A NEW PEDAGOGICAL APPROACH TO TEACHING PROBLEM SOLVING

by

SUSAN MARGARET CARROLL

B. A., Emporia State University, 1973

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

Approved by:

[illegible]

Major Professor

TABLE OF CONTENTS

Chapter I

INTRODUCTION

In this age of advanced technology, computers have become common pieces of equipment. They can be found in industry, medicine, education, and homes.

Along with the ever-increasing popularity of computers comes the burden of training people to program them. No longer is the general populace satisfied by having only a few computer experts; it now expects and demands enough knowledge about these machines to control them. Training people to write computer programs is the responsibility of computer scientists. Without proper guidance, the science of writing correct and understandable programs will be degraded to a level of incomprehensible code.

Good computer programs don't just happen--they must be developed. The developmental stage is commonly referred to as problem solving and is one of the most difficult areas of programming. It is during problem solving that the programmer must decide exactly what he intends to do and how he intends to do it. While this may sound simple, the fact is that most people may understand the problem but have no idea how to solve it. Therefore, the untrained programmer tends to write long, unstructured, and complicated programs. To overcome this situation, a proper problem-solving technique must be developed. Once the ability to solve computer problems is mastered, good computer programs will be produced.

I.I Background

In the past, problem solving was left to those who had the intuitive ability to solve problems. The traditional approach of teaching computer programming relied heavily on this intuitive ability because it only taught the syntax of a computer language. It totally neglected how to solve computer programs. While this method was acceptable for most of those with an analytical background, it was a miserable failure for those who did not. As more people from varied backgrounds became interested in computer programming, the ability to solve problems could no longer be left to the intuitive abilities of the individual but had to be taught.

A method of teaching computer programming was developed which included the concept of teaching problem solving instead of teaching only the syntax of a language as in the traditional approach. By combining these two subjects, it was hoped that even those people with a non-analytical background would be able to solve computer problems. Unfortunately, while this method did approach the area of problem solving, it often befuddled those for whom it was developed (Walker).

The latest method for teaching computer programming separates the topics of problem solving and language syntax. It is now an accepted fact that one must know how to solve the problem before he can write the computer program. But this method of teaching also can fail. Without a good problem-solving text or an instructor who is properly trained in the area, the latest approach to the teaching of computer programming will succumb to failure (Unger, Pattis).

I.II The Problem

As the demand for programming knowledge increases, the number of individuals who are qualified and willing to teach decreases. The result is large classes being taught by under-qualified instructors. There is no easy answer for this problem, but it is the desire of this author to help ease the situation by introducing a new method to teach problem solving.

I.III The Proposed Solution

Before a programmer can solve a computer problem, he must be aware of exactly what the problem is. Understanding the problem includes not only knowing what must be done but also knowing the surrounding circumstances. A method is needed which explains precisely what must be known before a programming problem can be solved.

Once the problem is understood, the next step is to produce a solution. Using the available facilities within computer languages, the programmer must interpret his thoughts into a computer program. The task is not an easy one, for it requires a good working relationship between the programmer and the programming concepts. However, once this relationship exists, the programmer is better prepared to produce a solution to a computer problem.

The Karel method is a robot-oriented approach to the teaching of problem solving. The method utilizes a mobile, programmable robot, called Karel, to teach problem solving through the use of computer language concepts.

Karel is a programmable robot who inhabits a great flat plane. He has five primitive instructions built into his vocabulary that allow him to move through his world. Like natural languages, Karel's language has a vocabulary, punctuation, and grammatical rules. The robot language also includes many of the concepts built into computer programming languages. Through the use of Karel's language, it is possible to teach the basic constructs of computer programming, instill a good problem-solving technique, and give the student an insight into computer language constraints.

The sequential thought process is one of the most crucial aspects of program development (Wirth). The student must learn how to think in a step-by-step fashion in order for the computer to perform the intended actions and produce the desired results. Rather than explain how the instructions must be arranged, the Karel method demonstrates how a robot program is executed. Simulation is the fundamental technique and explicitly shows the student how the instructions must be sequenced. Because the student can visualize Karel's movements, it is easier for him to comprehend why each step in a program is necessary. It is during simulation that the student learns that it is not what he means but, more importantly, what he says and when he says it.

The programs used during simulation also expose the student to other rudiments of programming. Reserved words, punctuation, and indentation are introduced so that the student can see the regimentation of computer languages. He is made fully cognizant of the fact that while reserved words and punctuation are governed by strict rules, indentation is solely for the benefit of the reader and does not affect program execution. After his exposure to the preliminary rules of grammar and punctuation, the student can build on this foundation as other language constructs are presented.

While programmers never intentionally put errors into their programs, they must acknowledge that errors do occur and be able to recognize them. Therefore, the Karel method defines and describes four basic categories of programming errors. The method also expounds the philosophy that programs should be assumed incorrect until proven otherwise. Throughout the course, the student is periodically reminded of this principle and urged to be on constant guard for violations of programming rules and special situations which could affect the correctability of his program.

Another important concept in computer programming is modularity (Ross). While other methods which teach problem solving often delay this concept, the Karel method conveys the idea of modularity as one of the fundamental issues. With only primitive commands to work with, the student is shown how to decompose a problem through stepwise refinement in order to construct a program. By introducing modularity as one of the primary concepts, the student learns to solve his programming problems in a way natural to his way of thought.

The computer language construct, block structuring, is discussed in conjunction with modularity. Because block structuring is integrally linked to several programming concepts, it is of vital importance that the student have a thorough understanding of why block structuring is necessary and when its use is required. Thus, block structuring is introduced and explained along with modularity.

To aid the student in his understanding of when block structuring is necessary, the idea of boxing is demonstrated. Boxing is defined as drawing boxes around the program components. After a program has been boxed, the student can easily see how the program will be executed and thus determine if block structuring is needed in order for the program to execute properly.

In addition to teaching how to solve computer problems correctly, the Karel method also stresses the importance of understandable programs. The student is informed that while modularity is of major importance, it is not enough. Descriptive names and indentation must also be included. To emphasize the value of understandability, the Karel method points out how writing understandable programs will benefit the programmer.

The remainder of the course is devoted to introducing various language constructs and their proper usage. The IF - THEN and IF - THEN - ELSE constructs are presented individually, and each is explained completely. Both forms of looping, iteration and conditional, are presented. In connection with these four constructs, the subjects of alternative conditions, negative conditions, nesting, and infinite looping are discussed. As a finale, the student is shown how Karel may be programmed to compute addition problems while using only the conditional and looping constructs.

I.IV Organization of Paper

This chapter introduced the problems encountered when teaching problem solving to beginning programming students and proposed a solution--the Karel approach. Chapter II describes the behavior objectives of the Karel method and explains why each of them is essential. The third chapter is a description of the work done toward the implementation of the Karel approach, and a discussion of the future work needed in this area is given in Chapter IV.

Chapter II

BEHAVIORAL OBJECTIVES

Knowing something and being able to teach it are two different things. As a programmer, I feel very comfortable with my knowledge of programming; however, I am very uncomfortable when it comes to teaching it.

Why? The answer is quite simple: I have been trained to perform the duties of a programmer, not those of a teacher. In fact, the more training I receive in my field, the more difficult it becomes for me to relate to the needs of the beginning student.

I should add that I am not alone. Most programming instructors become so involved in their fields that they think more about what they want to teach than they do about what they want their students to learn.

To be an effective teacher, one must determine exactly what it is that the student is to learn (Walbesser, Kurtz). For a computer programming course, the accepted objective has been to have the student learn to write computer programs. But this objective is too generalized, for it establishes no guidelines for the acceptability of the programs and completely ignores the issue of problem solving. It would be better to have many smaller objectives which establish specific goals.

The objectives for a computer programming course should be classified into two types: those which relate to problem solving and those which relate to programming. While problem solving and programming are integrally related, it is necessary to separate them during their instruction because problem solving pertains solely to the writing of generalized solutions for computer problems through the use of computer language concepts, and programming is the direct application of a specific language to the already formulated solution. Once this distinction has been made, it is relatively easy to set the objectives for each classification.

In problem solving, the emphasis is on how well the student learns to apply his knowledge rather than on simply learning the facts. For this reason, the objectives of problem solving must be defined as behavioral objectives. The behavioral objectives express what the student should be able to do after he has completed his instruction in problem solving.

II.I Be Able to Understand the Problem

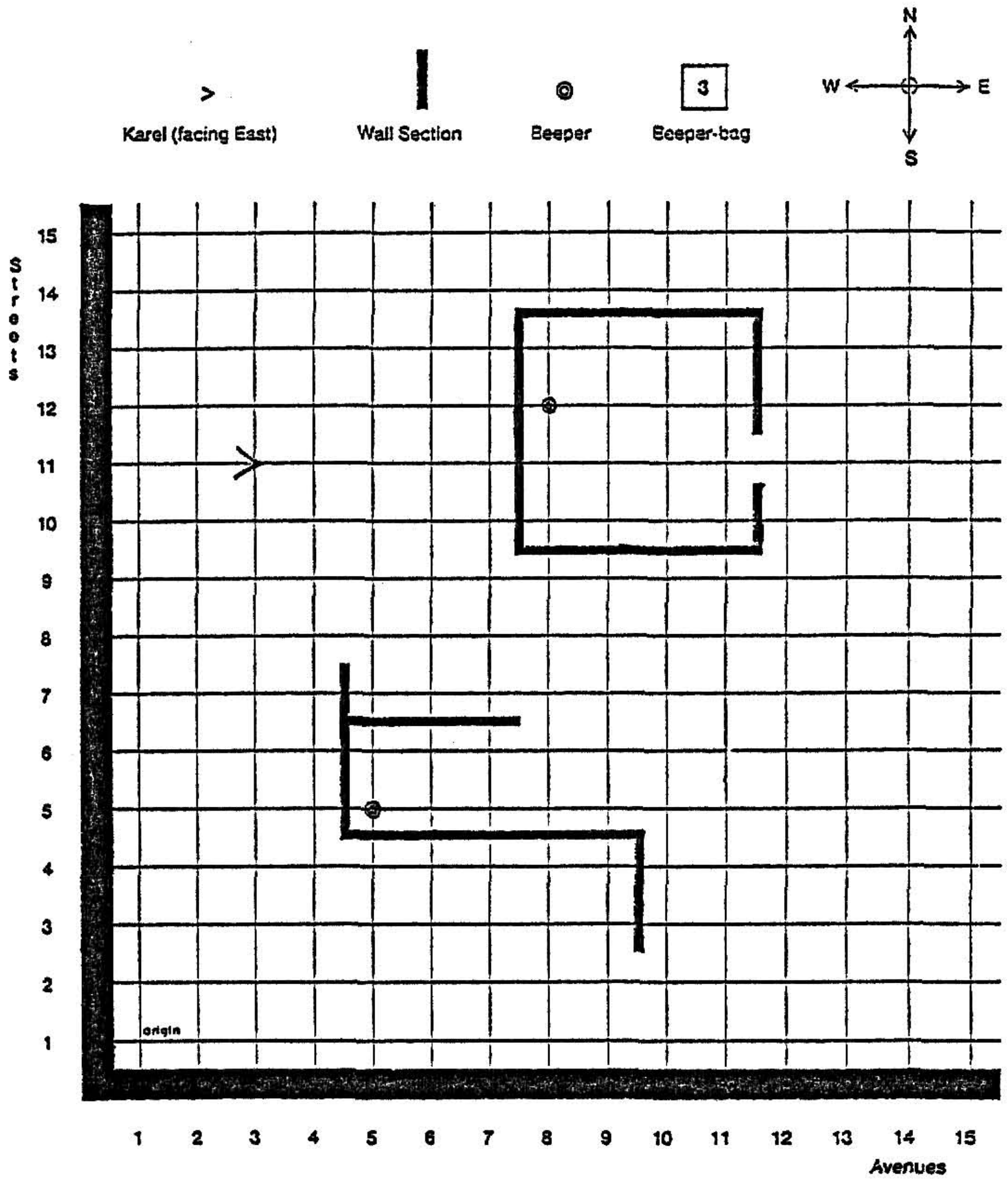
Although most people don't like to compute arithmetic problems, they dislike "word problems" even more. The principal reason for this dislike is that they do not understand "word problems." Unfortunately, all the problems in the real world, including computer problems, begin as "word problems"; to solve them, it is necessary to understand them.

The first behavioral objective of the Karel method is to have the student be able to understand robot problems. In order to accomplish this objective, it is first necessary to introduce Karel, his world, and the objects which may affect Karel's movement. After the student has become familiar with Karel and his world, it is then possible to define the necessary components for understanding a robot problem.

Figure 2-1 is a map illustrating the great, flat plane where Karel resides. The plane is bounded on the east and south by an impenetrable wall which restrains Karel from falling off. Criss-crossing Karel's world are horizontal streets and vertical avenues which extend indefinitely to the north and west. Both the streets and avenues are numbered, and the intersection of First Street and First Avenue is called the origin.

Figure 2-1

KAREL'S WORLD



Boundaries on South and West

Horizontal streets

Vertical avenues

Streets and avenues number consecutively

Wall sections

Beepers

Karel is not alone in his world. Two other objects, wall sections and beepers, may be found there. Wall sections are made from impenetrable metal and may be any desired length. "They are positioned sideways between adjacent street corners, effectively blocking Karel's direct path from one corner to the next. Beepers are small plastic balls that emit a quiet beeping noise. They are situated on street corners and can be picked up, stored, and put down by Karel" (Pattis). By using wall sections and beepers, separately or together, it is possible to alter Karel's world in order to create different robot problems.

Karel is a mobile robot which has the ability to perceive his immediate surroundings. He can move forward in the direction he is facing, and he can turn in place. Karel can see wall sections if they are within one-half block away, and he can hear beepers if he and the beeper are located at the same corner. To determine the direction he is facing, Karel has been given an internal compass. Because Karel must have the ability to transport beepers, he has been equipped with arms and a beeper bag. By reaching into the bag, Karel can determine if he has any beepers.

When the student is given a robot problem, he must be able to define it in terms of Karel and his world. The first step is to decide exactly what it is that Karel must do; this is known as the task. Once the task has been specified, the student must then be aware of the circumstances of the problem. An exact description of Karel's world, the situation, must be given and should include Karel's current position, the location and size of each wall section, and the location of each beeper. The situation before Karel starts (the initial situation) and the situation after Karel is finished (the final situation) will pictorially show the problem. Only after the student has identified the task and the initial situation has the problem been sufficiently defined.

