MATERIAL AND IDEAS TO TEACH AN INTRODUCTORY PROGRAMMING COURSE USING LOGO

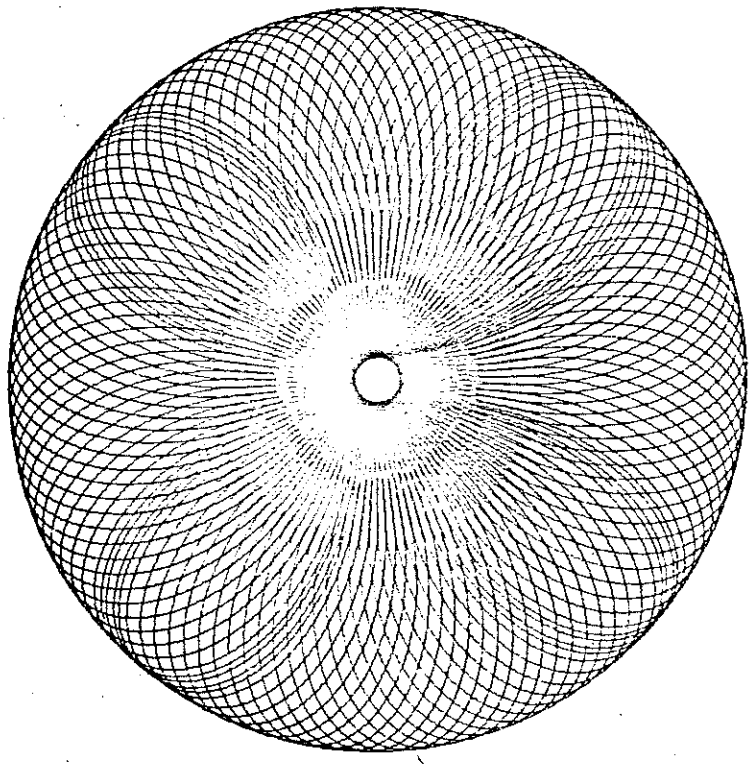
by

Gerhard Fischer

TECHNICAL REPORT #42 - JUNE 1973

June 1973

MATERIAL AND IDEAS TO TEACH AN INTRODUCTORY PROGRAMMING COURSE USING LOGO



Gerhard Fischer

UC Irvine

Department of Information and Computer Science

PREFACE

These notes represent a first draft for creating a problem book for an introductory LOGO course. It is primarily a collection of non-numerical problems which were written down in some hurry. We hope that by using these notes it will turn out which of the problems and ideas are interesting enough to think about and to work with them in further courses.

We would be pleased if you, the reader, would write down and address your comments, criticism and suggested changes:

Gerhard Fischer
ICS Department
UCI
Irvine, Calif. 92264

TABLE OF CONTENTS

1. INTRODUCTION

The following notes originated from two sources:

A) The course SOCIAL SCIENCE 15(SS 15), taught by John S. Brown with the assistance of Richard Burton, Steve Levin and myself.

B) A project for an ICS 190 Senior Seminar where six students and myself (as a teaching assistant for this class) attempted to gain insight into problems which arise from teaching High School students programming.

Both activities took place during Spring Quarter 1973 at UC Irvine, and they were based on previous experience in teaching SS 15, which is documented in [1]. These notes can be regarded as an extension of [1], and we assume that the reader is familiar with this document. There he can find some insight into what our goals have been and why we have chosen LOGO as our programming language (the reader, who is not familiar with LOGO will find a brief introduction in [1]).

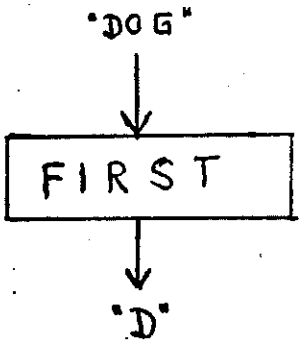
2.PROBLEMS

This section may be regarded as an initial step toward the creation of a problem book for an introductory LOGO course and will point out a few strategies which may be used to introduce new concepts.

2.1 PROBLEMS TO GET STARTED WITH LOGO

2.1.1 THE BOX THEORY

LOGO deals with words and sentences, so it was quite natural to start off with the operations which construct new items (WORD and SENTENCE) and those which get parts of existing items (FIRST,LAST,BUTFIRST and BUTLAST). We introduced the operations with the "box theory". For example, FIRST could be represented as follows (with a sample input):



This approach has many advantages which cannot be overemphasized, since it will make the student familiar with the following ideas:

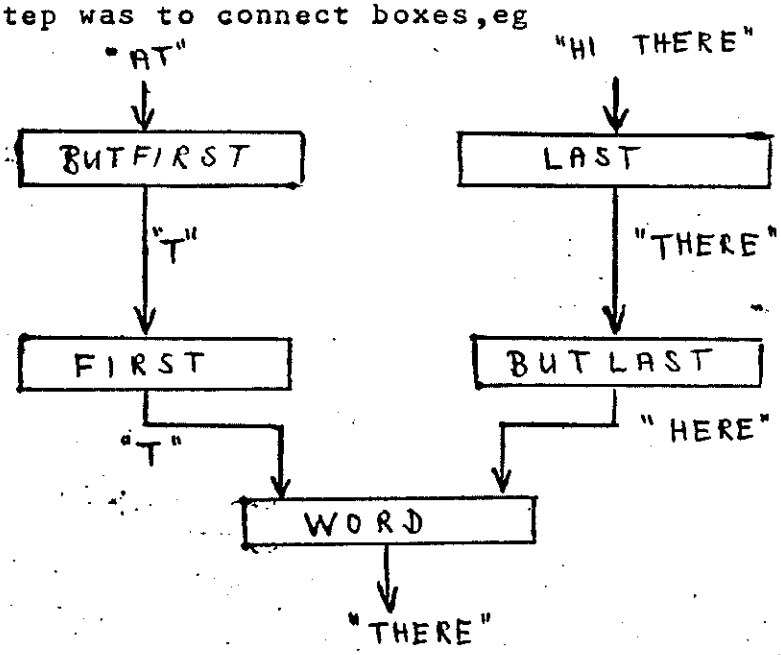
A) that a certain procedure (or operation) has a certain number of inputs and outputs (and this number stays the same for a box at all times)

B) it helps later to show how we can build big boxes from little ones

C) it clarifies that after having constructed a box, this box is available to the student for his further work.

With respect to A), we noticed that even after a few weeks it wasn't quite obvious to some students that this way of thinking (that you represent your procedures as boxes) was very helpful to construct new procedures (see later remarks on STOP and OUTPUT).

The next step was to connect boxes, eg



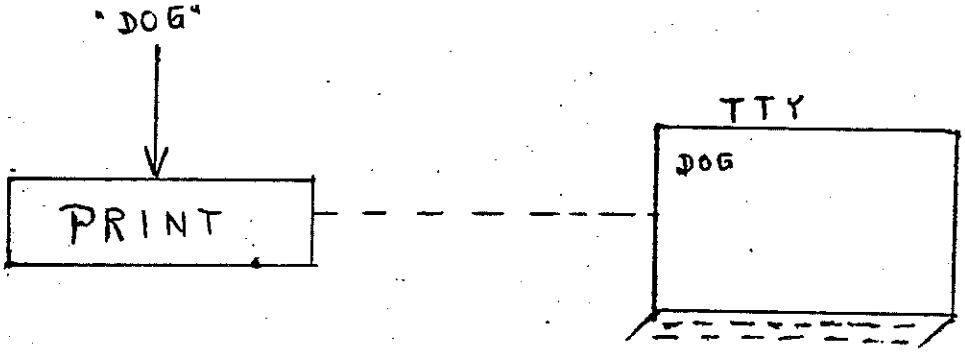
Some effort should be directed toward showing the student how he can transform this two-dimensional representation into a one-dimensional LOGO line, and how he can parse a more complex line such as

WORD FIRST BUTFIRST "AT" BUTLAST LAST "HI THERE"

to get back to the two-dimensional representation. We explained it roughly in the following way:

1) start to scan the line from the left
2) first element is WORD- requires two inputs - go on to find these two inputs
3) second element is FIRST - requires one input
4) .....and so on

The parsing of the LOGO line shows the importance of A). The next box introduced is the PRINT box



which has no output, but it has a side-effect: it spews out its input to the teletype.

At this time the student should know enough (after we showed him briefly, how to use the operating system to get into LOGO) that he can play with LOGO. Some other LOGO

operations can be explained briefly (eg what a predicate is); most LOGO operation names are well chosen so the student can guess what they are designed to do.

Richard Burton wrote an interesting program with which the students could test whether they had understood how the basic LOGO commands work. Here is a sample run:

```
SS15
HOW MANY EXAMPLES WOULD YOU LIKE ME TO GIVE YOU?
♦3

WHAT IS -- SENTENCE OF "LIFE IS JUST" AND "GLORK" -- ?

♦I DON'T KNOW
THAT'S NOT QUITE RIGHT. WATCH WHILE I DO IT.

'SENTENCE OF "LIFE IS JUST" AND "GLORK"
'SENTENCE OUTPUTS "LIFE IS JUST GLORK"

MY ANSWER IS -- LIFE IS JUST GLORK


WHAT IS -- BUTLAST OF "IYT THE KA" -- ?

♦IYT THE
VERY GOOD

WHAT IS -- LAST OF SENTENCE OF "MEKUZ" AND "AMONG FRIENDS" -- ?

♦DO YOU KNOW
THAT'S NOT QUITE RIGHT. WATCH WHILE I DO IT.

'SENTENCE OF "MEKUZ" AND "AMONG FRIENDS"
'SENTENCE OUTPUTS "MEKUZ AMONG FRIENDS"
'LAST OF "MEKUZ AMONG FRIENDS"
'LAST OUTPUTS "FRIENDS"

MY ANSWER IS -- FRIENDS
```

We also gave the first assignment:

A) What are the results? (If you don't know try it on the machine!)

```
PRINT LAST OF LAST OF LAST OF "THE CAT ATE THE MOUSE"
PRINT FIRST OF LAST OF LAST OF "THE CAT ATE THE MOUSE"
PRINT SUM OF 2 BUTLAST "1A"
PRINT SENTENCEP OF BUTLAST "THE CAT"
PRINT SENTENCEP BUTFIRST BUTFIRST "THE CAT"
PRINT SUM OF 12 COUNT 345
PRINT IS COUNT OF "THE BIG CAT" AND COUNT BUTFIRST OF "THE BIG CAT"
PRINT WORD SENTENCEP "THE MOUSE" "Y"
PRINT WORD FIRST BUTLAST LAST "THE CAT ATE" "FOO"
PRINT WORD 1 BUTLAST "A"
PRINT WORD LAST BUTLAST "THE CAT" "ATE"
PRINT BUTFIRST DIFFERENCE 2 BUTFIRST "X14"
PRINT SENTENCEP OF BUTFIRST OF "HI THERE"
PRINT WORDP OF BUTFIRST OF "HI THERE"
PRINT SENTENCEP OF LAST "HI THERE"
PRINT WORDP OF LAST "HI THERE"
PRINT IS LAST "HI THERE" BUTFIRST "HI THERE"
PRINT SUM OF FIRST 223 LAST 123
PRINT SUM OF WORD "-" 4 AND 4
PRINT COUNT OF "-123"
PRINT SUM OF 1 0
PRINT IS "-4"WORD "-" 4
PRINT LAST OF "A"
PRINT BUTLAST OF "A"
PRINT FIRST OF "A"
PRINT BUTFIRST OF "A"
PRINT EMPTYP OF BUTLAST OF "A"
PRINT IS LAST "A" FIRST "A"
PRINT IS EMPTYP "AA" "FALSE"
PRINT IS 04 4
PRINT WORD OF "A" WORD OF "B" "C"
PRINT WORD OF WORD "A" "B" "C"
PRINT IS ( WORD "A" WORD "B" "C" ) ( WORD WORD "A" "B" "C" )
PRINT IS FIRST "B" LAST "B"
PRINT IS BUTFIRST "B" BUTLAST "B"
PRINT SUM OF WORD 1 4 5
PRINT IS 0 00
PRINT ZEROP 0
PRINT ZEROP 00
PRINT COUNT WORD "" "ABC"
PRINT IS ( BUTFIRST BUTLAST "ABC" ) BUTLAST BUTFIRST "ABC"
```

This didn't work out too well for the following reasons:

a) nobody tried to figure out the results in advance; they just typed in the lines and watched what was printed

b) some of the lines introduced questions which they couldn't understand at this point in the course, eg

```
→P SENTENCEP BUTFIRST "HI THERE"
TRUE
```

whereas

```
→P SENTENCEP "THERE"
FALSE
```

This led us into endless discussions which were too advanced for this part of the course and which were confusing to quite a few students.

2.1.2 NON-RECURSIVE PROCEDURES

Next we introduced procedures or in other words showed them "how to construct their own boxes." The first few examples

were non-recursive, which limits the procedures to a small class, but it still helps to acquaint the student with a few important concepts (eg how to give a certain number of instructions a name so we can refer to them later by this name).

Here is a list of non-recursive procedures (the code for the trivial procedures is left out):

1) write a procedure BUTFIRST2 (BUTLAST2) which takes one input (word or sentence) and outputs the input without the first (last) two elements

```
→ P BUTFIRST2 "HELLO"
  LLO
```

2) write a procedure SECONDFIRST, SECONDLAST, THIRDFIRST....., which do the obvious thing (ie they output the second, secondlast, third .....element)

```
→ P THIRDLAST "DOWNTOWN"
  O
```

3) write a procedure GLUEIT, which takes two inputs (words) and glues them together

→ P GLUEIT "HE" "RE"

HERE

4) write a procedure TENSE, which takes two inputs. The first input is either the literal "PAST" or "PRES"; the second is an English verb

If the first input is "PAST", TENSE should remove "ED" from the verb, if the first input is "PRES", it should tag on a "S"

→ P TENSE "PAST" "HAPPENED"

HAPPEN

5) write a procedure ADD1 (SUB1), which takes one input, a number and outputs the sum (difference) of the number and 1.

