PLATO Lessons for a Data Structures Course

by

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana

Technical Report No. 15

PLATO Lessons for a Data Structures Course

Stuart C. Shapiro

August, 1974

This work was supported in part by the National Science Foundation under Grant No. NSF EC-41511.

PLATO LESSONS FOR A DATA STRUCTURES COURSE

Stuart C. Shapiro

1. Introduction

This paper discusses some PLATO [2] lessons that have been written for a course in data structures, the philosophy behind them and ideas for continued development of lesson material for a data structures course. These lessons are included in the Information Structures Area of a large collection of introductory computer science lessons [6].

2. Philosophical Bases

The author has previously stated [7] that one of the best uses of Computer Assisted Instruction (CAI) "is to present visually to the student a process that he has some control over and which he would not otherwise be able to observe." The algorithms used to manipulate complex data structures are just such processes. They are, therefore, appropriate material to be presented by PLATO lessons. Conversely, the PLATO system excels at the presentation of graphic and animated material. It allows for complete program control of the results of a keystroke, including which character, if any, appears on the screen. These features make the PLATO system particularly well suited to the implementation of student controlled and visually presented processes.

The lessons discussed in this paper were designed to be visual simulations of student controlled processes. Within this framework, various approaches were tried. As a result, I now feel that a good CAI lesson should contain the following:*

1. A small amount of textual material to explain the lesson, the diagrams used in it and the language the student will use.

2. An environment the student can manipulate by the use of some language. The environment is displayed on the screen. When the student types in a sentence of the language, the environment changes appropriately. Minimal time is spent in correcting or recovering from ungrammatical sentences. Grammatical but inappropriate sentences result in reasonable but unintended (by the student) changes in the environment. Only when necessary is an error message provided and the student protected from the consequences of the error. The difficulty of recovery from the error should be proportional to its relevance to the concepts being taught.

3. Three levels of direction should be provided, with the student able to choose the desired level at any time:

3a. Undirected experimentation allows the student to explore the environment on his/her own. Hopefully, (s)he will attain a facility with and discover the limits of the language/environment.

3b. Suggested experiments are listed by the lesson writer to encourage the student to explore all the facilities provided and to experience all the limitations. The student is not monitored to see

---

* Another good use of CAI is to present computer generated drills and evaluate the student's answer, provided that an arbitrary number of drills can be generated depending on the student's interest or progress.

if (s)he carries out these experiments successfully.

3c. <u>Monitored tasks</u> are assigned which the student is to carry out successfully. Help is provided when the student errs. Sometimes each step is monitored, but it is preferable for the student to monitor him/herself by observing the environment, although helpful suggestions should still be made when desired.

4. Two modes should be provided for experimentation, and the student should be required to carry out the monitored tasks in both.

4a. In the <u>immediate mode</u> each statement is executed when the student types it in. The decision as to what should be done next is made by the student while visually inspecting the environment. This is the better mode for the initial learning of how to accomplish the task.

4b. In the <u>delayed mode</u> the student uses the language, enhanced if necessary with test and branch instructions, to write a program which is executed step by step after it is finished. By observing the successive changes in the environment, the student can determine if the program is correct. In this mode, it can be determined if the student has consolidated what (s)he learned while in the immediate mode.

3. Discussion of Lessons

I will now discuss the data structure lessons written so far. None of them contains all the facilities outlined above, although each facility is contained in at least one lesson. High priority will be given to provide each lesson with all the facilities.

3.1 STACKS

3.1.1 Purpose

The purpose of the lesson, STACKS, is to teach the student what the concept of a stack is and what a stack may profitably by used for. The purpose is not to teach the student how to implement a stack on a computer. The lesson is intended to be used when a data structures course based on Knuth [5] would discuss Chapter 2.2.1, one based on Harrison [4], Chapter 2, and one based on Berztiss [1], Chapter 10. For these purposes, the major properties of a stack were considered to be its Last In, First Out (LIFO) character and the fact that only the top element is accessible. The two main operations that can be performed on a stack are pushing an element onto the top of the stack and popping an element off the top. Also, the top element (and only the top element) may be accessed. Figure 1 shows the title page of lesson STACKS and Figure 2 shows the index to the three sections of the lesson. Section 1 contains introductory material, allows the student undirected experimentation with the stack and contains suggested experiments. Section 2 gives monitored tasks dealing with evaluating Polish postfix expressions. Section 3 provides an enhanced environment in which the student may experiment with a more specific domain. These sections will be discussed below.

3.1.2 The STACKS Environment

The STACK environment and language are introduced in Section 1 of the lesson (Figures 3-10). Figures 3-6 show the introductory text. The basic environment contains an animated representation of a stack and three variables. An empty stack is represented with a raised platform (Figures 3-6). A non-empty stack has only the top number visible (Figure 8). The process of pushing and popping is animated. Popping an empty stack causes the platform to pop off as shown in Figure 9. Since any real implementation of a stack (certainly the implementation the lesson itself uses) must have finite size, the student's stack has a finite, though changing, size which is displayed prominently on the stack itself (see Figure 4). If the student attempts to push a number onto a full stack, the number just slides off as shown in Figure 10.

3.1.3 The STACKS Language

The STACKS language is presented to the student as shown in Figures 5 and 6. From then on, a summary of the syntax is displayed in the corner of the screen whenever it is legal for the student to type in an instruction. This is done because it is not the purpose of this lesson to teach the student this (or any) particular language. For the same reason, the student is protected from even typing a syntactically incorrect instruction. This is a basic feature of the STACKS parser and depends on the PLATO feature that allows a lesson program to intercept a student key press before any character is displayed on the screen. More specifically, the parser is a finite state automaton in which each state waits for a valid character to be pressed by the student and only then displays and processes the character. As shown in Figure 6, the STACKS language contains only a push statement and a pop statement, begun by the characters ↓ and ↑ , respectively. The pop arrow is followed by the name of the variable into which the top number in the stack is to be popped. The student can push onto the stack either the contents of any of the three variables or an integer constant. He can enter a positive or a negative integer or zero, but due to the physical size of the displayed representation of the stack, he is limited to a five-digit positive integer and to a four-digit negative integer. Since pushing an integer constant is the only variable length instruction, it is the only one for which the NEXT key is required to indicate the end of the statement. The ERASE key is specifically recognized by the parser which erases the last displayed character and returns to the appropriate previous state.

Section 1 ends by allowing the student unlimited time to experiment with the stack. The list of suggested experiments shown in Figure 7 is available via the HELP key. Figures 8-10 show some typical experimentation.

3.1.4 Evaluating Polish Postfix Expressions

Section 2 of STACKS contains a monitored task to use the stack to evaluate arithmetic expressions written in Polish postfix notation. The purpose of this task is to demonstrate to the student a use of the LIFO regime. Figure 11 shows the introduction to this section. Figures 12 and 13 show the introduction of a slightly enhanced environment including an expression, a cursor that shows progress in evaluating the expression and an area in which the instructions are to be written. The student can then choose to see the standard demonstration expression or randomly generated expressions.

Some stages in the evaluation of the standard expression are shown in Figures 14-16. This expression, the stack instructions used in evaluating it and the commentary made while the evaluation proceeds are all programmed explicitly into the lesson. The student controls the pace of the demonstration by pressing the NEXT key for each step. In this way (s)he has full opportunity to observe and comprehend the process. When the demonstration is over (Figure 17), the student can choose to do undirected experimentation, to return to the index or to continue on to the monitored task.

In the second subsection of the expression evaluation section, postfix expressions are generated by the lesson program. A flowchart of the generation algorithm is shown in Figure 18. The elements of the expression are stored in the array <u>expr</u>; <u>fexpr</u> is an index into that array; <u>rexpr</u> keeps track of the depth of the stack when the part of the expression thus far generated will have been evaluated. The generation algorithm is based on the following points:

1) For simplicity, the elements of the expression are single characters -- one digit positive integers and the operators + , - , / , and * .

2) The expression is to contain at least one operation.

3) The expression is to be a maximum of 11 characters long.

4) The expression cannot be complete unless rexpr = 1.

5) If rexpr = 1 and the expression is less than 11 characters long, we stop with probability 1/3.

6) Just before adding the fexpr+1<u>st</u> element, fexpr + rexpr - 2 is the smallest number of characters a completed expression can

have, and this minimum will be attained only if the rest of the characters are operators. If an operand is added, the minimal length will increase by 2. Therefore, once the minimal length exceeds 9, only operators may be added.

7) If rexpr = 1, the next element must be an operand.

8) If there is no specific reason to add either an operator or an operand, decide randomly with equal probability.

The constructed expression is both stored and displayed on the screen. The student can evaluate the expression him/herself, or have it evaluated automatically as (s)he repeatedly presses the NEXT key (Figure 19). The lesson contains an evaluator capable of evaluating any expression the generator is capable of generating. The evaluator also displays the instructions it is using, as did the demonstrator.

If the student evaluates the expression (Figures 20-23), (s)he is monitored by two basic monitoring routines. One monitors what (s)he does with operators, the other monitors what (s)he does with operands. Both compare the actual state of the environment after each stack instruction with the correct state. If the actual state is correct, the cursor advances and the task continues. If the state is incorrect, the monitor undoes the last operation, crosses it off and writes an appropriate message. This rigorous monitoring was chosen because it was felt that the results of a mistake would not be obviously wrong to the student.

3.1.5 Traversing a Binary Tree

Section 3 of STACKS introduces a binary tree and asks the student to traverse it. This material is more advanced that the rest of STACKS, corresponding to Section 2.3.1 of Knuth [5] and Chapter 12 of Harrison [4], but is included because it also demonstrates the LIFO character of a stack. The enhanced environment for this section is introduced as shown in Figure 24. It includes a binary tree, null pointers, one pointer variable and two new expressions for the STACKS language.

The student is presented with the environment as shown in Figure 25. There is one suggested task of trying to "visit" each node of the tree. The student is also given two additional instructions. Pressing "v" causes the node pointed to by _p_ to be checked as having been visited (see Figure 26). The student can press "t" to start all over with an empty stack and _p_ pointing to the root. As can be seen from Figures 25-28, the only node whose contents is visible is the one currently being pointed to. The purpose of this and other features of this environment is to force the student to use the stack as his/her only memory. The student has full freedom to enter any instruction of the enhanced STACKS language at any time. Popping a number into the variable _p_ causes _p_ to be moved appropriately. If the number of the address of a node of the tree, _p_ is changed to point to that node (Figure 26). If zero is popped into _p_, the result is as shown in Figure 27. If any other number is popped into _p_, the result is as shown in Figure 28.

Although internally, the nodes are addressed consecutively 1-13, these addresses are disguised for the student to prevent his/her learning them. The formula for the student address is

$$sa = \begin{cases} 0 & \text{if } ta = 0 \\ a*ta+c & \text{if } ta \neq 0 \end{cases}$$

where ta is the true address, a is a random number between 1 and 60, and c is a random number between 100 and 200. Whenever the student starts over by pressing "t," a and c are chosen anew.

No additional error messages are provided, as it is obvious when a node becomes unreachable, and the student can then start over at the root.

3.2 LISTER*

3.2.1 Purpose

The purpose of the LISTER lesson is to introduce linked list processing and the pointer concept. This is done by representing pointers both as addresses within an array and as arrows. This lesson presentation corresponds to Sections 2.1 and 2.2.3 of Knuth [5] and Section 10d of Berztiss [1]. The lesson allows undirected experimentation in the environment using a language designed to be executed in the delayed mode. The title page is shown in Figure 29.

3.2.2 The LISTER Environment

The LISTER environment is introduced to the student as shown in Figure 30. It consists of six variables and 25 words of list space. Each word of list space contains an information field and a next field. The contents of the variable, the information fields

* The initial design and programming of this lesson was done by Stephen F. Ziegler.

and the next fields are shown as numbers. An arrow is drawn from each next field to the word whose address is contained in that field (see Figure 34).

3.2.3 The LISTER Language

The LISTER language is defined in Figures 31 and 32. It is a very simple language, but rich enough to write interesting programs for the LISTER environment. The parser is designed the same way as the STACKS parser described above, so it is impossible for the student to enter a syntactically illegal statement. Since the language is designed for delayed mode execution, each statement is stored in the student's lesson memory and can be accessed by the processor via an array indexed by the statement step numbers. Since the environment is not cleared between programs, the student can get the effect of immediate execution by writing a series of one-line programs.

A typical initialization program is shown in Figure 33 just after entry of the stop statement. An arrow points to the next statement to be executed, which is done when the student presses the NEXT key. The student can observe the effect of each statement as it is executed and thereby learn how the environment works and also discover any mistakes (s)he has made. The results of executing the program in Figure 33 are shown in Figure 34.

Generality in drawing the arrows is attained while preserving readability by establishing a series of "lanes" to the right of the list space and storing which lanes are in use and where. This information is used to pick a clear lane for each new arrow.

3.3 NODER

3.3.1 Purpose

The NODER lesson allows students to gain experience manipulating singly-linked lists using the representation most common in texts and class lectures. Nodes are shown as rectangles divided into two fields, only one of which (the Next field) is currently used in NODER. A pointer is shown as an arrow, pointing from the Next field of one node to another node. The student can build any data structure that it is possible to construct with single-linked list structures. Rather than attempting to solve the general problem of how to draw any such list structure for best readability, the student is provided with the ability to move any node anywhere on the screen most convenient for himself. The title page of NODER is shown in Figure 35.

3.3.2 The NODER Environment

The NODER environment consists of 31 nodes and three pointer variables. Nodes in available space are not displayed, neither are pointer variables whose values are the null pointer. Nodes in use are displayed on the screen and each pointer variable is written above the node it points to. The ground symbol or an arrow pointing to another node emanates from the Next field of each node. If the Next field of a node points to a node in available space, the arrow points off the screen. Moving nodes around can cause other nodes to be partially or completely erased. To recover from this condition the student can ask for the entire display to be redrawn. Although the student is responsible for returning unneeded nodes to available space, the situation may arise where there are unreachable nodes on the screen. To recover from this situation, the student may call on an animated garbage collector.

The internal address of each node is an integer between 1 and 31. The null pointer is equal to 0. To manage the NODER environment the following data are stored for the ith node, $1 \leq i \leq 31$:

locx(i) and locy(i) are the x,y coordinates of the position of node i on the screen.

nxt(i) is the address of the node pointed to from the Next field of node i.

nvbl(i) is a 3 bit number indicating which of the three pointer variables point to node i.

prev(i,j),$1 \leq j \leq 31$, is one bit which is on if node j points to node i.

availnd(i) is one bit which is on if node i is in available space.

3.3.3 The NODER Language

The NODER language, executed in the immediate mode, is described in Figures 36 and 37. A summary of the language is visible while the student is in the environment and (s)he is able to review the detailed description at any time by pressing the HELP key.

Although the parser could have been implemented in the same way as the STACKS and LISTER parsers, the TUTOR arrow and judging commands were used to compare the techniques. For this purpose the three "words," "p," "q," and "r" were defined to be synonymous, as were the words consisting of from one to eight "n"s. The following are, therefore, the "understandable" sentences:

1. replot
2. sweep
3. move p
4. p <= avail
5. avail <= p
6. p <= p
7. p <= λ
8. p <= np
9. np <= p
10. np <= λ
11. np <= np

My current thoughts on the results of this comparison are that for small languages such as those used in STACKS, LISTER and NODER, the ease with which the parser can be implemented is about the same using each technique. Using the parsing automaton, the student can be prevented from entering a syntactically meaningless statement. Incorrect keypresses are ignored. Using the TUTOR judging commands, meaningless commands can be entered and the quality of the error message depends on whether or not the particular error was forseen by the lesson writer. It would seem that there is more danger of being distracted by irrelevant language errors when using the TUTOR judging commands than when using a specially designed parsing automaton. This conclusion might not generalize to lessons that do not deal mainly with manipulation of an environment.

Figure 39 shows the NODER environment just after the student's first instruction has been executed. The student is immediately placed in the move mode since that is almost invariably what (s)he would want to do next. In Figure 40, a small list structure has been built. In Figure 41, a circular list has been added, but there are now two garbage nodes on the screen.

Figure 41 shows the garbage collector in progress. The garbage collection algorithm is shown in Figure 38. The student can see this by pressing the HELP key while on the second page of the language description. Although it is not the purpose of this lesson to teach garbage collection, the student is allowed to observe it. Nodes on the screen that are reachable from pointer variables are checked in the proper order at the rate of approximately one every 0.5 seconds. Then the "sweeper" (thanks to the art work of W.J. Hansen) sweeps up the garbage nodes in the order of their internal addresses. Finally the screen is replotted.

4. Future Development

Additional work on the three lessons described above, STACKS, LISTER and NODER, would be beneficial. Experience with them has resulted in definite ideas about how the PLATO system can be used to construct interactive environments to teach data structures. These ideas are crystallized in the points made in Section 2 above. Additional data structures lessons will be written based on these ideas. Needless to say, extensive student experience on these lessons is needed before any final evaluation can be made.

Below, I briefly discuss some planned modifications to existing lessons and some planned additional lessons.

4.1 STACKS

STACKS is useable as it is. The most beneficial additions would be the ability to execute programs in delayed mode. Additional tasks might also be useful. The Polish postfix task needs some introductory material on the Polish postfix notation. This could develop into an independent lesson covering prefix and infix notations as well. The tree section will be the basis of an independent lesson on binary trees (see below).

4.2 LISTER

LISTER needs some work before it is useable, as it is still possible for the student to cause execution errors in the lesson. An editor for the LISTER language is also needed. In addition, there should be some suggested and some monitored tasks.

4.3 NODER

NODER is useable as it is. Some suggested and some monitored tasks should be added. Also, the ability to execute programs in delayed mode should be added.

4.4 Sequential Allocation

The environment would be a linear array on which the student could carry out the operations for manipulating deques, queues and stacks as discussed in Section 2.2.2 of Knuth [5].

4.5 Strings

A lesson on strings, written by Silas Warner for the Indiana University Computer Science Department is a good start on a lesson teaching how character strings can be represented and manipulated.

4.6 Structures with Two Links per Node

The techniques of NODER can be extended to an environment with two links per node. The most useful approach would be to implement the SLIP or LISP structures.

4.7 Sparse Matrices

The use of orthogonal list to represent sparse matrices is particularly difficult to explain in a static medium. A PLATO lesson demonstrating this technique would be useful.

4.8 Polynomials

A lesson on using lists to represent polynomials can also be written along the lines of NODER. The data structures of a language like SAC-1 [3] should be implemented.

4.9 Trees

The tree section of STACKS can be developed into an independent lesson on trees. Monitored tasks involving various traversal algorithms and tree building can be included. Generality can be attained by using the techniques of NODER on nodes with two link fields and one information field.

4.10 Graphs

The techniques of NODER can easily be generalized to arbitrary graphs, showing vertices as circles connected by directed or undirected edges. Graph theory, graph manipulation algorithms and applications of graphs can be taught in this way.

4.11 Garbage Collection

The environment would be a list space containing randomly generated lists and garbage. The student would have to find and collect the garbage using only the storage provided.

4.12 Dynamic Storage Allocation

The environment would be a large array of words, blank for unused, colored for in use. Several allocation schemes would be pre-programmed and the student would be allowed to write his own. The student could also provide a request sequence or use a lesson generated one.

5. Conclusion

Several lessons have been written on the PLATO system to teach data structures by providing the student with an interactive environment. Techniques and approaches developed while writing these lessons are readily applicable to other lessons on data structures and hopefully to other material as well.

References

1. Berztiss, A.T. Data Structures Theory and Practice, Academic Press, New York, 1971.

2. Bitzer, D.L.; Sherwood, B.A.; and Tenczar, P. Computer-based science education, CERL Report X-37, University of Illinois, 1972.

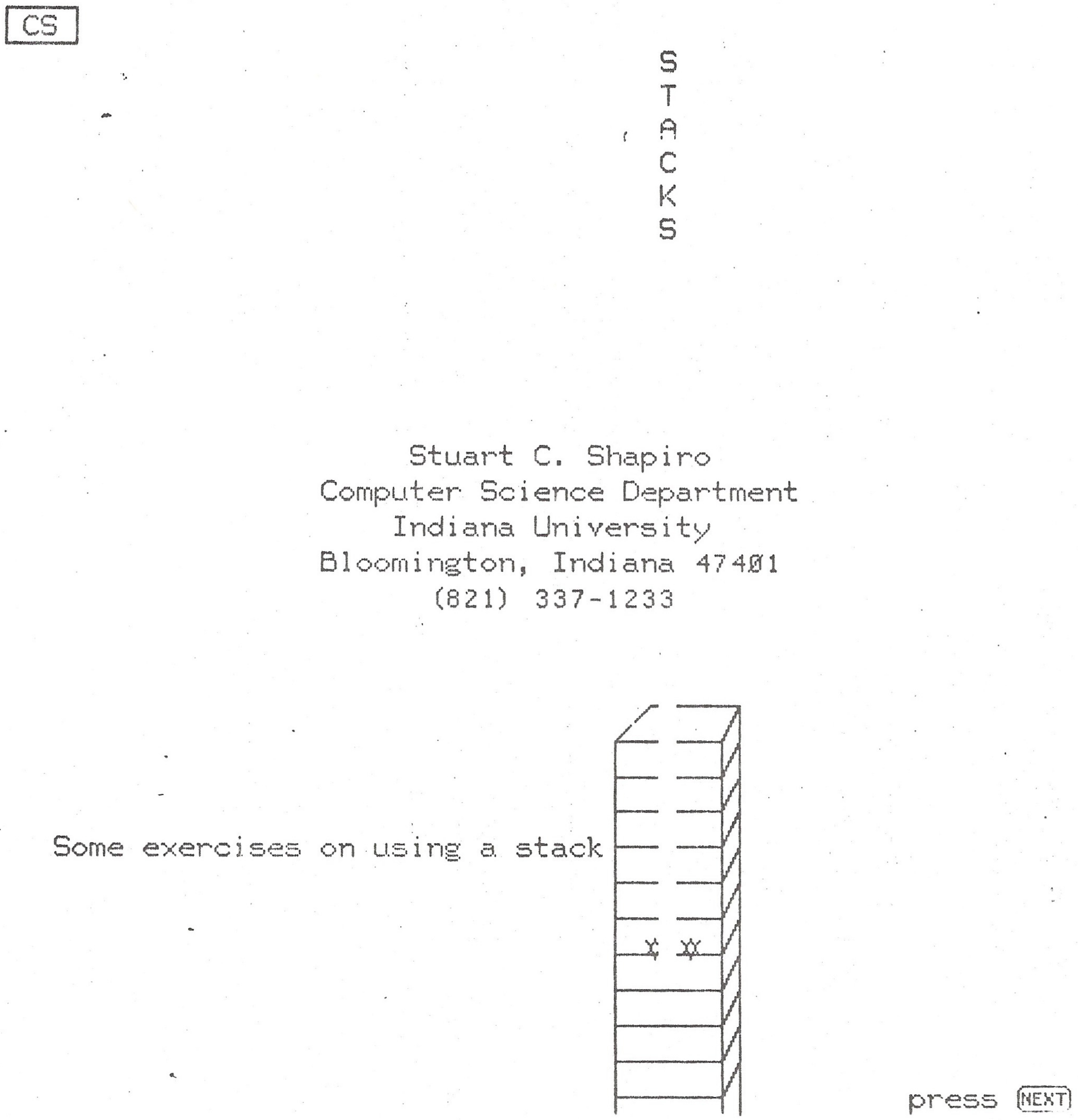
3. Collins, G.E. Computer algebra of polynomials and rational functions. Am. Math. M. 80, 7 (August-September, 1973), 725-755.