II.II Be Able to Think Sequentially

When people give instructions for performing an action, they usually rely heavily on the ability of the one performing the action to do the obvious without being told. This reliance leads to the deletion of obvious instructions. Unfortunately, computers are not endowed with this ability, so it is necessary to train people to give instructions in a step-by-step fashion.

Teaching the student to think sequentially is the second behavioral objective of the Karel method. After the student learns Karel's basic commands, it is possible for him to write a robot program. Once written, the program can then be manually simulated, thus enabling the student to observe the result of each instruction. Through simulation, the student can visually see why every step in a robot program is necessary; consequently, this makes it easier for him to understand.

Karel has only five primitive instructions built into his vocabulary (see Figure 2-2), but these five primitive instructions allow him to move through his world and handle beepers. The move instruction makes Karel move forward one block in the direction he is facing. By issuing a turnleft instruction, Karel will turn 90 degrees to the left. A pickbeeper instruction will make Karel pick up one beeper, while a putbeeper instruction tells him to leave one. The stop instruction makes Karel stop and turn himself off; it is always the last instruction of every program.

By using only Karel's primitive commands, it is possible for the student to write a complete robot program. As he writes the program, the student must remember to include an instruction for every action Karel is

Figure 2-2

KAREL'S PRIMITIVE INSTRUCTIONS

move	Move Karel one block forward
turnleft	Turn Karel 90 degrees to the left
pickbeeper	Karel puts beeper in beeper bag
putbeeper	Karel puts beeper on corner
stop	Karel turns himself off

to perform, because Karel will do only as he is commanded. Simulation of the program will allow the student to experience the results of inadvertently excluding an instruction and will aid his development of the sequential thought process. Once the student has the ability to think sequentially, he is better prepared to write robot programs.

II.III Be Able to Use Punctuation and Reserved Words Correctly

Rules and regulations are common in today's life. They govern everything from how people talk to how they treat their fellow men. But, as the saying goes, rules are meant to be broken. It is very doubtful that someone has ever lived his entire life and not broken a few rules.

As in the real world, there are many rules for writing computer programs; however, the rules for programming are strictly enforced. The fact that the rules are always in effect is very difficult for most people to accept. Because they are used to simply bending the rules whenever necessary, they find it very perplexing to adhere strictly to the rules of programming.

Within the Karel method, there are rules covering the use of punctuation and reserved words. These rules are simple, yet they must be obeyed. The student has to learn not only what the rules are, but he must learn that he is required to abide by them as well.

The only punctuation allowed in a robot program is the semicolon. It "serves to separate instructions: each instruction is separated from the next instruction by a semicolon" (Pattis). A common misinterpretation of this rule is to end each instruction with a semicolon. Although these two rules sound equivalent, they are slightly different in that instructions followed by a reserved word should not be ended by a semicolon.

The 15 reserved words in the Karel language are listed in Figure 2-3. Four of these words are present in every robot program: BEGINPROGRAM, BEGINEXECUTION, ENDEXECUTION, and ENDPROGRAM; therefore, they are introduced along with the semicolon. The remaining 11 reserved words are introduced throughout the course as new language concepts are explained. As each of these reserved words is presented, the rules pertaining to its usage are explained.

Following the stringent rules of programming is very difficult for the beginning student; however, he must learn to do so. The ability to write correct robot programs requires the ability to use punctuation and reserved words accurately.

II.IV Be Able to Make Effective Use of Indentation

When writing in the English language, indentation is a common way of expressing a change in thought. Similarly, when writing a computer program, indentation should be used to indicate the transition from one logic component to another.

The main difference between the two uses of indentation is that only the first line is indented when writing in English; however, when writing a program, complete logic components are set in. This results in the left margin of an English text remaining fairly constant, while the margin of a program is extremely flexible.

The only area of confusion concerning indentation lies in how it affects program execution. Although indentation has no influence on how the program will be executed, it often deceives the programmer by showing him what he wants instead of what he has said. The Karel method explains this problem and warns the student to be on constant guard against it.

Figure 2-3

KAREL'S RESERVED WORDS

AS

BEGIN

BEGINEXECUTION

BEGINPROGRAM

DEFINE-NEW-INSTRUCTION

DO

ELSE

END

ENDEXECUTION

ENDPROGRAM

IF

ITERATE

THEN

TIMES

WHILE

Because there are no computer enforced rules pertaining to the correct usage of indentation, it is up to the student to develop his own technique. The Karel method stresses the importance of "adopting a lucid programming style" (Pattis) and illustrates good indentation form. Using these examples as a guide, the student is then free to develop his own technique.

II.V Be Able to Recognize and Correct Programming Errors

"Programming requires an inhuman amount of precision, and although errors should not occur in principle, they occur abundantly in practice" (Pattis). Because errors are inevitable, it is necessary to be able to locate and correct them.

The Karel method classifies all possible errors into the four basic groups listed in Figure 2-4. Lexical errors are those errors which occur when Karel is confronted with a word or form of punctuation that he does not recognize. Syntactic errors occur when incorrect grammar or inaccurate punctuation is used. Both of these error types will be detected by Karel before he tries to execute the program. If an instruction is lexically and syntactically correct but Karel cannot perform the action, an execution error will result. Karel will detect this type of error and turn himself off prematurely. The last class of errors is the most insidious, because Karel cannot detect them. Intent errors occur when Karel successfully completes the program, but he does not perform the desired task.

Once an error is detected, it must be corrected. Both lexical and syntax errors can be rectified by simply applying the rules of programming. These rules pertain to the correct usage of reserved words, punctuation, and vocabulary. Execution and intent errors are more difficult to correct

Figure 2-4

KAREL'S ERROR GROUPS

Lexical	Word not in Karel's vocabulary
Syntactic	Incorrect grammar or inaccurate punctuation
Execution	Karel is unable to execute an instruction
Intent	Karel completes program but does not complete task

because they require the use of logic. In order to determine exactly where the program went wrong, it is necessary to simulate Karel's actions. After the location has been discovered, it is then necessary to rewrite that segment of the program.

To be a competent programmer requires the ability to recognize and correct each type of error. The Karel method appreciates this and stresses that all programs should be considered guilty of being incorrect until proven otherwise. Only after the student has adopted this belief and become proficient at finding and correcting errors will it then be possible for him to be considered as having an adequate knowledge of programming.

II.VI Be Able to Write Modular Programs

As stated previously, it is necessary, when writing computer programs, to give detailed instructions. However, most people do not think in such detail; instead, they tend to think in terms of large, compound instructions which they must then break down. Because it is more natural to think in the large, the computer concept of modularity was developed. By applying this concept, it is possible for people to think in a way natural for them and then break down their thoughts into the necessary detail required by the computer.

To implement the idea of modularity into the Karel language, a new construct is introduced:

```
DEFINE-NEW-INSTRUCTION <newname>
AS
          <instruction>
```

The new instruction name can then express the large, compound instruction desired, while the instruction gives the necessary detailed instructions required by Karel.

The rules governing the use of the DEFINE-NEW-INSTRUCTION construct are very simple. They are:

- The <newname> can be any word or set of hyphenated words except for already used names and reserved words.
- The <instruction> must contain only one instruction, and it must be one that Karel understands.
- All instruction definitions must be placed between the words BEGINPROGRAM and BEGINEXECUTION.
- Definitions must be separated by semicolons.

While these rules are not very complicated, they must be obeyed.

As the student learns to use the DEFINE-NEW-INSTRUCTION construct, he is encouraged to write his programs using only new instructions. These new instructions are then defined in terms of either primitive commands or other new instructions. This defining process must be repeated until all new instructions have been defined. After all the new instructions have been defined, the program must be organized according to the precedence rule for new instructions. By decomposing the problem in this fashion, the concept of stepwise refinement is introduced.

Because the Karel method teaches the concept of modularity before the student has had much practice in writing long programs, the student learns to break up his program, through stepwise refinement, in a way natural for him.

II.VII Be Able to Use Block Structuring When Necessary

When using the DEFINE-NEW-INSTRUCTION construct, it is obvious that giving one instruction a new name is of minimal value. Thus, the concept of block structuring is necessary, for it allows the creation of "one big instruction from a sequence of smaller ones" (Pattis). By using block structuring in combination with the new instruction definitions, it is possible to define a new instruction as a set of instructions.

The concept of block structuring has been included in the Karel language. It is accomplished by enclosing a sequence of instructions between the reserved words BEGIN and END. Because BEGIN and END are reserved words, they are not separated from other reserved words or instructions by a semicolon. Once a BEGIN - END block is being executed, all the instructions contained in the block will be executed unless there is an error. This method of block structuring is simple to understand, yet general enough to use with the complex instructions in the programming.

To use block structuring correctly, it is necessary for the student to not only know the syntax of the BEGIN - END block, he must also understand when its use is required. "The fundamental property of a BEGIN - END block is that Karel understands the entire block to represent one instruction" (Pattis). If there is only one instruction within the block, the block is still correct, although its use is superfluous. If the set of instructions is to be understood as one and the words BEGIN and END are omitted, Karel will understand each instruction separately and will not perform as desired. In order for the student to have a sufficient knowledge of block structuring, he must recognize both of these conditions and be able to determine when a BEGIN - END block is essential.

II.VIII Be Able to Box a Program

One of the most common mistakes of programming is the omission of a necessary BEGIN - END block. Although designed as a programming aid, indentation will often conceal this mistake by making the program look correct. Because the computer does only as instructed and does not comprehend indentation, it is necessary for the programmer to understand how the machine will interpret his program. By breaking the program into components the computer can recognize, the programmer will be able to detect any missing BEGIN - END blocks.

The Karel method defines boxing as the operation of drawing boxes around every program unit. A unit may be an instruction, any type of BEGIN - END block, or a new instruction definition. "The main geometric property of boxing is that boxes are either one inside the other or are adjacent; boxes never overlap" (Pattis).

To box a program, the primitive instructions are boxed first, and then the units that enclose them are boxed. In this way, larger units will be built out of smaller ones. Boxing should be started at the beginning of the program, with the largest possible units being boxed before proceeding further. Figure 2-5 illustrates a program which has been boxed.

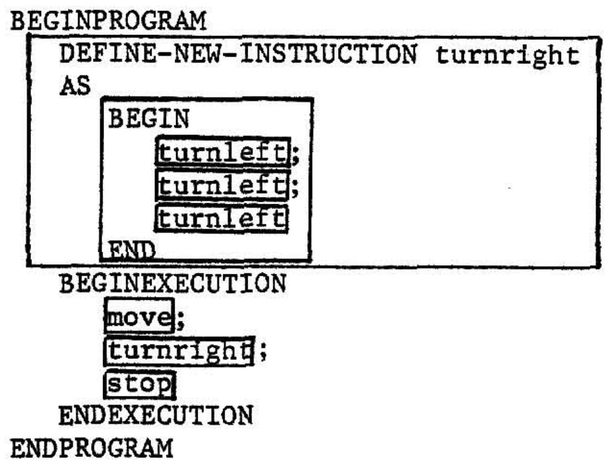
The use of boxing will aid the student in his quest to write correct programs. Because omitting a BEGIN - END block may cause either a syntax or intent error, it is necessary for the student to be able to detect a missing block. By allowing him to see how the program will be executed, boxing will help the student discover when a BEGIN - END block is necessary.

II.IX Be Able to Write Understandable Programs

There are three criteria for computer programs: the program runs, the program does as intended, and the program is understandable. The first two requirements are obviously important, for a program which does not work or does not perform the desired task is of negligible value. The

Figure 2-5

A BOXED PROGRAM



```
BEGINPROGRAM
   DEFINE-NEW-INSTRUCTION turnright
   AS
      BEGIN
         turnleft;
         turnleft;
         turnleft
      END
   BEGINEXECUTION
      move;
      turnright;
      stop
   ENDEXECUTION
ENDPROGRAM
```

significance of the third requirement, the program is understandable, is not as obvious; however, without it, fulfilling the first two criteria becomes extremely difficult.

If it is assumed that all programs initially contain errors, then it becomes necessary to be able to correct these errors. To correct a program with errors, one must be able to understand the program. Thus, the criteria of understandability is of an equal or greater importance than those concerning the correctness of a computer program.

The Karel method emphasizes the importance of understandability and gives two requisites for understandable programs: the program should be "the simple composition of easily understandable parts" (Pattis), and new instruction names should "provide a description of how the program accomplishes the task" (Pattis). The method explains that decomposing "a program into separate instructions, even if an instruction is executed once," (Pattis) will make the program more structured and more understandable. But, decomposing a program is not enough. The new instructions must be given meaningful names in order for the programmer to remember what the instructions do. If both of these requisites are met, the program will be understandable.

To entice the student into writing understandable programs, the Karel method explains how understandable programs will help him. When shown the same program written once in an understandable, structured fashion and once using only sequential primitive instructions, the student can observe that the understandable program is shorter, easier to check for correctness, easier to correct if necessary, and easier to modify if necessary. Because each of these characteristics will benefit the student, he can realize how understandable programs will be of value to him.

II.X Be Able to Write Conditional Statements

The conditional statement is an extremely valuable tool for a computer programmer, for it is the means which allows him to make decisions during program execution. According to the decision, based on the current situation, the programmer can either have instructions performed or not performed. This type of control is of great use to a programmer.

The Karel language has the facilities to implement two types of conditional statements. The two forms available are:

```
IF <test>                          IF <test>
   THEN                  and          THEN
     <instruction>                      <instruction 1>
                                      ELSE
                                        <instruction 2>
```

Both instructions perform a test on the condition and execute the instruction following the THEN only when the condition is true. The IF - THEN - ELSE instruction, however, allows the programmer to specify when action should be performed if the test fails. As in new instruction definitions, only one instruction is allowed in the THEN and ELSE clauses, but this is again overcome by the use of block structuring.

The conditions which Karel can test are limited to those which he can perceive through his sensory modes. These conditions are listed in Figure 2-6. "Each condition is represented in both its positive and negative forms" (Pattis); thus, when the positive condition is true, the negative form is false.

To determine when a test is true, certain conditions must be satisfied. "Karel's front, left, and right are respectively clear if there are no walls between Karel and the next corner over in the direction he is commanded to look. The next-to-a-beeper test is true when Karel is on the same corner

Figure 2-6

CONDITIONS WHICH KAREL CAN TEST

front-is-clear	front-is-blocked
left-is-clear	left-is-blocked
right-is-clear	right-is-blocked
next-to-a-beeper	not-next-to-a-beeper
facing-north	not-facing-north
facing-south	not-facing-south
facing-east	not-facing-east
facing-west	not-facing-west
any-beepers-in-beeper-bag	no-beepers-in-beeper-bag

as one or more beepers" (Pattis). Because the beeper bag is soundproof, Karel cannot hear the beepers he is carrying; therefore, these beepers will not affect the test. When testing the direction he is facing, Karel consults his internal compass. To test whether he has any beepers in his beeper bag, Karel reaches into his bag; if any beepers are present, Karel will always find them.

