Introducing novices to structured programming with a "tree" programming language and a "tree" editor

Alain Giboin, Alain Michard

To cite this version:

Alain Giboin, Alain Michard. Introducing novices to structured programming with a "tree" programming language and a "tree" editor. [Research Report] RR-0255, INRIA. 1983, pp.45. inria-00076303

HAL Id: inria-00076303

https://inria.hal.science/inria-00076303

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports de Recherche

N° 255

INTRODUCING NOVICES TO STRUCTURED PROGRAMMING WITH A "TREE" PROGRAMMING LANGUAGE AND A "TREE" EDITOR

Alain GIBOIN
Alain MICHARD

Décembre 1983

INTRODUCING NOVICES TO STRUCTURED PROGRAMMING WITH A "TREE" PROGRAMMING LANGUAGE AND A "TREE" EDITOR

Alain GIBOIN and Alain MICHARD

SUMMARY

A learning environment was proposed to 11-13 years old secondary school novices to acquire the basic notions of structured programming (mainly. stepwise refinement/ top-down development and modular development). This environment consisted of : (1) The tree diagram (to provide a notation convenient to the notions). (2) Two software tools we thought they lead the beginners to make use of the tree diagram and, consequently, to apply the notions : (a) a small programming language. Flip; and (b) a "syntax editor" of Flip programs, Luciflip. (Secondarily, a third tool was provided to novices : the Unix operating system, especially its file system because of its tree structure.) Two preliminaries studies, performed to see how well beginners used this environment, led us to conclude that the simultaneous use of Flip and of its syntax editor, within the context of the tree diagram, constitutes a refined environment to introduce novices to structured programming. the syntax editor allowing to ignore programming syntactic details and to focuse on semantic features of programming -which are the most important.

RESUME

On a proposé à des débutants de 11-13 ans, élèves de collège, un environnement d'apprentissage des principes de la programmation structurée (principalement : "raffinement pas à pas", "planification descendante" et"planification modulaire"). Cet environnement comprenait : 1) Le schéma de l'arbre (notation qui illustre les notions de manière adéquate). 2) Deux outils logiciels dont on pensait qu'ils conduiraient les débutants à utiliser le schéma de l'arbre et, partant, à appliquer les notions : a) un mini-langage de programmation, Flip; et b) un "éditeur"syntaxique" de programmes Flip, Luciflip. (Secondairement, un troisième outil fut fourni aux débutants : le système d'exploitation Unix, en particulier son système de gestion de fichiers qui présente l'avantage de posséder une structure arborescente.) Dans le but de déterminer si les débutants sauraient manipuler pertinemment cet environnement, deux études préliminaires ont été réalisées. De ces études nous concluons que l'utilisation simultanée de Flip et de son éditeur syntaxique, dans le contexte du schéma de l'arbre, représente un environnement épuré permettant aux débutants de mieux comprendre, parce qu'ils les cernent mieux, les principes de la programmation structurée. On souligne le rôle important que joue, dans cet environnement, l'éditeur syntaxique. Ce dernier, en effet, évite aux débutants de s'attarder sur les aspects syntaxiques de la programmation et les oblige à se centrer sur ses aspects sémantiques —qui sont les plus importants.

1. Introduction

Among the different steps involved in computer programming [1], coding is not considered, nowadays, as the most important one. It is the "preparatory work clarifying, organizing, structuring, and representing [the] solution" (Schneider, Weingart & Perlman, 1978, p. 2) which is considered as so. This preparatory work is often referred as _developing algorithms_, or, more generally (and preferably -we think so), as _planning or designing a solution_. A method proposed to help the programmer to do it is called _structured programming_.

1.1. STRUCTURED PROGRAMMING

Structured programming can be understood as "the application of a basic problem decomposition method to establish a manageable hierarchical problem structure" (Jensen, 1981, p. 31; see also Dijkstra, 1970, Wirth, 1974). It is "the formulation of programs as hierarchical, nested structures of statements and objects of computation" (Wirth, 1974). In the basic structured programming procedure, program construction consists of a sequence of refinement steps. Each step of the construction consists of breaking a given task into a number of subtasks (Wirth, 1971). The task is broken down into simpler tasks or steps that begin to describe how to accomplish it. If these simpler steps can be represented as "primitives", it is not need to refine them any further; if not, each of

these steps have to be refine into yet simpler steps. So structured programming is also called stepwise refinement or top-down development. The latter expression has not to be taken in its strict sense. In fact, the process of breaking down a given task into subtasks has not, each time, to be carried on to the very end, because some subtasks of the task have been already refined by other programmers or by oneself, so that it is possible to use them as primitives. What is important is to design and to recognize well-defined and distinct functional units.

This is why structured programming is also referred to, but sometimes distinguished from, modular development. This technique "partitions a program into units -modules- which perform defined tasks and can be coded, compiled, tested, and executed as independent subprograms" (Lemos 1980, p. 59).

The purpose of structured programming is to make programs easier to design, to read or to understand, to debug, and to maintain. Therefore it is recommanded, therefore it is taught. However, the environment chosen to teach it do not met always learners' requirements.

1.2. LEARNING STRUCTURED PROGRAMMING AND TEACHING IT

Our aim is to design an effective learning environment for introducing novices to the structured programming method, i.e. to lead them to learn its basic aspects. To do this, we have to know how they learn. We have to refer ourselves to a model of the psychological, especially cognitive, processes involved in learning. We have chosen the theory of meaningful learning (Bransford, 1979; Mayer, 1981; Carroll & Thomas, 1982). (a) Meaningful learning is a process, called assimilation, in which the learner connects new knowledge with knowledge that already exists in memory, called "knowledge structure" or schema. In other words, assimilation to schema means that "new concepts are typically expressed in terms of old concepts --at least initially" (Thomas & Carroll, 1981, p. 239). (b) This schema is used as a "structural template for further learning" (Thomas & Carroll, 1981, id.). This leads to generation of new cognition structures by metaphorical extension.