4. Harrison, Malcolm C. Data Structures and Programming, Scott, Foresman and Co., Glenview, Illinois, 1973.

5. Knuth, Donald E. The Art of Computer Programming Vol. 1/Fundamental Algorithms, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

6. Nievergelt, Jurg; Reingold, Edward M.; and Wilcox, Thomas R. The automation of introductory computer science courses. International Computing Symposium 1973, Davos, Switzerland, September, 1973.

7. Shapiro, Stuart C., and Witmer, Douglas P. Interactive visual simulators for beginning programming students, Fourth Symposium on Computer Science Education, SIGCSE Bulletin 6, 1 (February, 1974), 11-14.



CS

S
T
A
C
K
S

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana 47401
(821) 337-1233

Some exercises on using a stack

press NEXT

Figure 1

Press the number of the section you want.

When SHIFT NEXT is active, it can be used to
skip introductory text.

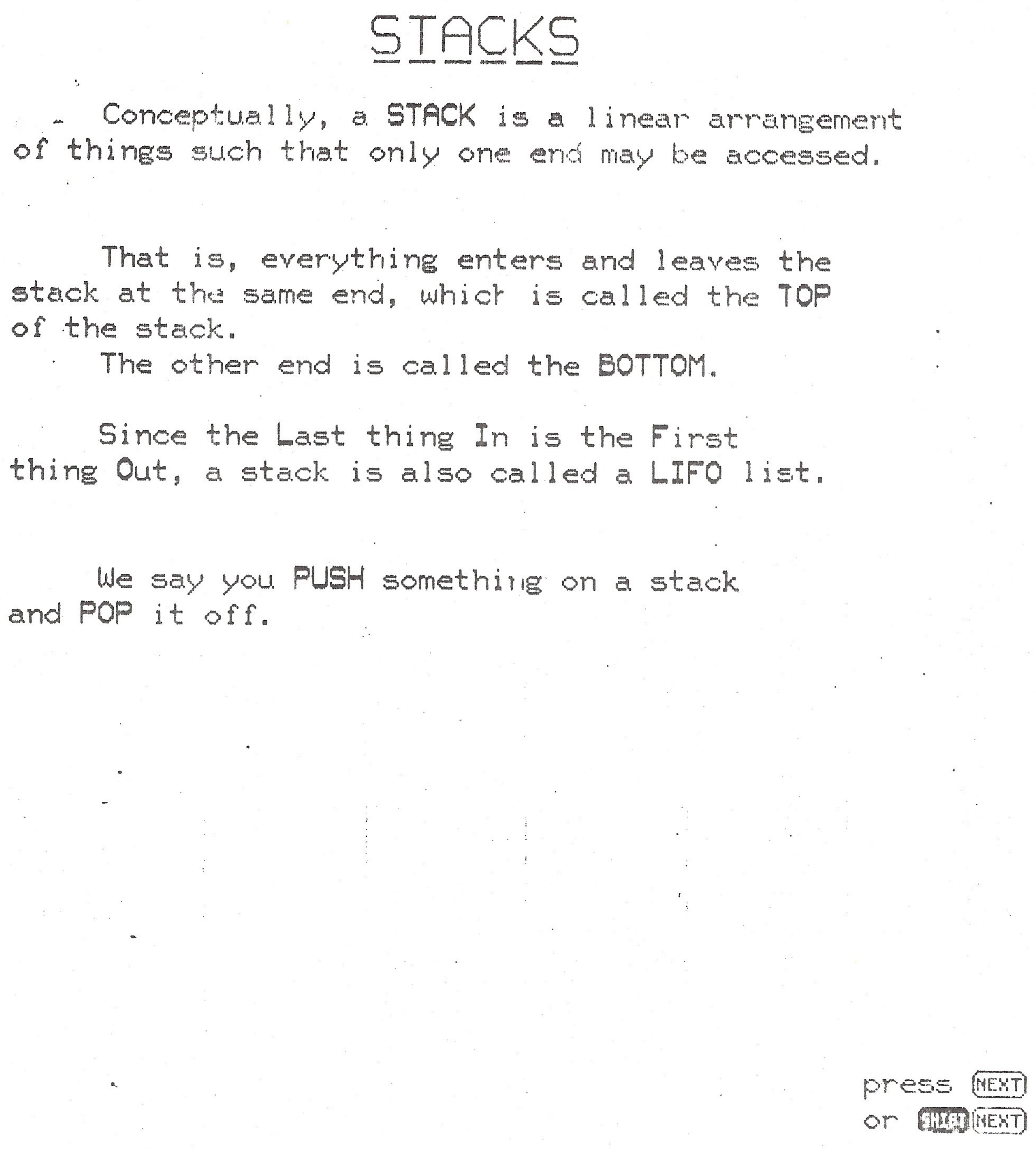
When you are finished with the lesson,
press SHIFT STOP.

1) Introduction to stacks.

2) Use of a stack to evaluate arithmetic
expressions in Polish postfix notation.

3) Traversing a binary tree using a stack.

Figure 2



STACKS

Conceptually, a **STACK** is a linear arrangement of things such that only one end may be accessed.

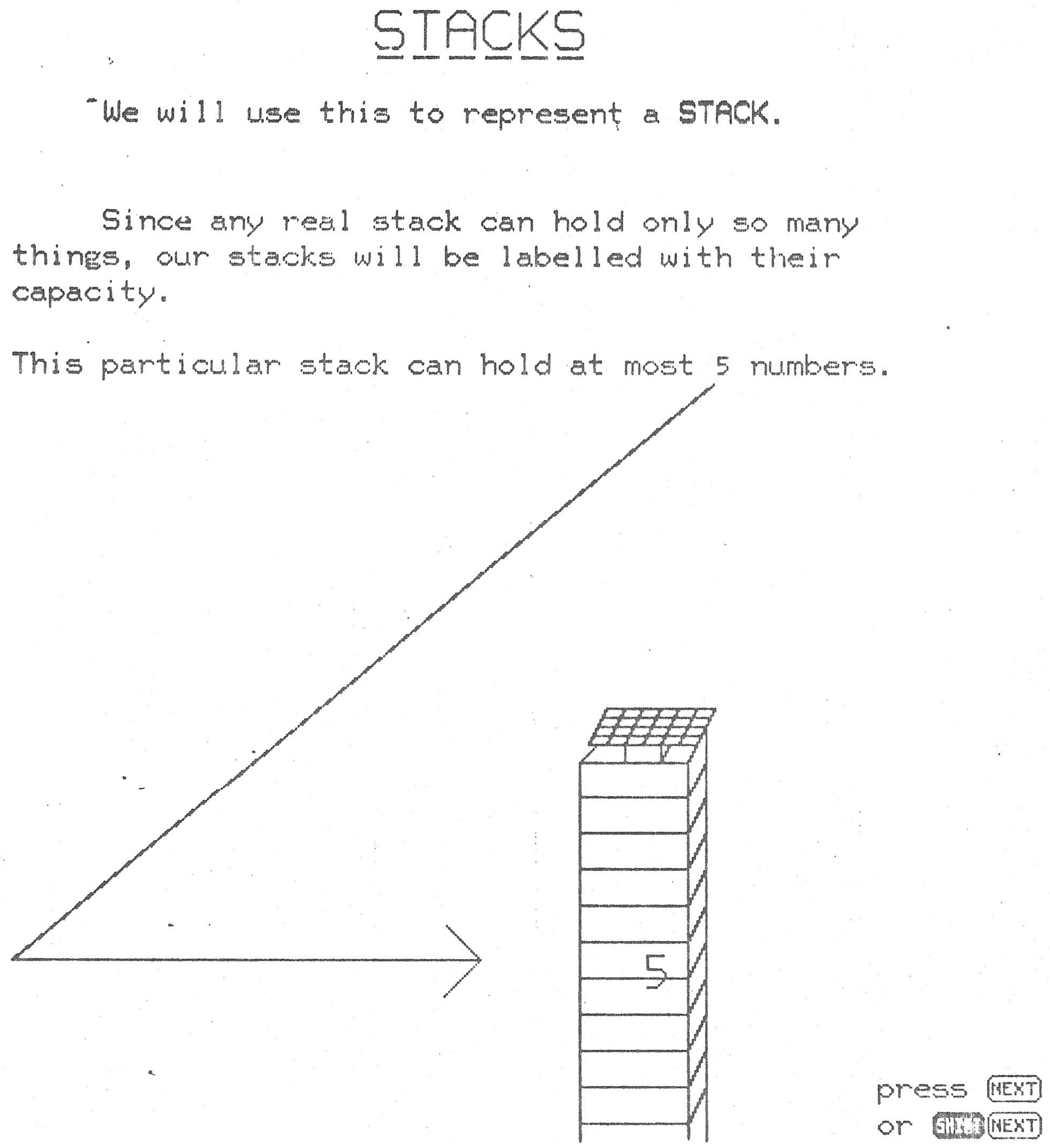
That is, everything enters and leaves the stack at the same end, which is called the **TOP** of the stack.
The other end is called the **BOTTOM**.

Since the Last thing In is the First thing Out, a stack is also called a **LIFO** list.

We say you **PUSH** something on a stack and **POP** it off.

press NEXT
or SHIFT NEXT

Figure 3



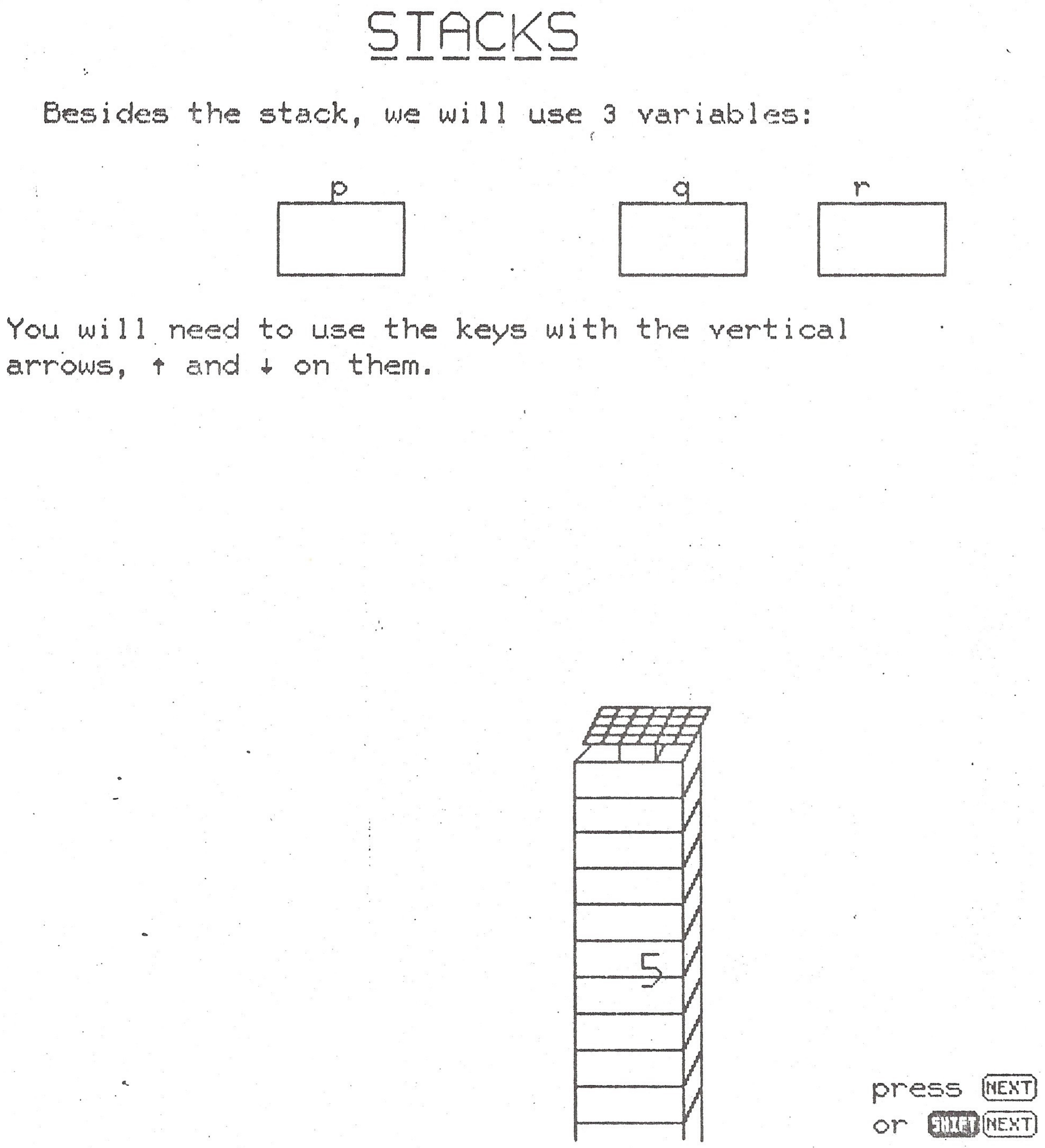
STACKS

We will use this to represent a STACK.

Since any real stack can hold only so many things, our stacks will be labelled with their capacity.

This particular stack can hold at most 5 numbers.

Figure 4

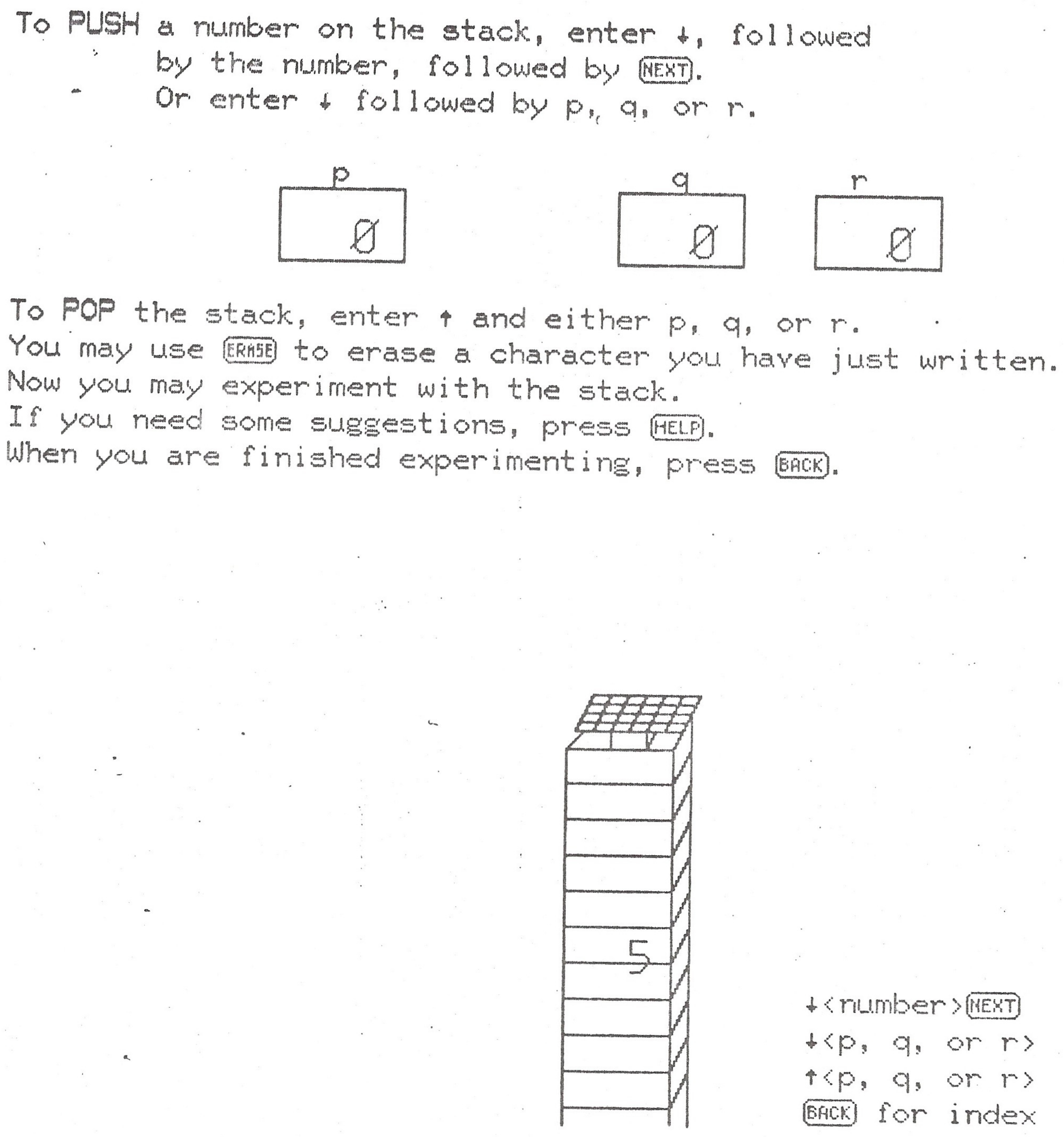


STACKS

Besides the stack, we will use 3 variables:

You will need to use the keys with the vertical arrows, ↑ and ↓ on them.

Figure 5



To PUSH a number on the stack, enter ↓, followed
by the number, followed by $\boxed{\text{NEXT}}$.
Or enter ↓ followed by p, q, or r.

To POP the stack, enter ↑ and either p, q, or r.
You may use $\boxed{\text{ERASE}}$ to erase a character you have just written.
Now you may experiment with the stack.
If you need some suggestions, press $\boxed{\text{HELP}}$.
When you are finished experimenting, press $\boxed{\text{BACK}}$.

↓<number>$\boxed{\text{NEXT}}$
↓<p, q, or r>
↑<p, q, or r>
$\boxed{\text{BACK}}$ for index

Figure 6

Some Suggestions for Experimeting with the Stack

1) How many numbers can you push onto the stack?

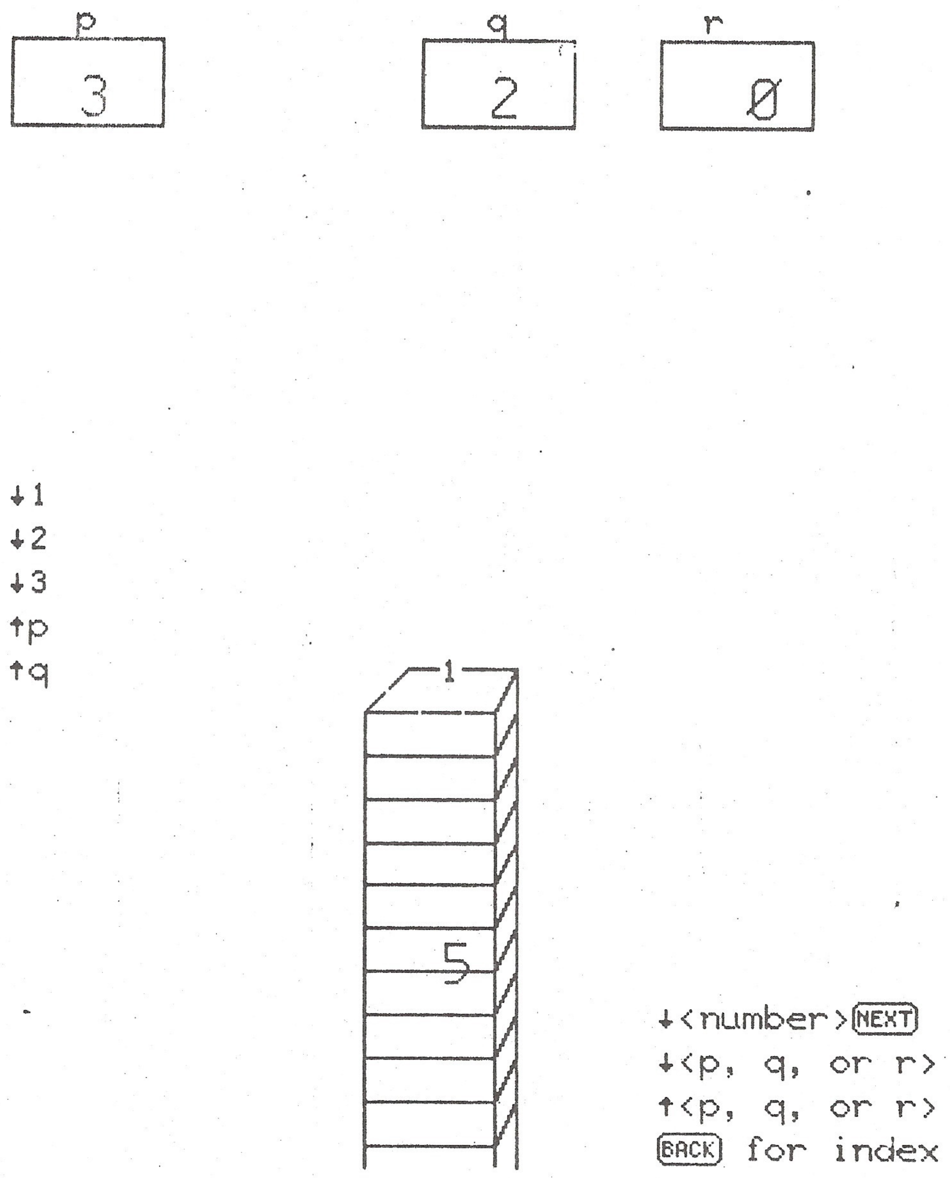
2) What happens if you try to push more than that on?