→ P SUB1 14

13

6) write a predicate LONGERP, which takes two inputs (words or sentences) and outputs true, if the first input has more elements than the second

```
→P LONGERP "HOW ARE YOU" "DOWNTOWN BOSTON"
  TRUE
```

This should be an example of what we mean when we talk about elements with respect to a word or sentence.

7) Write a predicate NEGATIVP, which takes one input (a number) and outputs true, if the number is negative.

```
→P NEGATIVP "-4"
  TRUE

  TO NEG1P :NUM:
  10 TEST IS FIRST OF :NUM: "-"
  20 IFTRUE OUTPUT "TRUE"
  30 IFFALSE OUTPUT "FALSE"
  END

  TO NEG2P :NUM:
  10 TEST IS FIRST OF :NUM: "-"
  20 IFTRUE OUTPUT "TRUE"
  30 OUTPUT "FALSE"
  END
```

```
TO NEG3P :NUM:
10 OUTPUT IS FIRST OF :NUM: "-"
END

TO NEG4P :NUM:
10 PRINT IS FIRST OF :NUM: "-"
END
```

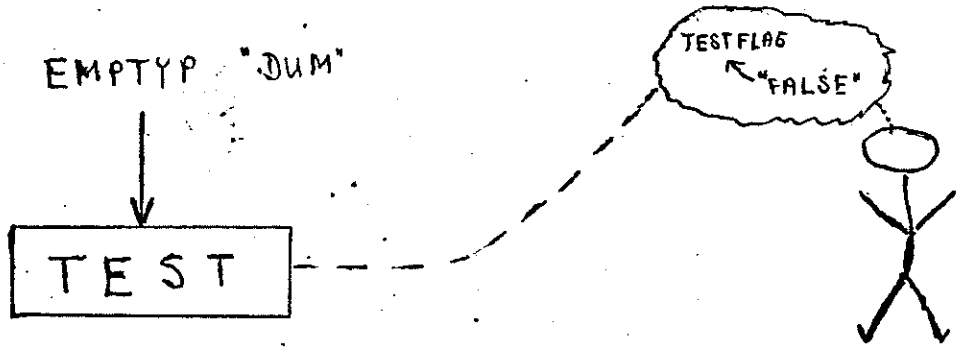
8) write a procedure ABS, which takes one input (a number N) and outputs N, if N>0, otherwise -N

```
→P ABS "-4"
4

TO ABS :NUM:
10 TEST NEG1P :NUM:
20 IFTRUE OUTPUT BUTFIRST OF :NUM:
30 OUTPUT :NUM:
END
```

The first three definitions for the NEGATIVP predicate (which are obviously all equivalent) may be used to introduce new elements of LOGO. The best way to start off is with definition 1, where we can explain the use of the

TEST command (similar to the explanation of recursion in [1]): TEST is a box with one input and no output



But it has the effect of setting a flag in the conceptual cloud of a little man according to the value of its input. The IFTRUE or IFFALSE boxes then look up this flag to determine whether the rest of the line where they appear should be executed.

OUTPUT is another box with one input and no output; its effect is to terminate the procedure and its input determines what comes out of the procedure box in which the OUTPUT box is embedded.

With this explanation it seems most natural to use definition 1 for NEGP. Then def 2 and def 3 are natural simplifications, if the students understand exactly what OUTPUT is doing.

Def 4 (which isn't a predicate any more, because predicates were defined as procedures which output either "TRUE" or "FALSE") should not be presented to the students at this

time. Unfortunately some of the students came up with solutions of this kind. The best answer at this moment is probably that the solution is correct, but it is not what we want because NEG4P can not be used instead of NEG1P in the procedure ABS.

9) Write a procedure EQP, which determines whether two numbers are equal.

```
TO EQP :F: :S:
10 OUTPUT IS (DIFFERENCE :F: :S:) 0
END
```

There are obvious cases where we want to use EQP instead of IS, eg

```
→P EQP 3 03
  TRUE
```

whereas

```
→P IS 3 03
  FALSE
```

10) write a procedure TWIST, which takes the first and the last element of a word(or sentence) and switches them

```
TO TWIST :W:
10 OUTPUT WORD OF LAST :W: WORD OF BUTFIRST BUTLAST :W:
              FIRST :W:
END
```

This procedure is an example where a student has to check whether his procedures work for all cases

```
→P TWIST "A"
AA
```

Most of these procedures seem to be trivial, but we should not forget that we had students who had used the computer just a little more than a week. In spite of this, it seems to be essential that the important features are explained well, without overemphasizing aspects which are too formal (in the sense that the work of the students didn't provide any motivation for them).

A big problem to a large number of students was the use of parameters in the declaration of a procedure. They had a hard time understanding the difference between "A" and :A:

(because we hadn't talked about names at this moment). The following was a big surprise for most students:

11) write a procedure BEGINP, which takes two inputs, a letter and a sentence, and checks whether the letter is the same as the first letter of the sentence.

```
TO BEGINP :L: :SENT:
10 OUTPUT IS :L: FIRST OF FIRST OF :SENT:
END
```

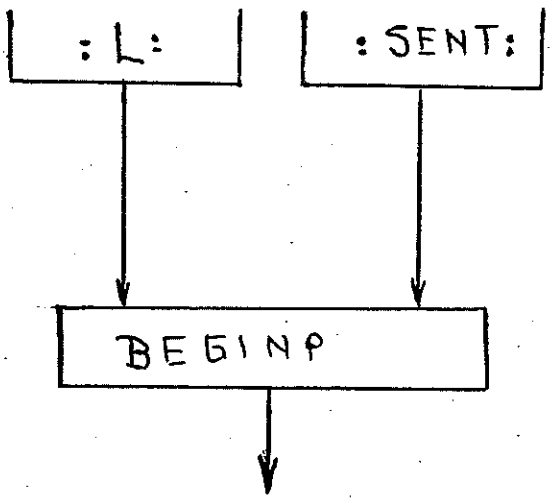
Some students used it in the following way:

```
→P BEGINP :H: :HI THERE:
 TRUE

→P BEGINP :H: :YOU DUMMY:
 TRUE
```

The second example returns true because both inputs by being specified as names have as their "thing" the empty word. At this point, we didn't want to talk about the MAKE command (see [1] for some arguments), so we tried to explain to the

students that they had to enclose the inputs to a procedure with quotes when they are executing a procedure whereas in define mode, they had to enclose the parameters in colons. The meaning of the parameters seems to be best explained as follows (to distinguish them right at the beginning from the concept of having names and things): in the box theory, we tag on a bucket to every arc leading in the box which we call a placeholder.



When we define a procedure, it is used to hold a place for the actual argument, which is filled in at the time that the procedure is invoked.

A short remark about the concept of numbers in LOGO: they can be introduced as a subset of LOGO words. For

simplicity, it is not necessary to surround positive numbers by quotes but to avoid confusion it is best to teach the students to do so. As an example of this, our EQP procedure (no 9) works fine in the case of positive numbers

```
→P EQP 3 3
   TRUE
```

but we are in trouble in the following case

```
→P EQP 3 -8
   THERE ARE 1 INPUTS MISSING FOR EQP.
```

And the students are faced with a very strange error message (which is due to the fact that our LOGO system has infix operators).
By always enclosing numbers in quotes this problem is completely avoided.

```
→P EQP "3" "-4"
   FALSE
```

Another word about the EQP procedure: a student came up with the following solution:

```
TO EQ1P :NUM:
10 TEST IS DIFFERENCE :FIRST: :SECOND: 0
20 IFTRUE OUTPUT "EQUAL"
30 IFFALSE OUTPUT "UNEQUAL"
END
```

To the beginner this appears to be certainly a better way to write the procedure (because it returns more information). With an example like the following (which determines, whether the remainder of the first input divided by the second input is equal to the third input) we show the students why the above version of NEG4P is not desirable.

```
TO REMP :F: :S: :T:
10 TEST EQ1P (REMAINDER :F: :S:) :T:
20 IFTRUE OUTPUT "TRUE"
30 OUTPUT "FALSE"
END
```

```
→P REMP 6 4 2
   TEST OF "EQUAL"
   INPUT MUST BE PREDICATE.
   I WAS AT LINE 10 IN REMP
```

We have shown that we would like to use EQP as a predicate (ie the input has either to be "TRUE" or "FALSE", because we want to use it as an input to the TEST box).

2.2 RECURSIVE PROCEDURES

Before introducing the student to recursion, it may be appropriate to talk about the editor, the error messages, and how to file his own procedures away so they can be used later. The student should be aware that he can extend the basic set of LOGO operations with his own procedures.

Recursion is a powerful concept which not only represents a certain way of programming but a way of thinking about a problem. Recursion cannot be represented in an appropriate way within the box theory, therefore we switched to the "little brother theory", where every little brother has its own conceptual cloud (we mentioned this idea briefly in connection with the TEST command). The explanation of how we introduced the first recursive procedure, MEMBERP, is described in great detail in [1],p7-p11, which the reader may refer to . A slightly modified reprint (which we handed out to our students) appears in the appendix.

The MEMBERP predicate seemed to all of us the best example to introduce recursion for the following reasons:

1) it is a predicate(ie it outputs)
2) it is the simplest form of recursion in the sense that if you come out of your recursion you just pass

the information back

3) it is a more natural problem than the FACTORIAL function, which is not a well known function to social science or high school students (and which fails in its usual version with respect to 2))

Another important feature of LOGO at this state is the trace feature. It helps the student to see exactly what is going on (eg, how many little brothers were generated) and it usually shows him what he is doing wrong when he writes the first few procedures on his own(eg if his process goes into an infinite recursion).

```
←P MEMBERP "A" "HAT"
MEMBERP OF "A" AND "HAT"
  MEMBERP OF "A" AND "AT"
  MEMBERP OUTPUTS "TRUE"
MEMBERP OUTPUTS "TRUE"
TRUE
←P MEMBERP "A" "WE"
MEMBERP OF "A" AND "WE"
  MEMBERP OF "A" AND "E"
    MEMBERP OF "A" AND ""
    MEMBERP OUTPUTS "FALSE"
  MEMBERP OUTPUTS "FALSE"
MEMBERP OUTPUTS "FALSE"
FALSE
←P MEMBERP "LIKE" "DO YOU LIKE TO WORK?"
MEMBERP OF "LIKE" AND "DO YOU LIKE TO WORK?"
  MEMBERP OF "LIKE" AND "YOU LIKE TO WORK?"
    MEMBERP OF "LIKE" AND "LIKE TO WORK?"
    MEMBERP OUTPUTS "TRUE"
  MEMBERP OUTPUTS "TRUE"
MEMBERP OUTPUTS "TRUE"
TRUE
```

Discussions which usually arise at this point are centered around the question of how to avoid a circular definition, During the process of defining a procedure, we are using the same procedure (before we have finished the definition). It may help some students if we show them how this process is related to the look-up procedure of a word in a dictionary. If we want to look up a word, we have to look up the first unknown word in its definition, before we can understand the whole definition. If we don't hit a trivial case (ie a case where we know the answer, which is the empty word in our MEMBERP predicate and some atomic information or words in the look-up procedure) then there is no way to come to an end.

With recursion the usual fact is true: you can't learn it by seeing how other people do it, you have <u>to do it yourself!</u> Therefore we gave the students a large number of different recursive procedures and encouraged them to write them.

They should be embedded in a meaningful task so that the students don't get the impression that they are only solving toy problems. The next section gives a set of projects where most of the recursive procedures below can be embedded.

1) Write a procedure HASNUMBERP which determines whether a word contains a number.

```
TO HASNUMBERP :W:
10 TEST EMPTYP :W:
20 IFTRUE OUTPUT "FALSE"
30 TEST MEMBERP FIRST OF :W: "0123456789"
40 IFTRUE OUTPUT "TRUE"
50 OUTPUT HASNUMBERP BUTFIRST :W:
END

→P HASNUMBERP "DOG3CAT"
TRUE
```

2) write a procedure VOWELP which determines whether a letter is a vowel

```
TO VOWELP :W:
10 OUTPUT MEMBERP :W: "AEIOU"
END
```

3) write a procedure CONSONANTP which determines whether a letter is a consonant

```
TO CONSONANTP :W:
10 OUTPUT NOT VOWELP :W:
END
```

All of the preceeding procedures obviously can be written by using our previously defined procedures. Not too many students understood this idea and some of them worked hard to achieve what could be done obviously in a much easier way (eg they didn't see how MEMBERP could be used for VOWELP because the two procedures have a different number of inputs; some others gave a list of all the consonants to CONSONANTP).

4) Write a procedure REVERSE, which reverses a word

```
TO REVERSE :W:
10 TEST EMPTYP :W:
20 IFTRUE OUTPUT ""
30 OUTPUT WORD LAST :W: REVERSE BUTLAST :W:
END

→ P REVERSE "UCI"
ICU
```

This is a more complicated example of a recursive procedure (eg it can't be replaced too easily by an iterative procedure), because it performs an operation after it comes out of the recursion.