To make the novices learn by metaphor's teaching method leads to carefully choose metaphors. Contrary to (Mayer, 1981), we do not provide them a model of the computer. The Mayer's model does not meet our goal. It applies to the acquisition of imperative programming languages. Our goal is to give to beginners a functional representation of the structured programming method, to lead them to understand it, in order to be able to use it and, later on, to apply it when dealing with more complex problems.

A representation often related to structured programming procedure is the <u>tree diagram</u>, which is emphasized as the "most effective technique [or] program development tool", for it "provides a clear graphic representation of the program structure and a notation amenable to the stepwise refinement process" (Jensen, 1981, p.43). To teach the structured programming method will consist of applying the tree diagram to the structure of the program and to the necessary process to obtain it.

1.3. TOOLS FOR LEARNING THE STRUCTURED PROGRAMMING TECHNIQUE

We propose two meaningful tools (which demand the structural point of view) to achieve this application :

(1) a hierarchical programming language (or "tree" language) : Flip; and

(2) a hierarchical "syntax editor" (or "tree" editor) : Luciflip (actually a mode of a general system for manipulating hierarchies called Ceyx).

2. The tree diagram used as a schema for structured programming learning

To make easier the learning of structured programming procedure, one has to use an adequate metaphor for representing it to the novice. The metaphor we used is the tree.

2.1. TREE DEFINITION

Generally, tree means a "branching" relationship between nodes. Formally defined, a tree is a structure consisting of a node, called the root of the tree, and of a finite set, eventually empty, of trees, called subtrees of the tree. This definition is recursive, i.e., the tree is defined in terms of trees. It follows from it that every node of a tree is the root of some subtree contained in the whole tree. A node is related to each of its subtrees by a branch. A node which has not itself subtrees is called a "terminal node" or a leaf. The nodes which are not leaves are said internal nodes or branch nodes. By analogy with the family trees, a subtree of a tree is also called his son; the former is inversely called the father of the latter, and two subtrees of the same tree are said brothers (Knuth, 1973, Meyer & Baudoin, 1978).

We shall assume that the tree structure reflects :

- a hierarchy of procedures that state how to obtain the result of a task; in this case, it is necessary to consider that the set of subtrees of a tree is arranged;

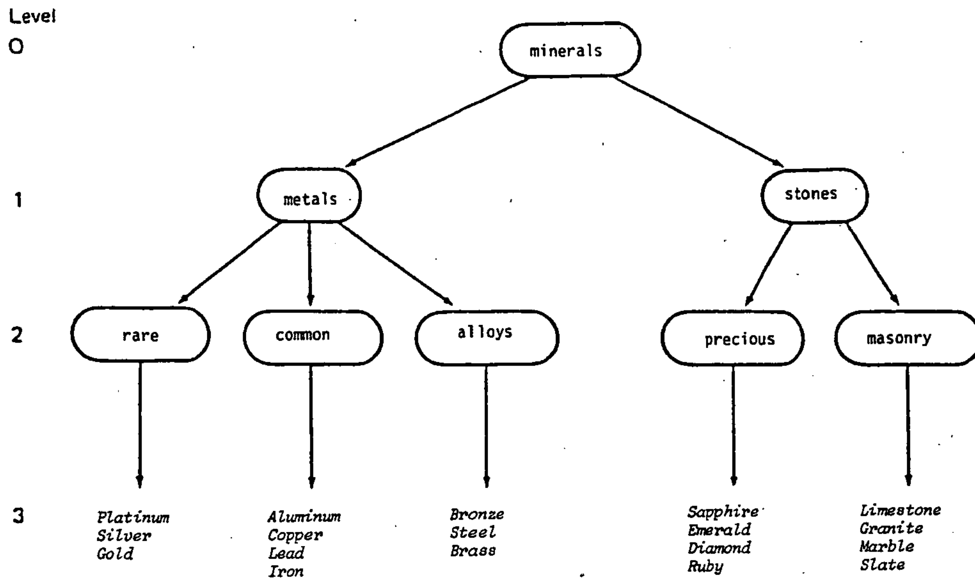
- a hierarchy of declarations that merely state what the result to obtain is; in this case. it is not necessary to consider that the set of subtrees is arranged.

We shall call procedural tree the first kind of tree, and declarative tree the second kind.

2.2. TREE REPRESENTATION : THE TREE DIAGRAM

Generally a tree is represented by a figure called tree diagram. They are many ways to draw diagrams of trees. We will consider two types of such diagrams (see figure 1) that we will call (a) graphic tree and (b) word tree (in which indentation is used to highlight the tree structure).

a) The "graphic tree"



b) The "word tree"

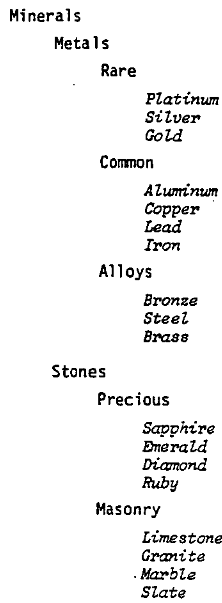


Figure 1. - Two types of tree diagram.

2.3. PREVIOUS KNOWLEDGE OF THE TREE DIAGRAM

The tree diagram is a knowledge that already exists in the memory of most individuals who met it in everyday life's activities : reading tables of contents in books, reading a royal family tree in a magazine or in a history book, parsing during a grammar exercise. and so on.

Anyway, we have observed in the two studies reported below that the subjects (12-13 years old secondary school students) seemed accustomed with this diagram and the terminology related to it. Particularly, the genealogic terminology raised immediate verbal reactions. As soon as we showed learners the pictures representing the tree diagram with genealogic terminology, we listened remarks such as :"It's a family tree, isn't it ?", or "It's like in a family. Can I say "sisters" instead of "brothers" ?"

The tree diagram was indeed a knowlege structure or schema. Doubtless, at the very beginning of learning, this schema was just about covert and rough. But the early observations we made in the first of our two studies proved that this schema could be made overt and refined.

In this first study, learners, because they had to communicate their programs to the computer, were first accustomed to some rudimentary notions of the Unix operating system they used, mainly that of "file system". A Unix file system "is arranged as a hierarchy of

directories, in the form of an upside-down tree. The source of the tree is the root, or the root directory". Each directory "can contain any number of directories, developing a branching structure" (Thomas & Yates, 1982, p. 74). So a Unix file system is a tree (see figure 2). In this "Unix tree", the leaves are the files (in which data and programs are written) and the nodes are the directories.