Without the conditional statement, programs must be written for specific situations; however, by using conditional statements, the program may be generalized in order to handle many similar situations. It is vitally important that the student learn to use the IF - THEN and IF - THEN - ELSE constructs correctly, for it is these constructs which allow the student to write versatile programs.

II.XI Be Able to Nest Conditional Statements

The concept of nesting, putting one or more conditional statements inside the THEN or ELSE clause of another conditional statement, is an important concept for the beginning programmer to learn. This concept allows the programmer to make decisions based on compound conditions. By testing one situation at a time, the programmer gains the ability of easily subdividing possible alternative situations, thus allowing him to handle each situation appropriately.

The nesting of conditional statements within the Karel language requires no new evaluation rules; however, a close adherence to the established rules is required. Because it is easy to lose track of exactly what is happening when the statements are nested several levels deep, it is necessary to understand precisely how the program will be executed; thus, the concept of boxing is used to explain to the student and aid his learning of how to interpret nested conditional statements. By boxing the nested statements, it is possible to insure that the rules pertaining to conditional statements and block structuring are being followed properly and that the statements are being evaluated as desired.

One major problem of nesting conditional statements concerns the dangling ELSE. The Karel method defines the problem as the nesting of an IF - THEN statement with an IF - THEN - ELSE statement. To solve the problem, the Karel method introduces a new rule of grammar which states that the first ELSE encountered must be matched with the last available IF - THEN statement. In the event that the ELSE should be matched with a previous conditional statement, block structuring is used as demonstrated in Figure 2-7.

The ability to test several conditions at one time is extremely useful because it permits the handling of various similar situations accordingly. The student must learn to use nested conditional statements properly in order to test more than one condition at a time.

II.XII Be Able to Use Iterate Loops

When writing a program, it is sometimes necessary to repeat an instruction, or a set of instructions, a fixed number of times. Of course, it is possible to accomplish this by writing out the instruction the number of times desired, but this is not a desirable method because it makes the program longer and allows the possibility of having the instruction repeated too few or too many times. Instead, the concept of iterative looping should be used. This concept permits the repeating of an instruction a specified number of times while requiring it to be written only once.

Figure 2-7

NESTED CONDITIONAL STATEMENTS

```
IF <test 1>
    THEN
       BEGIN
         IF <test 2>
             THEN
               <instruction 1>
       END
    ELSE
     <instruction 2>
```

The Karel language implements the concept of the iterate loop through the use of the ITERATE instruction. The general form is:

```
ITERATE <positive - integer> TIMES
    <instruction>
```

The <positive - integer> informs KAREL how many times to execute the one instruction that replaces <instruction>. If more than one instruction is to be repeated, block structuring must be used.

Similar to conditional statements, ITERATE instructions may be nested. When nesting, the innermost ITERATE instruction is executed in its entirety for each single execution of an outer ITERATE instruction.

While it is true that any program written using the ITERATE instruction could be written without it, the ITERATE instruction is of great benefit to a programmer and should be learned. By using the ITERATE instruction properly, the student will be making his program more structured and will decrease the chance for error.

II.XIII Be Able to Use Conditional Loops

While the use of the conditional statement allows programs to become more generalized and iteration simplifies the repetition of an instruction, it is the combination of these two features that produces one of the most powerful concepts in computer programming, the conditional loop. The conditional loop tests a condition in the current situation and, as long as the test remains true, repeats an instruction.

Within the Karel language, the conditional loop has the general form of:

```
WHILE <test> DO
  <instruction>
```

The <test> may be any of those stated for conditional statements listed in Figure 2-6. The <instruction> may be any single instruction which Karel understands. If more than one instruction is desired, block structuring is necessary.

To execute the WHILE statement, Karel tests the condition; if it is true, he executes the instruction. Karel then re-executes the entire WHILE instruction until the test becomes false. When the test is false, Karel is done with the WHILE instruction and continues with the statements following the entire WHILE loop. If the test is false the first time through the WHILE instruction, Karel does not execute the instruction inside the loop.

By using the WHILE instruction, the student will be able to solve problems which require repetition until a condition is met. The use of the conditional loop greatly increases the scope of programming problems which may be solved; therefore, the student should become familiar with it.

II.XIV Be Able to Avoid Infinite Loops

The ability to repeat an instruction as long as a condition remains true has its advantages, but it also adds to the programmer's responsibilities. The programmer must insure that the condition will eventually become false by the execution of the instruction inside the conditional loop. If he does not and the condition forever remains true, the programmer has written an infinite loop. An infinite loop is a loop that gets executed an infinite number of times. It is the programmer's responsibility to guarantee that every conditional loop he writes will terminate.

It is possible to write an infinite loop within the Karel language by the improper usage of a WHILE instruction. The student must be aware of infinite looping and is expected to be able to avoid it.

II.XV Be Able to Combine Computer Concepts to Produce Correct Solutions

After the student has been exposed to the different language constructs, he must learn to use them separately and together in order to derive solutions for computer problems. A frequent source of trouble involves IF instructions that are within the body of a WHILE loop. Because both instructions perform tests and have similar execution rules, the combination of the conditional statement and the conditional loop may cause confusion. Only through exposure to correct examples and practice will the student be able to understand how to combine the language constructs.

II.XVI Be Able to Test the Solution

It would be gratifying if every program written would work properly the first time; however, this is not the case. Even when the program is written through the use of stepwise refinement and meticulous care is taken when writing the constructs, there is always the possibility that the program will not perform the desired task. To insure that the program does as intended, it is necessary to test the program.

Usually, programming is guided by a few sample situations that seem to include all the different possibilities for the particular problem. But, as more is learned about the problem, previously overlooked situations might be disclosed. These special cases may cause trouble which could result in the changing of the program.

The student must learn to search for the special cases. "Good programmers become skilled at . . . finding dangerous situations that interfere with programs accomplishing their tasks" (Pattis).

Chapter III

RESULTS

The problem of teaching problem solving is only partially resolved after this approach has been accepted. The complete solution must also include a description of how the material should be introduced to the students. The Karel approach to teaching problem solving is a new technique and unfamiliar to instructors. For this reason, it is necessary to have not only a text but other materials as well.

III.I The Text

The idea of using a robot as a means to teach problem solving is not a new one. There are numerous texts which use the robot concept to illustrate computer problem-solving techniques.

In 1981, Karel, the Robot: A Gentle Introduction to the Art of Programming by Richard E. Pattis was published. This text utilizes the robot concept to teach problem solving in a step-by-step fashion. The robot language used is similar to PASCAL in grammar and punctuation. The software engineering goals of modifiability and understandability are emphasized. Incorporated throughout the text are the principles of modularity, confirmability, and completeness. It was a copy of the pre-published manuscript for this text that was used in the creation of Appendices A through C.

III.II The Lesson Plans

To aid instructors who are unfamiliar with the Karel approach, the material contained in Karel, the Robot has been outlined. Each major section in the outline is an individual lesson, although two or more small sections may be taught during one class period. The lesson plans are in Appendix A.

III.III The Transparencies

To further aid in the instruction of the Karel approach, a complete set of designs for transparencies has also been included. The transparency designs, found in Appendix B, follow the same sequence as the lesson plans and should be used to illustrate them.

III.IV The Script

So that the instructor does not simply read the transparencies to the students, a script has been provided in Appendix C. Each numbered section within the script explains the information seen on the corresponding transparency.

Chapter IV

CONCLUSION AND FUTURE WORK

Good problem-solving techniques are crucial in the development of computer programs. In the past, problem solving was left to those who had the intuitive ability to solve problems; however, it is now a recognized fact that the ability to solve problems can no longer be left to intuition but must be taught. Unfortunately, as the demand for programming knowledge increases, the number of qualified instructors decreases. Therefore, it has become necessary to find an easily understandable method to teach problem solving.

IV.I Conclusion

A new method for the teaching of problem solving has been introduced. This method utilizes a mobile, programmable robot called Karel to teach problem solving through the use of computer language constructs. The robot language has been broken down into its most basic parts, and each element is introduced separately in a logical order. By using the Karel approach, it is possible to teach problem-solving techniques, introduce the basic constructs of computer programming, and expose the student to the regimentation required in computer programming.

IV.II Testing

The Karel method has been used both at Stanford University and in an off-campus class offered by Kansas State University at Fort Riley. In both instances, it has been a highly successful method. The classes, however, were taught by instructors familiar with the Karel approach and who did not see any of the material found in Appendices A, B, and C. Therefore, before the Karel method is fully implemented through the use of the transparencies and script, it should be tested within a controlled environment for a minimum of one semester. After it has been tested and revised, it can then be used to teach the fundamentals of computer programming to all beginning students.

IV.III Future Work

Once the Karel method is fully implemented, it would be desirable for the method to be computerized. Of course, computerization would entail several costs but would solve the problem of too few instructors and would allow students to learn at their own speed.

Bibliography

Kurtz, Edwin B., Jr. "Help Stamp Out Non-Behavioral Objectives," The Science Teacher, Vol. 32, Number 1, January, 1965.

Pattis, Richard E. Karel, the Robot: A Gentle Introduction to the Art of Programming, John Wiley and Sons, Inc., 1981.

Ross, Douglass T.; Goodenough, John B.; and Irvine, C. A. "Software Engineering: Process, Principles, and Goals," IEEE Transactions on Software Engineering, Vol. SE-2, Number 4, December, 1976.

Unger, E. A., and Ahmed, Nasir. Computer Science Fundamentals, Charles E. Merrill Publishing Co., 1979.

Walbesser, Henry H. Constructing Behavior Objectives, The Bureau of Educational Research and Field Services, 1970.

Walker, Terry M. Introduction to Computer Science: An Interdisciplinary Approach, Allyn and Bacon, Inc., 1972.

Wirth, Niklaus. "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14, Number 4, April, 1971.

Appendix A

LESSON PLANS

I. Meet Karel, the Robot

A. Karel's World

1. Grid
   a. Boundaries - south and west, solid neutronium
   b. Streets - horizontal
   c. Avenues - vertical
   d. Origin - intersection of First Street and First Avenue

2. Wall Sections - neutronium obstacles which may be positioned sideways between adjacent street corners to block a path

3. Beepers - small plastic bells that emit a quiet beeping noise

B. Karel's Capabilities

1. Move
   a. Go forward in direction facing
   b. Turn in place

2. See walls if withinone-half block

3. Hear beepers if he and beeper are at same corner

4. Determine the direction he is facing

5. Pick up beepers

6. Put down beepers

7. Determine if he has a beeper

8. Receive set of instructions

9. Memorize set of instructions

10. Carry out set of instructions

C. Karel's Situations

1. Situation - Complete description of Karel's world
   a. Current position of Karel
   b. Location and size of each wall section
   c. Location of each beeper - including those in Karel's beeper bag

2. Initial situation - situation before Karel starts

3. Final situation - situation after Karel turns himself off

II. Learn How to Perform Tasks

A. Define Terms

1. Task - something we want done

2. Program - detailed set of instructions which explain how to perform the task

3. Programming language - language in which a program is written

B. Explain Method

1. Robot named Karel
   a. Limited world
   b. Limited set of instructions

2. Karel performs tasks

III. Understand Karel's Primitive Instructions

A. stop - Informs Karel that he has completed his task

1. Turns himself off

2. Will not resume action until restarted by another task

3. Must be the last instruction in every program

B. move - Causes Karel to go forward one block in the direction he is facing

1. Continues to face the same direction

2. Will not move if he sees a wall section or boundary wall between himself and the corner he is moving toward

C. turnleft - Causes Karel to turn 90 degrees to the left

1. Remains at the same corner

2. Can always turn

D. pickbeeper - Causes Karel to pick up a beeper at the corner where he is standing and put it in his beeper bag

1. Continues to face the same direction

2. Remains at the same corner

3. Picks up one and only one beeper even if there are several

4. Will not pick up a beeper if none are present

E. putbeeper - Causes Karel to take a beeper from his beeper bag and place it at the corner

1. Continues to face the same direction

2. Remains at the same corner

3. Will not deposit a beeper if he has none

IV. See How Karel Works

- A. Start Karel
  1. Set up initial situation
  2. Press Karel's start button
- B. Read Karel the Program
  1. Include each word
  2. Include all punctuation
- C. Karel Executes the Program
  1. Karel starts only after hearing ENDPROGRAM
  2. Karel continues until he turns himself off

V. Simulate Karel's Actions to Verify Program

A. Karel Executes Programs

1. Execute instructions between the words BEGINEXECUTION and ENDEXECUTION

2. Executes instructions sequentially (top to bottom)

3. Executes all instructions - does not leave any out

4. Continues until he turns himself off
   a. stop instruction
   b. error shutoff

B. Karel Executes Instructions

1. Performs action if possible

2. Shuts off if not possible

VI. Pay Attention to Grammar and Punctuation Rules

A. Karel's Vocabulary

1. Instructions - cause Karel to perform an action
   a. Instructions will always be written in lowercase letters

2. Reserved words - delimit different portions of the program
   a. Programs must start with BEGINPROGRAM and end with ENDPROGRAM
   b. Programs must contain BEGINEXECUTION and ENDEXECUTION
   c. Matching pairs of BEGIN-END
   d. Reserved words will always be written in uppercase letters

B. Karel's Punctuation

1. Semicolon - separates instructions
   a. Reserved words are not separated by semicolons
   b. Reserved words are not separated from instructions by semicolons

C. Karel's Indentation

1. Programs should be easily readable
   a. Karel cannot see indentation

VII. Don't Ask The Impossible

A. Error Shutoff - Karel Turns Himself Off When He Cannot Execute an Instruction

1. Move blocked by a wall section or boundary

2. pickbeeper - when Karel is not next to one

3. putbeeper - when Karel's beeper bag is empty

VIII. Know Your Enemies

A. Correct Program

1. Inhuman amount of precision

2. No errors "in principle"

3. Abundant errors "in practice"

B. Four Categories of Errors

1. Lexical error - Karel hears a word not in his vocabulary and turns himself off

2. Syntactic error - Karel hears incorrect grammar or inaccurate punctuation and turns himself off

3. Execution error - Karel is unable to execute an instruction and turns himself off

4. Intent error - Karel successfully executes the program but does not successfully complete his task

Remember: Karel does not know what you _want_ him to do; all he knows is what you _tell_ him to do.