5) Write a procedure ALLCOUNT which counts all the symbols in a sentence

```
TO ALLCOUNT :SENT:
10 TEST EMPTYP :SENT:
20 IFTRUE OUTPUT 0
30 OUTPUT SUM COUNT FIRST OF :SENT: AND
              ALLCOUNT BUTFIRST :SENT:
END

→P ALLCOUNT "HOW ARE YOU"
9
```

6) write a procedure PACK which takes a sentence and deletes all the blanks in it.

```
TO PACK :STR:
10 TEST EMPTYP :STR:
20 IFTRUE OUTPUT ""
30 OUTPUT WORD OF (FIRST OF :STR:) AND
            ( PACK OF BUTFIRST OF :STR:)
END
```

```
→P PACK "HOW ARE YOU"
  HOWAREYOU
```

7) write a procedure NTH which takes two inputs, a number and a word(or sentence) and gives the N-th element (counted from the end, if N 0) of the word(or sentence). This procedure is an obvious generalization of SECONDLAST, THIRDFIRST.....

```
TO NTH :NUM: :STR:
10 TEST EMPTYP :STR:
20 IFTRUE OUTPUT ""
30 TEST NEG1P :NUM:
40 IFTRUE OUTPUT NTH (ABS :NUM:)(REVERSE :STR:)
50 TEST EQP :NUM: 1
60 IFTRUE OUTPUT FIRST OF :STR:
70 OUTPUT NTH (SUB1 :NUM:)(BUTFIRST OF :STR:)
END

→P NTH -3 "GERD"
  E
```

8) write a procedure DELETE, which takes two inputs, and deletes all occurrences of the first input from the

```
second


TO DELETE :EL: :STR:
10 TEST EMPTYP :STR:
20 IFTRUE OUTPUT ""
30 TEST IS :EL: (FIRST OF :STR:)
40 IFTRUE OUTPUT DELETE OF :EL:
              AND (BUTFIRST OF :STR:)
50 OUTPUT SENTENCE (FIRST :STR:)
              (DELETE :EL: BUTFIRST :STR:)
END

→P DELETE "PRETTY" "PRETTY GIRLS ARE NOT PRETTY BOYS"
GIRLS ARE NOT BOYS
```

9) write a procedure REPLACE, which takes three inputs and replaces the occurrences of the first input by the second input within the third input

```
TO REPLACE :W1: :W2: :TEXT:
10 TEST EMPTYP :TEXT:
20 IFTRUE OUTPUT ""
30 TEST IS FIRST :TEXT: :W1:
40 IFTRUE OUTPUT SENTENCE :W2: REPLACE :W1: :W2:
```

```
(BUTFIRST OF :TEXT:)
50 OUTPUT SENTENCE FIRST :TEXT: REPLACE :W1: :W2:
BUTFIRST OF :TEXT:
END

→P REPLACE "UGLY" "PRETTY" "BOYS ARE UGLY"
BOYS ARE PRETTY
```

DELETE is now a special case of REPLACE, with the empty word as second input.

At this point, we may introduce the STOP command, because some procedures can be written more naturally if they are executed for a side-effect, rather than returning a value.

10) Write a procedure RECTANGLE, which takes two inputs, the number of rows and columns, and prints a rectangle of stars.

```
TO RECTANGLE :ROW: :COL:
10 TEST EMPTYP :ROW:
20 IFTRUE STOP
30 PRINT STARS :COL:
40 RECTANGLE (SUB1 :ROW:) :COL:
END
```

Before we can use RECTANGLE, we have to define STARS, which prints one row.

```
TO STARS :COL:
10 TEST EMPTYP :COL:
20 IFTRUE OUTPUT ""
30 OUTPUT WORD "*" STARS (SUB1 :COL:)

→ RECTANGLE 3 4
****
****
****
```

It may be interesting to make the students aware of the recursive structure of the arithmetic functions by showing them that we need only SUB1 and ADD1 to define addition, multiplication and exponentiation.

11) Write procedures MYADD, MYPROD, EXP, by using only ADD1 and SUB1

```
TO MYADD :X: :Y:
10 TEST ZEROP :Y:
20 IFTRUE OUTPUT :X:
```

```
30 OUTPUT MYADD (ADD1 :X:) (SUB1 :Y:)
END

→P MYADD 12 43
55

TO MYPRODUCT :X: :Y:
10 TEST IS :Y: 1
20 IFTRUE OUTPUT :X:
30 OUTPUT MYADD :X: (MYPRODUCT :X: SUB1 :Y:)
END

→P MYPRODUCT 21 5
105

TO EXP :A: :N:
10 TEST IS :N: 1
20 IFTRUE OUTPUT :A:
30 OUTPUT MYPRODUCT:A: (EXP :A: SUB1 :N:)
END

→P EXP 3 5
243
```

Some students liked to generate random numbers and studied the patterns which appeared. This lead into a discussion of how "random" these numbers really were and how they were generated.

12) Write procedures which generate a random number with a random number of digits between 0 and 99.

```
TO GEN :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
25 GEN SUB1 :N:
30 PRINT SENTENCES "THE" WORD :N: "." "NUMBER IS ---" SUPERRANDOM
END

TO SUPERRANDOM
10 OUTPUT BIGRANDOM WORD RANDOM RANDOM
END

TO BIGRANDOM :N:
10 TEST ZEROP :N:
20 IFTRUE OUTPUT RANDOM
30 OUTPUT WORD OF RANDOM AND BIGRANDOM DIFFERENCE :N: 1
END

←GEN 3
THE 1. NUMBER IS --- 635135479997397
THE 2. NUMBER IS --- 5829540222071369791
THE 3. NUMBER IS --- 897064174865139103 45

GEN 3
THE 1. NUMBER IS --- 690410013381416350698960445049164447783889212012
267715126620712756116454822764807117122869708099 6
THE 2. NUMBER IS --- 5794516507691266817414510965547251170868576985
THE 3. NUMBER IS --- 49399985515300607 76
```

13) A more difficult example of a recursive procedure is:

Write procedures which switch two words in a sentence

```
TO SWITCH :SEN: :F: :L:
10 TEST GREATERP :L: :F:
20 IFTRUE OUTPUT SWITCH1 :SEN: :F: :L:
30 OUTPUT SWITCH1 :SEN: :L: :F:
END

TO SWITCH1 :SEN: :LOW: :HIGH:
10 TEST IS :LOW: 1
20 IFFALSE OUTPUT SENTENCE FIRST :SEN: SWITCH1 BUTFIRST :SEN: (
   DIFFERENCE :LOW: 1 ) ( DIFFERENCE :HIGH: 1 )
30 OUTPUT SENTENCE SWITCH2 :SEN: :HIGH: BUTFIRST SWITCH3 :SEN: :HIGH:
   FIRST :SEN:
END

TO SWITCH2 :SEN: :HIGH:
10 TEST IS :HIGH: 1
20 IFTRUE OUTPUT FIRST :SEN:
30 OUTPUT SWITCH2 BUTFIRST :SEN: ( DIFFERENCE :HIGH: 1 )
END

TO SWITCH3 :SEN: :HIGH: :HOLD:
10 TEST IS :HIGH: 1
20 IFTRUE OUTPUT SENTENCE :HOLD: BUTFIRST :SEN:
30 OUTPUT SENTENCE FIRST :SEN: SWITCH3 ( BUTFIRST :SEN: ) ( DIFFERENCE
   :HIGH: 1 ) :HOLD:
END


←P SWITCH "A MOUSE IS BIGGER THAN AN ELEPHANT" 2 7
A ELEPHANT IS BIGGER THAN AN MOUSE
```

As mentioned before, these procedures should be embedded in more interesting problems so the students don't become bored by writing a large number of procedures for which they can see no use.

A good strategy may also be to give the students a large number of problems and let them choose those which they think are most interesting.

3. PROJECTS

This section describes a number of projects(we call a problem a project if it requires more than one or two simple procedures to solve it). To come up with good and interesting projects was one of the main tasks with respect to what we wanted to achieve in the two classes. Our goal was not programming per se but to make our students familiar with concepts like "world view" (or "conceptual cloud") and "representation of knowledge" and to teach them strategies in problem solving which had some degree of transferability. We tried to teach them debugging, handling of error messages and recognizing the structure of programs (which they didn't write themselve) as much as constructing a program.
In our opinion the following points are characteristic of a "good" project:

1) the problem should be interesting (ie a certain programming task should be embedded in some "relevant" problem) so that the students can see some reasons to solve the problem

2) it should be possible to start with a simple version of the problem. After the students have achieved

3. PROJECTS

This section describes a number of projects(we call a problem a project if it requires more than one or two simple procedures to solve it). To come up with good and interesting projects was one of the main tasks with respect to what we wanted to achieve in the two classes. Our goal was not programming per se but to make our students familiar with concepts like "world view" (or "conceptual cloud") and "representation of knowledge" and to teach them strategies in problem solving which had some degree of transferability. We tried to teach them debugging, handling of error messages and recognizing the structure of programs (which they didn't write themselve) as much as constructing a program.
In our opinion the following points are characteristic of a "good" project:

1) the problem should be interesting (ie a certain programming task should be embedded in some "relevant" problem) so that the students can see some reasons to solve the problem

2) it should be possible to start with a simple version of the problem. After the students have achieved

the first part, there should be natural ways to extend the problem in several directions (this feature would emphasize a functional programming style; the final problem has to have a structure which allows us to decompose it in subproblems)

3) the students should see some results already at an early stage of the project

The rest of the section describes projects which we found interesting and we hope that some of them fit in the structure of introductory LOGO courses.

3.1 VOWELS AND CONSONANTS

The basic part of the problem was already mentioned in [1]. We asked the student to write a number of procedures to remove all vowels from a sentence. Procedure REMOVEVOWEL (which removes the vowels from a word) can be written very similar to DELETE (somebody may have the idea to call DELETE five times with all the vowels).

```
←LIST REMOVEVOWEL

TO REMOVEVOWEL :W:
10 TEST EMPTYP :W:
20 IFTRUE OUTPUT :W:
30 TEST VOWELP FIRST OF :W:
40 IFTRUE OUTPUT REMOVEVOWEL BUTFIRST OF :W:
50 OUTPUT WORD OF ( FIRST OF :W: ) AND ( REMOVEVOWEL BUTFIRST OF :W: )
END
```

The next procedure would be SCAN, which takes as input a sentence and hands words over to REMOVEVOWEL.

```
TO SCAN :SENT:
10 TEST EMPTYP :SENT:
20 IFTRUE OUTPUT ""
50 OUTPUT SENTENCE OF ( REMOVEVOWEL FIRST OF :SENT: ) AND ( SCAN OF
   BUTFIRST OF :SENT: )
END
```

At this point it may be appropriate to recall a problem from our first assignment. We recurse down a given sentence by calling SCAN with BUTFIRST :SEN:. When we reach the level of recursion, where :SEN: consists of a single word, we would still like SENTENCEP of :SENT: to be true, otherwise we would get a result like

```
→P SCAN "HI THERE"
   H T H R
```

This example should be an explanation to the students of what seemed to be a contradiction in the first assignment. It is obviously not possible to anticipate all necessary checks on data attributes. Usually a procedure can be fixed fairly easily, if we realize that it doesn't work for certain special cases, eg if we invoke SCAN with a word:

```
→P SCAN "DUMMY"
   D M M Y
```

This problem can be fixed by inserting

```
30 TEST WORDP OF :SENT:
40 IFTRUE OUTPUT REMOVEVOWEL OF :SENT:
```

The problem described in this way may appear to be one of the useless problems which computer scientists invent. To avoid this we could also describe the problem in the following way:

> "A linguist came up with the following conjecture: if we remove all vowels from a sentence which contains ten words or more people are still able to recognize the sentence. Write some LOGO procedures to test whether this conjecture is true!"

The students could show the results to each other and find some evidence for or against the conjecture. A few more questions may be asked:

1) Are there certain words which are difficult to recognize?
   We found that the words which start with a vowel are usually hard to guess.
2) Are the languages English, French and German different with respect to the average number and the information content of their vowels?
3) What happens, if we remove all the consonants from a sentence? Can we still recognize some words?

Here are some examples (the original text is given in the appendix):

```
P SCAN :ENGLISH:
TH NMLS WR SHCKD BYND MSR T LRN THT VN SNWBLL CLD B GLTY F SCH N CTN .
THR WS CRY F NDGNTN , ND VRYN BGN THNKNG T WYS F CTCHNG SNWBLL F H SHLD
VR CM BCK . LMST MMDTLY TH FTPRNTS F PG WR DSCVRD N TH GRSS T LTTL DSTNC
FRM TH KNLL . THY CLD NLY B TRCD FR FW YRDS , BT PPRD T LD T HL N TH HDG
. NPLN SNFFLD DFLY T THM ND PRNNCD THM T B SNWBLL'S . H GV T S HS PNN
THT SNWBLL HD PRBBLY CM FRM TH DRCTN F FXWD FRM .
←
```

```
P SCAN :GERMAN:
BLD SLLT CH JN BLM BSSR KNNNLRNN . S HTT F DM PLNTN DS KLNN PRNZN MMR
SCHN BLMN GGBN , SHR NFCH , S NM NZGN KRNZ VN BLTNBLTTRN GFRMT ; S SPLTN
KN GRSS RLL ND STRTN NMNDN . S LCHTTN NS MRGNS M GRS F ND RLSCHN M BND .
BR JN N HTT NS TGS WRZL GSCHLGN , S NM SMN , WSS GTT WHR , ND DR KLN
PRNZ HTT DSN SPRSS , DR DN NDRN SPRSSLNGN NCHT GLCH , SHR GN BRWCHT . DS
KNNT N N RT FFNBRTBM SN . BR DR STRCH HRT BLD F Z WCHSN ND BGNN , N BLT
NZSTZN .
←
```

```
P SCAN :FRENCH:
C'ST DMNCH : DRRR LS DCKS , L LNG D L MR , PRS D L GR X MRCHNDSS , TT TR
D L VLL L Y DS HNGRS VDS T DS MCHNS MMBLS DNS L NR . DNS TTS LS MSNS ,DS
HMMS S RSNT DRRR LRS FNTRS ; LS NT L TT RNVRS , LS FXNT TNTT LR MRR T
TNTT L CL FRD PR SVR S'L FR B . LS BRDLS VRNT LRS PRMRS CLNTS , DS
CMPGNRDS T DS SLDTS . DNS LS GLSS , L CLRT DS CRGS , N HMM BT D VN DVNT
DS FMMS GNX . DNS TS LS FBRGS , NTR LS MRS NTRMNBLS DS SNS , D LNGS FLS
NRS S SNT MSS N MRCH , LLS VNCNT LNTMNT SR L CNTR D L VLL .
←
```

To count the vowels is a trivial problem and can be done as follows:

```
TO VOWELCOUNT :SENT:
10 OUTPUT DIFFERENCE ALLCOUNT :SENT:
                ALLCOUNT (SCAN :SENT:)
END
```