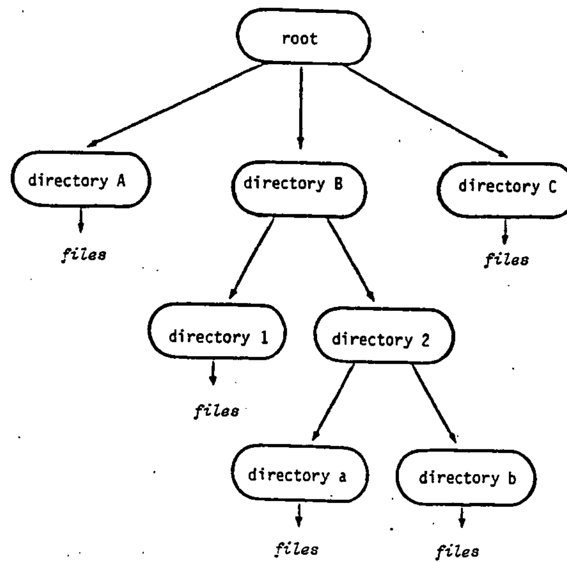


Figure 2. - The Unix tree.

Learners were taught how to move in the file system, how to change to a directory at different levels of the directory hierarchy : change to a subdirectory, change to another directory at the same level of the hierarchy, change to a superdirectory.

Indeed, thanks to the tree diagram, novices understood quickly the meaning of the Unix file system and they visualize how to move in the hierarchy of the directories. From the very beginning, to remember how to move in the Unix tree, they drew it and simulated with the finger the path they wanted to follow. Then they noted the path command to achieve it on the computer.

Discovering the Unix tree reveals to the beginners an instance of a declarative tree structure. Also it reveals to them an aspect of what is a structuring method :

- They can see that Unix users, because they have often many files, classify them by putting them into directories, and when these directories become numerous, put them into "super-directories", and so on. They were said that they will use themselves this technique later (in fact, they did not use it because they did not create a great number of files).

- They can see also that Unix users can organize themselves in groups, "subgroups" and "supergroups" within the Unix file system.

We then considered that the Unix tree belonged to the novices' previous knowledge of the tree diagram, that it represented for them another instance of the tree schema. In other words, novices generalized well the tree schema.

2.4. USING THE TREE DIAGRAM TO LEARN THE STRUCTURED PROGRAMMING METHOD

The tree diagram can then served as a structural template for structured programming learning. So we used it as a didactical diagram. Precisely, this diagram was used as a schema for learning how to program following the structured programming method and how to represent the structure of the program, for it is supposed to enhance the view that "a program is a hierarchy of structurally nested components" (Teitelbaum & Reps, 1981, p.7) and to force "a structured design process by prohibiting all but a hierarchical approach to the problem solution" (Jensen, 1981, p. 43). What we shall see in the two studies reported below.

3. Flip as a tool for learning structured programming

To learn the basic of the structured programming method. novices are invited to program the computer to design <u>flips</u> (i.e., color transparancies) by using a programming language called Flip (Kahn, 1981).

3.1. PRESENTATION OF FLIP

We have chosen Flip because :

(a) it is a small language which emphasizes hierarchical planning (or structured programming) in terms of manipulation of trees and which saves notions such as "variable", "type", "procedure", and so on;

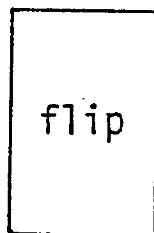
(b) it is a declarative (Du Boulay, O´Shea & Monk, 1981) or a descriptive (Green, 1980) language, as opposed to a procedural language, what meets the psychology of novices who had been showed to prefer instructions written in the form of a statement of the goal (e.g., "put all red things in box 1") rather than a procedure to achieve it (e.g., "if thing is red then put it in box 1") (Miller, 1975, cited by Du Boulay <u>et al</u>., 1981).

(c) it exists a syntax editor of Flip programs (see later).

3.1.1. What is a flip

With Flip, one can design "flips". In its simplest form, a flip is one rectangle with some graphical attributes (mainly. a frame-color, a ground-color, a text-color, a size, or an orientation). In figure 3.a is a simple flip with two graphical attributes : a black frame and a black text. More complex flips are composed of a set of horizontal, vertical, or superposed flips, as in figure 3.b.

a) A simple Flip



b) A complex Flip

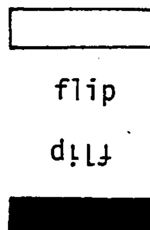


Figure 3. - Examples of Flips.

So a flip can be defined recursively as a tree : A flip tree is a structure composed of a flip and of a finite set, eventually empty, of other flip trees, called flip subtrees. Subtrees are delimitations. Leaves are graphical attributes. End leaves are text declarations.

3.1.2. The concrete syntax of Flip

In figure 4 is the concrete syntax of Flip, presented in a simplified form [2].

```
<FLIP> ::= <COLOR>
<FLIP> ::= <LINE>
<FLIP> ::= write <COLOR> <FLIP>
<FLIP> ::= frame <COLOR> <FLIP> frame_end
<FLIP> ::= paint <COLOR> <FLIP> end
<FLIP> ::= rotate <FLIP>
<FLIP> ::= horiz <NUMBER> : <FLIP> end
<FLIP> ::= vertic <NUMBER> : <FLIP> end
<FLIP> ::= cover <FLIP> with <FLIP>... end
```

Figure 4. - The concrete syntax of Flip (simplified).

3.1.3. An example of a Flip program

Programs written in Flip have the word tree form showed in figure 5.

```
horiz
  1:
    frame black
     -
    frame_end
  4:
    cover
      write black
        -flip
         -
    with
      rotate
      rotate
        -flip
         -
    end
  1:
    paint black
     -
    end
end
```

Figure 5. - A Flip word tree.