IX. Increase Karel's Vocabulary

A. Why Bother?

1. People think in one language and must program in another
   a. Turnright - requires three TURNLEFT instructions
   b. Move ten miles - requires 80 MOVE instructions

2. Programs are shorter
   a. Easier to write
   b. Easier to understand

3. It is better to have Karel learn new definitions than have us be slaves of the machine

B. It Is Easy

1. Give Karel a dictionary of useful instruction names and their definitions
   a. Definitions are built from simpler instructions which Karel already understands
   b. The first definitions would be built using primitive instructions

2. The definition mechanism defines a new instruction to have the same meaning as one other instruction
   a. Two new reserved words: DEFINE-NEW-INSTRUCTION, AS
   b. DEFINE-NEW-INSTRUCTION signals Karel that a new instruction is being defined
   c. AS separates the new instruction name from its definition
   d. Replace <new name> with the name of the new instruction
      1) Any word in lowercase letters or numbers
      2) May be hyphenated when a multiple word name is desired
      3) Cannot already be an instruction name
      4) Cannot be a reserved word
   e. Replace <instruction> with the definition of <new name>
      1) Any single instruction that Karel understands
      2) A primitive instruction
      3) A previously defined new instruction

3. The single instruction restriction on <instruction> is severe but useful
   a. Allows Karel to understand instructions in more than one language

X. Build Complex Commands

A. Block Structuring

1. Place sequence of instructions between reserved words BEGIN and END

2. Makes one big instruction out of a sequence of smaller ones

3. The sequence of instructions should be indented

B. Grammar Rules

1. Instructions inside the BEGIN-END block must be separated by semicolons

2. The reserved word BEGIN is <u>not</u> followed by a semicolon

3. There is no semicolon separating the last instruction from the reserved word END

4. May put from one to as many as needed instructions inside the BEGIN-END block

5. A BEGIN-END block is executed by executing the instructions sequentially

6. Once a block is being executed, all the instructions within the block will be executed unless Karel turns himself off

C. Fundamental Property

1. Karel understands the entire block to represent one instruction
   a. Define turnright
   b. Define move-mile

D. Meaning and Correctness

1. What's in a name?

2. Karel's only concept of a new instruction is the definition given to him
   a. Karel doesn't understand what the name means for him to do
   b. Karel does understand the definition and executes it

3. The name of a new instruction should specify "what" the instruction does, while the definition specifies "how"

XI. Give Karel the Definitions of New Instructions

A. Location

1. All definitions go between BEGINPROGRAM and BEGINEXECUTION

2. Nothing else can go there

3. This portion of the program is Karel's dictionary

B. Order

1. Each instruction must be defined before it is used

2. A lexical error will occur if an instruction is not defined prior to its use

C. Punctuation

1. The definitions of new instructions must be separated by semicolons

2. A semicolon must separate the last definition from the reserved word BEGINEXECUTION

D. Limitation

1. Karel's vocabulary reverts back to primitive instructions and reserved words each time he is started

XII. Learn to Box

A. Define Terms

1. Unit
    a. An instruction
    b. Any type of BEGIN-END block
    c. An entry in the dictionary

2. Boxing - drawing boxes around every unit in the program

B. Observations About Units and Boxing

1. The definition of a new instruction is the first box after the AS

2. Semicolons are placed between every pair of adjacent units

3. Units may be <u>nested</u> inside other units

C. Geometric Properties of Boxing

1. Boxes may be one inside the other

2. Boxes may be adjacent to one another

3. Boxes may not overlap each other

D. How to Box

1. Box primitive instructions first and work outward

2. Start at the beginning of the program

3. Build biggest possible boxes before starting boxes further down

XIII. Karel Finds Errors

A. Checks for Lexical and Syntax Errors While Program Is Being Read

B. Breaks Program Into Units to Check Grammar and Punctuation

1. Karel only knows program read and does not know indentation

2. Karel uses boxing to find syntax errors

XIV. Construct A Program

A. Stepwise Refinement - The Method to Use

1. Programs will be concise

2. Programs will be simple to read

3. Programs will be easy to understand

B. Steps to Follow

1. Write the sequence of instructions in the BEGINEXECUTION-ENDEXECUTION block using any instruction names desired

2. Write the definitions of the new instructions used in the BEGINEXECUTION-ENDEXECUTION block

3. Write the definitions of the new instructions used in BEGIN-END block of a new instruction definition

4. Repeat step three until all instructions are defined

C. Review Stepwise Refinement

1. Understand task

2. Break task into smaller, easier to understand, independent subtasks

3. Solve subtasks, thus solving main task

D. Verify Program

1. Stepwise refinement does not guarantee a correct program

2. "Programs are guilty of being wrong until they are proven correct"

XV. Write Understandable Programs

A. Understandability is as Important as Correctness

1. Understandable programs are easier to correct

2. Good programmers are separated from bad ones by their ability to write understandable programs

B. Criteria for Understandable Programs

1. "A good program is the simple composition of easily understandable parts"

2. Properly named instructions provide a description of how the program accomplishes the task
   a. After an instruction has been proven correct, only need to remember what it does

C. How Understandability Helps Us

1. Easier to read

2. Easier to verify

3. Easier to correct

4. Easier to modify

5. Easy to change a complete program into one instruction

XVI. Ask A Question

A. Karel Understands Two Similar IF Instructions

1. He tests his environment

2. He then executes the appropriate instruction depending upon the result of the test

B. The IF-THEN Instruction

1. Two new reserved words, IF and THEN

2. IF signals Karel that an IF instruction is present

3. The <instruction> is known as the THEN clause and is separated from <test> by the word THEN

4. The THEN clause is an instruction nested inside the IF instruction

C. How It Works

1. Karel checks <test> in the current situation

2. If <test> is true, Karel executes <instruction>

3. If <test> is false, Karel does not execute <instruction>

4. Karel continues executing the program

D. The Conditions Karel Can Test

1. front-is-clear . . . front-is-blocked

2. left-is-clear . . . left-is-blocked

3. right-is-clear . . . right-is-blocked

4. next-to-a-beeper . . . not-next-to-a-beeper

5. facing-north . . . not-facing-north

6. facing-south . . . not-facing-south

7. facing-east . . . not-facing-east

8. facing-west . . . not-facing-west

9. any-beepers-in-beeper-bag . . . no-beepers-in-beeper-bag

E. What Is True?

1. Karel's front, left, and right are respectively clear if there are no walls between Karel and the next corner in the direction he is commanded to look

2. Karel's next-to-a-beeper test is true when he and a beeper are located at the same corner. Any beepers in his beeper bag do not affect this test.

3. Karel's internal compass tells him what direction he is facing

4. Karel determines if he has any beepers in his beeper bag by poking around in the bag

XVII. Look At The IF-THEN Instruction

A. The harvest-a-furrow Instruction

1. Always use a BEGIN-END block in definitions

2. Know when necessary or redundant

B. The turn-around-only-if-blocked Instruction

1. Omit BEGIN-END block in THEN clause
   a. Karel not fooled by indentation
   b. Causes subtle intent error

2. Requires effort to understand meaning of an instruction
   a. Do not passively test an instruction
   b. Try to find special situations where the instruction might fail

XVIII. Box The IF-THEN Instruction

A. Steps To Follow

1. Box the instructions in the THEN clause

2. Box the nested THEN clause

3. Box the entire IF-THEN instruction

B. Pay Attention to Punctuation

1. Semicolons separating the IF-THEN instruction from other instructions

2. No semicolons after the last instruction in a BEGIN-END block

XIX. Do It Or Else

A. The IF-THEN-ELSE Instruction

1. ELSE - new reserved word
2. ELSE clause nested inside the IF instruction after the THEN clause
3. THEN and ELSE clauses are identically indented
4. No semicolon between <instruction 1> and ELSE

B. How It Works

1. Karel determines if <test> is true or false
2. If <test> is true, he executes <instruction 1>
3. If <test> is false, he executes <instruction 2>
4. Executes either <instruction 1> or <instruction 2>, but not both

XX. Look At The IF-THEN-ELSE Instruction

A. The race-stride Instruction

1. Equivalent IF-THEN-ELSE instructions
    a. Negate test
    b. Switch THEN and ELSE clauses
    c. More freedom

2. Suggested rule is to use the test that makes the THEN clause smaller
    a. Reader can see it is an IF-THEN-ELSE instruction
    b. Long THEN clause would put too much distance between the IF and the ELSE

3. Indentation helps

XXI. Box The IF-THEN-ELSE Instruction

A. Steps to Follow

1. Box the THEN clause

2. Box the ELSE clause

3. Box the entire IF-THEN-ELSE instruction

B. Possible Errors

1. Syntax error - BEGIN-END block omitted in THEN clause

2. Syntax error - semicolon between the THEN clause and the reserved word ELSE

3. Intent error - BEGIN-END block omitted in ELSE clause

C. Lessons to Learn

1. Tactical lesson - do not forget BEGIN-END blocks

2. Strategic lesson - the bigger an instruction, the more complicated it becomes

XXII. Observe Nested IF Instructions

A. IF Instructions Nested Inside a THEN or ELSE Clause of Another IF Instruction

1. No new evaluation rules

2. Closer attention to established rules required

3. Simulation is difficult

B. The Replant-Exactly-One Instruction

1. Notice boxing

2. Notice punctuation

3. Notice consecutive END instructions

C. Avoid Nesting More Than Two Levels Deep

D. Dangling ELSE Problem

1. Two IF-THEN instructions and one ELSE clause

2. ELSE clause boxed with most recent IF instruction

3. BEGIN-END block used to alter boxing

XXIII. Repeat That

A. Karel Has The Ability To Repeatedly Execute Any Instruction He Understands

B. The ITERATE Instruction

1. Shorthand notation to repeat another instruction a specified number of times

2. Two new reserved words, ITERATE and TIMES

3. The <positive-integer> informs Karel how many times to repeat <instruction>

4. The <instruction> is the body of the iterate instruction

5. New term ITERATE loop - instruction loops back and executes itself

C. How It Works

1. The body is executed the <positive-integer> number of times

D. How Many Times?

1. ITERATE instructions may be nested

2. The inner most ITERATE instruction is repeated the <positive-integer> number of times each time the outer one is repeated once

E. Box the ITERATE Instruction

1. Box the instructions in the body

2. Box the ITERATE instruction

3. If nested, start with inner most ITERATE instruction

XXIV. Learn WHILE, The Most Powerful Instruction

A. Programs Are Limited

1. Repeat instructions undetermined number of times

2. Need to repeat while a condition is true

B. The WHILE Instruction

1. Two new reserved words, WHILE and DO

2. WHILE starts the instruction

3. DO separates <tests> from the body of the WHILE instruction

C. How It Works

1. Karel checks if <test> is true or false

2. If <test> is true, Karel executes the body of the loop and then re-executes the entire WHILE instruction

3. If <test> is false, Karel does not execute the loop but continues execution at the first statement following the loop

D. A Formal Property of the WHILE Instruction

1. When Karel is finished executing the WHILE instruction, <test> is guaranteed to be false

2. If it is necessary for <test> to be true, write condition using <not-test>

XXV. Don't Write Infinite WHILE Loops

A. "An Infinite Loop Occurs When <Instruction> Does Not Cause Karel To Progress Toward His Goal"

1. WHILE instruction is the only one which has the ability to be executed indefinitely

2. A kind of intent error because Karel cannot detect it

XXVI. Know When To Perform The Test

A. Common Misconception

1. Karel checks <test> after each instruction is executed in the loop body

2. True only if loop body consists of one instruction

B. Correct Rule

1. Check <test> each time before all the instructions in the loop body are executed

2. Once in loop body, Karel is unaware of the <test>

3. <test> is rechecked only after the body is completely executed

XXVII. Learn To Repeat

A. Plan Ahead

1. Set up situation for repeating instruction

2. Be sure desired task is finished when loop terminates

3. Put special cases before loop for visibility

B. Use Block Structuring

1. Multiple instruction loop bodies need BEGIN-END blocks

2. Omitted BEGIN-END blocks cause errors
   a. Karel may recognize it as a syntactic error
   b. Most of the time, it will cause an intent error

C. IF Instructions In WHILE Loops

1. Both instructions perform tests and have similar execution rules

2. IF instructions in WHILE loops are easily simulated

3. Be careful with syntax
   a. WHILE instructions contain the reserved word DO
   b. IF instructions do not contain the reserved word DO
   c. Many programming errors are hard to find

XXVIII. See A Large Stepwise Refinement Programming Example

A. See How The Program Is Developed

1. Develop program in logical manner
2. Commit mistakes
3. Recognize mistakes
4. Rewrite program until correct

B. Task - Have Karel Escape From Any Rectangular Room With An Open Doorway Exactly One Block Wide

1. Difficult and elusive task
2. One possible initial situation
3. Beginning solution
4. Write definitions for new instructions

C. Beyond-The-Horizon Situation

1. Guided by a few sample situations that seem to cover all the facets of the problem
2. Uncover special trouble-causing cases
3. Good programmers become skilled at finding dangerous situations
4. Modify program segment where error occurs
5. Check for special trouble-causing cases
6. Repeat steps 4 and 5 until program is correct

D. An Invariant

1. A condition that must be true during the execution of a portion of a program
2. Karel's righthand side must be within one-half block of the room wall

E. Finish Program

1. Define last instructions
2. Assemble program

F. More Beyond-The-Horizon Situations

1. A very skinny room

2. A door in an unexpected place

3. Are there others where program fails?

XXIX. Graduate to Advanced Robot Programming

A. Two New Instructions

1. zig-nw moves Karel diagonally northwest

2. zig-se moves Karel diagonally southwest

3. Instructions will enable an easy solution to a novel set of beeper manipulation problems

B. The zig-nw Instruction

1. Simple definition

2. Eventually restrained by the western boundary wall

3. Precondition - Karel must be facing west

4. Precondition is invariant over instruction
   a. Karel is facing the same direction after the instruction ends as he was when he began the instruction