Assume we did not have all of the above routines available, we could write a routine directly:

```
TO COUNTVOWEL :SENT:
10 TEST EMPTYP :SENT:
20 IFTRUE OUTPUT "0 0"
30 TEST SENTENCEP :SENT:
40 IFTRUE OUTPUT ADD2 COUNTVOWEL FIRST :SENT: COUNTVOWEL BUTFIRST :SENT:
50 TEST MEMBERP FIRST :SENT: "AEIOU"
60 IFTRUE OUTPUT ADD2 "1 0" AND COUNTVOWEL BUTFIRST :SENT:
63 TEST MEMBERP FIRST :SENT: "!',;:()?"
65 IFTRUE OUTPUT ADD2 "0 0" AND COUNTVOWEL BUTFIRST :SENT:
70 OUTPUT ADD2 "0 1" AND COUNTVOWEL BUTFIRST :SENT:
END

TO ADD2 :X: :Y:
10 OUTPUT SENTENCE SUM FIRST :X: FIRST :Y: SUM LAST :X: LAST :Y:
END
```

If we add a small statistics program, we would get the results below for our three text examples. It was

interesting to find out that in all three text examples (which were chosen arbitrarily from books), the number of vowels to the number of consonants has the ratio 2:3. The word length in German turns out to be greater than in French and English.

```
STATISTICS :FRENCH:
HERE ARE SOME DATA ABOUT OUR TEXT:
THERE WERE :
--- 573 ---SIGNS
--- 222 ---VOWELS
--- 331 ---CONSONANTS
--- 20 ---SPECIAL SIGNS
THERE WERE --- 120 ---WORDS IN THIS TEXT
←STATISTICS :ENGLISH:
HERE ARE SOME DATA ABOUT OUR TEXT:
THERE WERE :
--- 479 ---SIGNS
--- 178 ---VOWELS
--- 292 ---CONSONANTS
--- 9 ---SPECIAL SIGNS
THERE WERE --- 104 ---WORDS IN THIS TEXT
←STATISTICS :GERMAN:
HERE ARE SOME DATA ABOUT OUR TEXT:
THERE WERE :
--- 530 ---SIGNS
--- 203 ---VOWELS
--- 312 ---CONSONANTS
--- 15 ---SPECIAL SIGNS
THERE WERE --- 97 ---WORDS IN THIS TEXT
←
```

3.2 THE WIZARD'S PROBLEMS

This problem is an example of the question of whether a computer can be used to settle a mathematical conjecture (it is similar to the conjecture about symmetric numbers in [1], except for the wizard's conjecture we have a mathematical proof).

When we gave the students in SS 15 this problem they had a difficult time understanding it. Therefore we wrote a handout for them which describes the problem and its solution by embedding it in more general ideas about problem solving. The handout appears in the appendix.

3.3 A QUESTION ANSWERER SYSTEM

The basic problem to start off with would be to ask the students to write a program which can do the following:

1) we can give some information to the system like

"X is/are Y"

eg

"Sophie is dog"

"Students are humans"

2) we can ask questions like

"What is/are X"

eg

"What is Sophie?"

3) we can delete information

"Forget x"

eg

"Forget Sophie"

In connection with this problem, we introduced the MAKE command, and the THING and REQUEST operation. We could also mention the standard LOGO conventions to distinguish different classes of boxes:

1) an operation is a built-in procedure which takes a

fixed number of things (possibly none) as inputs, and produces a new thing as an output (eg FIRST, WORD, REQUEST)

2) a command is a built-in procedure which has inputs but no output (eg PRINT, MAKE)

For the most basic version of this program, we could store the information "dogs are animals" in the form:

MAKE "DOGS" "ANIMALS"

Even in this most basic version quite a few students found it difficult to understand what they should do. Some of the problems were:

1) to understand the MAKE command and the thing operation; eg, if we would have the following procedure to add information to our data base:

```
TO ADDINFO :NA: :TH:
10 MAKE :NA: :TH:
END
```

and a question routine like

```
TO QUESTION :NA:
10 OUTPUT THING OF :NA:
END
```

The confusion was based on the fact that LOGO uses colons for "placeholders" (see page [16] and Appendix) and for names. Most students didn't see that the colons around a string have the same function as the THING operation, ie line 20 in ADDINFO should be read: make the entity which is denoted by the placeholder :NA: the name of the entity which is denoted by the placeholder :TH:.

Another difficult idea seemed to be that the thing represented by a name could be itself the name of another thing, eg

"freshmen are students"

"students are humans"

would result in that "student" is the thing of "freshmen" and the name of "humans".

2) How to use the REQUEST operation to keep on requesting information from the teletype, but also having the ability to stop.

The top level routine of this problem could be written as follows:

```
TO TALK
10 PRINT "PLEASE TELL ME YOUR NEXT SENTENCE"
20 TEST SPLITUPP REQUEST
30 IFFALSE STOP
40 TALK
END
```

and SPLITUPP would be the routine which classifies the information typed in from the user at the teletype and passes back the value false if the user wants to stop. SPLITUPP can be written in many different ways according to the formats of the input. A slightly more complex Q/A system would allow inputs like "everything" and "forgetall" and simple LOGO procedure would handle this:

```
TO EVERYTHING
10 LIST ALL NAMES
END

TO FORGETALL
ERASE ALL NAMES
END
```

A predicate could determine whether we have a "WH" question (eg who, which, what, why, ...)

```
TO WHP :W:
10 OUTPUT IS (WORD OF FIRST :W: SECOND :W:) "WH"
END
```

Another extension would be that more than one piece of information could be attached to a name, eg that we could assert: "students are humans" and "students are dummies". Some of the students did not see that the thing of a name could be a sentence as well as a word. The required modification shows that it pays off if the students program functionally. We only have to change ADDINFO:

```
TO ADDINFO :NA: :TH:
10 TEST EMPTYP OF THING OF :NA:
20 IFTRUE MAKE :NA: :TH:
30 IFFALSE MAKE :NA: SENTENCE THING :NN: :TH:
END
```

We could also add small routines which would

a) check whether the information is already known

b) handle a more English type of syntax, eg

"A freshman is a student"

c) print comments (eg "I UNDERSTAND")

This little system could handle a dialogue as shown:

```
TALK
PLEASE TELL ME YOUR NEXT SENTENCE
♦SOPHIE IS A DOG
I UNDERSTAND
I ADDED TO THE DATA BASE:------SOPHIE IS DOG


PLEASE TELL ME YOUR NEXT SENTENCE
♦SOPHIE IS BLACK
I UNDERSTAND
I ADDED TO THE DATA BASE:------SOPHIE IS BLACK


PLEASE TELL ME YOUR NEXT SENTENCE
♦INGRID IS A GIRL
I KNOW ALREADY,THAT----INGRID IS GIRL


PLEASE TELL ME YOUR NEXT SENTENCE
♦UCI IS A UNIVERSITY
I UNDERSTAND
I ADDED TO THE DATA BASE:------UCI IS UNIVERSITY


PLEASE TELL ME YOUR NEXT SENTENCE
♦WHO IS SOPHIE
SOPHIE IS----------DOG AND  BLACK

PLEASE TELL ME YOUR NEXT SENTENCE
♦FORGET SOPHIE
I FORGOT--- SOPHIE


PLEASE TELL ME YOUR NEXT SENTENCE
♦WHO IS SOPHIE
SOPHIE IS----------UNKNOWN TO ME.BUT YOU CAN TELL ME SOMETHING



PLEASE TELL ME YOUR NEXT SENTENCE
♦TELL EVERYTHING

:FRESHMAN: IS "STUDENT"
:SOPHIE: IS ""
:INGRID: IS "GIRL"
:GERMANS: IS "BEERTRINKERS"
:NIXON: IS "PRESIDENT"
:UCI: IS "UNIVERSITY"


PLEASE TELL ME YOUR NEXT SENTENCE
♦STOP
```

There are two nontrivial extensions to this problem which can be persued at this point or later, if the students seem to have difficulties with the previous form of the problem. The first one makes use of the DO command: SPLITUPP could be changed in the following way:

```
TO SPLITUPP :INPUT:
10 DO SENTENCES OUTPUT FIRST OF :INPUT:
            BUTFIRST OF :INPUT:
END
```

ie the first word of the input would determine, which routine gets invoked.

The second extension is to provide the Q/A system with some routines, which could deduce the answers to some questions from the data base (an example for this would be [3]). The first thing that we would like to do may be to store subset/superset relations, eg from

"A setter is a dog"

"A dog is an animal"

we probably would like to answer the questions

"is a setter an animal?" with "yes"

by following superset chains and

"Is an animal a setter?" with "sometimes"

by following subset chains. This may be combined with member/element relations, eg

"John is a student"

would result in combining with John the property that he is a member of the set of all students and with the set of all students that it contains John as an element. For a system which handles many more of these extensions (eg adjectives, prepositional phrases, generic subclasses, transitive relations), see [4]. This second extension (which can lead behind the scope of an introductory LOGO course) contains some ideas, which are important and not difficult to understand:

1) with respect to the implementation: if we want to retrieve more information, we have to store more information than in our basic version of the program, ie we have to attach to the name "student" the attribute/value pairs: element/John, superset/human, subset/freshman and so on. A good way to do this is to introduce a property list structure (like in LISP). In LOGO, this can be done by synthesizing new names from

the name and the property. The following two procedures store and retrieve information:

```
TO PUTPROP :NAME: :PROP: :VALUE:
10 MAKE (WORD OF :NAME: (WORD OF "$" :PROP:)) :VALUE:
END

TO GETPROP :NAME: :PROP:
10 OUTPUT THING OF (WORD OF :NAME: (WORD OF "$" :PROP:))
END
```

The "$" sign is inserted to identify these names as property list structure names and we can separate the two parts of it. Defined like this,

```
PUTPROP "STUDENT" "SUBSET" "FRESHMEN"
```

would result in the LOGO name "STUDENT$SUBSET" which would have the thing "freshman". A call

```
GETPROP "STUDENT" "SUBSET"
```

would return the value "freshman". Having these two functions available, we do not have to worry any more about how the structure is implemented.

2) The second idea is to show the students that this representation of knowledge has a lot of advantages. If we later assert some information about "students", we know that these facts also hold for all sets on the subset chain of "students" and for all elements of these sets. In a large system this feature gives us the option of not having to store a large amount of information explicitly (it seems also that this hierarchical structure is not too different from the way humans store information).

3.4 PIGGY AND UNPIGGY

The first part of this problem is to translate english into pig latin by using the procedure PIGGY which contains the following rules:

1) if the word begins with a vowel add NAY to the end of the word.
2) if the word begins with a consonant then take all the consonants off the front of the word up to the first vowel, stick them on the end of the word and add AY to the end.

An example is:

"translate english into piglatin"

becomes

"anslatetray englishnay intonay igpay ationlay"

This is not too difficult to do. The problem becomes more interesting if we add an UNPIGGY procedure which translates pig latin back into English. The problem is that PIGGY is not a one-to-one mapping, so UNPIGGY of PIGGY gives us in general more than the original version

of the English sentence. This brings up a few interesting questions:

1) how can we resolve the ambiguities, eg with help from the user or by generating all possible UNPIGGY translations and comparing it with the original sentence (if given)

2) this is a good point to talk about certain properties of functions (eg when we like to have one-to-one functions, like the function which maps people to driver's license numbers or phone numbers). It can be shown by a more obvious example (eg by mapping the integers into even and odd) that we cannot retrieve x from f(x).

3.5 THE POETRY PROBLEM

Most students liked this problem very much. The basic version is not as demanding as a programming task, but it can be extended in interesting ways. It also has the desirable feature that the students could use their previously defined procedures.

The basic form only requires the following procedures:

1) a procedure POEM which determines the number of sentence to generate

```
TO POEM :NUM:
10 TEST ZEROP :NUM:
20 IFTRUE STOP
30 PRINT SENT
40 POEM (SUB1 :NUM:)
END
```

2) a procedure SENT which generates one sentence

```
TO SENT
10 OUTPUT SENTENCES OF
            RANDOMSELECT :ART:
            RANDOMSELECT :ADJ:
            RANDOMSELECT :NOUN:
        ..............
END
```

3) a procedure RANDOMSELECT which selects at random an element from a list (eg noun, verb, etc)

```
 TO RANDOMSELECT :LIST:
10 OUTPUT NTH (ADD1 RANDOM) :LIST:
END
```

The interesting points up to here are:

a) to design the structure of the program and the idea that we can use NTH in RANDOMSELECT

b) the creation of interesting lists of nouns, verbs etc

We have to pay attention that we call NTH with the appropriate first argument (otherwise we may go into an infinite recursion). In the above version, our lists

had to be of length 10.

The first extension may be to eliminate this restriction; we will talk about this more in the following section. The next step would be to allow a more flexible sentence structure, eg we can determine with another random number how many adjectives (possibly none) and adverbs our sentence should contain. That would allow us to generate sentences like follows:

"The old wise beautiful horse swims slowly"

"a rainbow cries a big cat"

A prepositional phrase could be a preposition+ article+noun or a relative clause would be the same as the original sentence where the subject gets substituted by a pronoun (this may show the students some of the recursive features of English).

Another extension may be to use the DO command. We could write procedures SENT1,SENT2,..... (where SENTn may consist of the concatenation of SENTm and SENTk) which return sentences or part of sentences of different syntactic structures. The only change would be line 30 in POEM:

```
30 PRINT DO SENTENCE "SENT" (ADJUST RANDOM)
```

3.6 RANDOM NUMBERS

This problem came up in connection with the poetry program where we were faced with the following questions:

a) How can we write a procedure which returns a random number within a certain range (eg between 1 and 25)

b) How can we find out how "random" our random numbers really are?

The second question could be embedded in the following story:

"A casino has a roulette table with numbers between 0 and 18. It determines the numbers, which will occur by the following LOGO procedure:

```
TO METHOD1
10 OUTPUT SUM OF RANDOM AND RANDOM
END
```

If you would play at this table which numbers would you bet?"