3.2. USING THE STRUCTURED PROGRAMMING METHOD TO DESIGN A FLIP PROGRAM

The logic of the Flip programming language induces the programmer to think in a hierarchical manner and to use the tree schema as we shall see by exposing the steps involved in Flip programming :

(1) Defining the problem.- The novice programmer has first to state the flip he wants or the teacher wants the computer to perform, e.g., the flip shown in figure 3.b.

(2) Planning the solution : designing the Flip graphic tree.- In the Flip context, the novice can see the sense of step-wise refinement. In fact, designing a flip consists of determining a hierarchy of declarations, precisely :

- delimitations of horizontal, vertical, or superposed rectangles;
- insertions of graphical attributes into the rectangles.

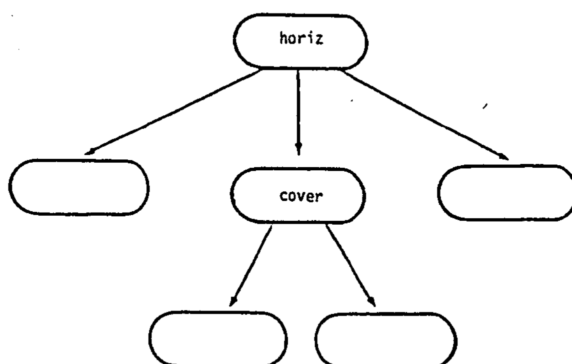
Because a flip can be defined as a tree (a) it can be represented as a tree diagram (the graphic tree), and (b) to conceive a Flip program, the programmer has to perform a hierarchical decomposition task consisting of delimitation refinements which can be represented too by the graphic tree. Each subtree of the tree can be seen as a sub-delimitation of a higher order delimitation. When the

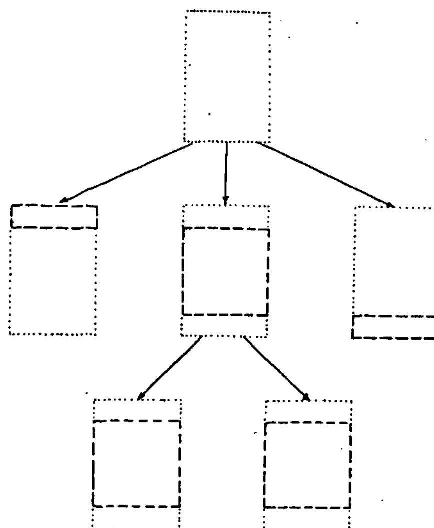
problem has been fully refined, i.e., when all the delimitations have been performed, all the leaves of the tree will contain the graphical attributes.

For instance, if he wants to program the flip in figure 3.b, the programmer has to follow the process represented in figure 6.a and 6.b.

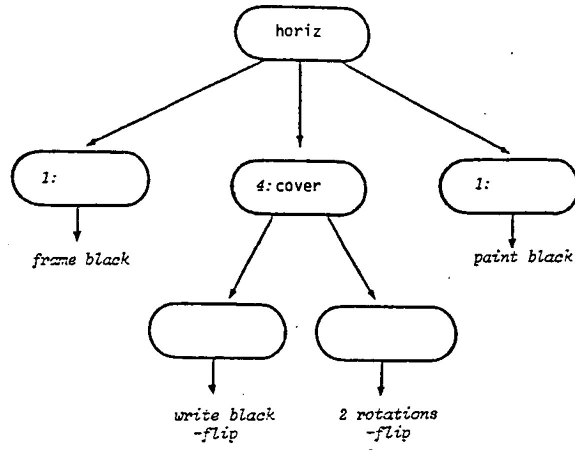
Figure 6. - A Flip graphic tree.

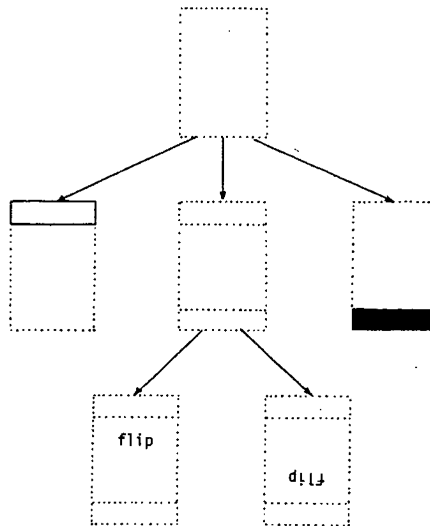
a) Hierarchical delimitation of rectangles





b) Hierarchical insertion of attributes.





At the end of the decomposition process, the flip is viewed as a hierarchy of structurally nested declarations. Also, one can design Flip programs of different complexity. Complexity depends mainly upon the number of levels of the Flip tree : 0-order flips are those composed of one

rectangle, 1st-order flips are those composed of only one set of 0-order flips, and so on.

Moreover, in the Flip context the novice can see also the sense of the modular programming technique. Because it is possible to split the flip into subflips already programmed. Call (insertion) of flips inside a flip is possible, so that recursion (insertion of a flip in itself) is, in a way, possible.

(3) Coding : designing the flip word tree.- The programmer writes his solution in the Flip programming language. In other words, he transforms his graphical tree in the word tree in which he makes use of indentation to indicate program structure (see figure 5).

(4) Editing and syntactic debugging.- To communicate the program to the computer for it executes it, the programmer has to edit it via an editor (here an Emacs-like screen editor). Here, he has to edit the Flip word tree and to write it in a file. Then he tests the program for its syntax, thanks a "pretty-print" program which allows him to have his program in the correct indented form; in other words, to have the correct word tree. If necessary, he makes corrections to the program.

(5) Execution and semantic debugging.- Finally. the novice programmer asks to the computer to execute the program on a plotter or on a color screen. If necessary, he modifies the program to have a convenient graphic tree.

3.3. A PRELIMINARY STUDY OF THE EFFECTIVENESS OF FLIP AS A TOOL FOR LEARNING STRUCTURED PROGRAMMING

This study consisted of an observation of the effectiveness of Flip for novices to acquire the structured programming procedure. The aim of the study was not to collect data obtained in conditions rigourously defined; it was mainly to follow the novices' mode of thinking, or, in other words, to disclose the cognitive representations and operations they used, whether they succeed or they fail.