3) What happens if you pop an empty stack?

4) Can you use p, q and r to store numbers?

5) Be sure to try the $\boxed{\text{ERASE}}$ key.

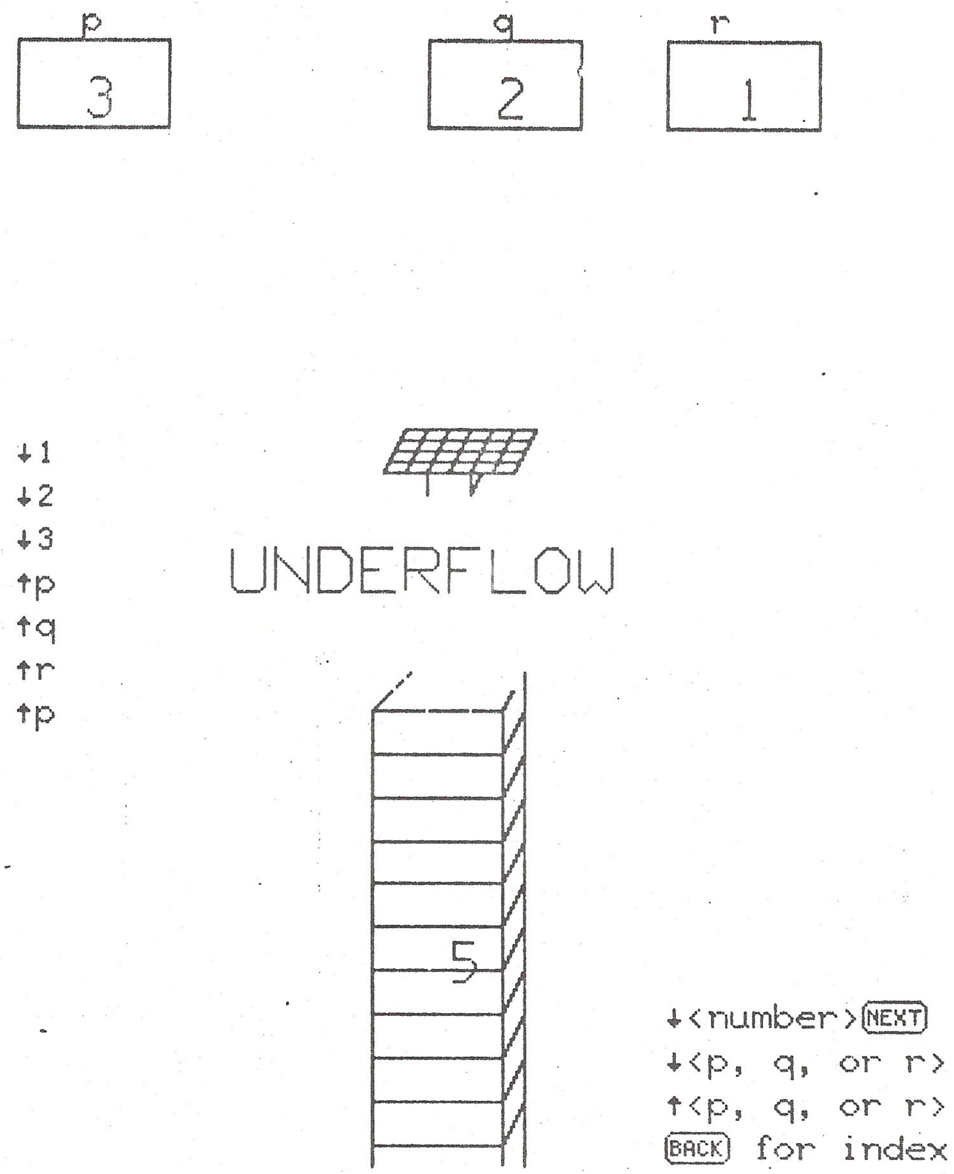
Figure 7



```
↓1
↓2
↓3
↑p
↑q
```

```
↓<number>NEXT
↓<p, q, or r>
↑<p, q, or r>
BACK for index
```

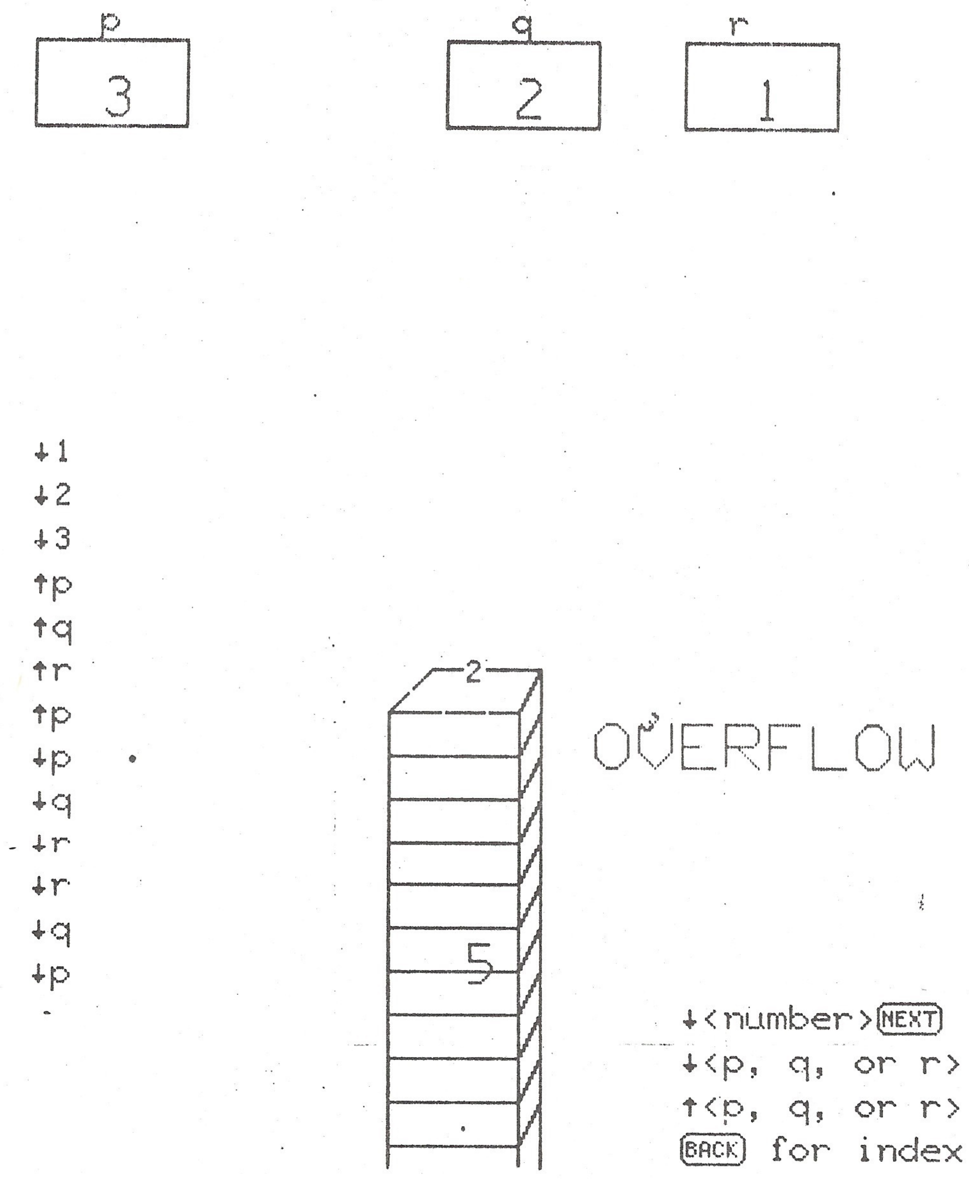
Figure 8



```
↓1
↓2
↓3
↑p
↑q
↑r
↑p
```

UNDERFLOW

Figure 9

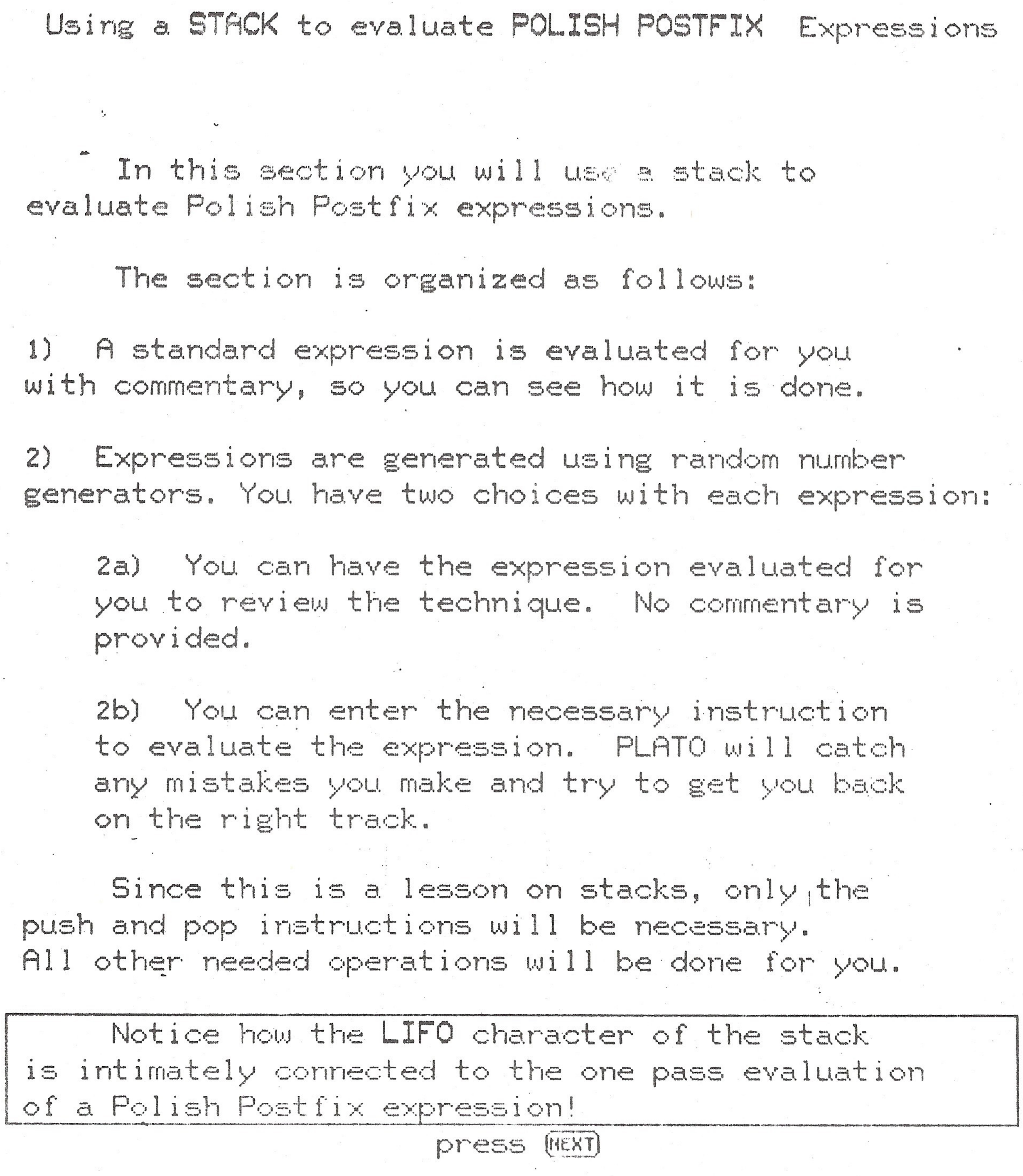


```
↓1
↓2
↓3
↑p
↑q
↑r
↑p
↓p
↓q
↓r
↓r
↓q
↓p
```

OVERFLOW

```
↓<number>NEXT
↓<p, q, or r>
↑<p, q, or r>
BACK for index
```

Figure 10



Using a **STACK** to evaluate **POLISH POSTFIX** Expressions

In this section you will use a stack to evaluate Polish Postfix expressions.

The section is organized as follows:

1) A standard expression is evaluated for you with commentary, so you can see how it is done.

2) Expressions are generated using random number generators. You have two choices with each expression:

2a) You can have the expression evaluated for you to review the technique. No commentary is provided.

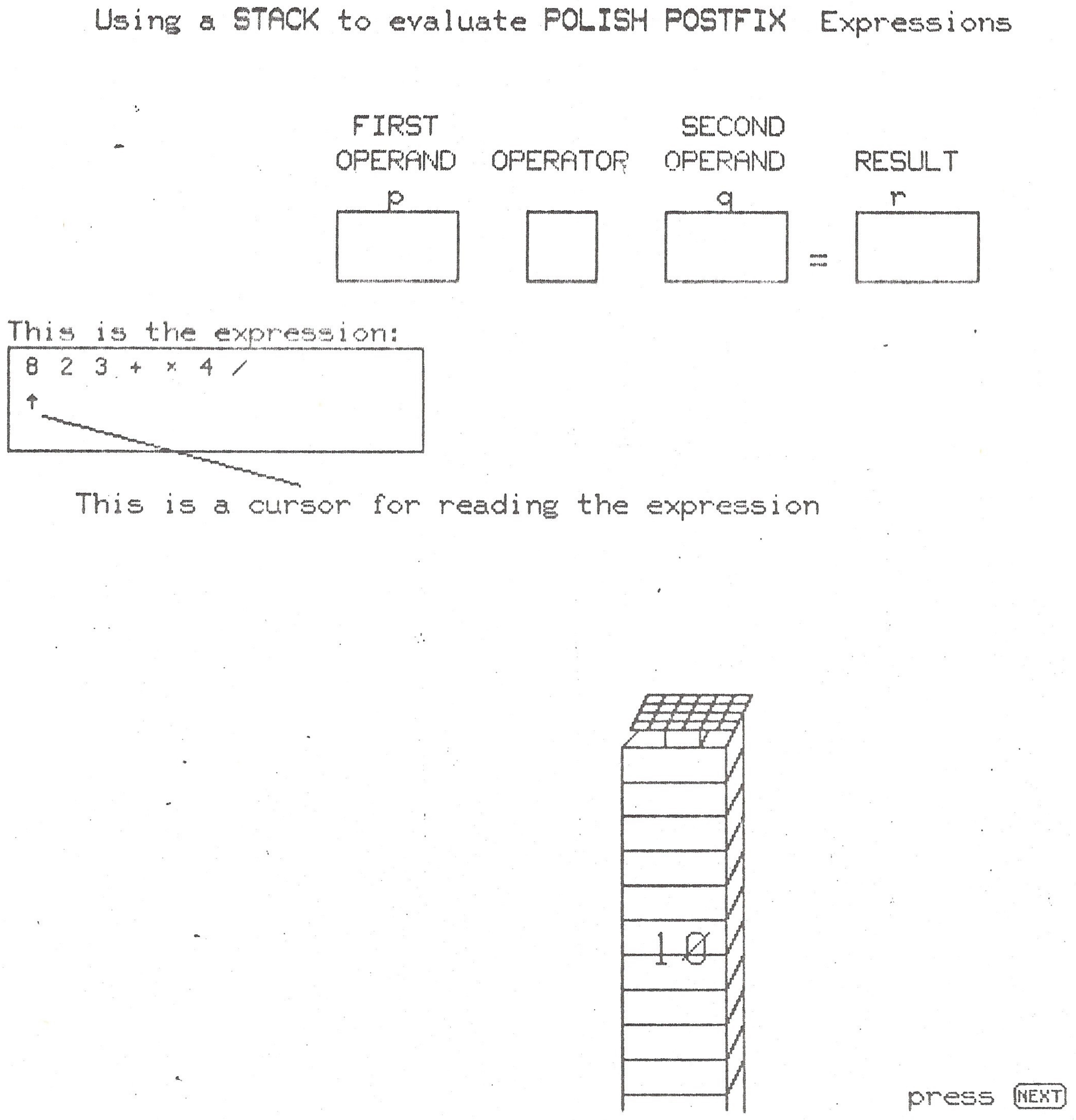
2b) You can enter the necessary instruction to evaluate the expression. PLATO will catch any mistakes you make and try to get you back on the right track.

Since this is a lesson on stacks, only the push and pop instructions will be necessary. All other needed operations will be done for you.

Notice how the **LIFO** character of the stack is intimately connected to the one pass evaluation of a Polish Postfix expression!

press NEXT

Figure 11



Using a STACK to evaluate POLISH POSTFIX Expressions

FIRST OPERAND p | OPERATOR | SECOND OPERAND q | = | RESULT r

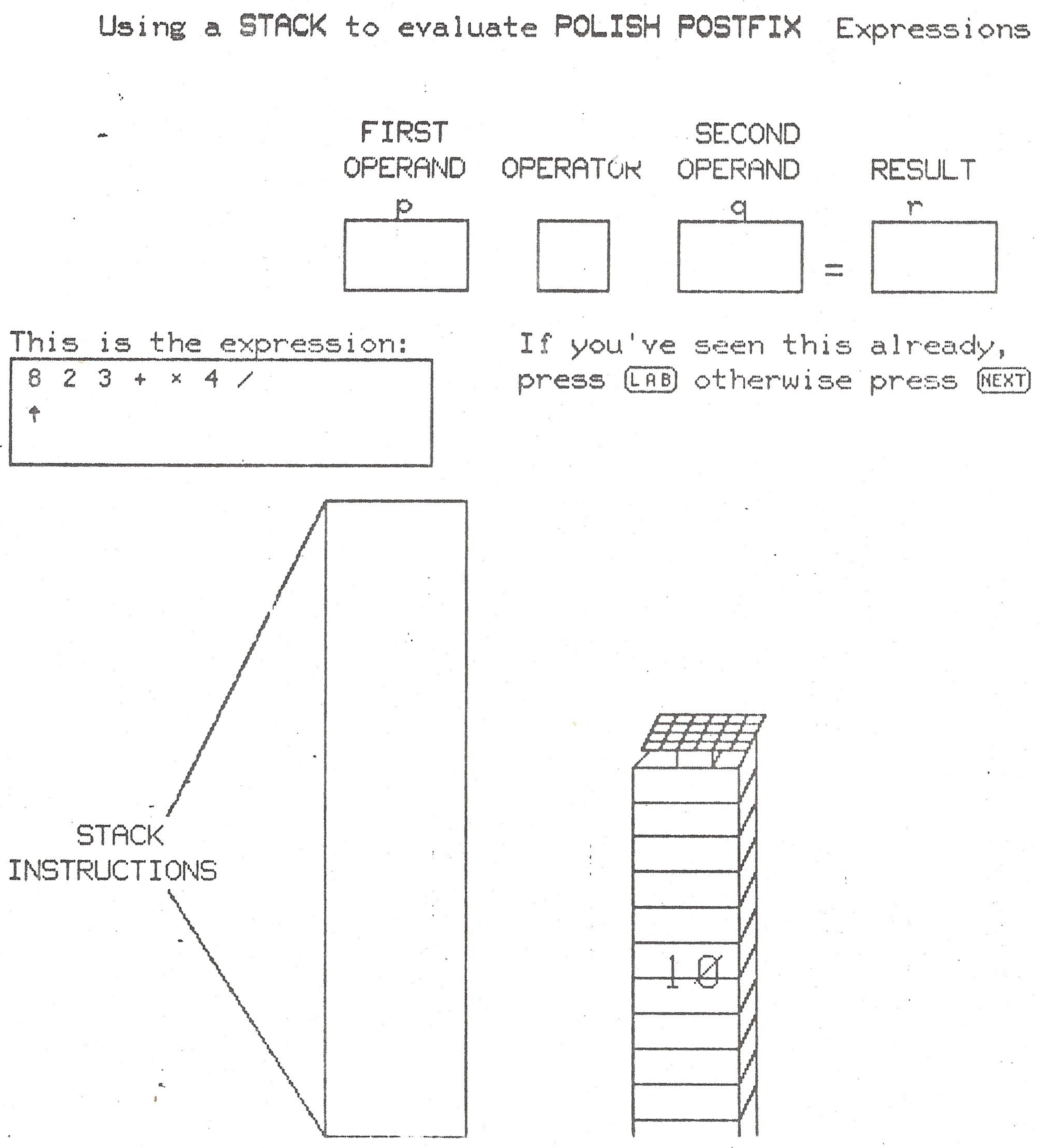
This is the expression:

```
8 2 3 + × 4 /
↑
```