Obviously we get a normal distribution and not an

uniform one in this case (eg 0=0+0 and 18=9+9 can be generated only in one way, whereas 9 (0+9, 1+8,....) can be generated in many different ways).

The "correct" LOGO procedure would be (in the sense that every number appears with the same probability):

```
TO METHOD2
10 OUTPUT WORD OF RANDOM AND RANDOM
END
```

(that this version works is based on the assumption, that the operation RANDOM in LOGO gives a uniform distribution)

This causes the problem again that we get numbers between 0 and 99, when we want them to be between 0 and 18. The easiest way to solve this problem would be to generate random numbers and throw away all the ones which are greater than 18. It may be worthwhile to write a few LOGO procedures which adjust a random number to a certain range (the lower limit can be taken zero; otherwise we just have to add or to subtract something to the generated numbers). The procedures are based on the following ideas (19 is chosen as a concrete example):

1) determine how many digits the number n (=the upper limit of the range) contains (eg COUNT 19=2)

2) generate a random number, say r, with the same number of digits (method2)

3) take n and compute the highest multiple of it (minus 1) which is smaller than 999...9 (5*19-1=94 99)

4) if r is greater than h, throw it away and generate another random number, otherwise output the remainder of r and n.

```
TO METHOD1
10 OUTPUT SUM OF RANDOM AND RANDOM
END

TO ADJUST :NUM:
10 MAKE "NUMBER" METHOD2 COUNT :NUM:
20 TEST GREATERP :NUMBER: MULTIPLE BIGGEST COUNT :NUM: :NUM:
30 IFFALSE OUTPUT REMAINDER :NUMBER: :NUM:
40 OUTPUT ADJUST :NUM:
END

TO METHOD2 :LEN:
10 TEST ZEROP :LEN:
20 IFTRUE OUTPUT ""
30 OUTPUT WORD RANDOM METHOD2 DIFFERENCE :LEN: 1
END

TO MULTIPLE :UPPER: :NUM:
10 OUTPUT DIFFERENCE ( PRODUCT :NUM: FIRST OF DIVISION :UPPER: :NUM: ) 1
END

TO BIGGEST :LEN:
10 TEST ZEROP :LEN:
20 IFTRUE OUTPUT ""
30 OUTPUT WORD 9 BIGGEST DIFFERENCE :LEN: 1
END
```

By adding a few more procedures, we could show the difference between the two methods (the number of the generated number was too small to have a very good distribution, but the basic difference shows up clearly):

```
←
←TEST1 500
18 --- ♦♦♦♦♦
17 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦
16 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
15 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
14 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦
13 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
12 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
11 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
10 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
9 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
8 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
7 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
6 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
5 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
4 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
3 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
2 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
1 ---- ♦♦♦♦♦♦
0 ---- ♦♦♦♦♦♦♦♦
←
←
←DOIT 500
18 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
17 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
16 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
15 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
14 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
13 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
12 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
11 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
10 --- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
9 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
8 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
7 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
6 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
5 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
4 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
3 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
2 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
1 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
0 ---- ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
←
```

METHOD1

METHOD2

3.7 THE ANIMAL PROGRAM

The sample run below should give an idea, what the program is supposed to do.

```
GUESSANIMAL
WOULD YOU LIKE TO PLAY 'GUESS THE ANIMAL' WITH ME?
+YES
PLEASE THINK OF AN ANIMAL !
HAVE YOU THOUGHT OF ONE YET? 'NO' GIVES YOU A CHANCE TO DO SOMETHING
ELSE!
+YES
DOES IT HAVE A TAIL?
+YES
IS IT A HORSE ?
+NO
TOO BAD! I DIDN'T GET THAT ONE!
WOULD YOU PLEASE TYPE IN THE NAME OF THE ANIMAL!
+CAT
COULD YOU PLEASE GIVE ME A QUESTION THAT DISTINGUISHS YOUR ANIMAL FROM A
HORSE ?
+DO PEOPLE RIDE OM\M\N IT?
AND WHAT WOULD BE THE ANSWER IN THE CASE OF A HORSE ?
+YES
THANK YOU FOR GIVING ME THIS INFORMATION----I WILL REMEMBER IT!


PLEASE THINK OF AN ANIMAL !
HAVE YOU THOUGHT OF ONE YET? 'NO' GIVES YOU A CHANCE TO DO SOMETHING
ELSE!
+YES
DOES IT HAVE A TAIL?
+NO
IS IT A SNAKE ?
+NO
TOO BAD! I DIDN'T GET THAT ONE!
WOULD YOU PLEASE TYPE IN THE NAME OF THE ANIMAL!
+MONKEY
COULD YOU PLEASE GIVE ME A QUESTION THAT DISTINGUISHS YOUR ANIMAL FROM A
SNAKE ?
+DOES IT CRAWL ON THE GROUND?
AND WHAT WOULD BE THE ANSWER IN THE CASE OF A SNAKE ?
+YES
THANK YOU FOR GIVING ME THIS INFORMATION----I WILL REMEMBER IT!
```

```
PLEASE THINK OF AN ANIMAL !
HAVE YOU THOUGHT OF ONE YET? 'NO' GIVES YOU A CHANCE TO DO SOMETHING
ELSE!
+YES
DOES IT HAVE A TAIL?
+YES
DO PEOPLE RIDE ON IT?
+NO
IS IT A CAT ?
+YES
I GUESSED IT! I AM A SMART MACHINE?!--AT LEAST SOMETIMES!!!
PLEASE THINK OF AN ANIMAL !
HAVE YOU THOUGHT OF ONE YET? 'NO' GIVES YOU A CHANCE TO DO SOMETHING
ELSE!
+NO


IF YOU HAVE ENOUGH,TELL ME 'GOODBYE'!
IF YOU WANT TO SEE WHAT ELSE I CAN DO,TYPE IN 'MORE'!
+MORE


HERE ARE SOME OTHER QUESTIONS,WHICH I CAN ANSWER(TYPE IN THE QUESTIONS
LIKE SHOWN)!
WHAT ANIMALS DO YOU KNOW?--TYPE 1 FOR THAT!

WHAT QUESTIONS DO YOU KNOW?--TYPE 2 FOR THAT!

3)WHAT ANIMALS DO YOU KNOW,WHICH HAVE A YES OR NO ANSWER TO A CERTAIN
QUESTION?
THE FORMAT FOR 3) SHOULD BE:
    3 'THE QUESTION' 'YES-OR-NO'
PLEASE TYPE IN YOUR QUESTION.'NO' GETS YOU BACK!
+1
HORSE CAT SNAKE MONKEY


+3 DOES IT HAVE A TAIL? YES
HORSE CAT


+NO
PLEASE THINK OF AN ANIMAL !
HAVE YOU THOUGHT OF ONE YET? 'NO' GIVES YOU A CHANCE TO DO SOMETHING
ELSE!
+NO


IF YOU HAVE ENOUGH,TELL ME 'GOODBYE'!
IF YOU WANT TO SEE WHAT ELSE I CAN DO,TYPE IN 'MORE'!
+GOODBYE
THANK YOU FOR PLAYING WITH ME! COME BACK ANY TIME!!
DO YOU WANT TO SAVE WHAT YOU DID FOR YOUR NEXT SECTION!
+YES
SEE YOU NEXT TIME! AUF WIEDERSEHEN!!
+
```

Before we talked about the animal program, we discussed the procedure to walk a binary tree (see [1], Appendix 2), so the students had seen how to use a binary tree to represent a certain problem.

The animal problem was interesting with regard to the following features:

a) it is required to write programs to manipulate a binary tree

b) we could talk about the abstract concept of a node in a tree (and how information can be attached to a node)

c) the program shows some kind of learning behavior in the sense that it "knows more" after we played for a while. We hoped that this feature would be of special interest to Social Science students. After they had solved the problem, we gave them a copy of [2] to gather some evidence as to whether they got more out reading a paper like this after having programmed the animal problem.

We used our previous defined property list functions to attach information to a node in our tree, eg a node consisted of four things:

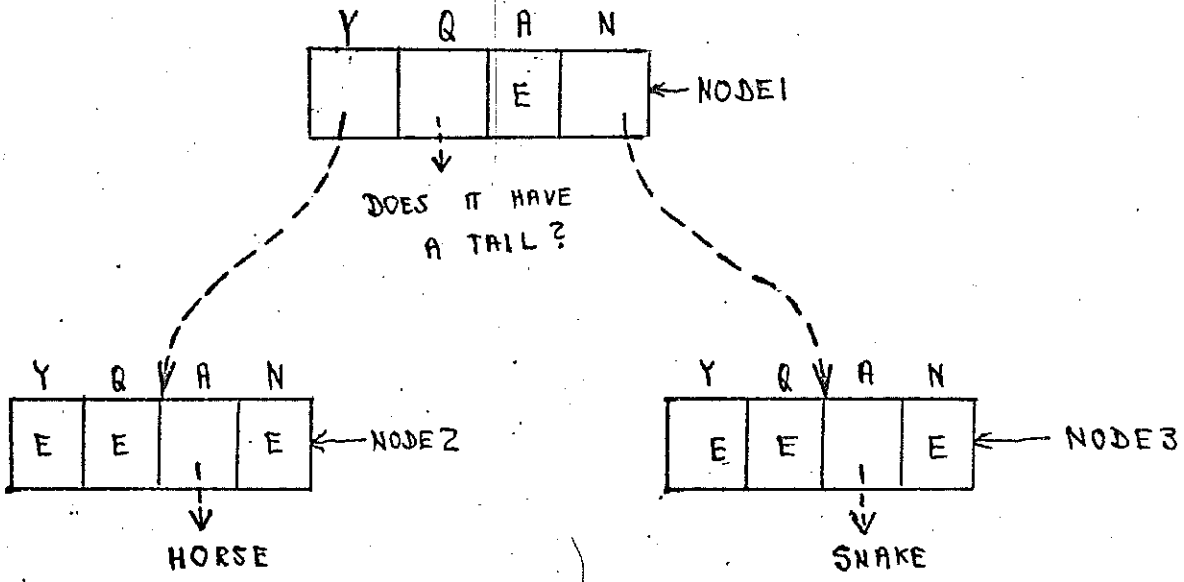
a) a YESBRANCH (a pointer to the yes-subtree)

b) a NOBRANCH (a pointer to the no-subtree)

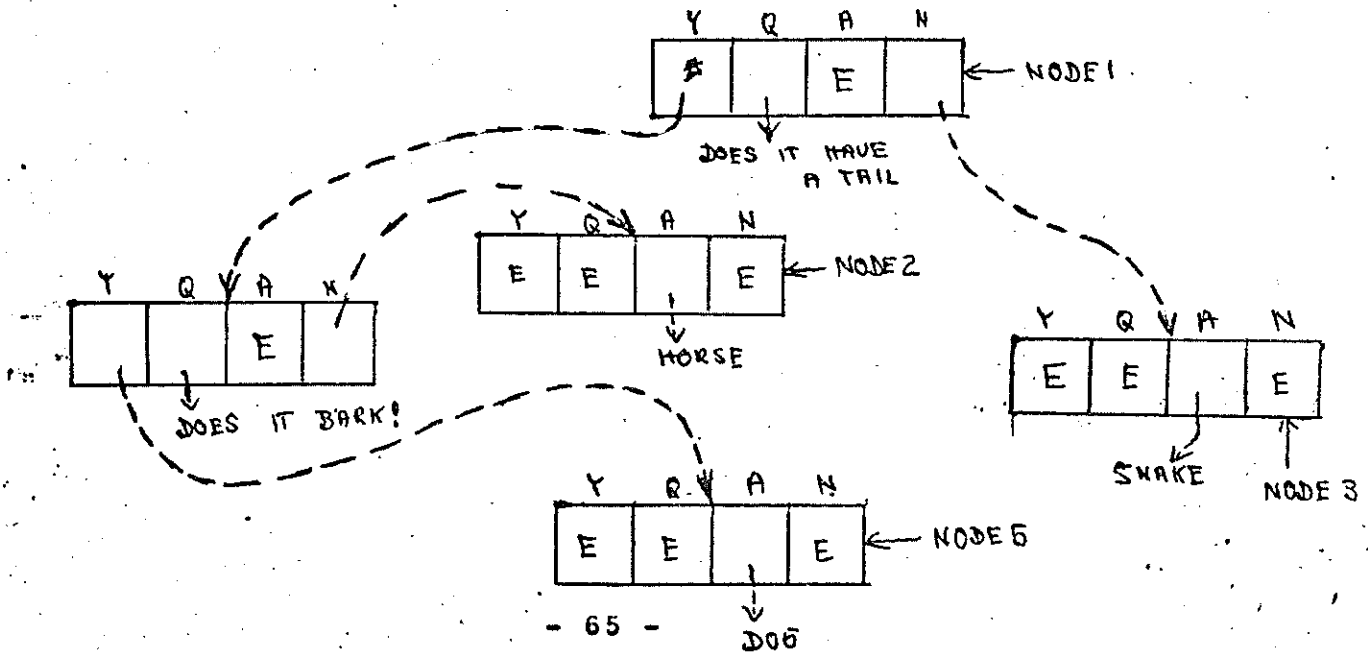
c) a QUESTION

d) an ANIMAL

As an example we may have the following tree (E stands for empty):



If we play the program and we don't guess the animal, we have to add new nodes to our tree (lets say, the new animal is a dog and the question is: "does it bark?")



There is an alternative way to update the tree by changing the content of nodes and leaving the linkage between the nodes. This method would have the advantage that we could work more locally on the tree, ie we don't have to look back to the previous node. The way described has the advantage that existing nodes are never changed.