3.3.1. Method

Subjects were 3 secondary school girls (12-13 years old) who had no previous experience of computer programming.

Materials.- We used a 68000 based microcomputer with a Unix operating system, a 8-color graphic plotter and a 8-color screen for flip output, and a classical video terminal for editing. We used also : 1 illustrated Flip manual; 1 list of some commands of the Emacs-like editor; 1 list of some Unix commands; 1 series of flips illustrating the Unix tree and how to move in it.

Sessions.- The sessions were collective and six in number, one each week. (A further session took place where students used the Luciflip syntax editor. See later.) Each session lasted two hours and consisted of an alternance of theoretical periods and practical periods. The first theoretical period was devoted to how to move in the hierarchical file system of the Unix operating system the beginners were using. They were provided three tree diagrams : the first represented an "abstract" tree with the words root, node and leaf; the second represented an "abstract" Unix tree with the words root, directory and file (see figure 2); the third represented a "concrete" Unix tree where the words "directory" and "file" where replaced by the names given to the directories and files of the Unix system used. Novices learned mainly how to move in this hierarchy, i.e. to go down, up, left, or right in the file tree. The remaining periods were devoted to the introduction to structured programming with Flip, as exposed earlier, with the aid of tree diagrams. We never used explicitely the words "stepwise refinement", "top-down development", or "modular development". In the practical periods, the learners were asked to apply what they have learned in the theoretical periods by solving exercises : mainly to design Flip programs of increasing complexity (from 0-order flips to n- order flips). In two exercises, however, they were invited to retrieve the structure of a program they had not conceived. They were helped by the teacher-observer when necessary. In these practicals periods. beginners also learned to use the Emacs-like editor.

3.3.2. Results and discussion

Data collection.- Data collected were : (a) notes on learners simultaneous and subsequent verbalization [3]; (b) drawings and diagrams relative to the flip design; (c) listings of programs created or modified by the subjects (which were asked to record the successive versions of a same program in different files).

Data analysis.- To determine if Flip can be an effective tool for learning the basic of structured programming method, we have to point out how this method became novices' property, i.e., how well they understood it. Practically, we have to show how fluently and correctly they used it in Flip programming. This can be done by specifying indicators revealing that learners understood the structured programming method in Flip programming and did not misuse it. So we have to define what is comprehension of structured programming method.

Understanding of the structured programming method can be split into : (1) understanding of stepwise refinement or top-down development, and (2) understanding of modular development.

(1) Understanding of stepwise refinement or top-down development is revealed here by the ability to decompose the flip drawing into hierarchically nested rectangles with their respective graphical attributes. This ability can be broken down into two main aspects :

(a) An ability to dissociate the delimitation phase from the insertion phase, that is to say, an ability to abstract the rectangles as the units which will compose the hierarchy. Indicators are : dotted lines in flip drawings (representing the rectangles limits), presence of two steps in graphic tree diagramming (first, delimitation; second, insertion).

(b) An ability to hierarchize the declarations of delimitations and to nest them. The indicator is the correlation of the position of rectangles in the flip drawing with the graphic tree or word tree (the inclusion relationships between rectangles must be reflected in the relationships between nodes in the graphic tree or the blocks in the word tree).

(2) Understanding of modular development is revealed here by the ability to design independent Flip programs and to call them in a bigger program. Indicators are : call of flips and names given to program files (some of these names must denote a part of a bigger program file).

Understanding of modular development

All subjects understood call of flips, although they used it little.

Understanding of stepwise refinement or top-down development

(a) All subjects are able to dissociate the delimitation phase from the insertion phase. However, it must be noted that, at the beginning of learning, they confused delimitation of rectangles and frame insertion : they confused drawing of limits with dotted lines and drawing a frame. For instance, one subject believed that she would obtain a frame only by delimiting a rectangle. This was because we had not emphasized enough the "invisibility" of dotted lines which delimit the rectangle.

(b) Subjects were not able immediately to hierarchize delimitations and to nest them. In the first times, we observed that the delimitations were juxtaposed instead of nested. In the case of the graphic tree, subjects drew juxtaposed graphic trees instead of a unique tree. In the case of the word tree, subjects did not succeed to indent the coded program to form hierarchical and nested blocks.

To find the hierarchy of delimitations was all the more difficult since it revealed a dissymmetry. In this case, subjects preferred to bypass the difficulty instead of overcoming it, by modifying their flip so that it revealed a symmetry. However, this difficulty was surmounted whenever we asked the learner to apply really top-down development by really decomposing the flip into subflips.

However, learners were progressively able to hierarchize, for they designed more and more complex Flip programs. But they wanted

to stop when they judged the task too difficult and a work rather than a game. The more complex Flip programs they designed were of 3rd-order. Difficulty mainly stayed in the boring aspect of the coding and editing phases.

An obstacle : the coding and editing phases.-

We observed that novices mixed up graphic tree and word tree syntax, e.g. :

- to be able later to place correctly the "end" and the "frame_end" in the word tree, one subject decided to put them in the graphic tree;
- all the three subjects wrote the low levels nodes of the graphic tree in the word tree syntax.

So we can say that subjects, when they were designing the graphic tree, were anticipating and preparing the coding and editing phases. We can say also that these phases put a brake on the novices' activities, hence on their interest or motivation. The learner has to respect the syntax of the Flip programming language, what was not necessary in the planning phase, without being prevented from lexical and syntactic errors. From which, sometimes, a reaction of annoyance such as one subject's remark that syntactic features "end" and "frame_end" are superfluous in the word tree.

Coding and text editing favours also a linear mode of thinking.

Subjects composed their programs line after line, not in a structural way. The "fliprint" program was little used to verify the correctness of indentation, but often to verify if the program will run or not. E.g., even if the the indented program was not the one expected, subjects ran their programs. Coding and editing phases distracted novices from the main features of the method we wanted to teach them. They consumed time and reduced interest. Contrary to Habrias, Levasseur & Liscouet (1982) who, observing that the graphic step seems to be a brake rather than an aid in designing computer programs, proposed to economize it by a pseudo-code step (i.e., writing a text in a code similar to the programming language), we think that it is the coding and editing phases which are a brake, and which must be economized.