5. Must only be executed when Karel's front is clear

C. The zig-se Instruction

1. Simple definition

2. Eventually restrained by the southern boundary wall

3. Precondition - Karel must be facing south

4. Precondition is invariant over instruction

5. Must only be executed when Karel's front is clear

D. Task - Find A Beeper

1. Obvious solution won't work

2. Use a search pattern involving zig-nw and zag-se
   a. The search pattern will work

3. See the program which solves the task

XXX. Do Arithmetic

A. Karel Can Add

1. Pickbeeper on question corner - Sth Street and Ath Avenue

2. Putbeeper on answer corner - First Street and (S + A)th Avenue

B. How To Do Addition

1. Phase one - locate and pick up beeper

2. Phase two - compute sum and deposit beeper

C. Phase One

1. Use previous example

2. Pickbeeper

D. Phase Two

1. Use zag-se instruction

2. The invariant is: the sum of the street number and avenue number that Karel is on is always S + A

3. Karel moves south until he is on First Street

4. Position is A + S - 1 so must move 1 block east

5. Putbeeper

E. See Example

1. Karel must be facing south to perform zag-se properly

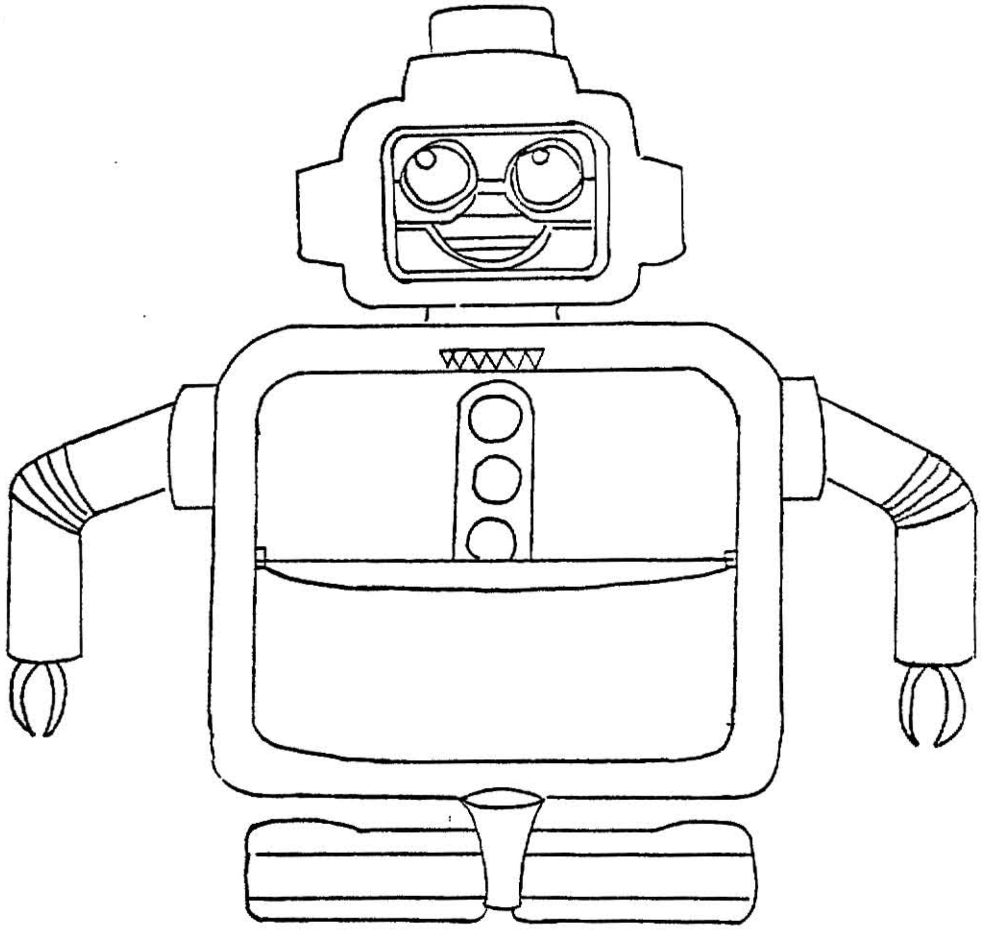
2. Karel must be facing east for last move instruction

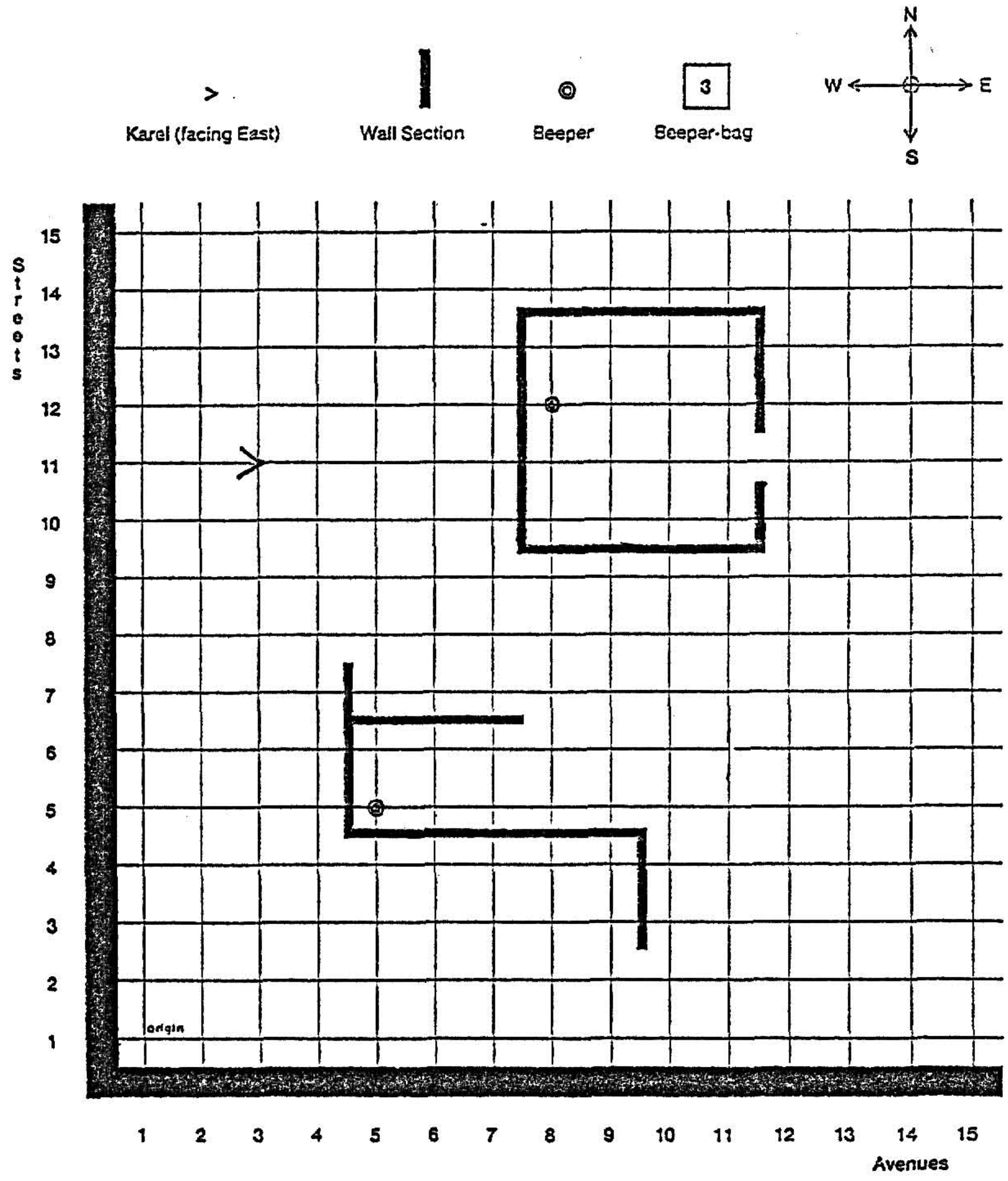
XXXI. Put It All Together

- A. Solve the Task
    1. Understand the task
    2. Break large task into small, independent sub-tasks
    3. Solve sub-tasks within language constraints
    4. Combine sub-task solutions to produce task solution
- B. Correct the Program
    1. Eliminate lexical errors - unknown words
    2. Eliminate syntax errors - incorrect grammar and punctuation
- C. Simulate the Program
    1. Program should execute
    2. Program should perform intended task - no intent errors