The following procedures would traverse the tree and handle the case when we find an animal:

```
LIST TRAVERSE

TO TRAVERSE :NODE: :PN: :FLAG:
10 TEST ANIMALP :NODE:
20 IFTRUE HANDLEANIMAL :NODE: :PN: :FLAG:
30 IFTRUE STOP
40 PRINT QUESTION :NODE:
50 TEST IS REQUEST "YES"
60 IFTRUE TRAVERSE ( YESBRANCH :NODE: ) :NODE: "YES"
70 IFTRUE STOP
80 TRAVERSE ( NOBRANCH :NODE: ) :NODE: "NO"
END

←LIST HANDLEANIMAL

TO HANDLEANIMAL :NODE: :PN: :FLAG:
10 PRINT SENTENCE$ "IS IT A" ( GETPROP :NODE: "ANIMAL" ) "?"
20 TEST IS REQUEST "YES"
30 IFTRUE FOUNDIT
40 IFTRUE STOP
50 GATHERINFO :NODE: :PN: :FLAG:
END
```

GATHERINFO is the procedure which gathers information from the user and then calls UPDATETREE which gets new nodes from GENNAME, fills in the information and updates the tree.

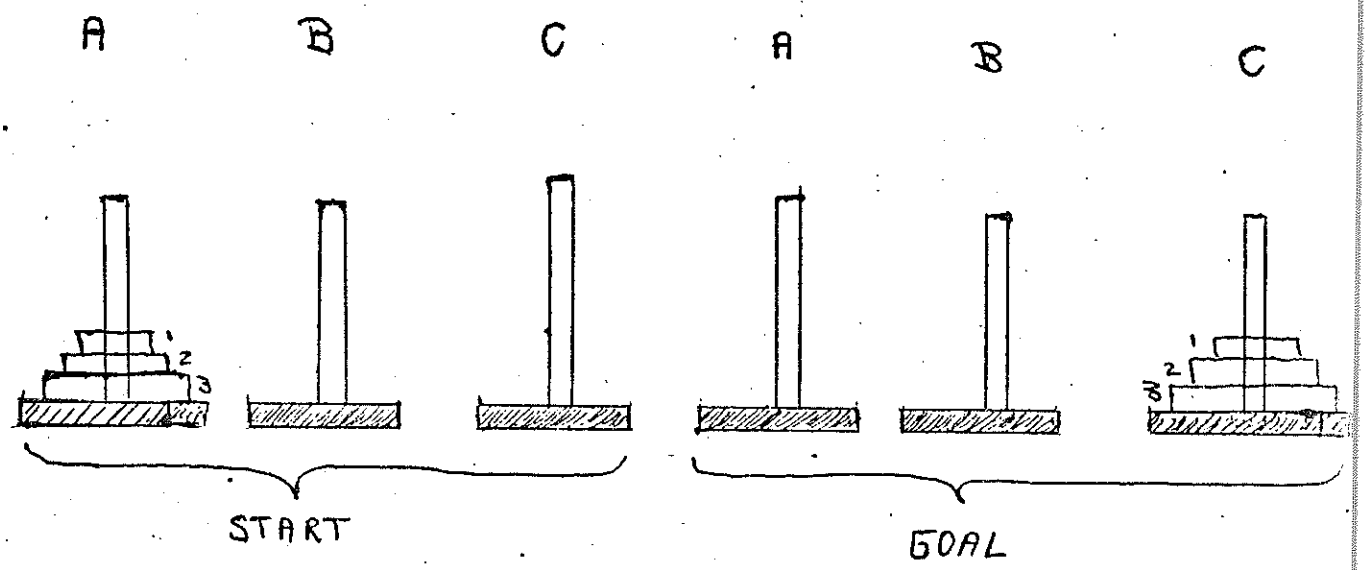
Extensions to this problem could be to provide answers to the questions shown in the test run (ie "What animals do you know?" or "Which animals have a tail?").

3.8 THE TOWER OF HANOI

A short description of the problem would be:

> "There are three pegs A, B, C and N disks of different size. The disks have holes in their centers so that they can be stacked on the pegs.
> Initially the disks are all on peg A, ordered by size with the largest one at the bottom.
> We have to move all the disks to peg C by moving one disk at a time. Only the top disk on a peg can be moved, but it can never be placed on a smaller disk."

the figure below shows the initial and goal configurations:



The solution in LOGO and a sample test run for N=3:

```
TO HANOI :N: :S: :I: :D:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 HANOI ( DIFFERENCE :N: 1 ) :S: :D: :I:
40 PRINT SENTENCES "MOVE PIECE NUMBER" :N: "FROM" :S: "TO" :D:
50 HANOI ( DIFFERENCE :N: 1 ) :I: :S: :D:
END
```

```
←HANOI "3" "A" "B" "C"
MOVE PIECE NUMBER 1 FROM A TO C
MOVE PIECE NUMBER 2 FROM A TO B
MOVE PIECE NUMBER 1 FROM C TO B
MOVE PIECE NUMBER 3 FROM A TO C
MOVE PIECE NUMBER 1 FROM B TO A
MOVE PIECE NUMBER 2 FROM B TO C
MOVE PIECE NUMBER 1 FROM A TO C
←
```

The solution is remarkably short, if we see the right way to approach the problem. A computer scientist who wants to design a data structure for the problem may have a difficult time solving the problem, whereas if we see the problem as follows:

> "to solve the problem for N disks, lets assume we would know how to move N-1 disks from peg A to peg B. Then we could move the largest disk from A to C

and then the N-1 disks from B to C."

the solution in LOGO is then an exact translation from this description into LOGO code.

An extension of the problem would be to ask the students to think about how many moves we need to solve the puzzle for n disks (the proof is a nice example for a simple mathematical proof by induction). It may be also interesting to let them calculate how long the output would be for 30,40,50 disks!! (One line per move, 300 lines per yard)

3.9 THE LOGO INIT FILE

Every LOGO user will find that he would like to have available all the time some of his own procedures (eg SUB1, ADD1, NTH, the property list functions). To achieve this, we create a special file, called INIT which contains some of these procedures. It is up to the personal taste of every user which procedures he wants to keep in his INIT file. A possible set is shown below:

```
TO MEMBERP :ELEMENT: :SET:
10 TEST EMPTYP :SET:
20 IFTRUE OUTPUT "FALSE"
30 TEST IS :ELEMENT: ( FIRST OF :SET: )
40 IFTRUE OUTPUT "TRUE"
50 OUTPUT MEMBERP OF :ELEMENT: AND BUTFIRST OF :SET:
END

TO SUB1 :X:
10 OUTPUT DIFFERENCE :X: 1
END

TO ADD1 :X:
10 OUTPUT SUM OF :X: AND 1
END

TO SPACING :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 PRINT ""
40 SPACING SUB1 :N:
END

TO BLANKS :N:
10 TEST ZEROP :N:
20 IFTRUE OUTPUT :EMPTY:
30 OUTPUT WORD :BLANK: BLANKS SUB1 :N:
END
```

```
TO MYERASE :S:
10 TEST EMPTYP :S:
20 IFTRUE STOP
30 DO SENTENCE "ERASE" FIRST OF :S:
40 MYERASE BUTFIRST :S:
END

TO MYTRACE :S:
10 TEST EMPTYP :S:
20 IFTRUE STOP
23 TEST IS FIRST OF :S: "MYTRACE"
30 IFFALSE DO SENTENCE "TRACE" FIRST OF :S:
40 MYTRACE BUTFIRST :S:
END

TO MYLIST :SEN:
10 TEST EMPTYP :SEN:
20 IFTRUE STOP
30 DO SENTENCE "LIST" FIRST OF :SEN:
40 MYLIST BUTFIRST :SEN:
END
```

SPACING: to print N empty lines

BLANKS: to print N blanks

MYERASE: to erase a list of procedures

MYTRACE: to trace a list of procedures. The LOGO name CONTENTS which returns a sentence of all procedures names in workspace can be used as input to trace all procedures

MYLIST: to list a number of procedures

4. ERRORS

This chapter will give some examples of typical errors which occured in the students programs. The examples may give some hints

1) to see where the students had difficulties to understand something
2) to improve the error messages and the error handling

It turned out that it was important to explain the meaning of the error messages to the students. If they followed our advice to write short procedures, traced their procedures and understood the error messages they developed good abilities to debug their programs (we mentioned earlier that we believe that this ability is important).

A list of the most common errors appears below (this list is not ordered; in most cases the problems, where the error occured, are mentioned).

1) Wrong number of inputs:

a)→P WORD "THE CAT" "IN THE HAT"

WORD OF "THE CAT" AND "IN THE HAT"

INPUTS TO WORD CANNOT BE SENTENCES

b)→PRINT SUM 4 5 6

9

"6" IS EXTRA

2) wrong kinds of inputs

a) from the vowel problem:

TO ALLCOUNT :SENT:

10 TEST EMPTYP :SENT:

20 IFT OUTPUT :SENT:

30 OUTPUT SUM COUNT FIRST :SENT:

ALLCOUNT BUTFIRST :SENT:

END

→P ALLCOUNT "I HATE TO WORK"

SUM OF "4" AND ""

INPUTS MUST BE NUMBERS

I WAS IN LINE 30 IN ALLCOUNT

This error is a little bit more settled, but if the

students turn on a trace, it should not be too difficult to find out where the error occured.

It would be helpful in a case like this, if LOGO would print out the line in which the error occured.

B)
```
TO ONEP :NUM:
10 OUTPUT IS :NUM: 1
END

TO REVERSE :W:
10 TEST ONEP :W:
20 IFTRUE OUTPUT :W:
30 OUTPUT WORD OF (LAST OF :W:)
                  (REVERSE BUTLAST OF :W:)
END
```

REVERSE goes into an infinite recursion, because we should have tested COUNT :W: equal to 1. If we fix this error, there is still a small problem with this version of REVERSE(in contrast with the version in 2.2): when we call REVERSE "", we still go into an infinite recursion.

C) another example would be that the inputs to the TEST box have another value besides "TRUE" or "FALSE". See page 18 where we talked about this.

3) The students had a hard time to understand the difference between PRINT (whose only function is to spew out its input to the teletype), STOP (which terminates a procedure) and OUTPUT (which terminates a procedure and passes its input back to the procedure which called the current procedure):

a) from the wizard problem:

```
TO DIV3P
10 -UTPUT IS 0 REMAINDER ADDIG :NUM: 3
20 OUTPUT IS 0 REMAINDER :NUM: 3
END
```

This error is probably difficult to detect, because the student won't get an error message; line 20 will just never be executed.

B) from the Q/A problem:

```
TO ADDINFO :NA: :TH:
10 TEST EMPTYP OF THING OF :NA:
20 IFTRUE MAKE :NA: :TH:
30 MAKE :NA: SENTENCE THING OF :NA: :TH:
END
```

This bug has the consequence that when the thing of :NA: is empty, the first element will be inserted twice.

C) from the INIT file

```
TO SPACING :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 PRINT ""
40 SPACING SUB1 :N:
END
```

```
→ P SPACING 4
```

SPACING CAN'T BE USED AS AN INPUT. IT DOES NOT OUTPUT

This is a good example for discussing the question of where during execution an error occurs. SPACING prints the five empty lines and the error occurs when the PRINT box outside discovers that it doesn't get an input because SPACING doesn't output.

4) Problems with recursive procedures:

We noticed that some students used pattern matching

methods to write their recursive procedures (ie, they tried to make the appropriate changes in the procedures which we showed them in class). This method didn't succeed too often and the students were generally confused if something went wrong

a) an example for the above behaviour:

```
TO VOWELP :L:
10 TEST IS :L: FIRST OF "AEIOU"
20 IFTRUE OUTPUT "TRUE"
30 OUTPUT VOWELP :L: BUTFIRST OF "AEIOU"
END
```

This procedure was written early in the course and it is obviously modeled after MEMBERP. It contains several mistakes. LOGO doesn't discover that VOWELP is used with the wrong number of inputs in line 30, because before it comes to the second argument , it will go into an infinite recursion.

b) Terminating the recursion at the wrong level:

```
TO STARS :N:
10 TEST IS :N: 1
20 IFTRUE OUTPUT ""
30 OUTPUT WORD OF "*" AND STARS SUB1 :N:
END

P STARS "5"
****
```

5. THE HIGH SCHOOL PROJECT

The project was planned hastily because we did not know until the beginning of the first week of the Spring quarter that we wanted to do the project. John Brown brought up the idea and the ICS Department provided the money. There were less than ten weeks left to find some interested students from the ICS senior seminar and some high school students who were willing to participate. Because of this time pressure we (ie the six students from ICS 190 and myself) did not have very much time to find a group of high school students who could have been selected along certain criteria (eg age, no previous programming experience, strong or weak interest in mathematics). Nevertheless the project worked out well for all participating groups.

5.1 THE "TEACHERS"

At the beginning of the project none of the six students had very much teaching experience and none of them had programmed in LOGO very much (some of the students had implemented LOGO at a SIGMA 7 computer the quarter before, but it turned out that this was a different task as compared with programming in LOGO).

We spent the first few weeks to study the work that other people did in this area (eg at BBN and the LOGO group at MIT). The most difficult task in our opinion was to come up with good problems and good projects.

Every student was asked to prepare a lecture and a handout (because we did not give a LOGO manual to the high school students) about a certain topic. Most students regarded the preparation of a lecture and the speaking in front of a group of students as a very important experience. They noticed to some extent the teaching effect ("you never learn something, until you have to teach it") and they became aware that it was more difficult to talk to the high school students about certain concepts and ideas because they didn't know the "computer science vocabulary".

5.2 THE HIGH SCHOOL STUDENTS

We were worried at the beginning that we may get students who would show some initial interest but that this interest would soon disappear when they realized that programming requires a lot of time and effort if you want to do more than just the most trivial things. Moreover the students could come out to the Computing Center only after their regular school hours.

We were totally surprised when we realized that more than half of them spent almost all their spare time at a terminal during the time period of our project.

As mentioned earlier, we did not have too much choice in selecting the group. It turned out that several students had previous programming experience (mostly BASIC and FORTRAN). They also showed a great deal of interest learning something about the operating system. Unfortunately they did not use their knowledge only to do constructive things so that within a few weeks they had lost the sympathy of the Computing Center staff. We did not anticipate these problems, because we assumed that we would get mostly beginners, and we were therefore partly responsible for the problems by answering their questions about the operating system.