4. Luciflip as a tool for learning structured programming

If neither coding nor editing represent the main activities of programming, they represent necessary activities if the programmer wants its program to be run by the computer. But we have seen that these two activities could be very tedious and consequently might slow down learning of structured programming method. To remove this obstacle should be very interesting.

It exists a kind of editor -called syntax editor- that "spares the user from mundane and frustrating syntactic details while editing programs" (Teitelbaum & Reps, 1981, p. 563). Moreover, with this editor, editing can be similar to structured programming's stepwise refinement because it uses the syntactic structure or "tree form" of the programs "to direct their creation or alteration" (Allison 1983, p. 454). The designers of such a syntax editor -the Cornell Program Synthetizer- claimed that one of its goals is to "promote[ ] programming by stepwise refinement" (Teitelbaum & Reps, 1981, p. 563). The latter, used for planning, can also be used for entering programs. So that the use of the syntax editor -called then, sometimes, tree editor-, demands also a structural point of view. Syntax editing serves the learners as a mean of perceiving what programming and program structure should be like. It allows them to think in terms of the problem to be solved, without having to deal with the syntax of the programming language -which comes close to the finding that an expert programmer encodes and processes information semantically, ignoring programming language syntactic details (Shneiderman & Mayer, 1979).

Accordingly the syntax editor can be used as a part of an environment for structured programming instruction and can be considered as a tool for beginners to learn structured programming. As it exists a "syntax editor", actually a mode of a general system for manipulating hierarchies called Ceyx (Hullot, 1982, 1983 a and b), which helps to edit programs written with the Flip programming language : Luciflip, we used it to observe its effectiveness for beginners to acquire the structured programming method.

4.1. USING THE STRUCTURED PROGRAMMING METHOD TO EDIT A FLIP PROGRAM VIA THE LUCIFLIP "SYNTAX EDITOR"

Syntax editing of a Flip program resembles structured programming of a flip. It forces its user to think hierarchically and to use the tree schema. Editing a Flip program using a Luciflip editor involves a mechanism near to the mechanism involved in the planning of the Flip program itself. In other words, the passage from the graphic tree to the word tree is made covert, or using the syntax editor almost suppresses the coding phase.

Because it forces the programmer to enter the program according to sentential forms (indentation being automatic), it is not possible to enter a syntactically incorrect program, which allows the programmer to never get lost in syntactic details. So it makes implicit the coding phase and enhances the planning phase.

4.1.1. Stepwise refinement or top-down development

Luciflip is an aid for the beginner to better understand the sense of stepwise refinement, for programs are created top-down by inserting declarations at a cursor position. As a syntax editor, Luciflip "allows a programmer to interactively develop programs in a top-down fashion" (Ince, 1983, p. 687).

Luciflip maintains a pointer to the current node; the pointer moves within nodes, in terms of the branches of the tree : move down to son; move up to father; move left or right to brother. there are one-letter to three-letter commands to move, to create nodes, and to define graphical attributes. Moreover, at the execution phase, not only the whole Flip tree can be executed, but any Flip subtree can also be executed. It can be noted here that execution can follow editing without delay. for the program is interpreted during editing.

Because portions of the program can be left out and filled in later, the novice can firstly design the hierarchy of delimitations of rectangles and, to see if this has been done well, ask to the computer to draw them. Secondly, he can insert the attributes into the rectangles.

If he wants, for instance, to design and edit the program of the flip in figure 3.b, he will follow the procedure given in figure 7.

Figure 7. - The edition of a Luciflip word tree

a) Hierarchical delimitation of rectangles

NOVICE		SCREEN	
GOAL	KEYBOARD	DIALOG AREA	WORKING AREA
To design a program called "flip"	(...)	(...)	*#&flip*
To delimit three horizontal rectangles	esc 3 H		*(horiz #& #& #&)*
To go to the second "horiz" son : *a) Go down to the first "horiz" son*	↓		*(horiz #& #& #&)*
b) Go right to the brother	→		*(horiz #& #& #&)*
To delimit two superposed rectangles	esc 2 C		*(horiz* *#&* *(cover #& #&)* *#&)*

b) Hierarchical insertion of attributes

NOVICE		SCREEN	
GOAL	KEYBOARD	DIALOG AREA	WORKING AREA
To insert the proportion of the second "horiz" son	%	proportion : _	*id.*
	4 < >	proportion : 4 < >	*(horiz* *#&* *(% 4* *(cover #& #&))* *#&)*
To go down to the first "cover" son	↓		*(horiz* *#&* *(% 4* *(cover #& #&))* *#&)*

To insert the following text : *-flip*	A	align : _	*id.*
	flip < >	align : flip < >	*id.*
		align : _	*id.*
	< >	align : < >	*id.*
		align : _	*id.*
	< >	align : _	*(horiz* *#&* *(% 4* *(cover* *(align* *"flip"* *" ")* *#&))* *#&)*
To insert the color of the text above :	w	color : _	*id.*
	black < >	color : black < >	*(horiz* *#&* *(% 4* *(cover* *(write black* *(align* *"flip"* *" "))* *#&))* *#&)*
To go right to the second "cover" brother	→		*(horiz* *#&* *(% 4* *(cover* *(write black* *(align* *"flip"* *" "))* *#&))* *#&)*
To insert the following text : *-flip*	(see above)	(see above)	*(horiz* *#&* *(% 4* *(cover* *(write black* *(align* *"flip"* *" "))* *(align* *"flip"* *" ")))* *#&)*
To rotate two times the text above	r		*(horiz* *#&* *(% 4* *(cover* *(write black* *(align* *"flip"* *" "))* *(Rot 1* *(align* *"flip"* *" "))))* *#&)*

	r		(horiz #& (% 4 (cover (write black (align "flip" " ")) (Rot 2 (align "flip" " ")))) #&)
To go up to the immediate father	↑		(horiz #& (% 4 (cover & &)) #&)
To go left to the first "horiz" brother	←		(horiz #& (% 4 (cover & &)) #&)
To insert no text	A	align : _	id.
	<>	align : <>	(horiz (align) (% 4 (cover & &)) #&)
To insert a black frame	f	color : _	id.
	black <>	color : black <>	(horiz (frame black (align)) (% 4 (cover & &)) #&)
To go right to the third "horiz" brother	→		(horiz (frame black (align)) (% 4 (cover & &)) #&)
	→		(horiz (frame black (align)) (% 4 (cover & &)) #&)
To insert no text	(see above)	(see above)	(horiz (frame black (align)) (% 4 (cover & &)) (align))