Appendix B

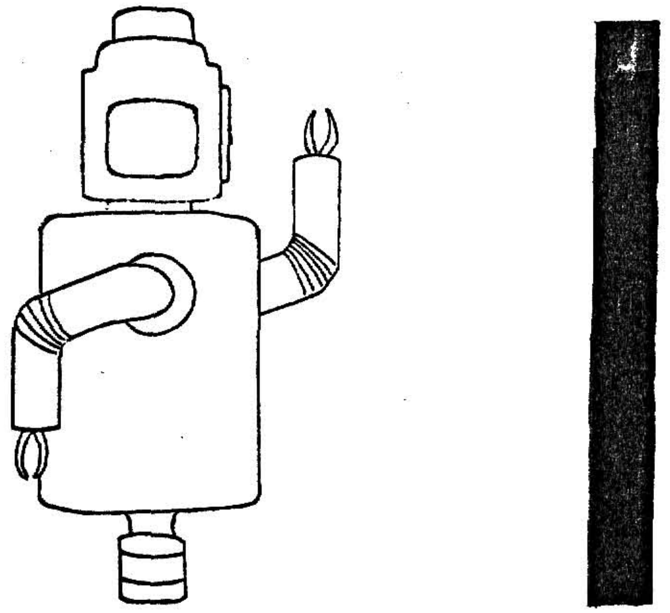
1

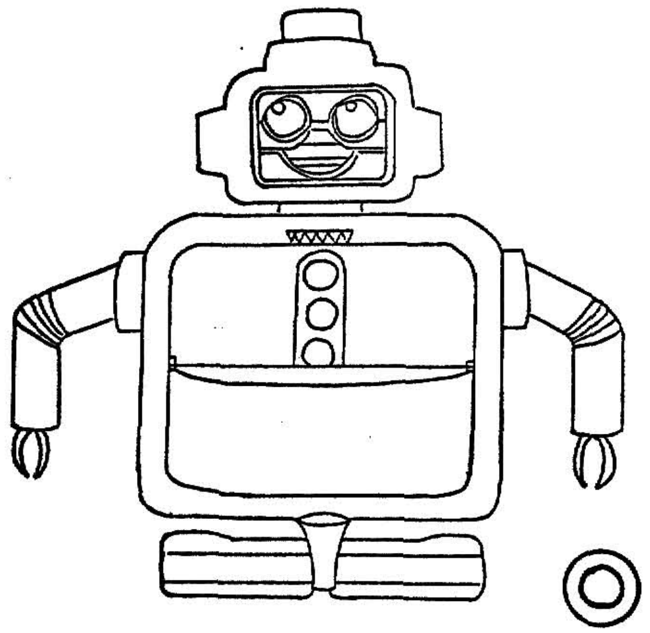


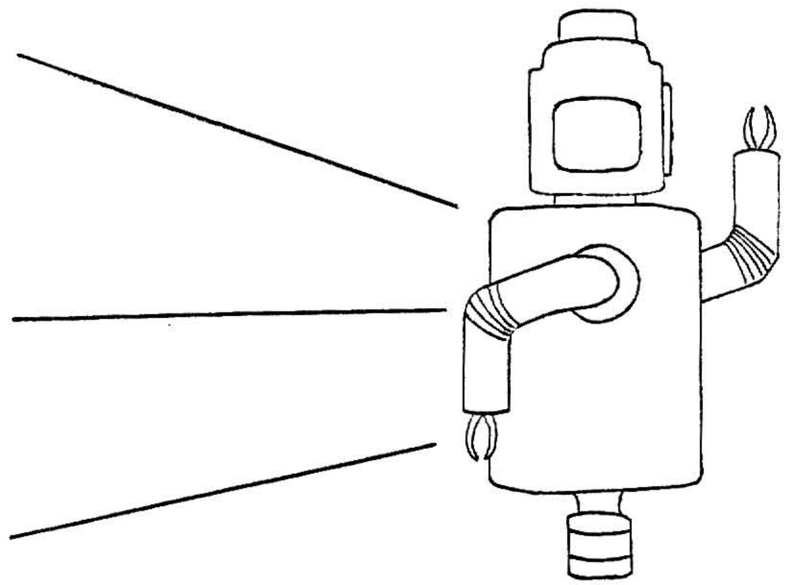


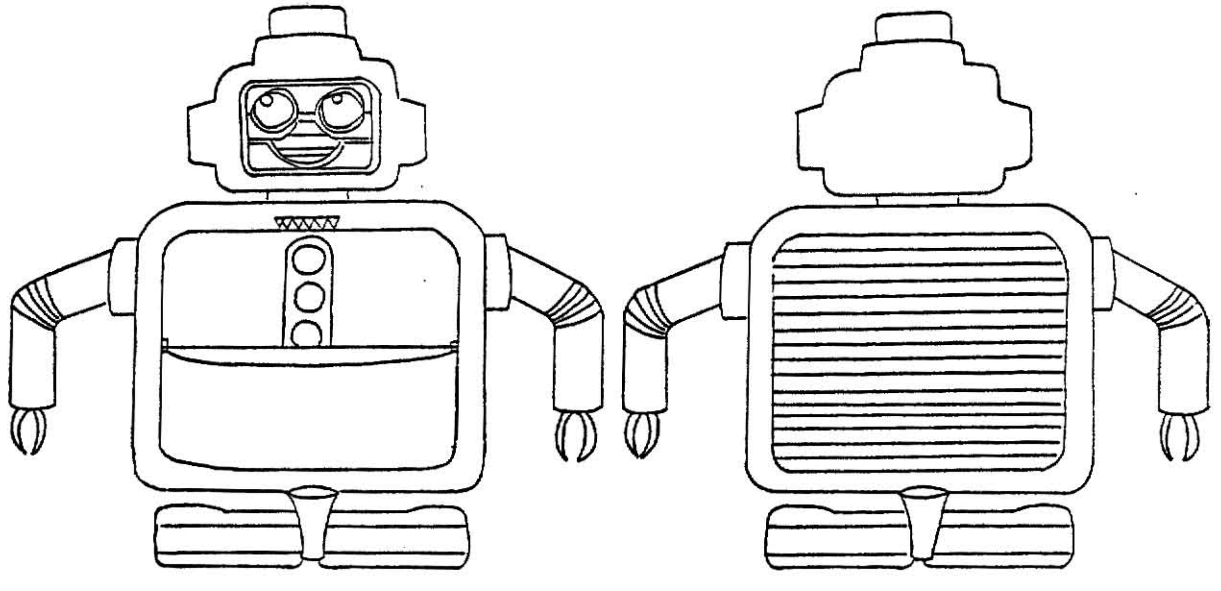
The Structure of Karel's World

3

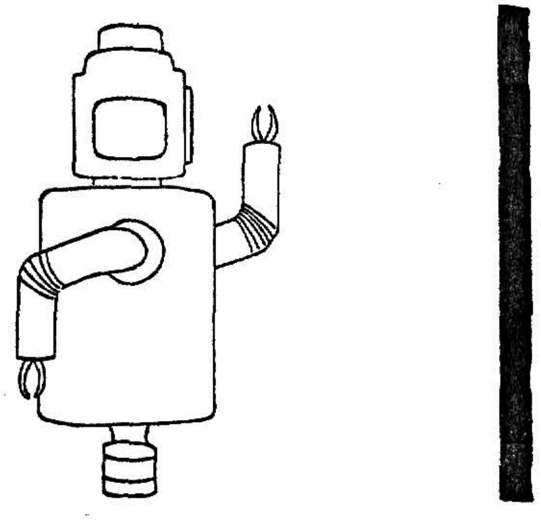


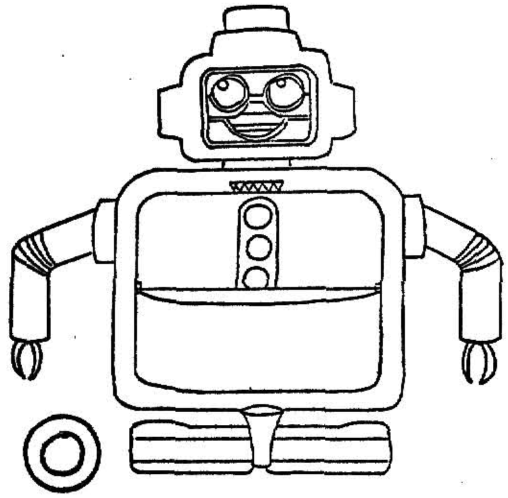


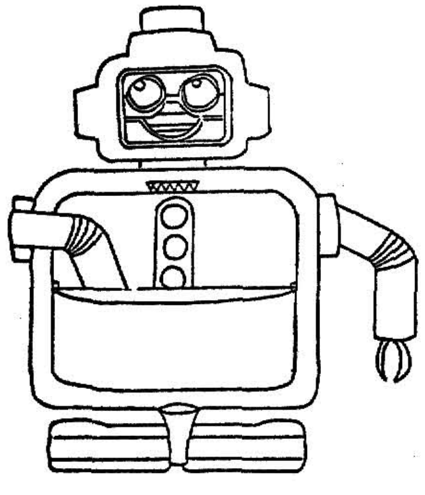




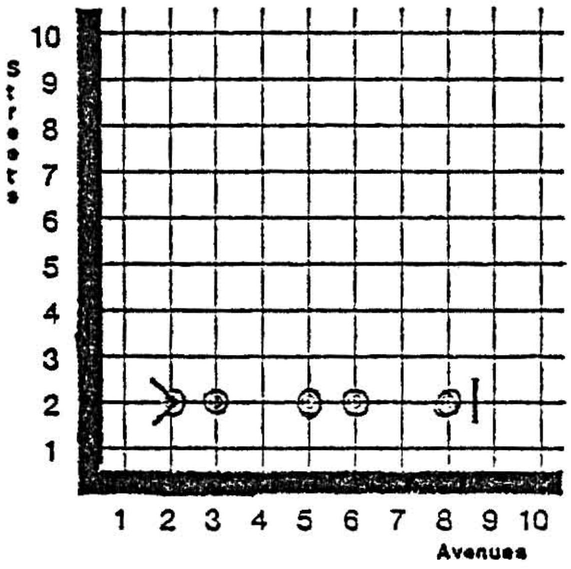
5



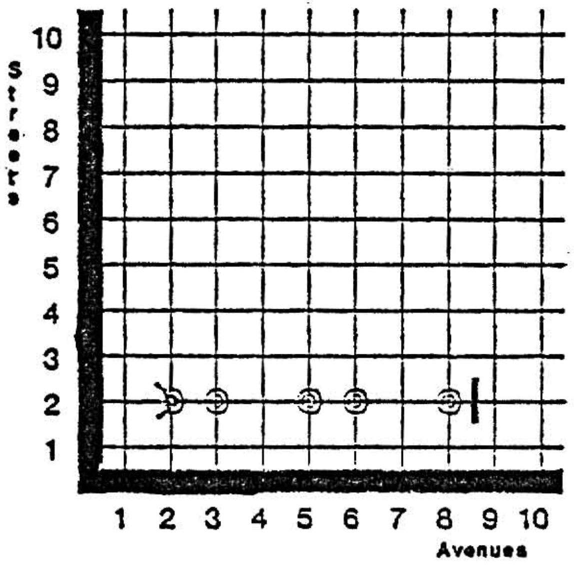




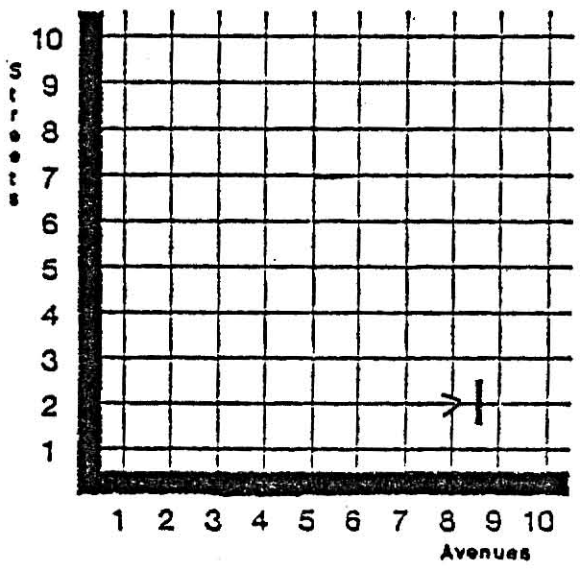
6



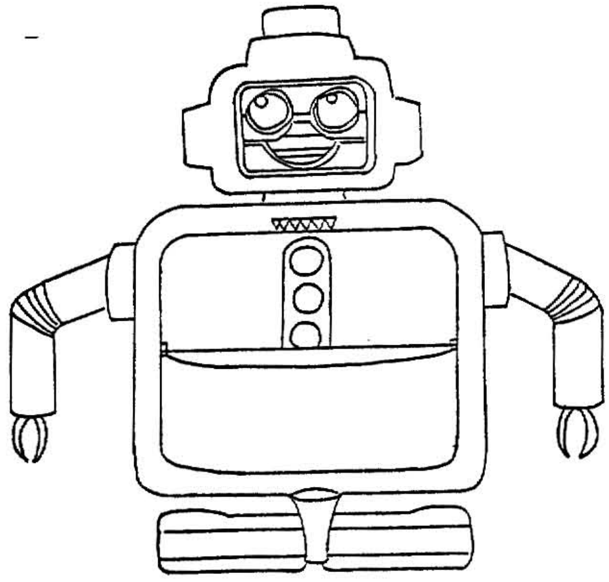
7

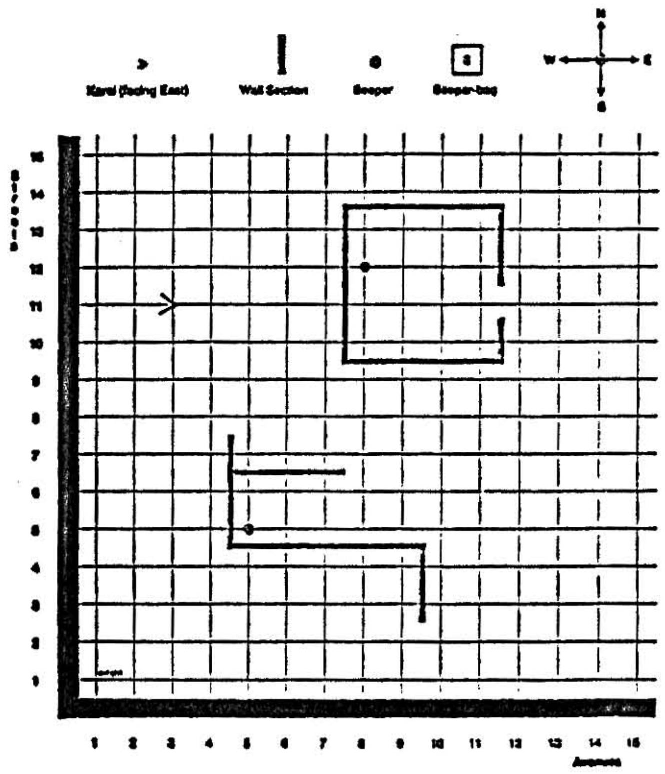


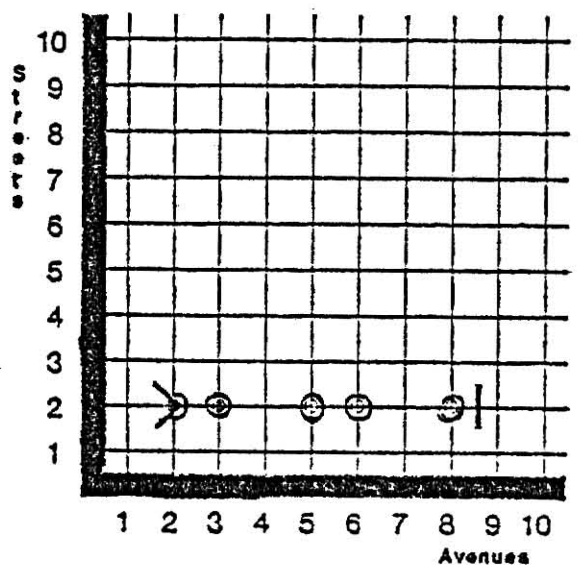
Initial Situation



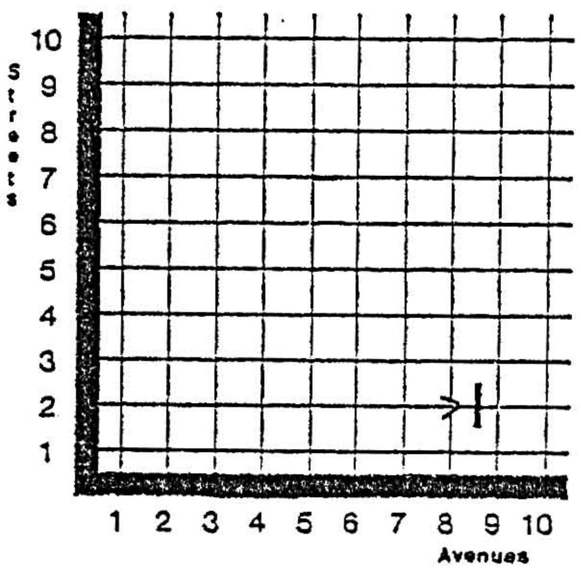
Final Situation







Initial Situation



Final Situation

9

TASK --- Something we want done

PROGRAM --- Detailed set of instructions which explain how to perform the task

PROGRAMMING LANGUAGE --- Language in which a program is written

---

10

We will be using Karel, his limited world, and a limited set of instructions to perform tasks.

11

stop --- informs Karel that he has completed his task

1. Karel turns himself off.
2. Karel will not resume action until restarted by another program.
3. Stop must be the last instruction in every robot program.

12

move --- causes Karel to go forward one block in the direction he is facing

1. Karel continues to face the same direction.
2. Karel will not move if he sees a wall section or a boundary wall between himself and the corner he is moving toward.

13

turnleft --- causes Karel to turn 90 degrees to the left

1. Karel remains at the same corner.
2. Karel can always turn.

14

pickbeeper --- causes Karel to pick up a beeper at the corner where he is standing

1. Karel continues to face the same direction.
2. Karel remains at the same corner.
3. Karel picks up one and only one beeper even if there are several.
4. Karel will not pick up a beeper if none are present.

15

putbeeper -- causes Karel to take a beeper from his beeper bag and place it at the corner

1. Karel continues facing the same direction.
2. Karel remains at the same corner.
3. Karel will not deposit a beeper if he has none.

16

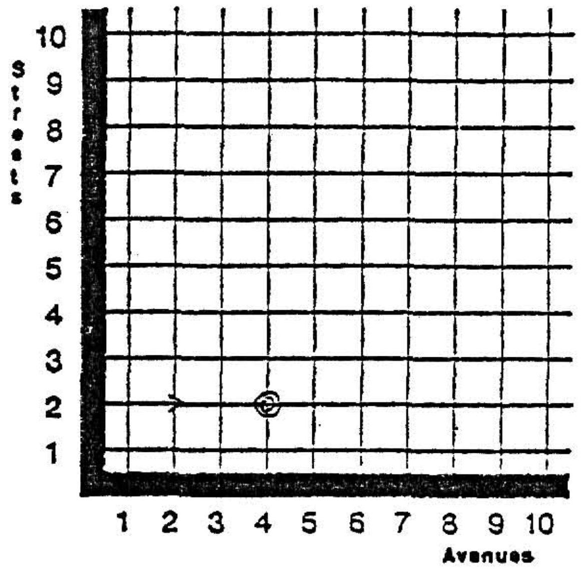
Karel's Five Primitive Instructions

1. stop
2. move
3. turnleft
4. pickbeeper
5. putbeeper

17

TASK --- Move a beeper from Second Street and Fourth Avenue to Fourth
Street and Fifth Avenue.