We felt very strongly that we should not suppress the curiosity and interest (one of the most encouraging experiences about this experiment) of the high school students. Although even we would have preferred if they would have spent all their time with LOGO (the good students did not have any difficulties with most of the problems which we gave them, but they did not work too much on their final projects), we let them go ahead to do some other things (eg some students liked to use the plotter; the two plots at the first and last page of this report are an example for this). But they also did some additional work in LOGO:

the students who did not have any programming experience enjoyed at the beginning to print funny messages, happy faces, etc. The better students wrote game playing programs (NIM, GUESS THE NUMBER etc). One student got hold of a LOGO manual and created a file which contained the most important commands and copied it to the accounts of the other students. They just had so many ideas!!

5.3 THE ORGANIZATION OF THE COURSE

It was an experience for all of us and we believe that we would avoid some mistakes if we would do it again. It seemed to most of us that it is impossible to have a strict course outline for an experimental class like we had. There were just too many unknown variables (eg how much do the students know, how much do they work, what ideas interest them, etc) that the only possibility was to choose an adaptive strategy. This means that the "teachers" met before every class to discuss the previous class and, based on how it went, what they should do next. The last two weeks were the most fruitful part of the course, because we split the students in small groups so that every "teacher" worked together with one or two students on a project. Unfortunately the time was too short to get too far with the projects.

APPENDIX A: THE LITTLE BROTHER THEORY

Since LOGO contains only a few primitive procedures (we use the term "procedure" interchangeably with "function"), it was reasonable to ask students to create some new ones:

> Write a predicate to be called MEMBERP which is to have two arguments and which checks to see if its first argument is contained in its second. If it is, then MEMBERP should output "TRUE". Otherwise it should output "FALSE".

The purpose of this assignment was twofold. First, it exposed the student to the simplest form of recursion. Second, it called to their attention the possibility of adding new predicates, as well as operators, to the language. We also established the naming convention that any procedure which is to behave as a predicate (i.e. outputs "TRUE" or "FALSE") should have a "P" as the last letter in its name. This helped the students to remember which functions could follow a TEST command.

A solution to this problem might be:

```
TO MEMBERP /ELEMENT/ /SET/
10  TEST EMPTYP /SET/
20  IF TRUE OUTPUT "FALSE"
30  TEST IS /ELEMENT/ FIRST OF /SET/
40  IF TRUE OUTPUT "TRUE"
50  OUTPUT MEMBERP OF /ELEMENT/ AND BUTFIRST OF /SET/
END
```

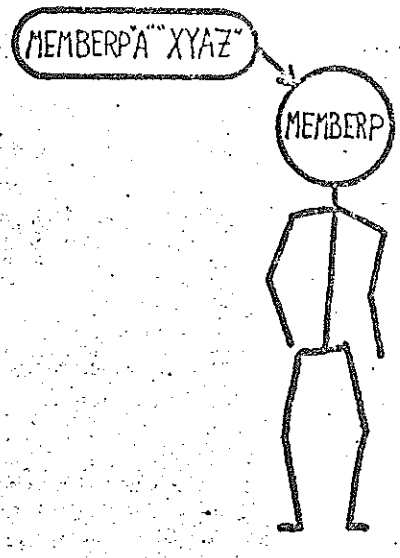
CHECKS FOR TERMINATING CONDITION OF THE RECURSION

CHECKS IF THE CURRENT FIRST ELEMENT OF /SET/ EQUALS THE DESIRED ELEMENT

RECURSES WITH THE CURRENT /SET/ MINUS ITS FIRST ELEMENT

We consider the "MEMBERP" predicate to be the name of a "little brother" who has numerous identical twin brothers -- all called by the same name, MEMBERP. This family of MEMBERP brothers works as follows: suppose we make a request of a MEMBERP brother, i.e.

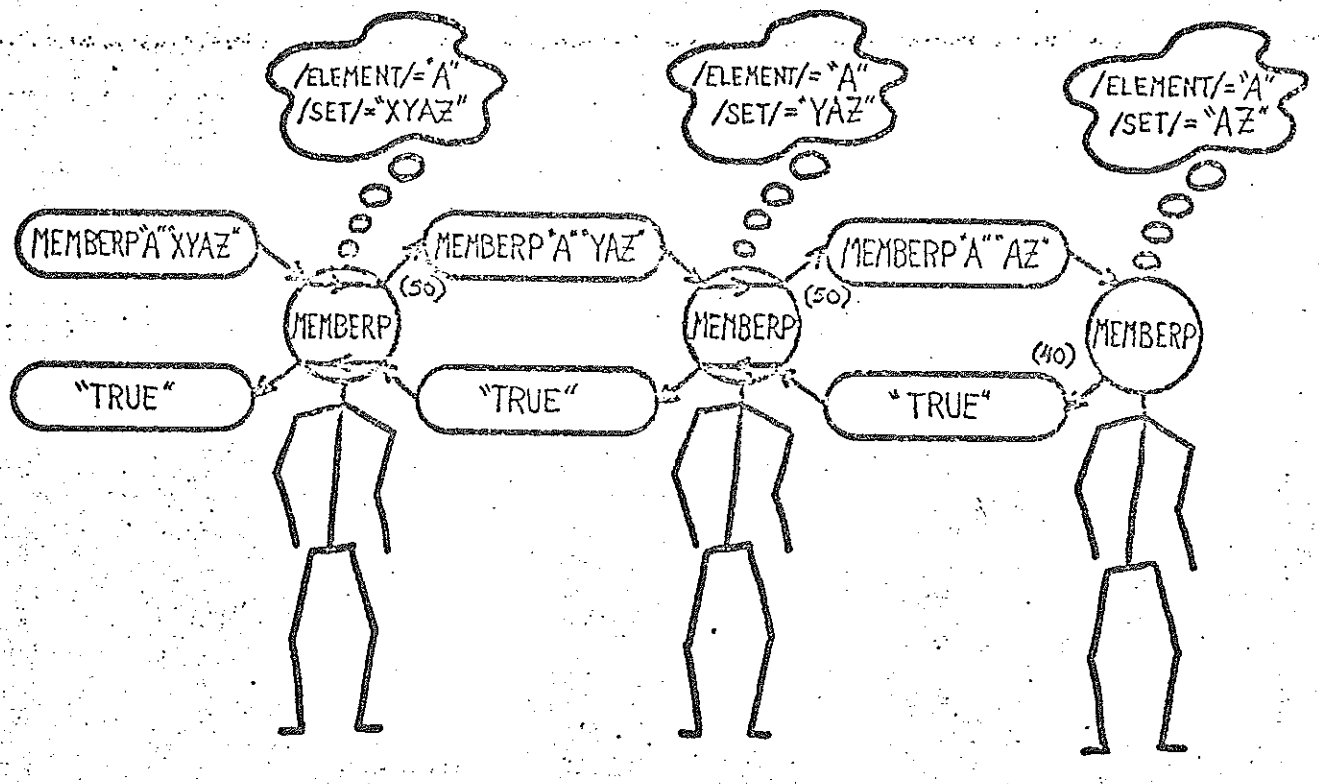
Figure 1 -- A MEMBERP Brother



The first MEMBERP brother executes his definition by first testing if /SET/ has any elements. It is not empty, so he tests if /ELEMENT/ (i.e. "A") is first of "XYAZ". Since "A" is not equal to "X", "IS" outputs "FALSE" to "TEST" (line 30) causing line 40 to fail. We are now at line 50. But in order for this first little brother to complete line 50, he must call for assistance from one of his twins. He requests that his brother tell him the answer to a slightly simpler problem; he asks him to compute: MEMBERP "A" "YAZ". This process continues with each brother calling on another brother to do a slightly simpler task

until finally a brother is called who can complete his simpler task (possibly the null task). This last brother then sends his answer back to the brother that called him enabling that brother in turn to finish, (i.e. complete his line 50), and so on:

Figure 2 -- A Chain of MEMBERP Brothers



The explanation omits one very important construct which we dub "conceptual clouds." A conceptual cloud is used to determine the "world-view" of a particular brother. That is, it defines what he knows or what meanings he ascribes to the names in his particular world. Each MEMBERP brother has a conceptual cloud that looks like those above the men in Figure 2. So as far as the first brother is concerned, the meaning of /SET/ (what /SET/ denotes, i.e. the THING OF "SET") is the string "XYAZ". His next brother in line has a different world-view: in his conceptual cloud /SET/ has the meaning "YAZ".

APPENDIX B: NOTE ABOUT NAMES AND PLACEHOLDERS

This is a sample of the handouts which we gave to the high school students. It was written by David Walker.

NAMES THINGS AND PLACE-HOLDERS

To begin with, let's not talk about LOGO. Instead, we'll discuss a language called OGOL. OGOL is very similar to LOGO; in fact, there is only one difference between the two languages. (There is no real language called OGOL; I just made it up.)

OGOL has three ways of specifying the value of something. The first is simply to type the value in double quotes (a THING):

"THIS IS A SENTENCE" or,

"AWORD"

The second way is to type the NAME of the value:

:X:

:AVERYLONGNAME:

These first two ways of specifying a value are very similar to the way that we specify values in LOGO. The third way to specify a value in OGOL can also be used in LOGO, but it looks different. This method is called a PLACE-HOLDER. See if you can guess what a PLACE-HOLDER is from the following OGOL procedure.

```
TO GREET <PERSON>
10 PRINT SENTENCE OF "HI THERE," AND <PERSON>
END
```

(PLACE-HOLDERS are surrounded by angle brackets: <>)

Appendix B continued:

PLACE-HOLDERS are only used in procedures. When OGOL encounters a PLACE-HOLDER, it looks at the line that started the procedure to find out what value is really wanted. So, using the OGOL procedure GREET from above, if someone typed:

GREET "JOHN"

OGOL would type: HI THERE, JOHN

Perhaps you can see how LOGO handles PLACE-HOLDERS. LOGO uses colons (:) to surround NAMES and PLACE-HOLDERS. Therefore, GREET would be written in LOGO as follows:

```
TO GREET :PERSON:
10 PRINT SENTENCE OF "HI THERE," AND :PERSON:
END

GREET "GEORGE"

HI THERE, GEORGE
```

APPENDIX C: TEXTS FOR THE VOWEL PROBLEM

These are the original texts of the texts which appear on page 39

P :FRENCH:
C'EST DIMANCHE : DERRIERE LES DOCKS , LE LONG DE LA MER , PRES DE LA GARE AUX MARCHANDISES , TOUT AUTOUR DE LA VILLE IL Y A DES HANGARS VIDES ET DES MACHINES IMMOBILES DANS LE NOIR . DANS TOUTES LES MAISONS ,DES HOMMES SE RASENT DERRIERE LEURS FENETRES ; ILS ONT LA TETE RENVERSEE , ILS FIXENT TANTOT LEUR MIROIR ET TANTOT LE CIEL FROID POUR SAVOIR S'IL FERA BEAU . LES BORDELS OUVRENT A LEURS PREMIERES CLIENTS , DES CAMPAGNARDS ET DES SOLDATS . DANS LES EGLISES , A LA CLARTE DES CIERGES , UN HOMME BOIT DU VIN DEVANT DES FEMMES A GENOUX . DANS TOUS LES FAUBOURGS , ENTRE LES MURS INTERMINABLES DES USINES , DE LONGUES FILES NOIRES SE SONT MISES EN MARCHE , ELLES AVANCENT LENTEMENT SUR LE CENTRE DE LA VILLE .
←

P :ENGLISH:
THE ANIMALS WERE SHOCKED BEYOND MEASURE TO LEARN THAT EVEN SNOWBALL COULD BE GUILTY OF SUCH AN ACTION . THERE WAS A CRY OF INDIGNATION , AND EVERYONE BEGAN THINKING OUT WAYS OF CATCHING SNOWBALL IF HE SHOULD EVER COME BACK . ALMOST IMMEDIATELY THE FOOTPRINTS OF A PIG WERE DISCOVERED IN THE GRASS AT A LITTLE DISTANCE FROM THE KNOLL . THEY COULD ONLY BE TRACED FOR A FEW YARDS , BUT APPEARED TO LEAD TO A HOLE IN THE HEDGE . NAPOLEON SNUFFLED DEEPLY AT THEM AND PRONOUNCED THEM TO BE SNOWBALL'S . HE GAVE IT AS HIS OPINION THAT SNOWBALL HAD PROBABLY COME FROM THE DIRECTION OF FOXWOOD FARM .
←

P :GERMAN:
BALD SOLLTE ICH JENE BLUME BESSER KENNENLERNEN . ES HATTE AUF DEM PLANETEN DES KLEINEN PRINCEN IMMER SCHON BLUMEN GEGEBEN , SEHR EINFACHE , AUS EINEM EINZIGEN KRANZ VON BLUTENBLATTERN GEFORMT ; SIE SPIELTEN KEINE GROSSE ROLLE UND STORTEN NIEMANDEN . SIE LEUCHTETEN EINES MORGENS IM GRASE AUF UND ERLOSCHEN AM ABEND . ABER JENE EINE HATTE EINES TAGES WURZEL GESCHLAGEN , AUS EINEM SAMEN , WEISS GOTT WOHER , UND DER KLEINE PRINZ HATTE DIESEN SPROSS , DER DEN ANDEREN SPROSSLINGEN NICHT GLICH , SEHR GENAU UBERWACHT . DAS KONNTE EINE NEUE ART AFFENBROTBAUM SEIN . ABER DER STRAUCH HORTE BALD AUF ZU WACHSEN UND BEGANN , EINE BLUTE ANZUSETZEN .
←

APPENDIX D: NOTES ABOUT THE WIZARD'S PROBLEM

This little handout tries to show some general ideas about solving a problem with the help of LOGO and the computer. As an example, we choose the following conjecture:

Last night (late at the computing center) a mad wizard appeared and proposed a ridiculous problem:

"John," he said, "How can you tell if a number is divisible by 3?"