To insert a black background	p	color : _	*id.*
	black < >	color : black < >	*(horiz (frame black (align)) (% 4 (cover & &)) (paint black (align)))*
To go up to the top of the flip tree	esc <		*(horiz & & &)*
THE PROGRAM IS DESIGNED AND EDITED			

Finally he will obtain the word tree shown in figure 8.

```
(horiz
  (frame black
    (align))
  (% 4
    (cover
      (write black
        (align
           "flip"
           " "))
      (Rot 2
        (align
           "flip"
           " "))))
  (paint black
    (align)))
```

Figure 8. - A Luciflip word tree.

Furthermore, a feature of the editor allows the learner to see the subtree being edited and its context (father and brothers). This is holophrasting. Luciflip maintains on the screen an holophrasted representation of the hierarchy manipulated, i.e., it provides "an overall view of the program by unparsing or displaying text down to a certain level of detail and eliding the rest. The user may then select a smaller area to zoom in on" (Allison, 1983, p. 457).

a) Holophrasting of levels ≥ 2

```
(horiz
  (frame black
    (align))
  (% 4
    (cover & &))
  (paint black
    (align)))
```

b) Holophrasting of levels ≥ 1

```
(horiz & & &)
```

c) Holophrasting of levels ≥ 0

```
&flip
```

Figure 9. - Holophrasting of the Luciflip word tree.

4.1.2. Modular development

In the context of Luciflip editing, the beginner can see the sense of the modular programming technique. He can design modules in different buffers, then call for these modules when editing the definitive program. Later, when holophrasting his word tree, the learner can see his program as a set of modules because nodes are named, i.e., they appeared with the form : "&name".

4.2. A PRELIMINARY STUDY OF THE EFFECTIVENESS OF LUCIFLIP AS A TOOL FOR LEARNING STRUCTURED PROGRAMMING

This study consisted also of an observation of the effectiveness of the Luciflip "syntax editor" for introducing beginners to structured programming method.

4.2.1. Method

Subjects were two of the three subjects of study I and a new one (11 years old).

The Ceyx environment is implemented on a Multics operating system. It can be noted that Multics files are hierarchically organized just as are the Unix files. So our "old" subjects were not disorientated by the Multics context. The "old" students continued their course with the syntax editor, the "new" one was introduced to the course directly with Luciflip. Subjects were taught how to edit with

Luciflip using the family tree terminology (father, brother, and son). They were invited to edit their programs immediatly after having planned it.

4.2.2. Results and discussion

Subjects programmed more effectively with the syntax editor than with the classical text editor. They designed not only more programs, but more complex programs : in the Luciflip session, the "old" novices wrote, on average, 5 Flip programs -among which the most complex was of 4th-order- whereas in the best of the Flip sessions they wrote,always on average, 2 Flip programs -among which the most complex was of 3rd-order; for its first contact with Flip and a computer, the "new" beginner designed 5 Flip programs, one of which was of 5th-order, and embodied all the variety of declarations of delimitation and of insertion.

The reasons for this improvement lie in the fact that syntax editing allows (1) to ignore programming syntactic details and (2) to focuse on semantic features of programming -which are the most important.

(1) Indeed, subjects greatly appreciated that Luciflip suppresses the "clerical" aspect of editing (it reduces the number of touches, it prevents from syntactic mistakes, and so on). It can be noted that novices were not disturbed by the change in the concrete syntax of the language. Another aspect of the syntax editor

which was appreciated, is that it eliminates delay between edition and execution of programs. The fact that execution of a flip immediately follows its edition, does not tax novices' attention. Consequently, subjects found the syntax editor easy-and-satisfactory-to-use.

(2) Because editing of a Flip program with Luciflip involves, we assume. cognitive representations and treatments near to the cognitive representations and treatments involved in planning the Flip program, it illuminates the basic aspects of structured programming : stepwise refinement/top-down development and modular development.

Because of lack of time. modular development was little used in the context of Luciflip : only the two "old" subjects used it, each in one of their programs. Nevertheless, the subjects who develop their program in this way clearly perceived why the editor displayed "&name" when they inserted an already designed Flip program in the program they were going to plan.

Concerning stepwise refinement and top-down development, we have to emphasized that beginners did not think editing in linear terms but in structural terms. They thought editing in terms of planning through their manipulation of Luciflip trees. On this subject, it can be noted that :

- "Old" subjects referred explicitely to the way they "travelled" in the Unix tree to travel in the Luciflip trees.

- At the beginning, all the three novices did not succeed to move in the Luciflip word tree because the orientation to move in it is not always the same as the orientation to move in the graphic tree (e.g., sometimes to go down in the graphic tree means to go right in the word tree). It follows that some confusions occurred. They were avoided later because subjects, when they moved in the word tree, had in mind a representation of the graphic tree.

- Holophrasting was well understood. Beginners saw that the "&" sign represents a subtree which. eventually, contains other subtrees. So that, when they wanted to modify a Flip program, they succeeded to localize and to retrieve the subtree concerned by the modification, even though the program appeared to them in an holophrasted form.

Really, it was reasonable to think that editing of a Flip program involves a cognitive mechanism near to the mechanism involved in planning the Flip program. Besides, so near seem to be these mechanisms that one subject decided and succeeded to edit directly a program, in other words to edit it while designing it.

5. Conclusion

The didactical environment we have proposed here seems to be adapted to novices of age 11-13 years, as confirmed by the two observations performed to see how well those beginners used the environment, i.e., if they managed to apply the basic notions of structured programming :

Novices had a more or less refined mental representation of the tree diagram. They used it and refined it as they programmed with Flip or they edited with Luciflip (and also when they "travelled" in the Unix file system).