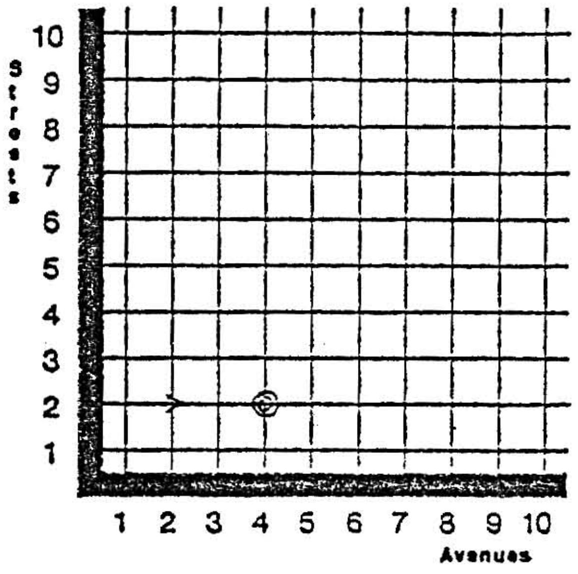
18



Initial Situation

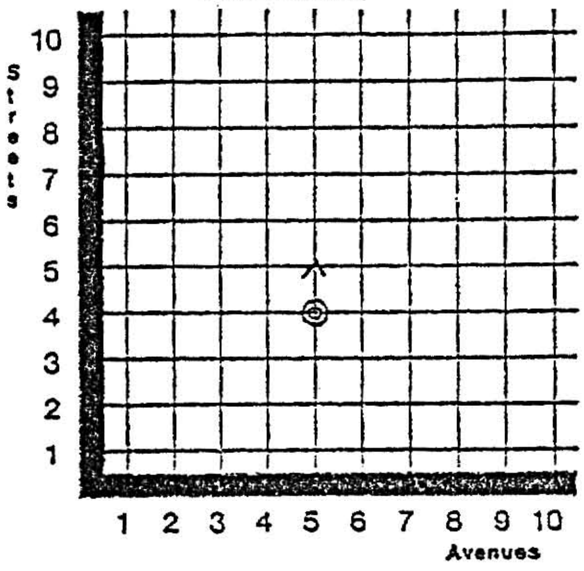
19

```
BEGINPROGRAM
   BEGINEXECUTION
      move;
      move;
      pickbeeper;
      move;
      turnleft;
      move;
      move;
      putbeeper;
      move;
      stop
   ENDEXECUTION
ENDPROGRAM
```



Initial Situation

```
BEGINPROGRAM
   BEGINEXECUTION
      move;
      move;
      pickbeeper;
      move;
      turnleft;
      move;
      move;
      putbeeper;
      move;
      stop
   ENDEXECUTION
ENDPROGRAM
```



Final Situation

21

Rules for Reserved Words

1. Every program must start with BEGINPROGRAM.
2. Every program must end with ENDPROGRAM.
3. Every program must contain a BEGINEXECUTION and an ENDEXECUTION.
4. Every BEGIN must have a matching END.

Note: Reserved words will always be written in uppercase letters.

Punctuation Rules

1. The semicolon is the only allowed punctuation mark.
2. The semicolon separates instructions.
3. The semicolon does not separate reserved words.
4. The semicolon does not separate reserved words from instructions.

Indentation

1. It makes the program easier to read.
2. It aids in the detection of errors.
3. Karel cannot hear indentation.

22

Error shutoff --- Karel turns himself off when he cannot execute an instruction.

1. move blocked by a wall section or boundary
2. pickbeeper when Karel is not next to one
3. putbeeper when Karel's beeper bag is empty

23

Programming requires an inhuman amount of precision. In principle, there should never be any errors since we know all the rules. In practice, there is an abundance of errors.

24

Four Categories of Errors

1. Lexical error --- Karel hears a word not in his vocabulary and turns himself off.

2. Syntactic error --- Karel hears incorrect grammar or inaccurate punctuation and turns himself off.

3. Execution error --- Karel is unable to execute an instruction and turns himself off.

4. Intent error --- Karel successfully executes the program but does not successfully complete his task.

25

Remember ---

Karel does not know what you _want_ him to do; all he knows is what you _tell_ him to do.

26

```
turnright = turnleft;
            turnleft;
            turnleft
```

27

```
move ten miles = move;
                 move;
                 move;
                   .
                   .   (154 move instructions)
                   .
                 move;
                 move
```

28

PROBLEM: People think in one language and must program in another.

29

Give Karel a dictionary of useful instruction names and their definitions.

1. Definitions are built from simpler instructions.
2. The first definitions would be built using primitive instructions.

30

```
DEFINE-NEW-INSTRUCTION turnright
AS

           turnleft
           turnleft
           turnleft

DEFINE-NEW-INSTRUCTION movemile
AS

             move
             move
             move
             move
             move
             move
             move
             move
```

31

```
DEFINE-NEW-INSTRUCTION <newname>
AS

         <instruction>
```

32

Replace <newname> with the name of the new instruction.

1. Any word in lowercase letters or numbers.
2. Cannot already be an instruction name.
3. Cannot be a reserved word.
4. May be hyphenated.

33

Replace <instruction> with the definition of <newname>.

1. Any single instruction that Karel understands.
2. A primitive instruction.
3. A previously defined new instruction.

34

```
DEFINE-NEW-INSTRUCTION evence
AS
     move

DEFINE-NEW-INSTRUCTION tourne-a-gauche
AS
     turnleft
```

35

```
BEGIN
   <instruction>;
   <instruction>;
   <instruction>;
         .
         .
         .
   <instruction>;
   <instruction>
END
```

36

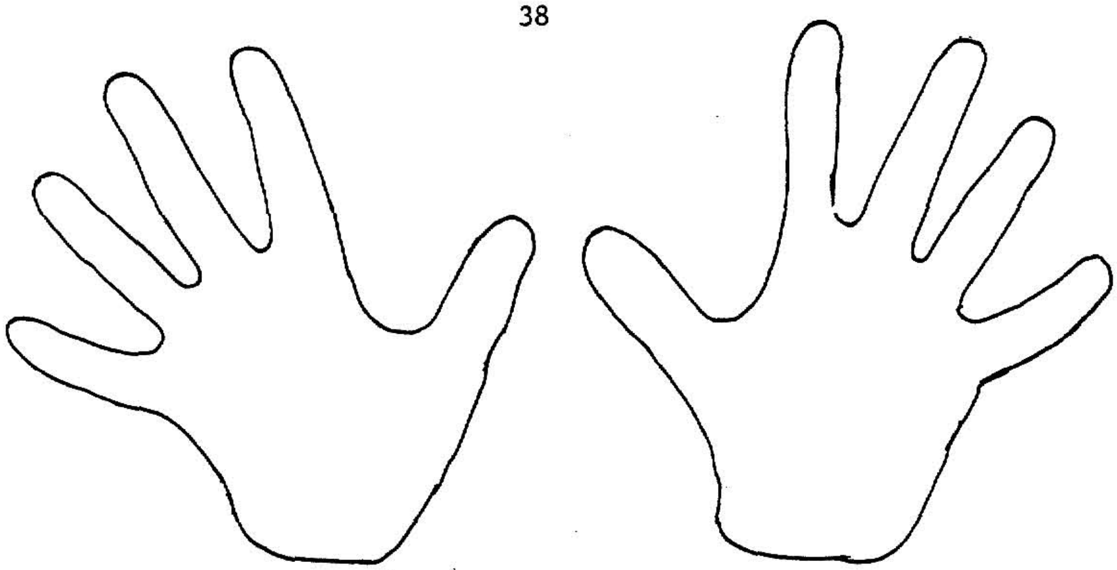
Fundamental property of the BEGIN - END:

Karel understands the entire block to represent one instruction.

37

```
DEFINE-NEW-INSTRUCTION turnright
AS
 BEGIN
   turnleft;
   turnleft;
   turnleft
 END

DEFINE-NEW-INSTRUCTION movemile
AS
 BEGIN
   move;
   move;
   move;
   move;
   move;
   move;
   move;
   move
 END
```

38



39

```
DEFINE-NEW-INSTRUCTION turnright
AS
 BEGIN
   turnleft;
   turnleft
 END
```

40

RULE: Karel executes a newly defined instruction by executing its definition.

41

RULE: The instruction name should specify "what" the instruction does, while the definition specifies "how".

```
BEGINPROGRAM
  DEFINE-NEW-INSTRUCTION turnright
  AS
   BEGIN
     turnleft;
     turnleft;
     turnleft
   END;

  DEFINE-NEW-INSTRUCTION climb-north-east
  AS
   BEGIN
     turnleft;
     move;
     turnright;
     move
   END;

  BEGINEXECUTION
    climb-north-east;
    pickbeeper;
    climb-north-east;
    pickbeeper;
    climb-north-east;
    pickbeeper;
    stop
  ENDEXECUTION
ENDPROGRAM
```

IMPORTANT

1. All definitions go between the reserved words BEGINPROGRAM and BEGINEXECUTION.
2. Only instruction definitions may be located between BEGINPROGRAM and BEGINEXECUTION.
3. This portion of the program is called Karel's dictionary.
4. Each instruction must be defined before it is used.
5. Instruction definitions must be separated by semicolons.
6. The last definition must be separated from the reserved word BEGINEXECUTION by a semicolon.
7. Every program must contain a complete set of definitions for all new instruction names that it uses.

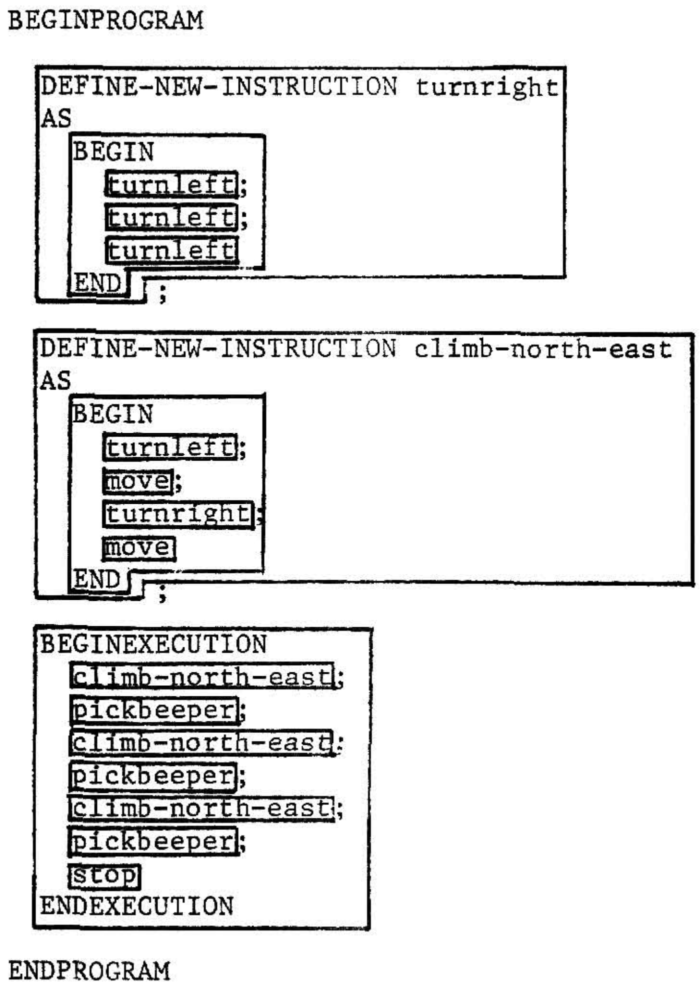
44

```
DEFINE-NEW-INSTRUCTION turnright
AS
   turnleft;
   turnleft;
   turnleft
```

45

UNIT: An instruction, any type of BEGIN - END block, or an entry in Karel's dictionary.

46



```
BEGINPROGRAM

  DEFINE-NEW-INSTRUCTION turnright
  AS
    BEGIN
      turnleft;
      turnleft;
      turnleft
    END;

  DEFINE-NEW-INSTRUCTION climb-north-east
  AS
    BEGIN
      turnleft;
      move;
      turnright;
      move
    END;

  BEGINEXECUTION
    climb-north-east;
    pickbeeper;
    climb-north-east;
    pickbeeper;
    climb-north-east;
    pickbeeper;
    stop
  ENDEXECUTION

ENDPROGRAM
```

47

Geometric property of boxing:

Boxes may be one inside the other, nested, or they may be adjacent. Boxes may <u>never</u> overlap.

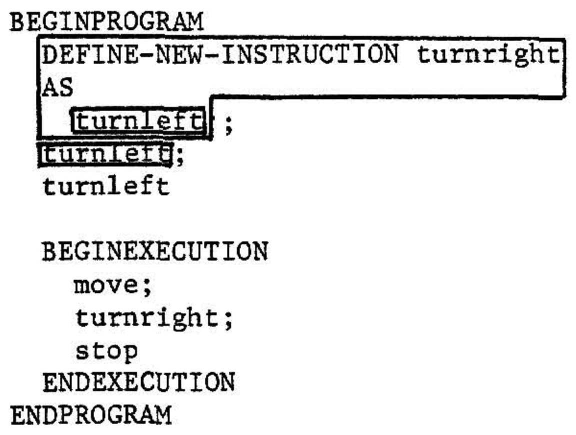
48

How to box:

1. Box primitive instructions first and work outward.
2. Start at the beginning of the program.
3. Build the biggest possible boxes before starting boxes further down.

49

```
BEGINPROGRAM
  DEFINE-NEW-INSTRUCTION turnright
  AS
    turnleft;
    turnleft;
    turnleft

  BEGINEXECUTION
    move;
    turnright;
    stop
  ENDEXECUTION
ENDPROGRAM
```



```
BEGINPROGRAM
  DEFINE-NEW-INSTRUCTION turnright
  AS
    turnleft ;
  turnleft;
  turnleft

  BEGINEXECUTION
    move;
    turnright;
    stop
  ENDEXECUTION
ENDPROGRAM
```

50

Primitive instructions

```
    stop
    move                      DEFINE-NEW-INSTRUCTION <newname>
    turnleft                  AS
    pickbeeper                  <instruction>
    putbeeper
```

```
BEGIN
  <instruction>;
  <instruction>;
       .
       .
       .
  <instruction>;
  <instruction>
END
```

Types of errors

- syntax
- lexical
- intent
- execution

51

STEPWISE REFINEMENT

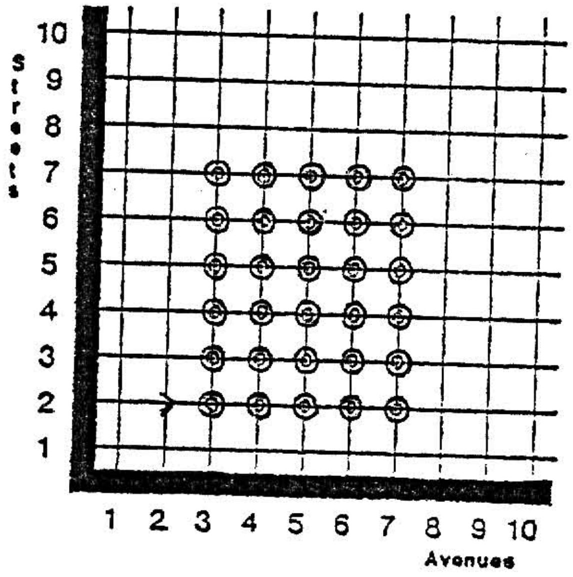
52

Steps to Follow:

1. Write the sequence of instructions in the BEGINEXECUTION - ENDEXECUTION block.
2. Write the definitions of the new instructions used in the BEGINEXECUTION - ENDEXECUTION block.
3. Write the definitions of the new instructions used in the BEGIN - END blocks of the new instruction definitions.
4. Repeat step 3 until all instructions are defined.

53

TASK: Harvest a field of beepers.