How absurd I thought, but the wizard claims he has some method to his madness:

He claimed: Take the number and add up all its digits. If the resulting number is divisible by three then so was the original number.

ex.: 471 = 4+7+1 = 12 and 12/3 = 4 so 471 must be divisible by 3 - or so he claims:

Dashing to a LOGO terminal, I thought I would quickly disprove him.

Assignment: Using LOGO try to determine if the wizard is crazy.

Hint: Use the REMAINDER procedure in LOGO to build a predicate DIV3P and then build a little man to add up the digits of a number and so on. Of course you will also need a little man who simply vomits forth the integers starting at 3.

Extra credit: Can you prove it???

Further examples:	(1)	(2)	(3)	
Number:	99	100	101	.....
Add up the digits:	18	1	2	
Sum of the digits divisible by 3:	yes	no	no	

Step 1: Do We Understand the Problem?

This is certainly the first question which we should ask, i.e. what does the wizard propose in our example.

Given a certain number, e.g. 4738, we would like to know whether the number is divisible by 3. We can determine this in the "normal way" by dividing the number by 3:

```
  1 5 7 9
3 /4 7 3 8
  3
  1 7
  1 5
    2 3
    2 1
      2 8
      2 7
        1
```

We are left with a remainder of 1, i.e. the number is not divisible by 3.

The whole procedure is certainly not too difficult but the wizard has a better idea. He claims that if we add up the digits, in our case $4 + 7 + 3 + 8 = 22$, the resulting number is (not) divisible by three if the original number was (not) divisible by three. In our example, you would say that 4738 and 22 have the same remainder when divided by three. Adding up the digits is easy and can be done fairly quickly, and it is certainly easier to determine that 22 is not divisible by three than if we had to figure the same for 4738.

The question arises: Is the wizard right, i.e. can we do this for any number (in our above example, he was right, because 4738 and 22 both leave the remainder 1).

For a correct solution, we also have to pay attention to the following fact: The two procedures (i.e. the "wizard's" method and the "normal" method) have to be equivalent (in mathematical terms: we have an "if and only if" conjecture).

Step 2: Make a plan how to solve the problem.

If you make a plan, you probably start off with figuring out the overall or global structure and ignoring the details at the beginning. Our problem is to show whether the wizard is right or wrong. In LOGO terms, this problem may be described in the following way:

We have to write a predicate COMPAREP, which compares the output of the following predicates:

a) DIV3P, which divides a number by 3 in the "normal" way and outputs true, if it is divisible by 3, otherwise false.
e.g. DIV3P "46" ⇒ FALSE; DIV3P "48" ⇒ TRUE

b) STRANGEDIVP, which divides a number by 3 in the "wizard's" way and outputs true,if it is divisible by 3 in this way, otherwise false.
e.g. STRANGEDIVP "46" ⇒ FALSE; STRANGEDIVP "48" ⇒ TRUE

Here is a version of COMPAREP in LOGO:

```
TO COMPAREP :N:
10 OUTPUT IS DIV3P :N: STRANGEDIVP :N:
END
```

COMPAREP returns true, if (and only if) the "normal" division and the "wizard's" division end up with the same result, otherwise false.

But we have not solved the problem so far, because we can't use COMPAREP at this point, because it contains the two procedure-calls DIV3P and STRANGEDIVP and we have not defined them.

But we reduced the original problem to two "smaller" problems, which we have to solve next.

Remark: This approach to problem solving is usually called a "top-down" approach, because we start with the most important problem first, and we decompose or reduce it to subproblems.

We hope that all of the subproblems are "easier" to solve (that may not necessarily be true). Can you see the similarity to the "little brother theory", where the next little brother always had a slightly easier problem to solve?

Step 3: Solve the subproblems and fill in the little details.

LOGO contains a function REMAINDER, which has the obvious effect: it takes two numbers as inputs and outputs the remainder. Therefore, we can write DIV3P in the obvious way:

```
TO DIV3P :NUM:
10 OUTPUT IS ( REMAINDER :NUM: 3 ) 0
END
```

STRANGEDIVP can now be written in two ways, if we observe that the number which gets divided is in this case the sum of the digits of the original number.

```
TO STRANGEDIVP :N:
10 OUTPUT ZEROP REMAINDER ( ADDUP :N: ) 3
END
```

In the second version, we use the procedure DIV3P just defined:

```
TO STRANGEDIVP :N:
10 OUTPUT DIV3P ( ADDUP :N: )
END
```

You probably noticed that we did the same "trick" again; we didn't worry for the moment how we would add up the digits. We generated a new subproblem, which we have to solve next: to write a procedure which adds up the digits of an arbitrary large number.

You probably realized immediately that this is a "little brother" problem and you may come up with the following solution:

```
TO ADDUP :NUM:
10 TEST EMPTYP :NUM:
20 IFTRUE OUTPUT ""
30 OUTPUT SUM OF ( FIRST OF :NUM: ) ( ADDUP BUTFIRST OF :NUM: )
END
```

We didn't have to generate a new subproblem, so the first version of our solution is found, assuming all our procedures will work.

Step 4: Debugging and testing of the LOGO-procedures.

This time, we probably start at the other end: we first make sure that the small procedures work, before we try to debug the procedures, which depend on many of the small ones (this method is in general called: "bottom-up debugging"). So let us start with ADDUP:

```
←ADDUP "4378"
SUM OF "8" AND ""
INPUTS MUST BE NUMBERS.
I WAS AT LINE 30 IN ADDUP
```

The error message should tell us probably enough to see what we did wrong, but we may turn on a TRACE to get even more insight why the procedure blew up:

```
ADDUP "4378"
ADDUP OF "4378"
  ADDUP OF "378"
    ADDUP OF "78"
      ADDUP OF "8"
        ADDUP OF ""
        ADDUP OUTPUTS ""
SUM OF "8" AND ""
INPUTS MUST BE NUMBERS.
I WAS AT LINE 30 IN ADDUP
←
```

We change line 20 of ADDUP and then our procedure works on several cases:

```
TO ADDUP :NUM:
10 TEST EMPTYP :NUM:
20 IFTRUE OUTPUT 0
30 OUTPUT SUM OF ( FIRST OF :NUM: ) ( ADDUP BUTFIRST OF :NUM: )
END
```

Similarly we test the other procedures:

```
←P ADDUP "4378"
22
←P DIV3P "4378"
FALSE
←P STRANGEDIVP "4378"
FALSE
←P COMPAREP "4378"
TRUE
←P ADDUP "567"
18
←P DIV3P "567"
TRUE
←P STRANGEDIVP "567"
TRUE
←P COMPAREP "567"
TRUE
```

They all seem to work and our wizard seems to be right on the two examples tested.

You hopefully noticed, that it pays off well to structure your programs, i.e. that we wrote several short procedures (whose task should be indicated by a mnemonic name). This approach (called "functional programming") shows the structure of the problem and makes debugging much easier.

Step 5: Have we solved all parts of the problem?

Our original problem was to show that the wizard is right or wrong. If we would have one couterexample, we could say for sure: "The wizard is wrong." But so far, he is right, even though evidence is based only on two numbers. We would certainly feel much better if we could show that he is right on hundreds or thousands of numbers.

For this purpose, let's write another procedure which generates the number for us (hopefully, nobody would be willing to type a thousand times: P COMPAREP ....."). A possible version may be:

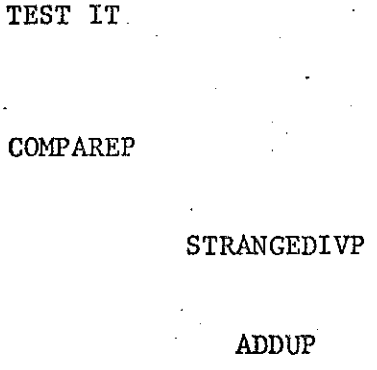
```
TO TESTIT :NUM:
10 TEST ZEROP :NUM:
20 IFTRUE OUTPUT "CONJECTURE SEEMS TO BE RIGHT"
30 TEST COMPAREP :NUM:
40 IFFALSE OUTPUT "CONJECTURE IS FALSE"
50 OUTPUT TESTIT ( DIFFERENCE OF :NUM: AND 1 )
END
```

and maybe we run it for a thousand numbers:

```
←P TESTIT 1000
CONJECTURE SEEMS TO BE RIGHT
```

After having this result we are much more inclined to say the wizard is right.

Now, we can represent our solution by a problem-solving tree:



Step 6: How can we extend the problem or improve our solution?

a) Let us assume for the moment that the wizard is right. Also let us assume that we don't have a REMAINDER function in LOGO. Can we still test whether a number is divisible by 3?

There is an obvious extension of the wizard's idea:

Why should we stop after one step, i.e., after going in our example from 4738 to 22. We could apply the wizard's method again and again, till we end up with a one digit number. And from a one digit number, we can decide fairly easily whether it is divisible by 3 because it should be a member of the set {0, 3, 6, 9}.

Using our previously constructed MEMBERP procedure, we can write the following LOGO procedure:

```
TO DIVBYMEMBP :N:
10 TEST IS COUNT OF :N: 1
20 IFTRUE OUTPUT MEMBERP :N: "0369"
30 OUTPUT DIVBYMEMBP ( ADDUP :N: )
END
```

and see how it works:

```
←P DIVBYMEMBP "12345678998767876787666"
DIVBYMEMBP OF "12345678998767876787666"
  DIVBYMEMBP OF "143"
    DIVBYMEMBP OF "8"
    DIVBYMEMBP OUTPUTS "FALSE"
  DIVBYMEMBP OUTPUTS "FALSE"
DIVBYMEMBP OUTPUTS "FALSE"
FALSE
```

b) One question remains - what does it mean: Can you prove it. Most of us probably have proven a theorem in mathematics. So we may ask: Isn't it good enough for a proof what we have done so far? We have fairly big evidence (a thousand numbers), that the wizard is right. But what happens if somebody comes along and says: I believe that the wizard is wrong, if we take really large numbers (e.g. over one million, or over one billion). Then we are still in trouble, because we all know that there are potentially infinite many integers, i.e. somebody may give us an arbitrary large number and we still can find a larger number by adding one to this number.

You may see now some evidence of why mathematicians try to prove theorems. They are faced with the problem of saying something about infinite structures.

Here is an outline of a mathematical proof:

Every number can be represented in the following way:

(*) $(a_n \times 10^n)+(a_{n-1} \times 10^{n-1})+(a_{n-2} \times 10^{n-2})+ \ldots +(a_1 \times 10^1)+(a_0 \times 10^0)$

(In our example: $4738 = (4 \times 10^3)+(7 \times 10^2)+(3 \times 10^1)+(8 \times 10^0)$)

If we add up the digits, we just leave out the powers of 10 and we get:

(**) $a_n + a_{n-1} + a_{n-2} + \ldots + a_0$ $\quad (4 + 7 + 3 + 8)$

Now we have to know a little bit about how to calculate with remainders:

First let us make the following abbreviation:

a = 10 mod 3 means a is the remainder of 10 divided by 3. Then, the following rules are valid.

(i) 10 mod 3 = 1
(ii) (10 X 10) mod 3 = (10 mod 3) X (10 mod 3) = 1 X 1 = 1
i.e. 10^n mod 3 = (10 mod 3) X . . . X (10 mod 3) = 1 X ...X 1 = 1
(iii) (a + b) mod 3 = (a mod 3) + (b mod 3)

Applying these rules to (*) gives us:

$(a_n \times 10^n + \ldots + a_0) \bmod 3 = (a_n \times 10^n) \bmod 3 + \ldots + (a_0 \times 10^0) \bmod 3$

$= (a_n \bmod 3)(10^n \bmod 3) + \ldots + (a_0 \bmod 3)(10^0 \bmod 3)$

$= (a_n \bmod 3)(1) + \ldots + (a_0 \bmod 3)(1)$ (***)

And applying these rules to (**):

$(a_n + a_{n-1} + \ldots + a_0) \bmod 3 = (a_n \bmod 3) + (a_{n-1} \bmod 3) + \ldots +$

$(a_0 \bmod 3)$ (****)

(***) is the same as (****), which proves the wizard's conjecture.

Step 7: Try to invent problems yourself!

After you've solved the wizard's original problem, play wizard and invent problems for yourself. The wizard still has some ideas. Here is his next problem:

"How can you tell whether a number is divisible by 9?"

I solved it in the following manner:

a) Given any number, cross out all the nines. For example, 47398719 becomes 473871.

b) Add up the digits. If the sum, y, is greater than or equal to 9 then reduce y to the remainder of y - 9. For example, 4 + 7 = y and y $\geq$ 9; y - 9 = 2 so now y = 2. Continue adding up the digits; 2 + 3 + 8 = 13 $\geq$ 9; y is reduced to y - 9 or now y = 4. Continuing, 4 + 7 = 11, so now y = 2 and 2 + 1 = 3.

If you reach the end of the number and you are left with y = 0, then the original number was divisible by 9; if you end up with a number between 1 and 8, then the original number was not divisible by 9

Here's another example: 834174189

a) 83417418 (after crossing out all the nines)

b) $8 + 3 = 11 \rightarrow 2 + 4 + 1 + 7 = 14 \rightarrow 5 + 4 = 9 \rightarrow 0 + 1 + 8 = 9 \rightarrow 0$
The number is divisible by nine because y = 0.

Once again, how can you determine whether the wizard is right or wrong?

References:

[1] J. Brown and R. Rubinstein:
"Recursive functional programming for students in the Humanities and Social Sciences"
ICS Technical Report #27, 1973,
UC IRVINE

[2] E. Feigenbaum (in "Computers and Thought", p 297-310):
"The simulation of verbal learning behaviour"

[3] B. Raphael (in Minsky: Semantic Information Processing, p 33-145):
"SIR: A computer program for semantic information retrieval"

[4] K. Wales, D. Reidlinger, G. Fischer:
"Eliza question/answering system"
Final project for CS 522, april 1972, UBC, Vancouver

Note: The two plots at the first and last page were produced by two of the high school students