Although an essential aspect of top-down development -i.e., nesting of delimitations as opposed to their juxtaposition- was not immediately understood. beginners learned progressively to design Flip programs following the basic notions of structured programming. However, learning was slowed down because of the use of a classical text editor which compelled to deal with the syntax of the programming language. This even discouraged novices from designing rather complex programs.

Thanks to the Luciflip syntax editor, novices could edit their programs in the same way they planned it, i.e., by following the basic notions of structured programming. Syntax editing allows (1) to ignore programming syntactic details and (2) to focuse on semantic features of programming -which are the most important. So

learning was made faster and programming more effective. Flip programs became more numerous and, above all, more complex.

Finally we can conclude that the simultaneous use of Flip and of its syntax editor, within the context of the tree diagram, delimits the basic notions of structured programming. It constitutes a refined environment to introduce novices to these notions. But this conclusion is limited to 11-13 years old secondary school novices. More systematic studies (Sheil, 1981) have to be performed to determine, for instance, individual differences in the adaptability of this introductory environment, or to see if the beginners so introduced to programming will transfer the structured programming technique to any language.

References

ALLISON, L. (1983). Syntax directed program editing. _Software-Practice and Experience_. _13_, 453-465.

BRANSFORD, J.D. (1979). _Human cognition: Learning, understanding and remembering_. Belmont, California: Wadsworth.

CARROLL, J.M. & THOMAS, J.C. (1982). Metaphor and the cognitive representation of computing systems. IEEE Transactions on Systems, Man, and Cybernetics. 12, 107-116.

DIJKSTRA, E.W. (1970). Notes on structured programming. T.H. Report 70-WSK-03, Technological University Eindhoven, Netherlands.

DU BOULAY, J.B.H., O'SHEA, T. & MONK J. (1981). The black box inside the glass box: presenting computing concepts to novices. International Journal of Man-Machine Studies. 14, 237-249.

GREEN, T.R.G. (1980). Programming as a cognitive activity. In H.T. Smith & T.R.G. Green (Eds), Human interaction with computers. London, Academic Press, pp. 271-320.

HABRIAS, H., LEVASSEUR, P. & LISCOUET, M. (1982). Représentations graphiques, aides ou obstacles à la conception de programmes ? Paper presented to the 18th Congrès de la Société d'Ergonomie de Langue française, Paris, France.

HOC, J.M. & LEPLAT, J. (1983). Evaluation of different modalities of verbalization in a sorting task. International Journal of Man-Machine Studies. 18, 283-306.

HULLOT, J.M. (1982). CEYX : fils de LUCIFER, manuel de référence. On-line manual [under a Multics operating system], I.N.R.I.A., Rocquencourt.

HULLOT, J.M. (1983 a). CEYX : A multiformalism programming environment. In Les éditeurs dirigés par la syntaxe. I.N.R.I.A., Aussois.

HULLOT, J.M. (1983 b). CEYX, a multiformalism programming environment. Paper presented at IFIP 83, 9th World Computer Congress, International Federation for Information Processing, Paris.

INCE, D.C. (1983). A software tool for top-down programming. Sotware-Practice and Experience. 13, 687-695.

JENSEN, R.W. (1981). Structured programming. Tutorial series 6, Computer. 14, 31-48.

KAHN, G. (1981). Flip : Manuel de référence. I.N.R.I.A., Rapport technique No 2.

KNUTH, D.E. (1973). The art of computer programming. Vol. 1/ Fundamental algorithms. Second Edition, Reading : Massachussets, Addison-Wesley Publishing Company.

LEMOS,R.S. (1980). Methods, styles, and attitudes in the programming language classroom. Computer. 13, 58-65.

MAYER, R.E. (1981). The psychology of how novices learn computer programming. Computing Surveys. 13, 121-141.

MELESE, B. (1982). METAL, un langage de spécification pour le système MENTOR. Technique et Science informatique. 1, 275-285.

MEYER, B. & BAUDOIN, C. (1978). Méthodes de programmation. Paris, Eyrolles.

SCHNEIDER, G.M., WEINGART, S.W. & PERLMAN, D.M. (1978). An introduction to programming and problem solving with PASCAL. John Wiley & Sons, New York.

SHEIL, B.A. (1981). The psychological study of programming. Computing Surveys. 13, 101-121.

SHNEIDERMAN, B. & MAYER, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences. 7. 219-239.

TEITELBAUM, T. & REPS, T. (1981'. The Cornell Program Synthetizer: A syntax-directed programming environment. Communications of the ACM. 24, 563-573.

THOMAS, J.C. & CARROLL, J.M. (1981). Human factors in communication. IBM Systems Journal. 20, 237-263.

THOMAS, R. & YATES, J. (1982). A user guide to the UNIX system. Osborne/McGraw-Hill, Berkeley: California.

TRACZ, W.J. (1979). Computer programming and the human thought process. Software-Practice and Experience. 9, 127-137.

WIRTH, N. (1971). Program development by stepwise refinement. Communications of the ACM. 14, 221-227.

WIRTH, N. (1974). On the composition of well-structured programs. Computing Surveys. 6, 247-259.

NOTES

1. The number and the content of these steps differ from one author to another, e.g., (a) for Weinberg (1971, cited by Tracz 1979), these steps are : understanding the problem, planning (designing) the solution in machine independent terms, translating the plan into code; (b) for Schneider _et_ _al_. (1978), they are : defining the problem, outlining the solution, selecting and representing algorithms, coding, debugging, testing and validating, documenting, and program maintenance; and (c) for Wertz (1982) : conceptualization of the activity the computer is supposed to perform, conceptualization of a method by which the intended activity may be performed, expressing this method in terms of a programming language, and communicating the program to the computer.

2. See Melese (1982) for the complete description of the concrete syntax of Flip.

3. In simultaneous verbalization, "the subject is asked to verbalize what he says to himself while performing the task"; in subsequent verbalization "he is asked to verbalize what he said to himself at first without recall aids, afterwards in front of

the record of his anterior behavior" (Hoc & Leplat, 1983, p. 283).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique