude. The 7 subjects in the DRLP group had a mean Math SAT of 557 and the 5 subjects in the LISP group had a mean Math SAT of 576.

In the previous section we briefly described the DRLP environment. It may be useful to say a few words about the structured editor the LISP group used (see also Corbett & Anderson, 1990). The structured editor does the following: (a) It automatically balances brackets in LISP expressions for the subject. (b) It displays 'nodes' corresponding to the arguments of a LISP function. The number of nodes will correspond to the number of arguments the function accepts. If the function takes variable number of arguments the editor will generate additional nodes as the subject goes along. (c) It will highlight the nodes one at a time and allow them to type in the code.

During the experiment the LISP group had to read through the first three chapters in a LISP book (Anderson, Corbett and Reiser, 1987). The first chapter is an introduction to LISP; the second is concerned with function definition; and the third is concerned with conditionality. As they read through the chapter, they were asked to do exercises at different points in the chapters. The exercises were a subset of the exercises used in the LISP tutor. The first three chapters of the LISP book were changed so that the examples and descriptions used the DRLP notation instead of LISP. The basic content was not changed, except for omission of material which was not relevant to people using DRLP. The DRLP group read through this material and did the same exercises as the LISP group.

Subjects took from 3 to 7 sessions to complete the experiment. The length of each session was approximately 2 hours. The LISP group used the structured editor environment and worked out the exercises given in the chapters. The DRLP group used the DRLP environment to create the same programs. Both groups were asked to do the exercises by themselves without interacting with others. However if a subject got stuck while working out the exercises, the experimenter would intervene and give some hints. The subject was also allowed to ask questions when in doubt.