This is a cursor for reading the expression

press NEXT

Figure 12



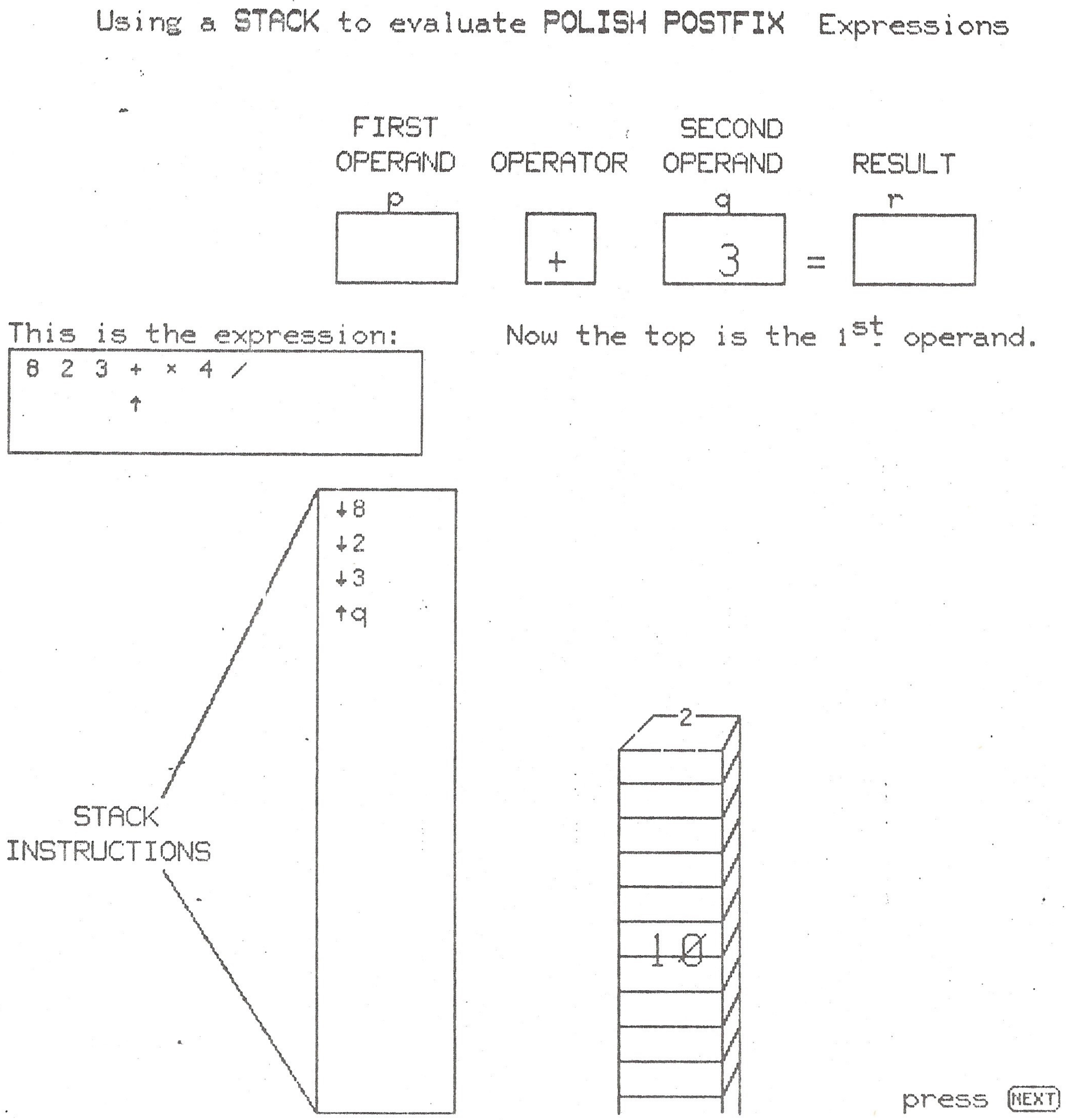
Using a STACK to evaluate POLISH POSTFIX Expressions

This is the expression:

```
8 2 3 + × 4 /
↑
```

If you've seen this already, press LAB otherwise press NEXT

Figure 13



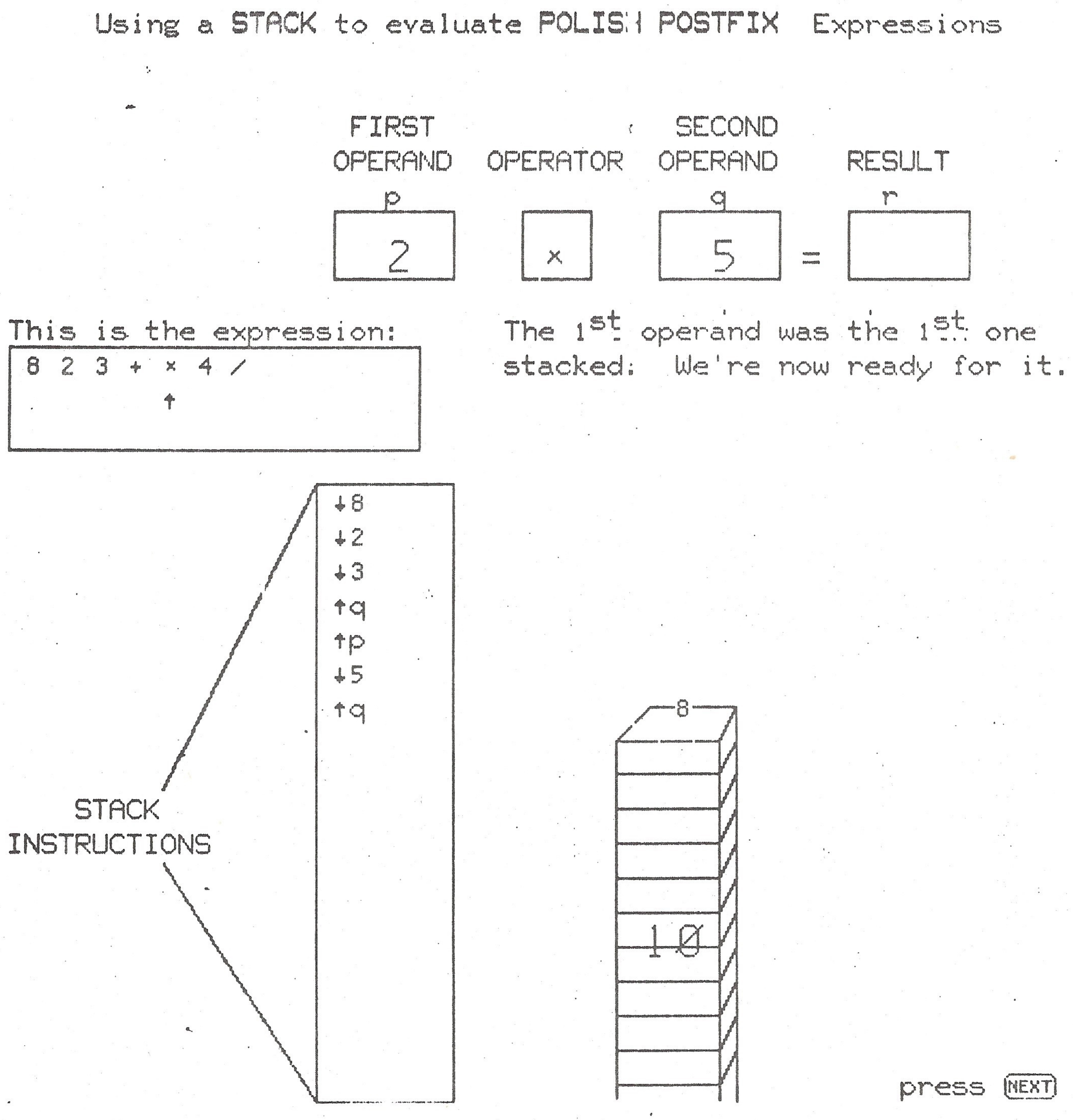
Using a STACK to evaluate POLISH POSTFIX Expressions

This is the expression:

```
8 2 3 + × 4 /
      ↑
```

Now the top is the 1st operand.

Figure 14



Using a STACK to evaluate POLISH POSTFIX Expressions

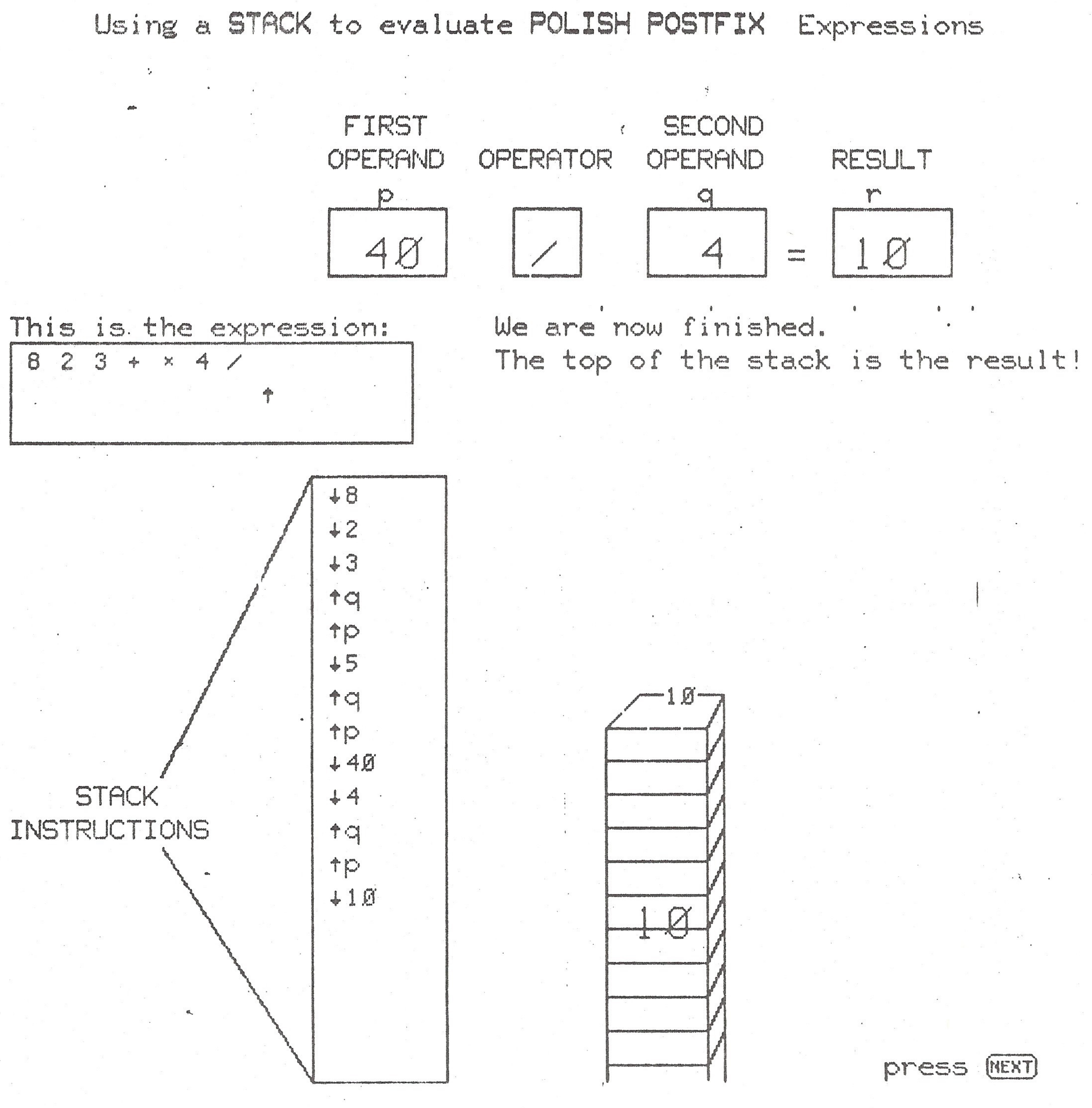
This is the expression:

```
8 2 3 + × 4 /
        ↑
```

The 1st operand was the 1st one stacked: We're now ready for it.

```
↓8
↓2
↓3
↑q
↑p
↓5
↑q
```

Figure 15



Using a STACK to evaluate POLISH POSTFIX Expressions

FIRST OPERAND p	OPERATOR	SECOND OPERAND q		RESULT r
4Ø	/	4	=	1Ø

This is the expression:

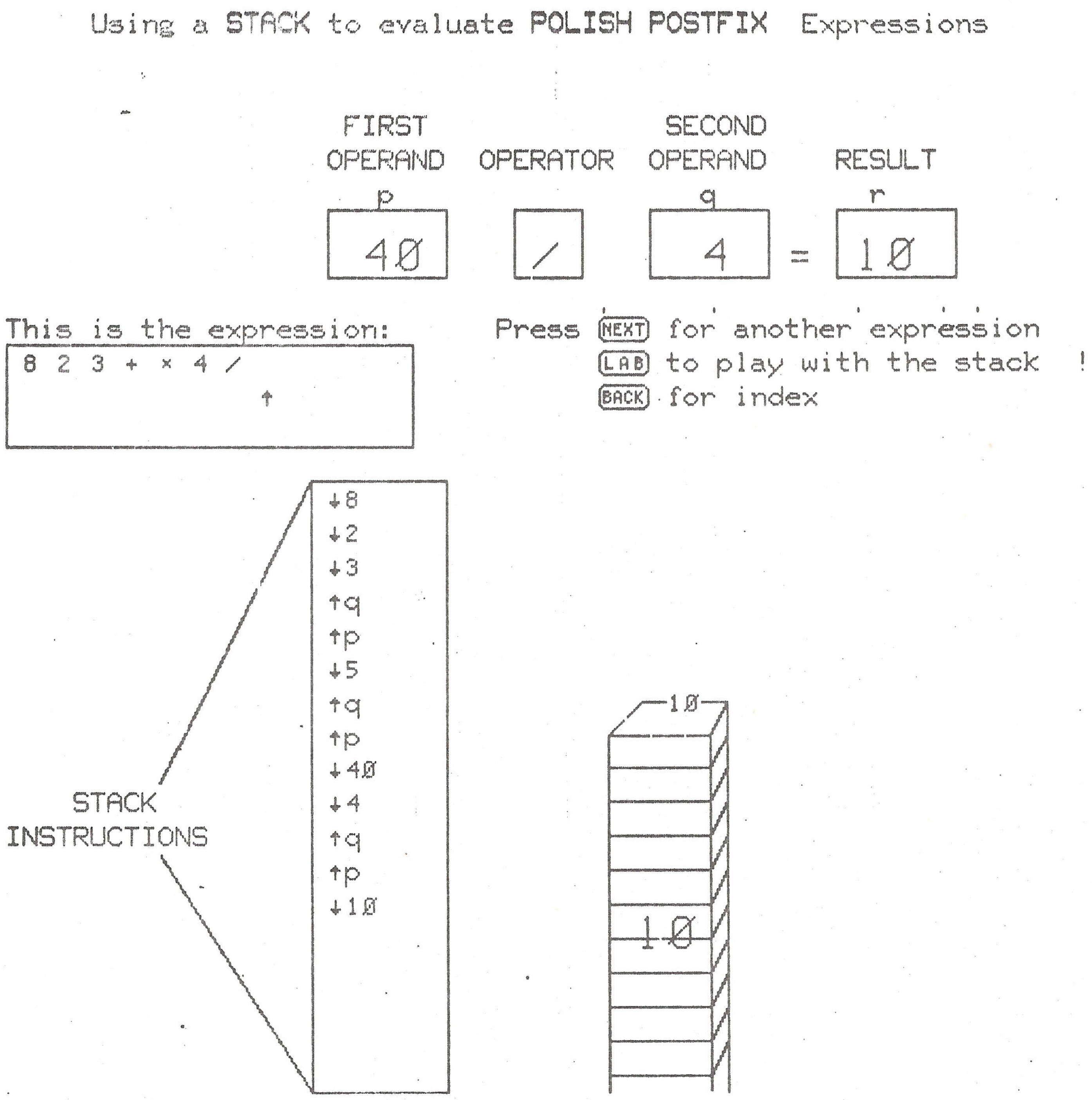
```
8 2 3 + × 4 /
            ↑
```

We are now finished.
The top of the stack is the result!

STACK INSTRUCTIONS

```
↓8
↓2
↓3
↑q
↑p
↓5
↑q
↑p
↓4Ø
↓4
↑q
↑p
↓1Ø
```

Figure 16



Using a STACK to evaluate POLISH POSTFIX Expressions

FIRST OPERAND p	OPERATOR	SECOND OPERAND q		RESULT r
40	/	4	=	10

This is the expression:

```
8 2 3 + × 4 /
            ↑
```

Press NEXT for another expression
LAB to play with the stack !
BACK for index

STACK INSTRUCTIONS

```
↓8
↓2
↓3
↑q
↑p
↓5
↑q
↑p
↓40
↓4
↑q
↑p
↓10
```

Figure 17

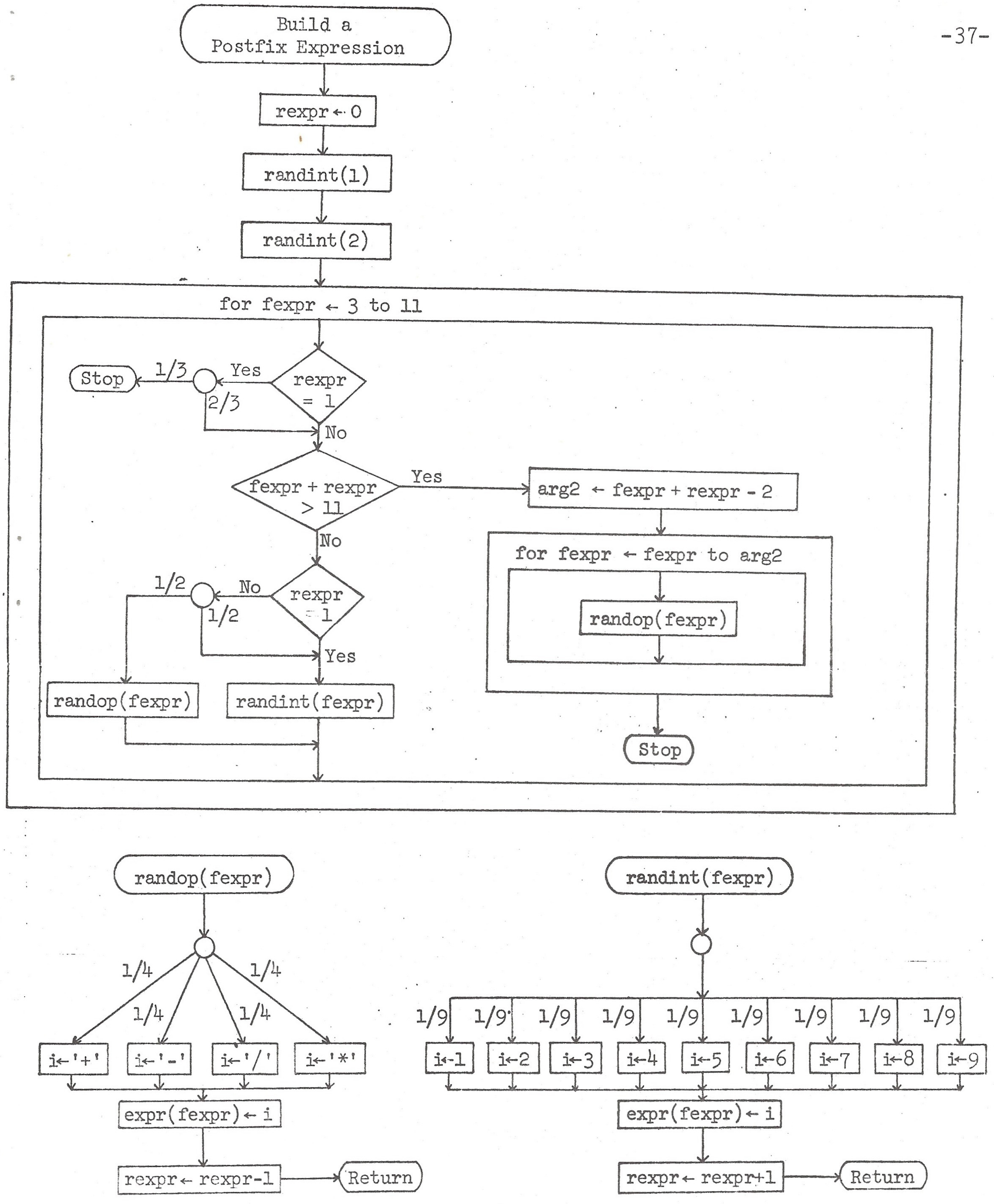
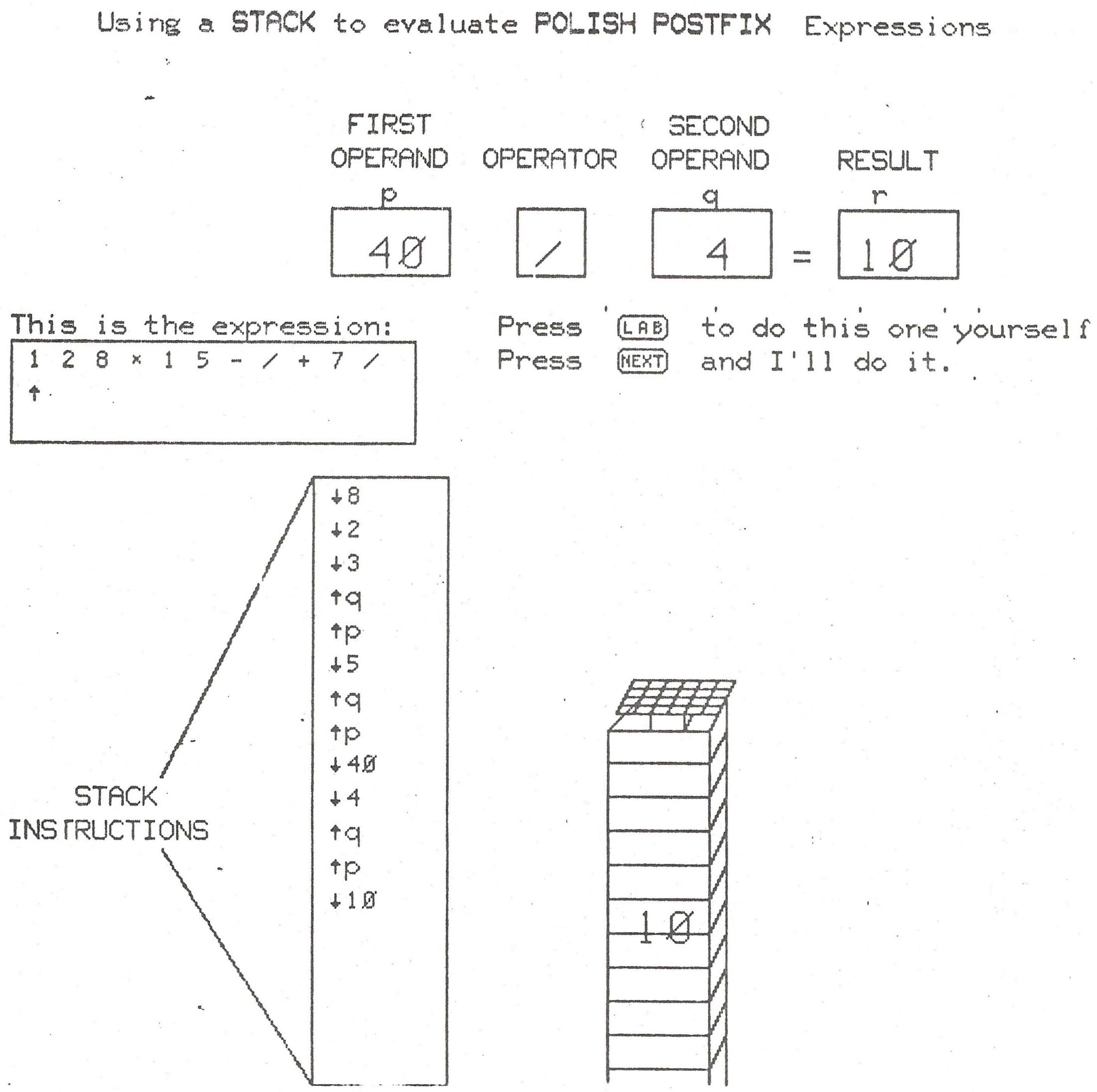

Build a Postfix Expression
rexpr ← 0
randint(1)
randint(2)
for fexpr ← 3 to 11
rexpr = 1
Yes
No
1/3
2/3
Stop
fexpr + rexpr > 11
Yes
No
arg2 ← fexpr + rexpr - 2
for fexpr ← fexpr to arg2
randop(fexpr)
Stop
rexpr = 1
No
Yes
1/2
1/2
randop(fexpr)
randint(fexpr)
randop(fexpr)
1/4
i←'+'
i←'-'
i←'/'
i←'*'
expr(fexpr) ← i
rexpr ← rexpr-1
Return
randint(fexpr)
1/9
i←1
i←2
i←3
i←4
i←5
i←6
i←7
i←8
i←9
expr(fexpr) ← i
rexpr ← rexpr+1
Return

Figure 18



Using a STACK to evaluate POLISH POSTFIX Expressions

FIRST OPERAND p	OPERATOR	SECOND OPERAND q		RESULT r
4Ø	/	4	=	1Ø

This is the expression:

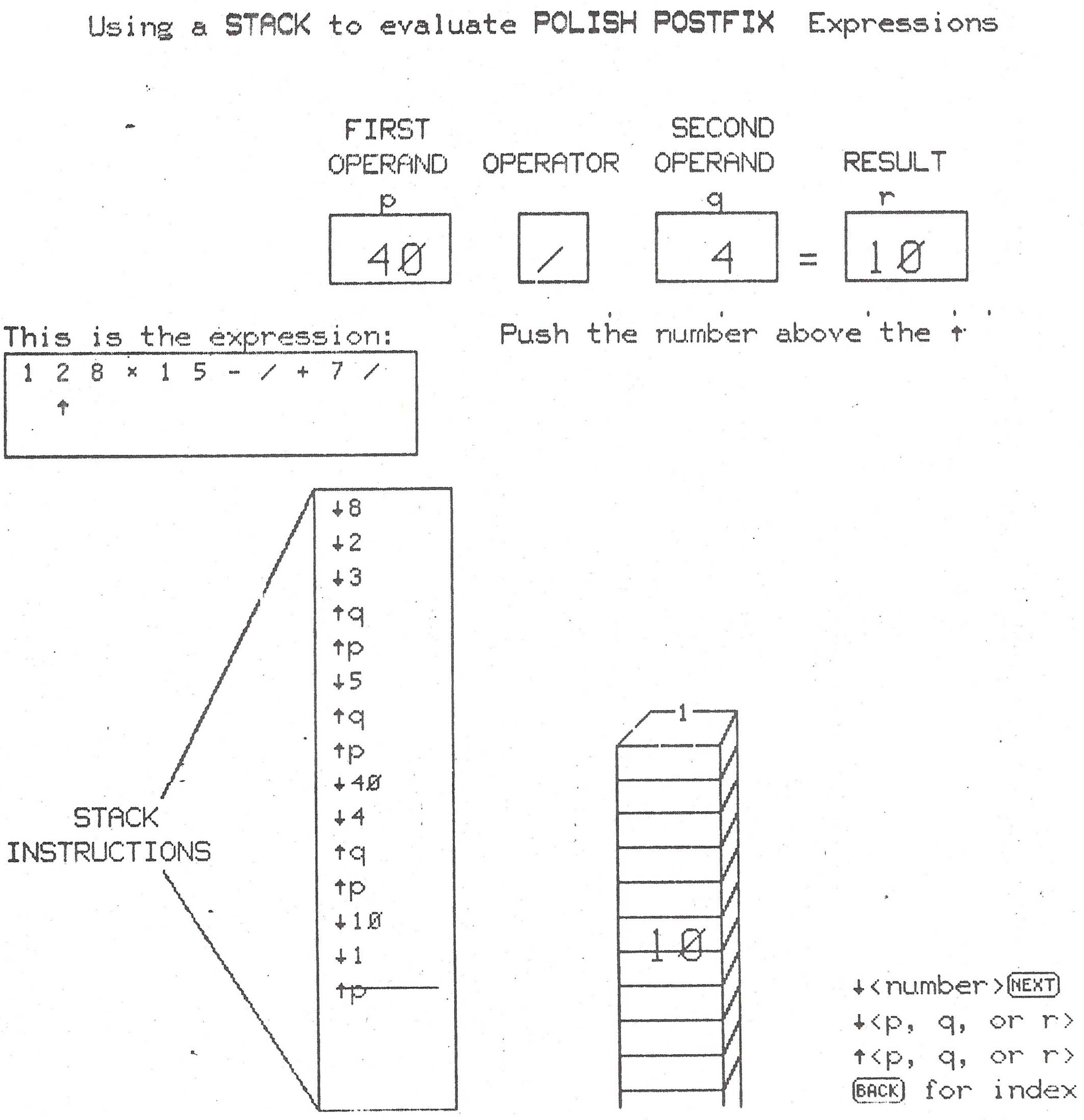
```
1 2 8 × 1 5 - / + 7 /
↑
```

Press LAB to do this one yourself
Press NEXT and I'll do it.

STACK INSTRUCTIONS

```
↓8
↓2
↓3
↑q
↑p
↓5
↑q
↑p
↓4Ø
↓4
↑q
↑p
↓1Ø
```

Figure 19



Using a **STACK** to evaluate **POLISH POSTFIX** Expressions

FIRST OPERAND p	OPERATOR	SECOND OPERAND q		RESULT r
4Ø	/	4	=	1Ø

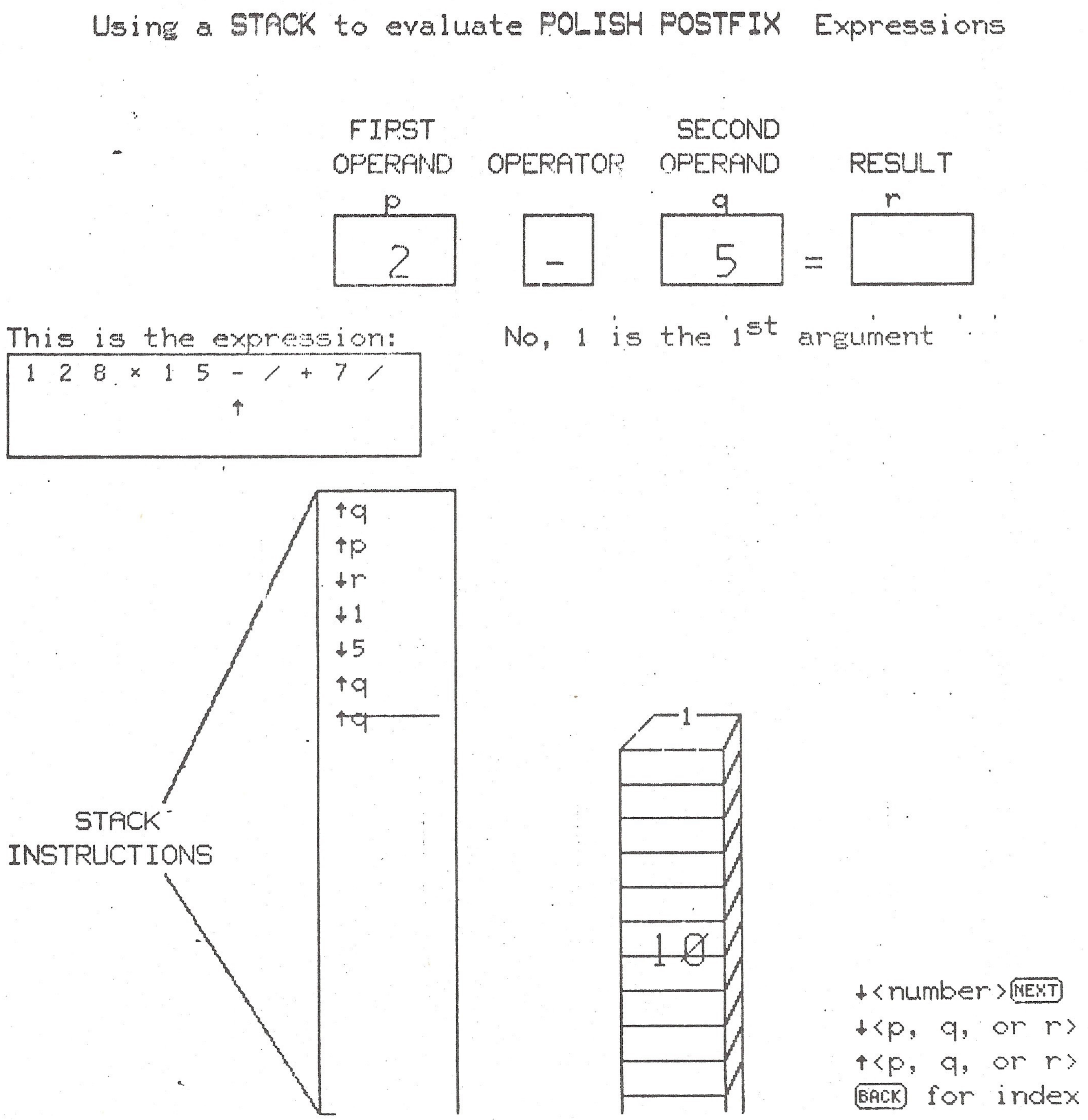
This is the expression:

```
1 2 8 × 1 5 - / + 7 /
  ↑
```

Push the number above the ↑

↓<number> NEXT
↓<p, q, or r>
↑<p, q, or r>
BACK for index

Figure 20


Using a STACK to evaluate POLISH POSTFIX Expressions

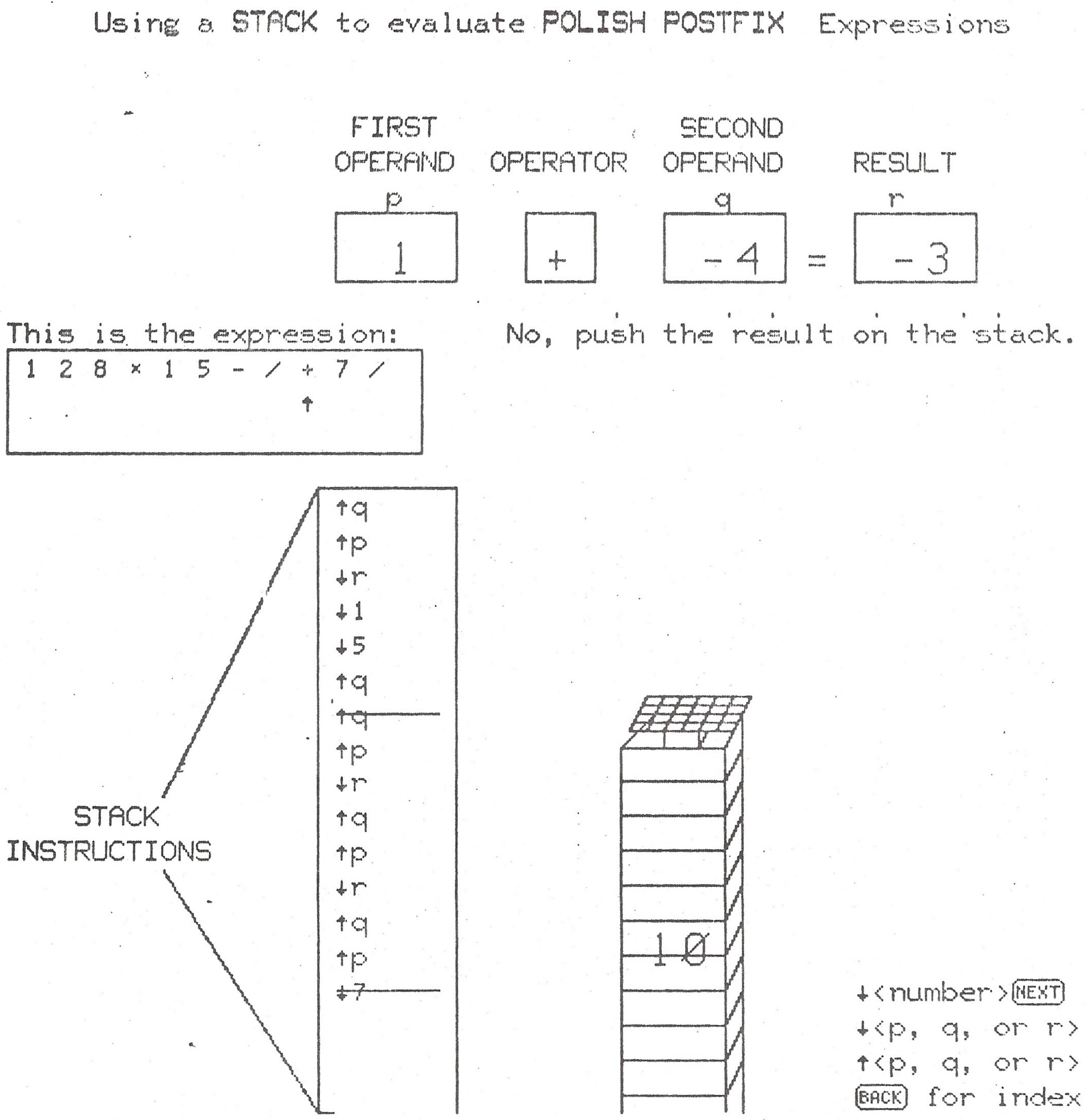
This is the expression:

1 2 8 × 1 5 - / + 7 /

No, 1 is the 1st argument

↓<number>NEXT
↓<p, q, or r>
↑<p, q, or r>
BACK for index

Figure 21



Using a STACK to evaluate POLISH POSTFIX Expressions

FIRST OPERAND	OPERATOR	SECOND OPERAND		RESULT
p		q		r
1	+	-4	=	-3

This is the expression:

```
1 2 8 × 1 5 - / + 7 /
                ↑
```

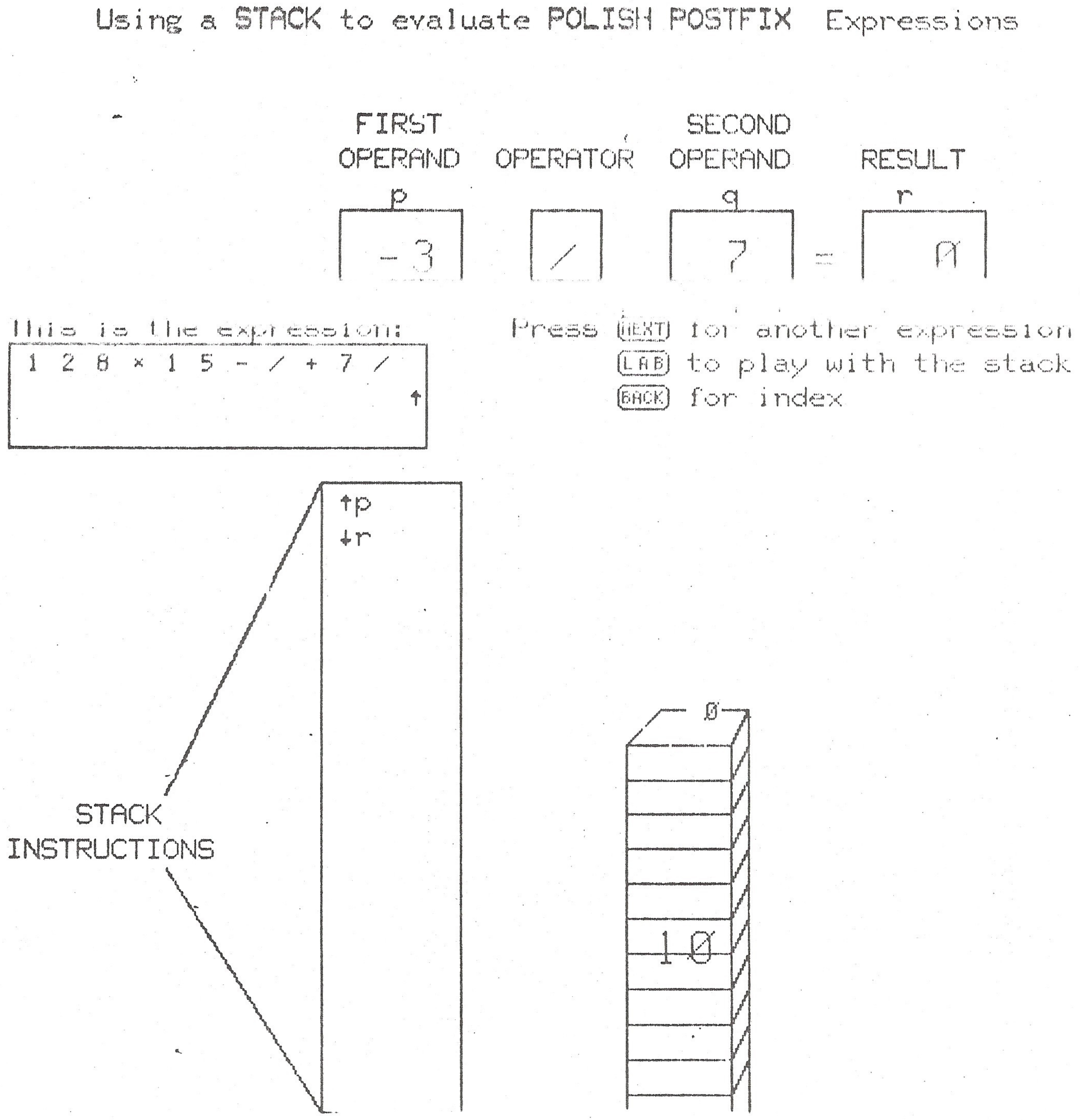
No, push the result on the stack.

STACK INSTRUCTIONS

↑q
↑p
↓r
↓1
↓5
↑q
~~↑q~~
↑p
↓r
↑q
↑p
↓r
↑q
↑p
~~↓7~~

↓<number> NEXT
↓<p, q, or r>
↑<p, q, or r>
BACK for index

Figure 22


Using a STACK to evaluate POLISH POSTFIX Expressions

This is the expression:

```
1 2 8 × 1 5 - / + 7 /
```

Press NEXT for another expression
LAB to play with the stack
BACK for index

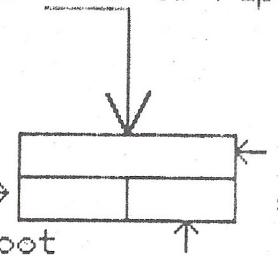
Figure 23

Using a STACK to Traverse a BINARY TREE

- A **BINARY TREE** is either **EMPTY**,

or consists of a **ROOT NODE** and two binary **SUB-TREES**.

We wil. use this to represent a NODE.



This field will contain some INFORMATION.

This field will point to the root of the LEFT sub-tree.

This field will point to the root of the RIGHT sub-tree.

This ⏚ will represent a pointer to the empty tree.

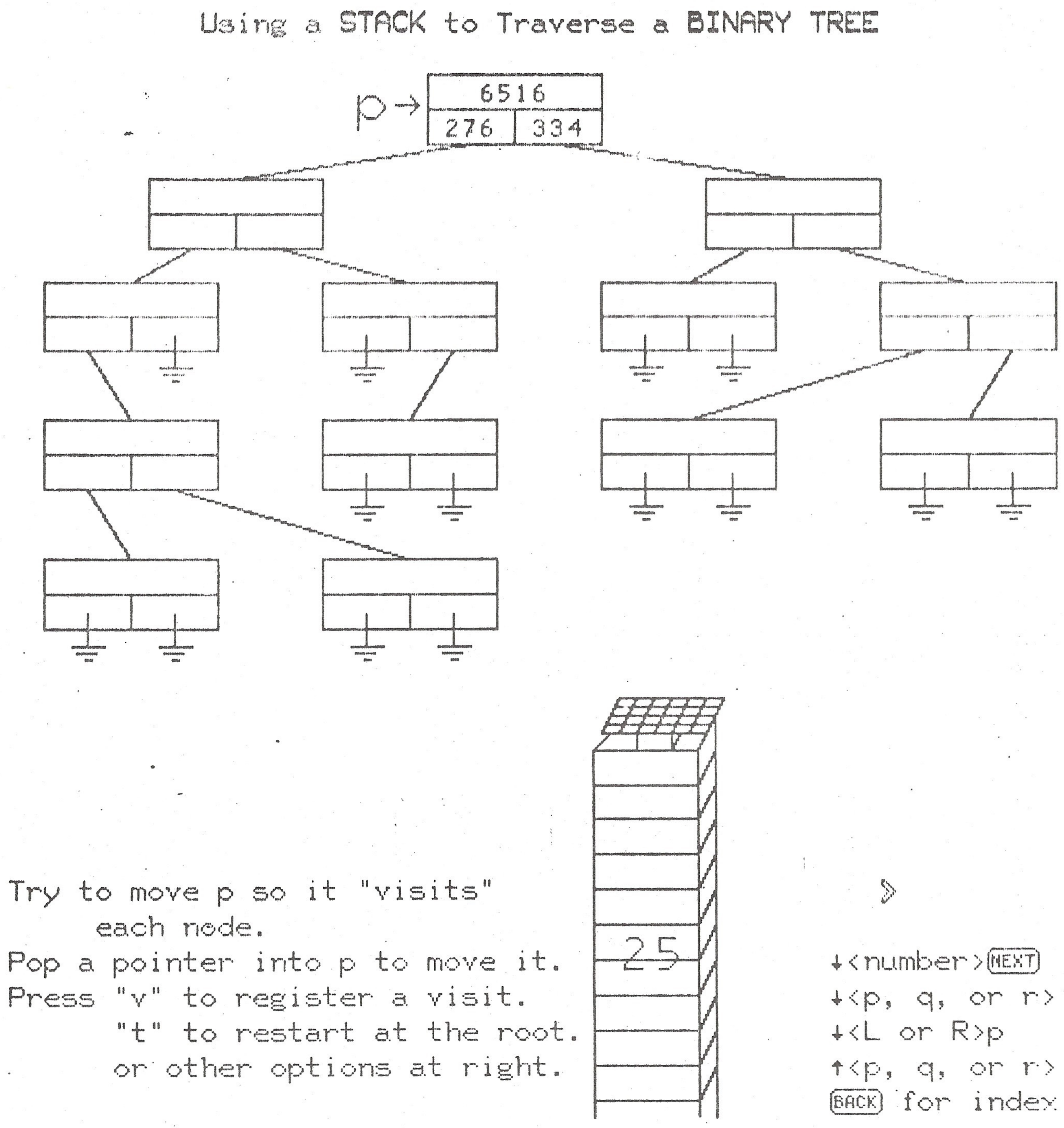
We will use p as a pointer variable.

Lp will refer to the Left subtree of p.

Rp will refer to the Right subtree of p.

press NEXT

Figure 24

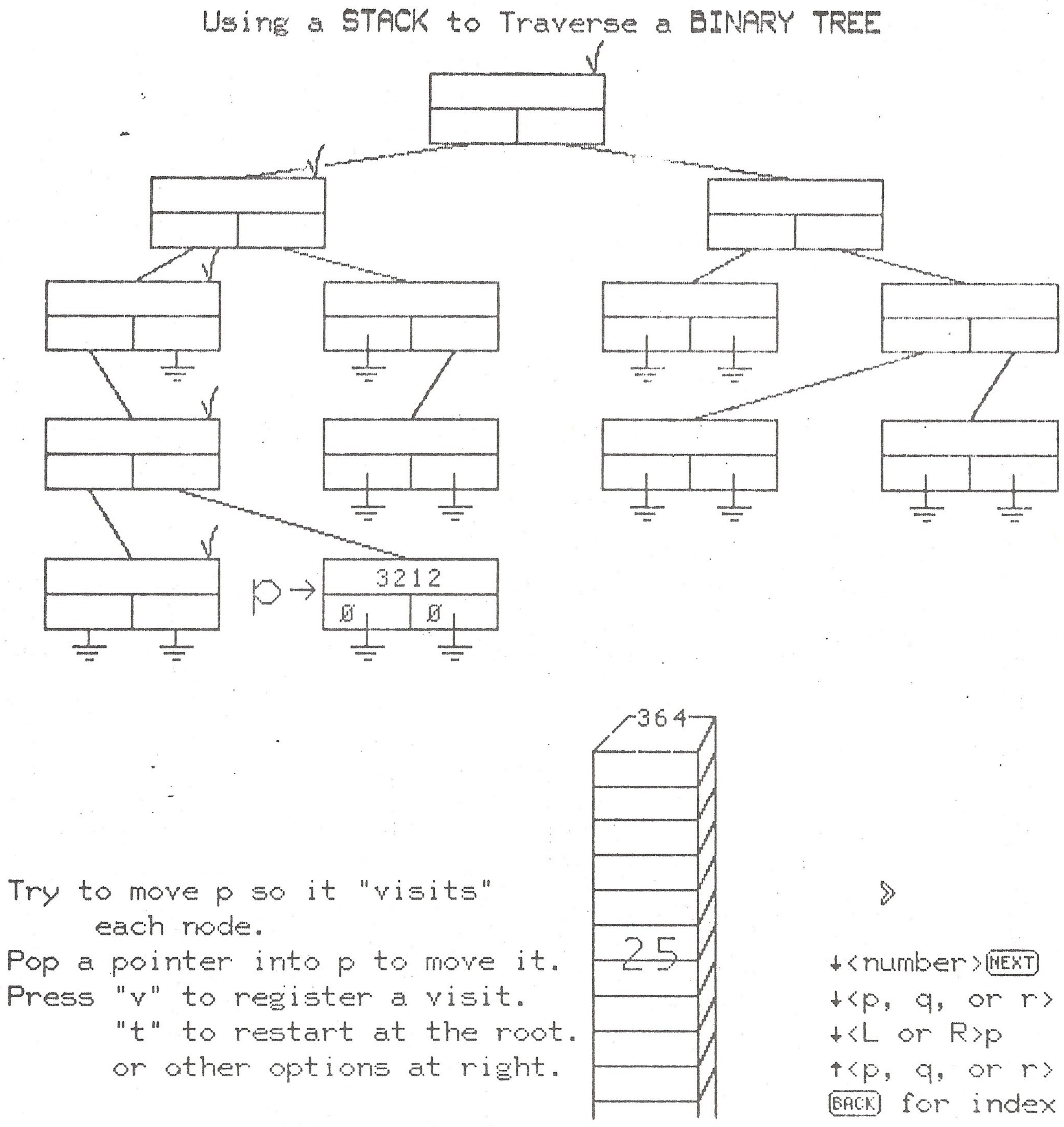


Using a STACK to Traverse a BINARY TREE

Try to move p so it "visits"
each node.
Pop a pointer into p to move it.
Press "v" to register a visit.
"t" to restart at the root.
or other options at right.

↓<number>NEXT
↓<p, q, or r>
↓<L or R>p
↑<p, q, or r>
BACK for index

Figure 25



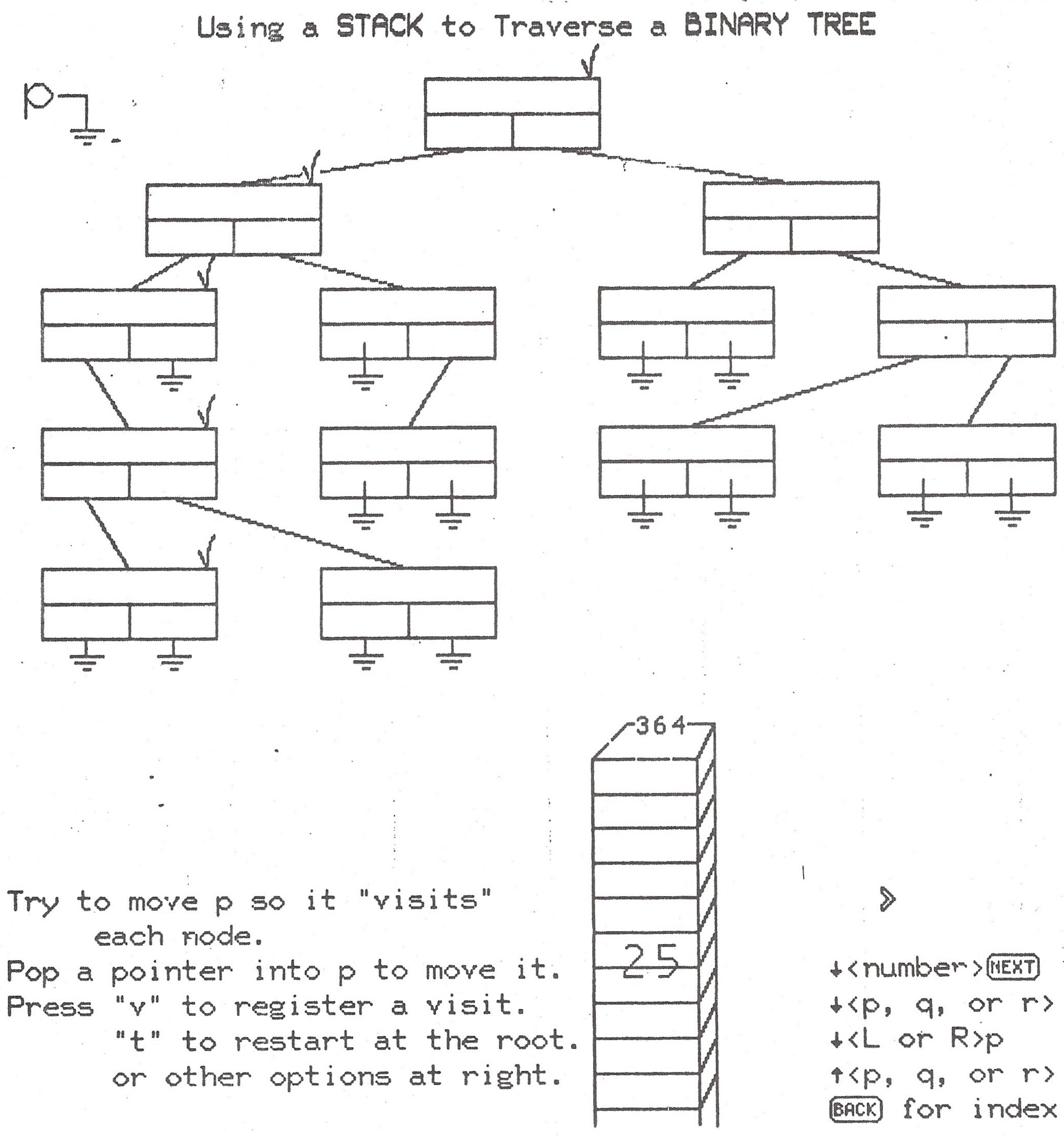
Using a STACK to Traverse a BINARY TREE

Try to move p so it "visits"
each node.
Pop a pointer into p to move it.
Press "v" to register a visit.
"t" to restart at the root.
or other options at right.

»

↓<number>NEXT
↓<p, q, or r>
↓<L or R>p
↑<p, q, or r>
BACK for index

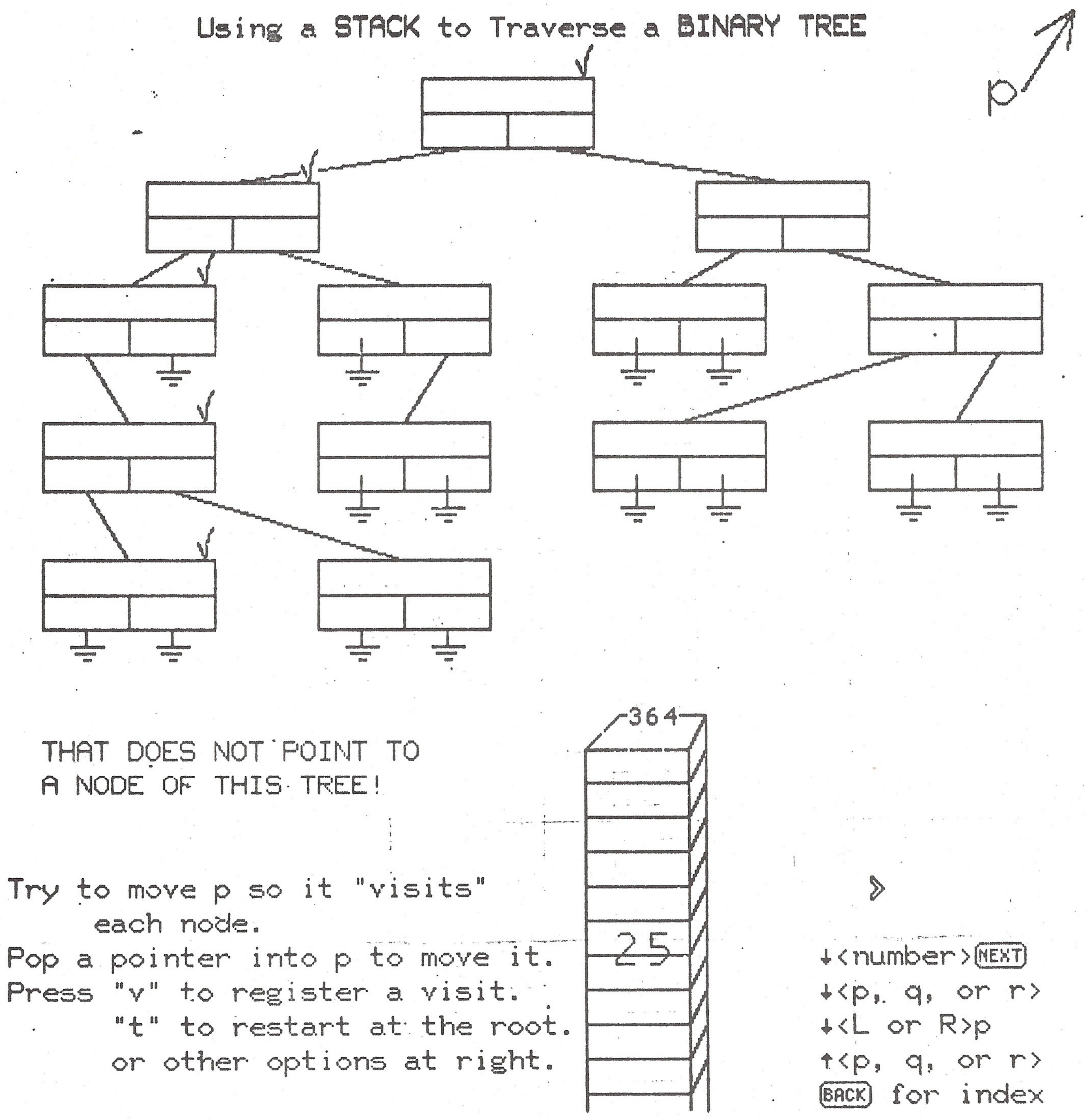
Figure 26



Using a STACK to Traverse a BINARY TREE

Try to move p so it "visits"
each node.
Pop a pointer into p to move it.
Press "v" to register a visit.
"t" to restart at the root.
or other options at right.

Figure 27



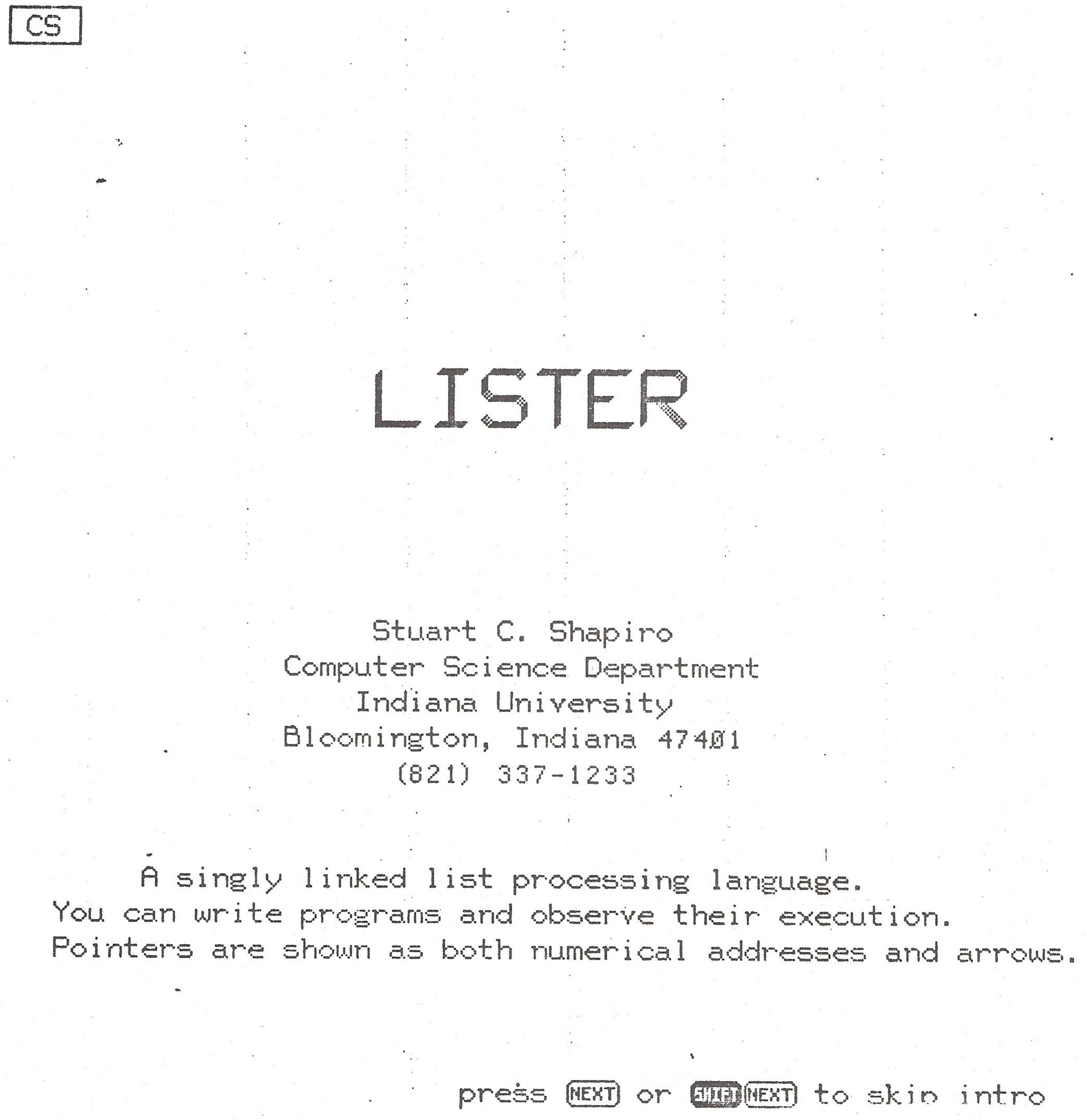
Using a STACK to Traverse a BINARY TREE

THAT DOES NOT POINT TO
A NODE OF THIS TREE!

Try to move p so it "visits"
each node.
Pop a pointer into p to move it.
Press "v" to register a visit.
"t" to restart at the root.
or other options at right.

↓<number> NEXT
↓<p, q, or r>
↓<L or R>p
↑<p, q, or r>
BACK for index

Figure 28



CS

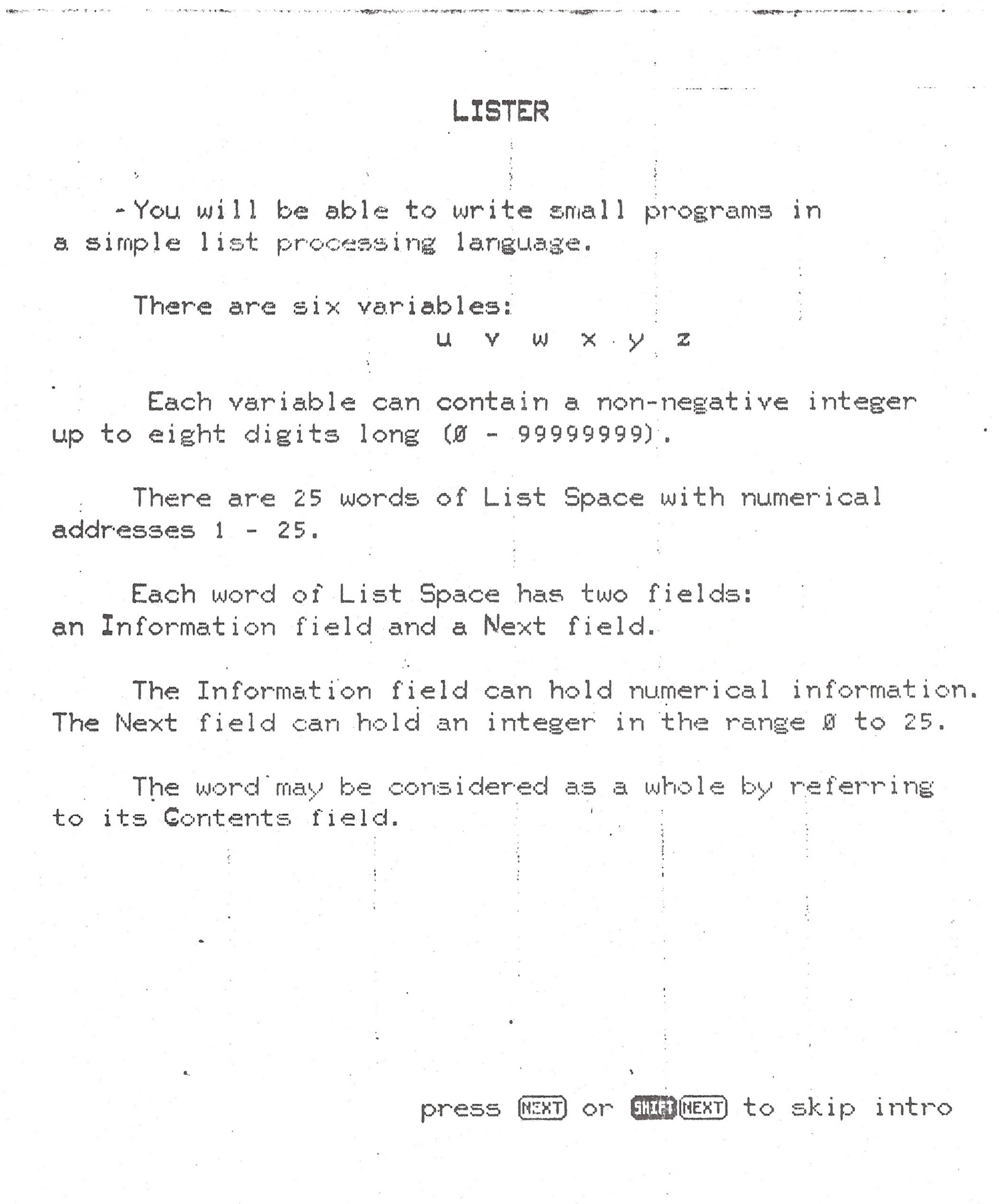
LISTER

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana 47401
(821) 337-1233

A singly linked list processing language.
You can write programs and observe their execution.
Pointers are shown as both numerical addresses and arrows.

press NEXT or SHIFT NEXT to skip intro

Figure 29



LISTER

-You will be able to write small programs in a simple list processing language.

There are six variables:

u v w x y z

Each variable can contain a non-negative integer up to eight digits long (Ø - 99999999).

There are 25 words of List Space with numerical addresses 1 - 25.

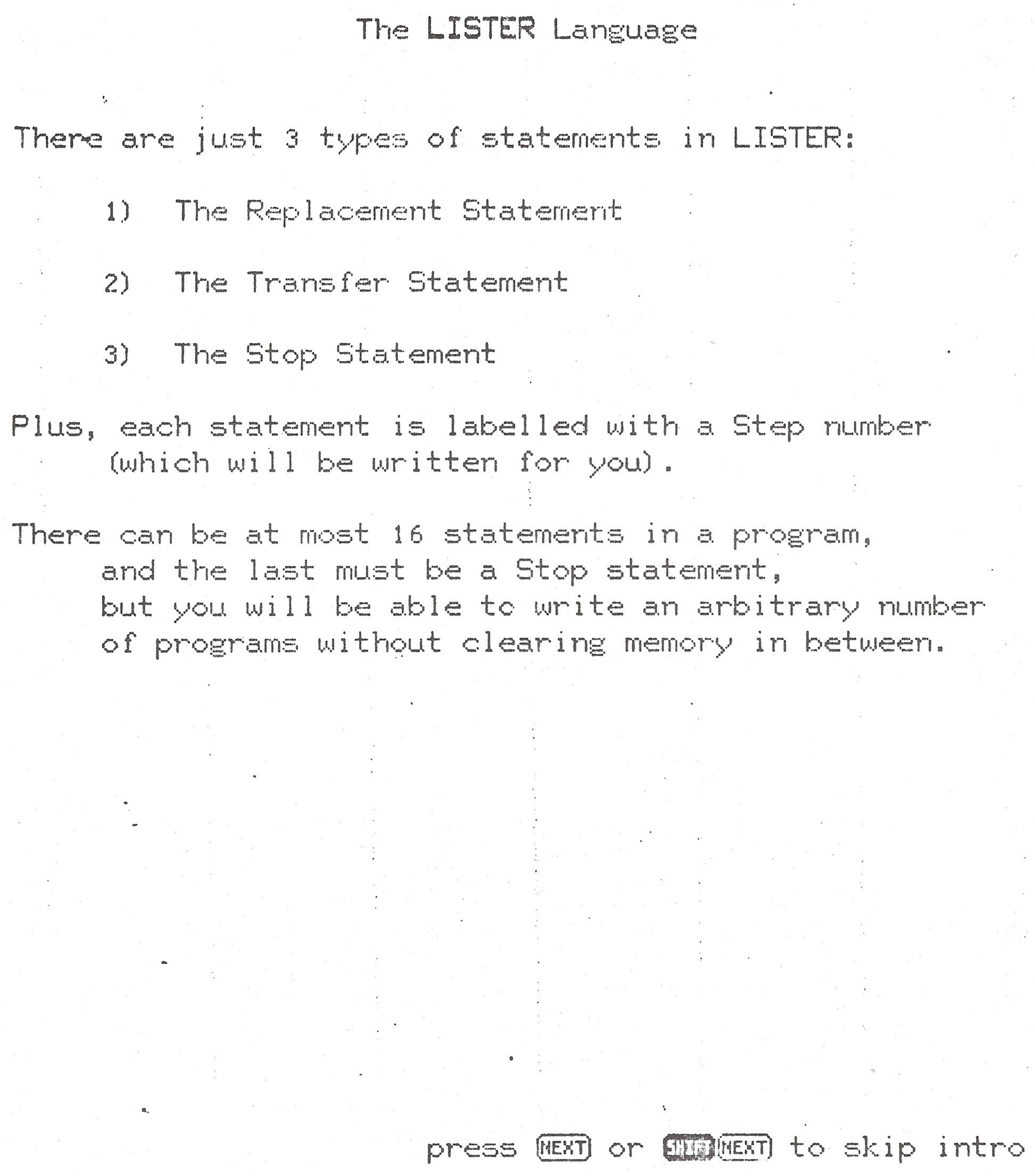
Each word of List Space has two fields: an Information field and a Next field.

The Information field can hold numerical information. The Next field can hold an integer in the range Ø to 25.

The word may be considered as a whole by referring to its Contents field.

press $\boxed{\text{NEXT}}$ or $\boxed{\text{SHIFT}}\boxed{\text{NEXT}}$ to skip intro

Figure 30



The **LISTER** Language

There are just 3 types of statements in LISTER:

1) The Replacement Statement

2) The Transfer Statement

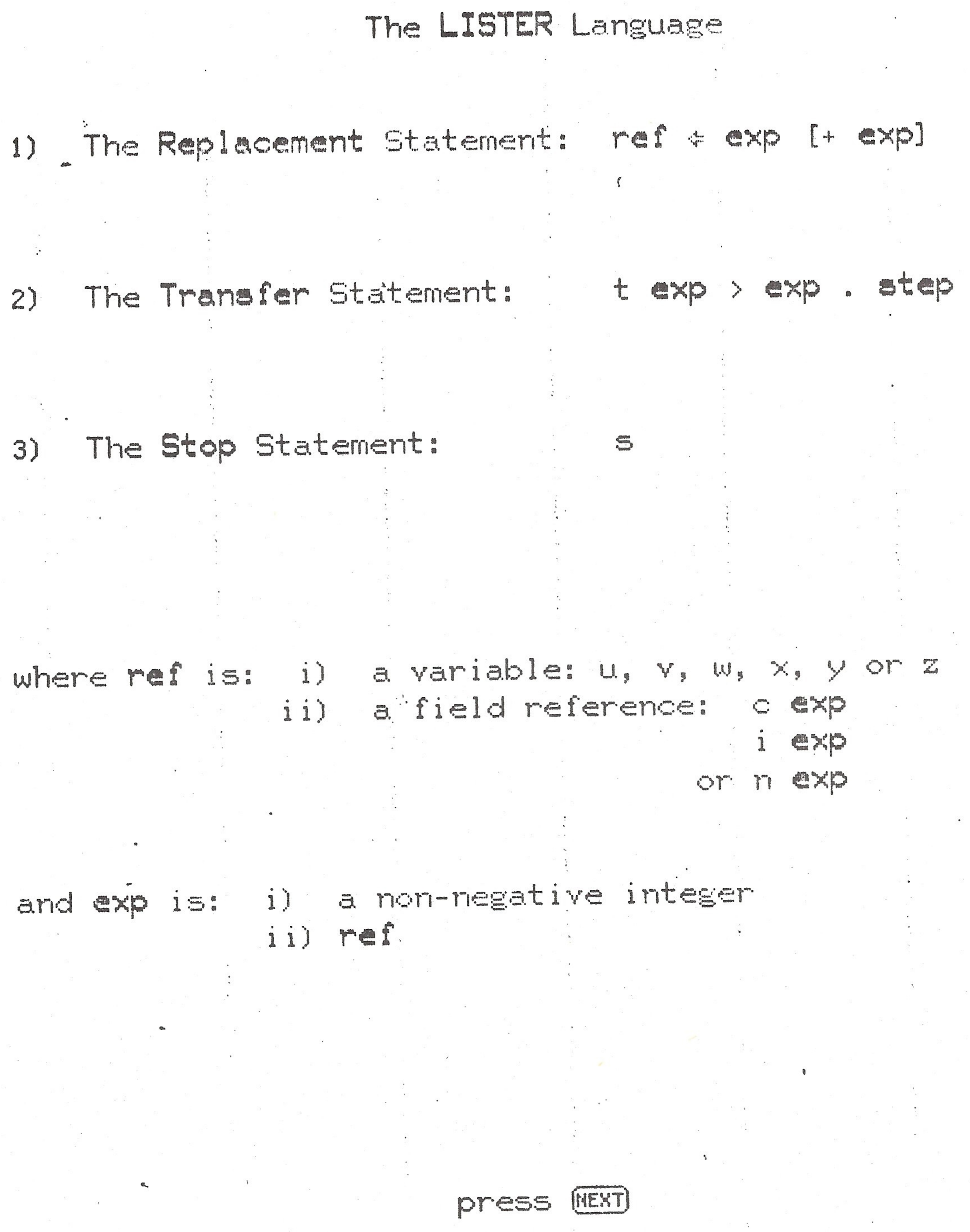
3) The Stop Statement

Plus, each statement is labelled with a Step number
(which will be written for you).

There can be at most 16 statements in a program,
and the last must be a Stop statement,
but you will be able to write an arbitrary number
of programs without clearing memory in between.

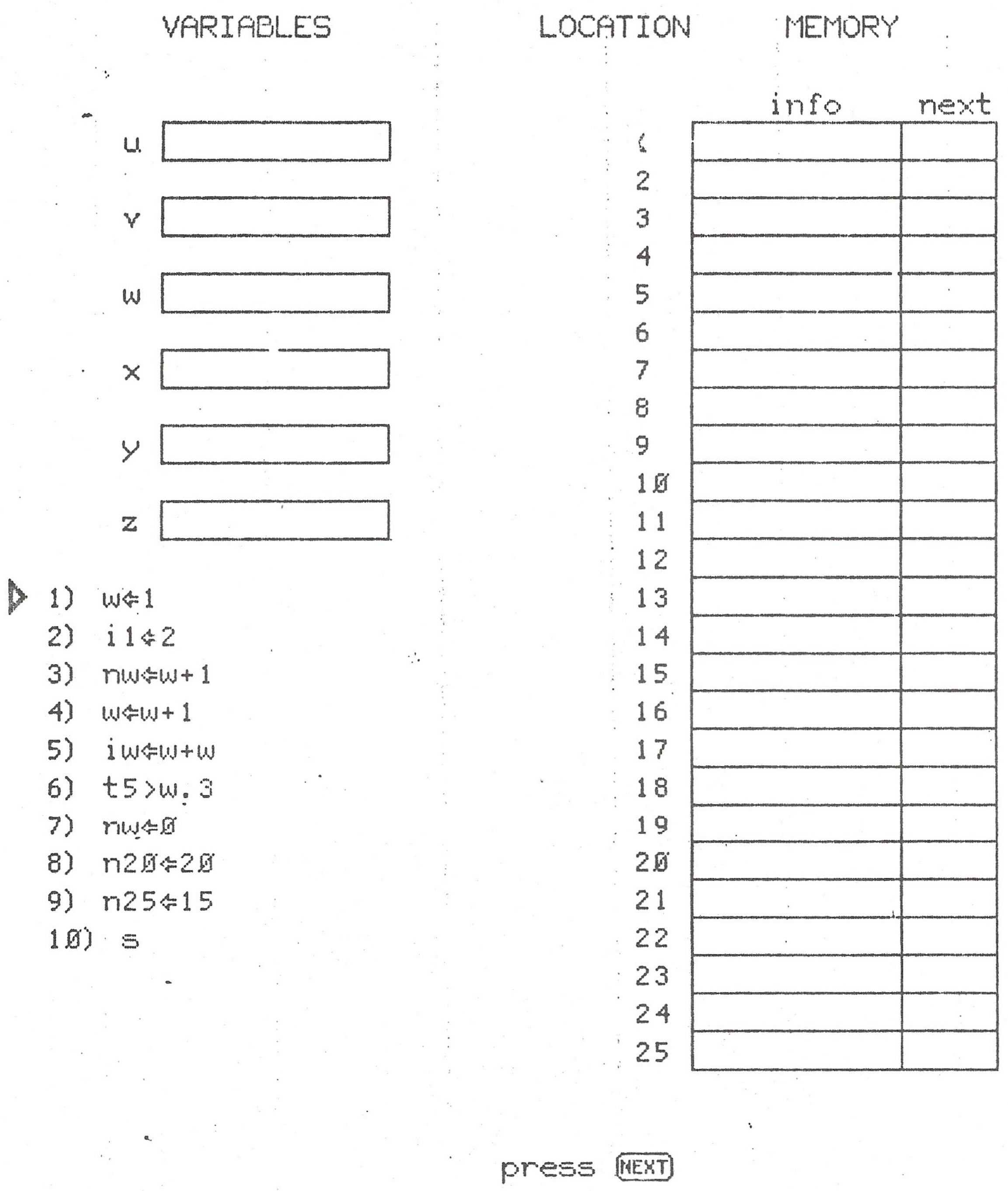
press NEXT or SHIFT NEXT to skip intro

Figure 31



```
                    The LISTER Language

1)  The Replacement Statement:   ref ← exp [+ exp]


2)  The Transfer Statement:      t exp > exp . step


3)  The Stop Statement:          s




where ref is:   i)  a variable: u, v, w, x, y or z
               ii)  a field reference:  c exp
                                        i exp
                                     or n exp


and exp is:   i)  a non-negative integer
             ii)  ref




                       press NEXT
```

Figure 32



VARIABLES

u	
v	
w	
x	
y	
z	

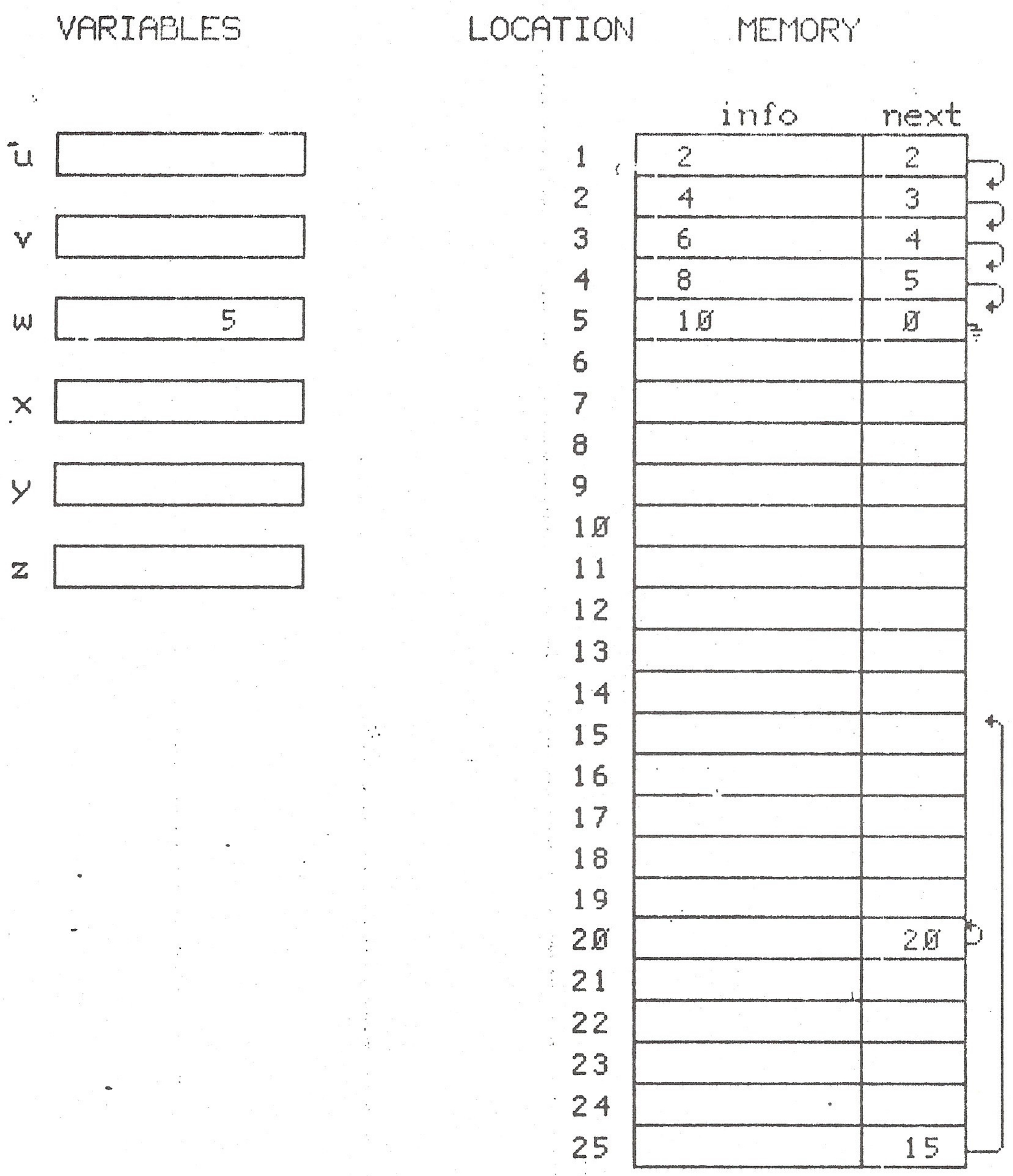
```
▷ 1)  w⇐1
  2)  i1⇐2
  3)  nw⇐w+1
  4)  w⇐w+1
  5)  iw⇐w+w
  6)  t5>w,3
  7)  nw⇐Ø
  8)  n2Ø⇐2Ø
  9)  n25⇐15
  1Ø)  s
```

LOCATION MEMORY

	info	next
1		
2		
3		
4		
5		
6		
7		
8		
9		
1Ø		
11		
12		
13		
14		
15		
16		
17		
18		
19		
2Ø		
21		
22		
23		
24		
25		

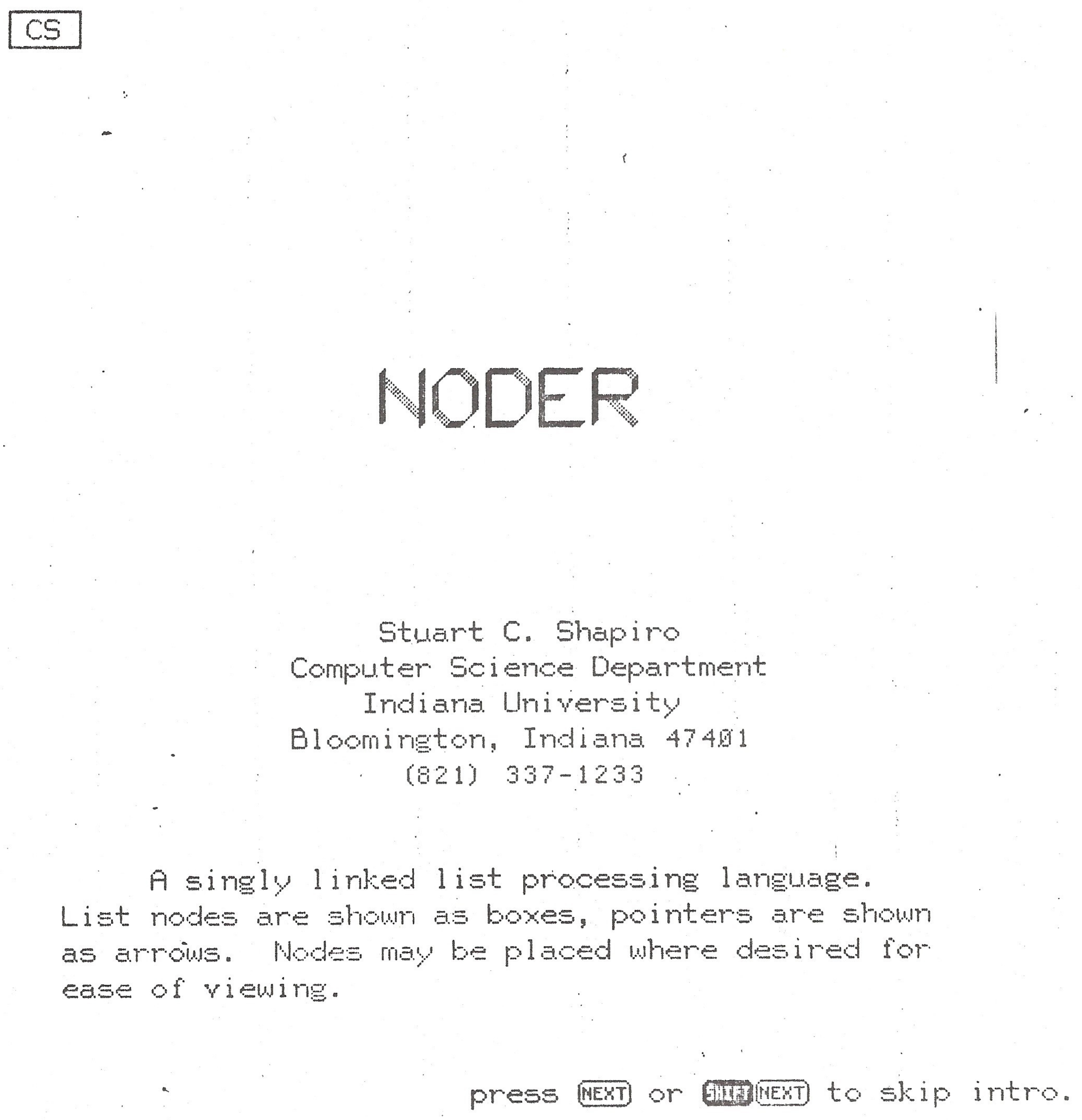
press NEXT

Figure 33


VARIABLES
LOCATION
MEMORY
info
next
u
v
w
5
x
y
z
1 2 2
2 4 3
3 6 4
4 8 5
5 1Ø Ø
6
7
8
9
1Ø
11
12
13
14
15
16
17
18
19
2Ø 2Ø
21
22
23
24
25 15

1)

Figure 34



CS

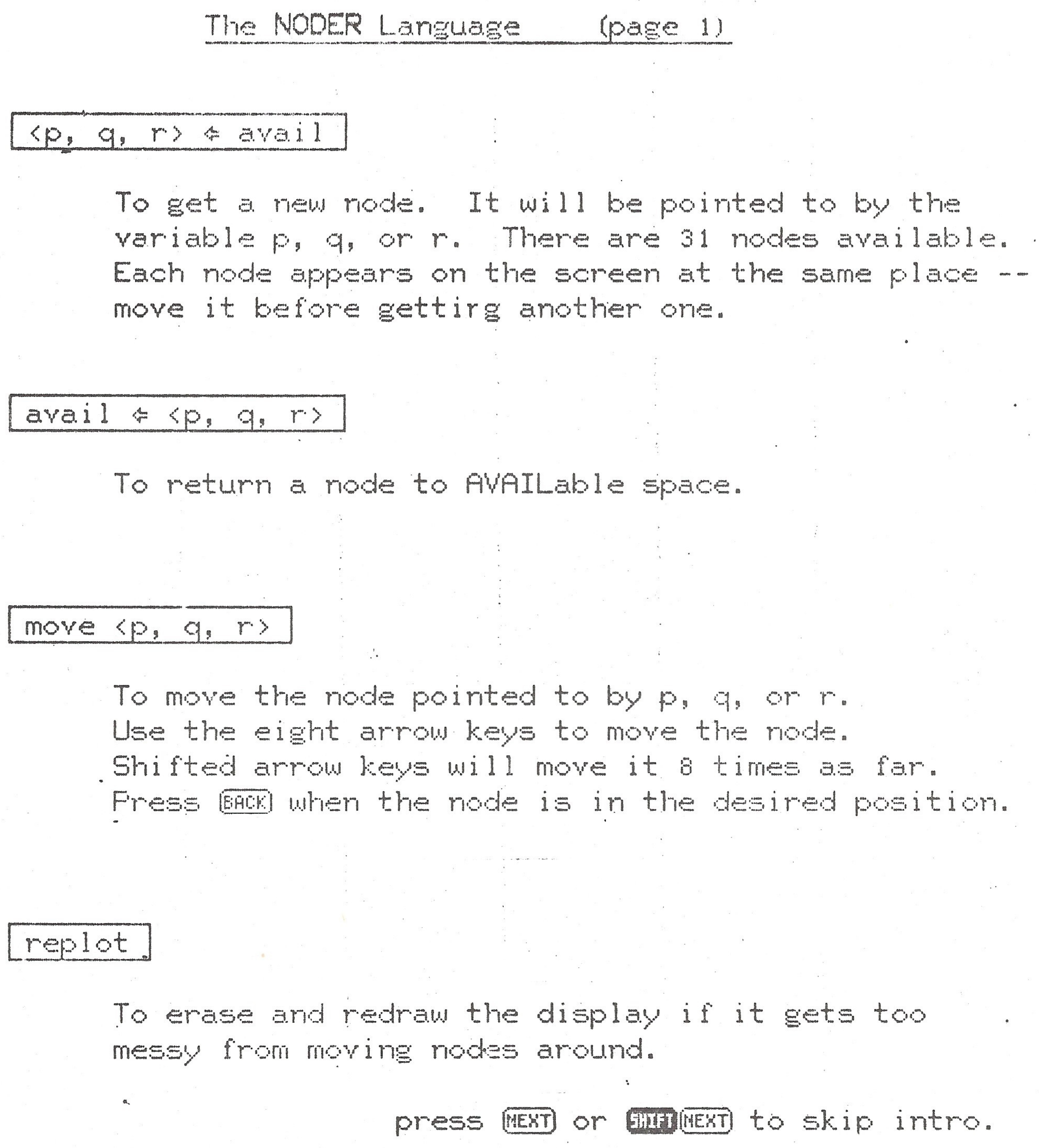
NODER

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana 47401
(821) 337-1233

A singly linked list processing language. List nodes are shown as boxes, pointers are shown as arrows. Nodes may be placed where desired for ease of viewing.

press NEXT or SHIFT NEXT to skip intro.

Figure 35



The NODER Language (page 1)

`<p, q, r> ⇐ avail`

To get a new node. It will be pointed to by the variable p, q, or r. There are 31 nodes available. Each node appears on the screen at the same place -- move it before gettirg another one.

`avail ⇐ <p, q, r>`

To return a node to AVAILable space.

`move <p, q, r>`

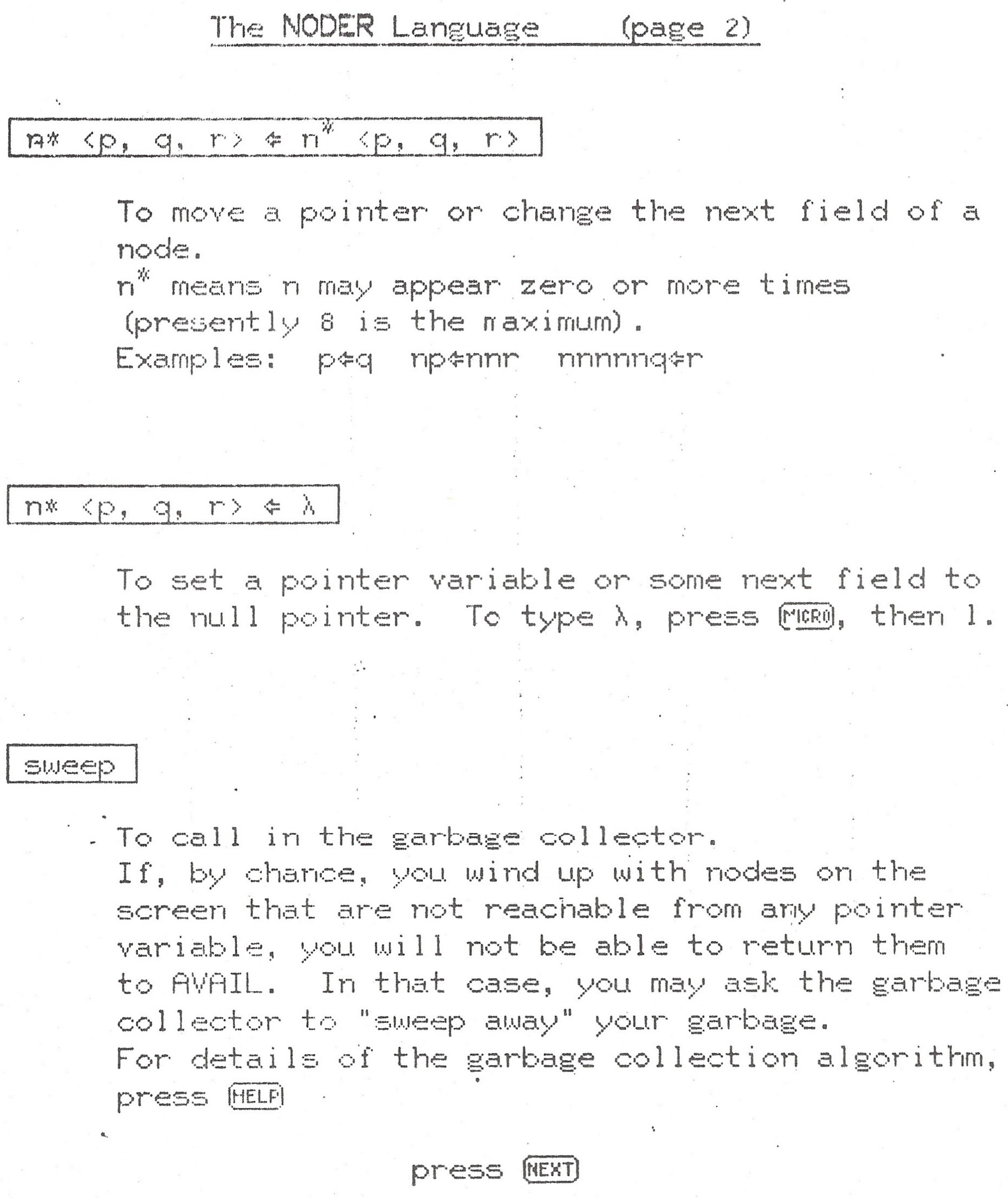
To move the node pointed to by p, q, or r.
Use the eight arrow keys to move the node.
Shifted arrow keys will move it 8 times as far.
Press BACK when the node is in the desired position.

`replot`

To erase and redraw the display if it gets too messy from moving nodes around.

press NEXT or SHIFT NEXT to skip intro.

Figure 36



The NODER Language (page 2)

n* <p, q, r> ⇐ n^* <p, q, r>

To move a pointer or change the next field of a node.
n^* means n may appear zero or more times (presently 8 is the maximum).
Examples: p⇐q np⇐nnr nnnnnq⇐r

n* <p, q, r> ⇐ λ

To set a pointer variable or some next field to the null pointer. To type λ, press MICRO, then l.

sweep

To call in the garbage collector.
If, by chance, you wind up with nodes on the screen that are not reachable from any pointer variable, you will not be able to return them to AVAIL. In that case, you may ask the garbage collector to "sweep away" your garbage.
For details of the garbage collection algorithm, press HELP

press NEXT

Figure 37

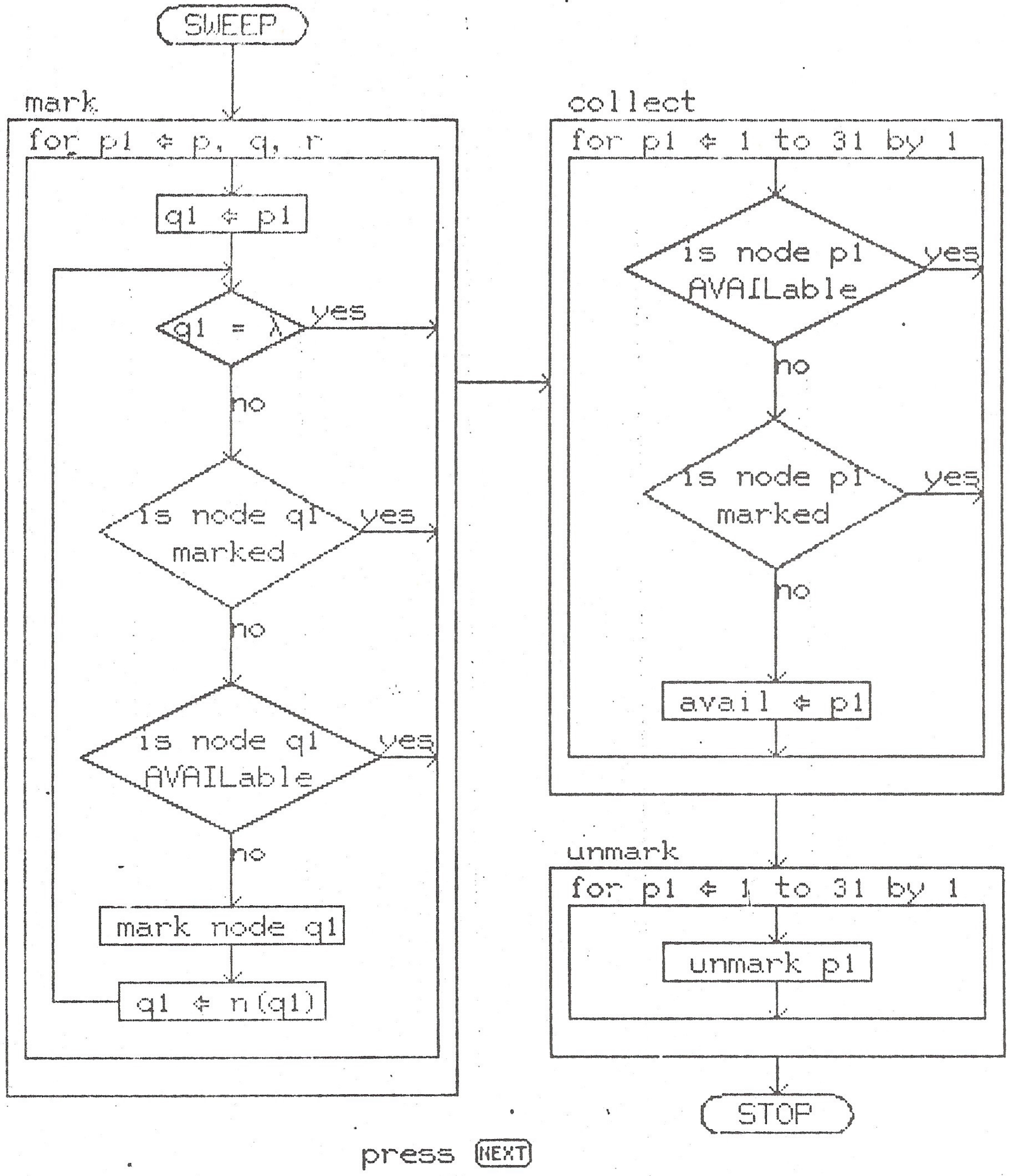
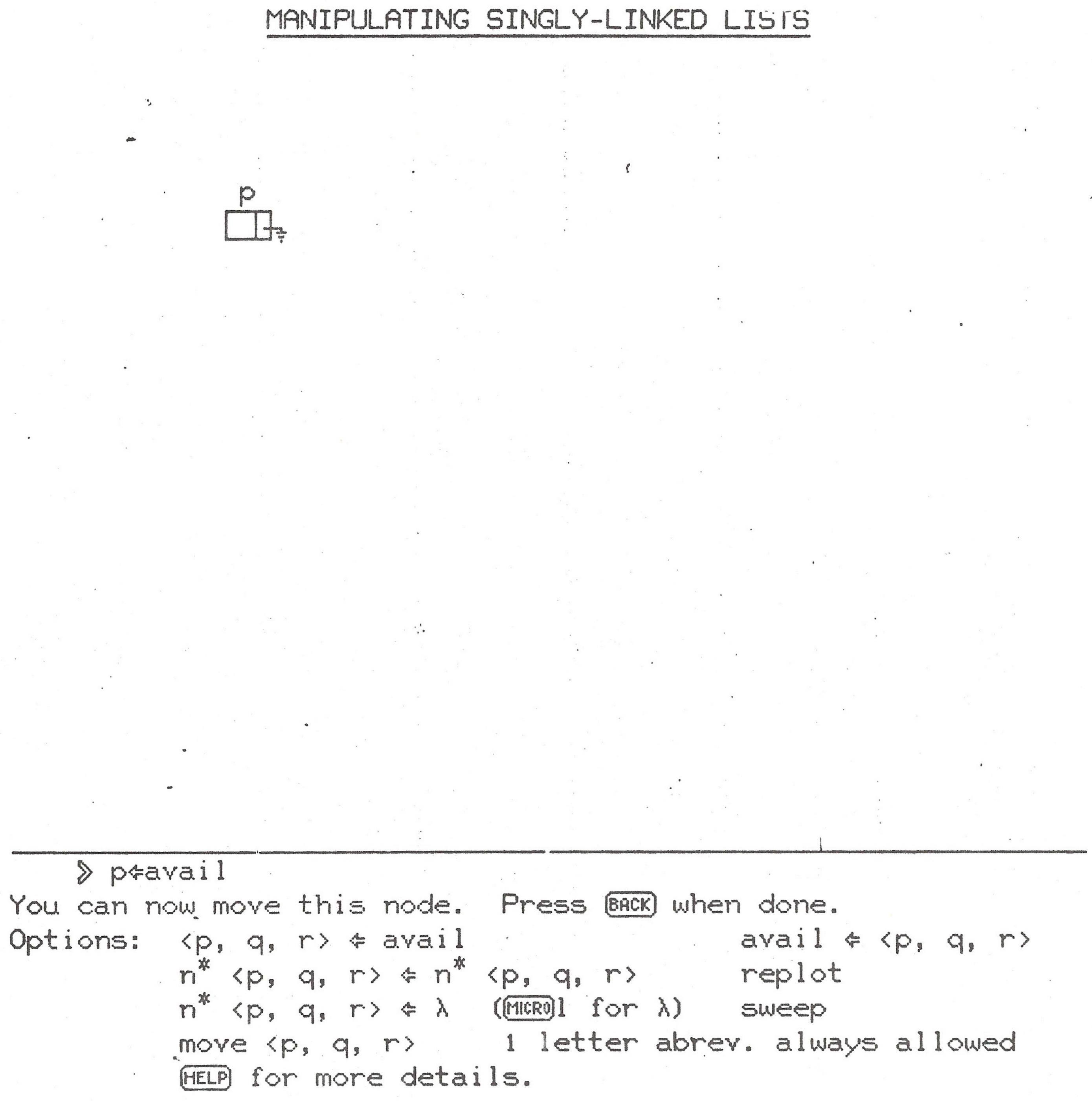

SWEEP
mark
for p1 ⇐ p, q, r
q1 ⇐ p1
q1 = λ
yes
no
is node q1 marked
yes
no
is node q1 AVAILable
yes
no
mark node q1
q1 ⇐ n(q1)
collect
for p1 ⇐ 1 to 31 by 1
is node p1 AVAILable
yes
no
is node p1 marked
yes
no
avail ⇐ p1
unmark
for p1 ⇐ 1 to 31 by 1
unmark p1
STOP
press NEXT

Figure 38

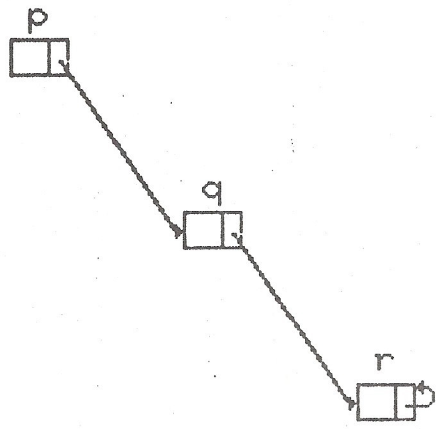


MANIPULATING SINGLY-LINKED LISTS

```
    ≫ p⇐avail
You can now move this node.  Press BACK when done.
Options:  <p, q, r> ⇐ avail                     avail ⇐ <p, q, r>
          n* <p, q, r> ⇐ n* <p, q, r>           replot
          n* <p, q, r> ⇐ λ   (MICRO l for λ)    sweep
          move <p, q, r>     1 letter abrev. always allowed
          HELP for more details.
```

Figure 39

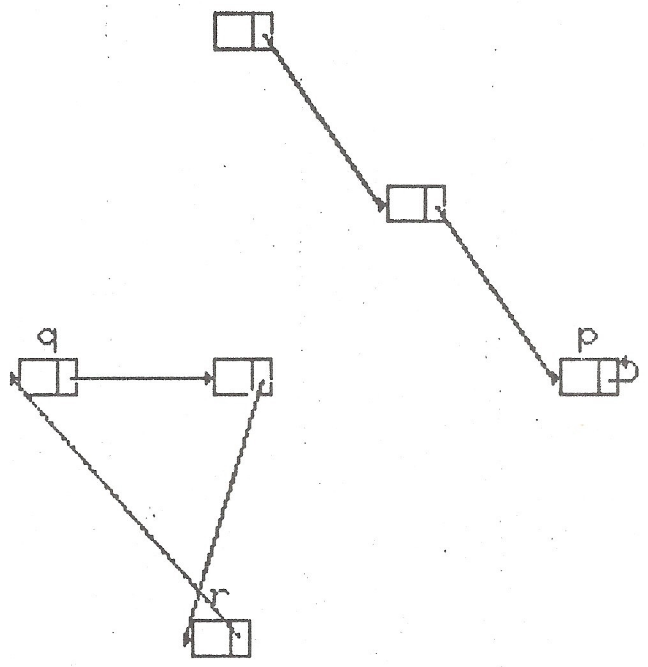
MANIPULATING SINGLY-LINKED LISTS



```
  »

Options:  <p, q, r> ⇐ avail                     avail ⇐ <p, q, r>
          n* <p, q, r> ⇐ n* <p, q, r>          replot
          n* <p, q, r> ⇐ λ   (MICRO l for λ)   sweep
          move <p, q, r>     1 letter abrev. always allowed
          HELP for more details.
```

Figure 40

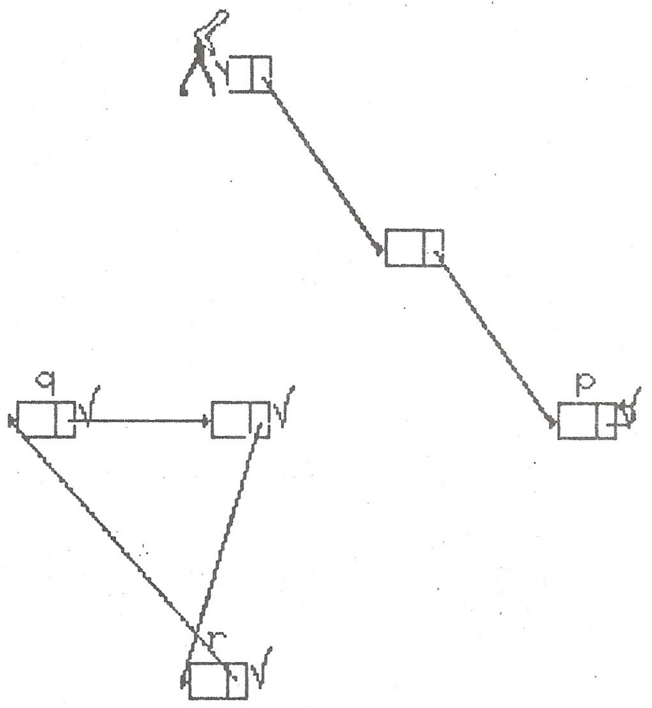
MANIPULATING SINGLY-LINKED LISTS



```
   ≫

Options:  <p, q, r> ⇐ avail                    avail ⇐ <p, q, r>
          n* <p, q, r> ⇐ n* <p, q, r>          replot
          n* <p, q, r> ⇐ λ   (MICRO l for λ)   sweep
          move <p, q, r>     1 letter abrev. always allowed
          HELP for more details.
```

Figure 41

MANIPULATING SINGLY-LINKED LISTS



```
    ≫ sweep

Options:  <p, q, r> ⇐ avail                   avail ⇐ <p, q, r>
          n* <p, q, r> ⇐ n* <p, q, r>         replot
          n* <p, q, r> ⇐ λ   (MICROl for λ)   sweep
          move <p, q, r>      1 letter abrev. always allowed
          HELP for more details.
```

Figure 42