Initial Situation

54

```
BEGINEXECUTION
   move;
   harvest-2-furrows;
   position-for-next-2;
   harvest-2-furrows;
   position-for-next-2;
   harvest-2-furrows;
   move;
   stop
ENDEXECUTION
```

55

```
DEFINE-NEW-INSTRUCTION harvest-2-furrows
AS
 BEGIN
   harvest-a-furrow;
   go-to-next-furrow;
   harvest-a-furrow
 END

DEFINE-NEW-INSTRUCTION position-for-next-2
AS
 BEGIN
   turnright;
   move;
   turnright
 END
```

56

```
DEFINE-NEW-INSTRUCTION harvest-a-furrow
AS
 BEGIN
   pickbeeper;
   move;
   pickbeeper;
   move;
   pickbeeper;
   move;
   pickbeeper;
   move;
   pickbeeper
 END

DEFINE-NEW-INSTRUCTION go-to-next-furrow
AS
 BEGIN
   turnleft;
   move;
   turnleft;
 END

DEFINE-NEW-INSTRUCTION turnright
AS
 BEGIN
   turnleft;
   turnleft;
   turnleft
 END
```

```
BEGINPROGRAM
  DEFINE-NEW-INSTRUCTION turnright
  AS
   BEGIN
     turnleft;
     turnleft;
     turnleft
   END;

  DEFINE-NEW-INSTRUCTION position-for-next-2
  AS
   BEGIN
     turnright;
     move;
     turnright
   END;

  DEFINE-NEW-INSTRUCTION go-to-next-furrow
  AS
   BEGIN
     turnleft;
     move;
     turnleft
   END;

  DEFINE-NEW-INSTRUCTION harvest-a-furrow
  AS
   BEGIN
     pickbeeper;
     move;
     pickbeeper;
     move;
     pickbeeper;
     move;
     pickbeeper;
     move;
     pickbeeper
   END;

  DEFINE-NEW-INSTRUCTION harvest-2-furrows
  AS
   BEGIN
     harvest-a-furrow;
     go-to-next-furrow;
     harvest-a-furrow
   END

  BEGINEXECUTION
     move;
     harvest-2-furrows;
     position-for-next-2;
     harvest-2-furrows;
     position-for-next-2;
     harvest-2-furrows;
     move;
     stop
  ENDEXECUTION
ENDPROGRAM
```

58

STEPWISE REFINEMENT

1. Understand task

2. Break task into smaller, easier to understand, independent subtasks

3. Solve subtasks

59

"Programs are guilty of being wrong until they are proven correct."

60

Understandability is as important as correctness.

61

Criteria for understandable programs:

1. A simple composition of easily understandable parts

2. Instructions properly named

62

Understandability helps the programmer by making the program:

1. easier to read

2. easier to verify

3. easier to correct

4. easier to modify

5. easier to change into one instruction

63

```
IF - THEN

IF - THEN - ELSE
```

64

```
IF <test>
 THEN
     <instruction>
```

65

How executed:

1. Check <test> in the current situation.
2. If <test> is true, execute <instruction>.
3. If <test> is false, do not execute <instruction>.

66

```
IF next-to-a-beeper
 THEN
    pickbeeper;
move
```

67

Conditions Karel can test for:

front-is-clear	front-is-blocked
left-is-clear	left-is-blocked
right-is-clear	right-is-blocked
next-to-a-beeper	not-next-to-a-beeper
facing-north	not-facing-north
facing-south	not-facing-south
facing-east	not-facing-east
facing-west	not-facing-west
any-beepers-in-beeper-bag	no-beepers-in-beeper-bag

68

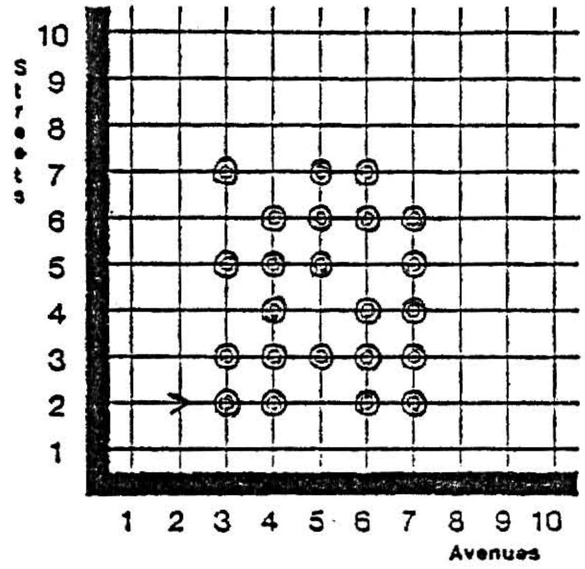
Conditions are TRUE when:

front, left, and right are respectively clear if there are no walls between Karel and the next corner in the direction he is commanded to look

next-to-a-beeper is true when Karel and beeper are located at the same corner

Karel's internal compass tells him what direction he is facing

Karel determines if he has any beepers by poking around in his beeper-bag


Streets
10
9
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9 10
Avenues

```
BEGINPROGRAM
  DEFINE-NEW-INSTRUCTION turnright
  AS
   BEGIN
     turnleft;
     turnleft;
     turnleft
   END;

  DEFINE-NEW-INSTRUCTION position-for-next-2
  AS
   BEGIN
     turnright;
     move;
     turnright
   END;

  DEFINE-NEW-INSTRUCTION go-to-next-furrow
  AS
   BEGIN
     turnleft;
     move;
     turnleft
   END;

  DEFINE-NEW-INSTRUCTION harvest-a-furrow
  AS
   BEGIN
     pickbeeper;
     move;
     pickbeeper;
     move;
     pickbeeper;
     move;
     pickbeeper;
     move;
     pickbeeper
   END;

  DEFINE-NEW-INSTRUCTION harvest-2-furrows
  AS
   BEGIN
     harvest-a-furrow;
     go-to-next-furrow;
     harvest-a-furrow
   END

  BEGINEXECUTION
     move;
     harvest-2-furrows;
     position-for-next-2;
     harvest-2-furrows;
     position-for-next-2;
     harvest-2-furrows;
     move;
     stop
  ENDEXECUTION
ENDPROGRAM
```

71

```
DEFINE-NEW-INSTRUCTION harvest-a-furrow
AS
 BEGIN
   pickbeeper-if-present;
   move;
   pickbeeper-if-present;
   move;
   pickbeeper-if-present;
   move;
   pickbeeper-if-present;
   move;
   pickbeeper-if-present
 END
```

72

```
DEFINE-NEW-INSTRUCTION pickbeeper-if-present
AS
 BEGIN
   IF next-to-a-beeper
     THEN
       pickbeeper
 END
```

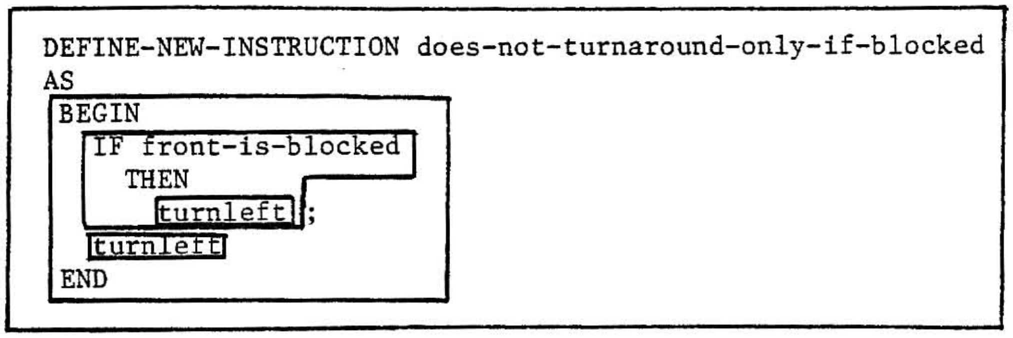
73

```
DEFINE-NEW-INSTRUCTION turnaround-only-if-blocked
AS
 BEGIN
   IF front-is-blocked
     THEN
       BEGIN
         turnleft;
         turnleft
       END
 END
```

74

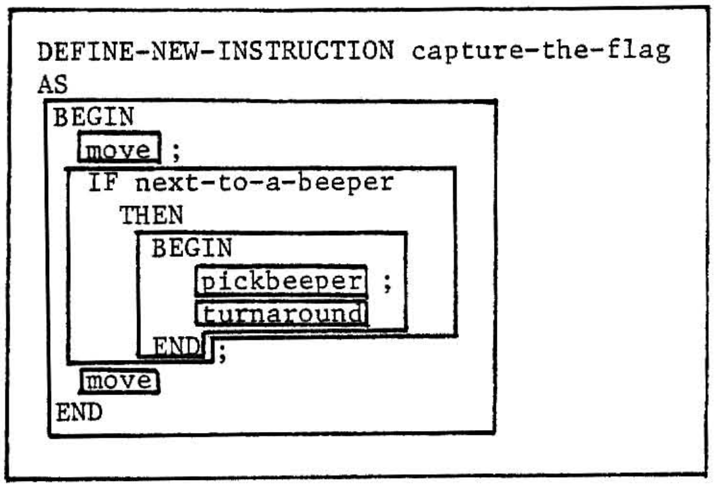
```
DEFINE-NEW-INSTRUCTION does-not-turnaround-only-if-blocked
AS
 BEGIN
   IF front-is-blocked
     THEN
       turnleft;
       turnleft
 END
```

75



```
DEFINE-NEW-INSTRUCTION does-not-turnaround-only-if-blocked
AS
 BEGIN
   IF front-is-blocked
     THEN
       turnleft;
   turnleft
 END
```

76



```
DEFINE-NEW-INSTRUCTION capture-the-flag
AS
 BEGIN
   move ;
   IF next-to-a-beeper
     THEN
       BEGIN
         pickbeeper ;
         turnaround
       END;
   move
 END
```

77

```
IF <test>
  THEN
   <instruction>
  ELSE
   <instruction>
```

78

How executed:

1. Check <test> in the current situation.
2. If <test> is true, execute THEN clause.
3. If <test> is false, execute ELSE clause.

79

TASK: Run a mile hurdle race with wall sections representing the hurdles.

80

```
DEFINE-NEW-INSTRUCTION race-stride
AS
 BEGIN
    IF front-is-blocked
      THEN
        jump-hurdle
      ELSE
        move
 END
```

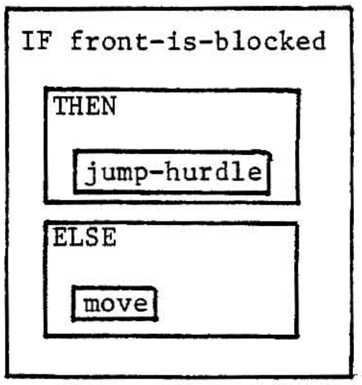
81

```
IF front-is-blocked            IF front-is-clear
  THEN                           THEN
    jump-hurdle                    move
  ELSE                           ELSE
    move                           jump-hurdle
```

82

```
DEFINE-NEW-INSTRUCTION incorrect-race-stride
AS
 BEGIN
   IF front-is-blocked
     THEN
       turnleft;
       move;
       turnright;
       move;
       turnright;
       move;
       turnleft
     ELSE
       move
 END
```

83



```
IF front-is-blocked
  THEN
    jump-hurdle
  ELSE
    move
```

84

```
DEFINE-NEW-INSTRUCTION incorrect-race-stride
AS
 BEGIN
   IF front-is-blocked
     THEN
       turnleft ;
    move ;
    turnright ;
    move ;
    turnright ;
    move ;
    turnleft
   ELSE
   move
 END
```

85

```
DEFINE-NEW-INSTRUCTION incorrect-race-stride
AS
 BEGIN
   IF front-is-clear
     THEN
       move
     ELSE
       turnleft ;
     move ;
     turnright ;
     move ;
     turnright ;
     move ;
     turnleft
 END
```

86

Tactical Lesson: Never forget BEGIN - END blocks.

Strategic Lesson: The bigger the instruction, the more complicated it becomes.

87

```
IF <test1>
  THEN
   <instruction1>
  ELSE
    IF <test2>
      THEN
       <instruction2>
      ELSE
       <instruction3>
```

88

```
DEFINE-NEW-INSTRUCTION replant-exactly-one
AS
 BEGIN
    IF not-next-to-a-beeper
      THEN
         putbeeper
      ELSE
        BEGIN
           pickbeeper ;
           IF not-next-to-a-beeper
             THEN
                putbeeper
        END
 END
```

89

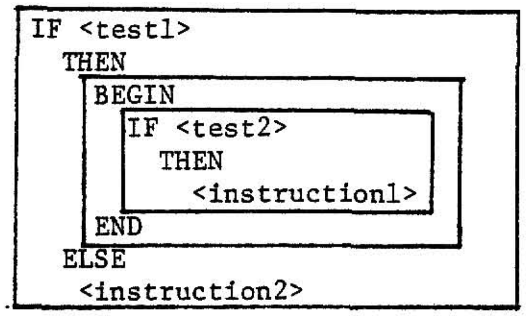
```
IF <test1>
   THEN
     IF <test2>
        THEN
          <instruction1>
        ELSE
          <instruction2>
```

```
IF <test1>
   THEN
     IF <test2>
        THEN
          <instruction1>
   ELSE
    <instruction2>
```

90

Rule: The ELSE clause is always boxed with the first possible preceding IF instruction.

91



```
IF <test1>
  THEN
    BEGIN
      IF <test2>
        THEN
          <instruction1>
    END
  ELSE
   <instruction2>
```

92

REVIEW

How the IF instruction is executed:

1. Check <test> in current situation.

2. If <test> is true, the THEN clause is executed.

3. If <test> is false, the ELSE clause is executed if one is present.

93

```
IF <test>
  THEN
   <instruction>


IF <test>
  THEN
   <instruction>
  ELSE
   <instruction>
```

```
IF <test1>
  THEN
    IF <test2>
      THEN
       <instruction1>
      ELSE
       <instruction2>

IF <test1>
  THEN
   <instruction1>
  ELSE
    IF <test2>
      THEN
       <instruction2>
      ELSE
       <instruction3>

IF <test1>
  THEN
    BEGIN
      IF <test2>
        THEN
          <instruction1>
    END
  ELSE
   <instruction2>
```

94

ITERATE

WHILE

95

```
ITERATE <positive-integer> TIMES
   < instruction >
```

96

```
DEFINE-NEW-INSTRUCTION turnright
AS
  BEGIN
   ITERATE 3 TIMES
     turnleft
  END
```

97

```
DEFINE-NEW-INSTRUCTION face-east      DEFINE-NEW-INSTRUCTION face-east
AS                                    AS
 BEGIN                                 BEGIN
    IF not-facing-east                    ITERATE 3 TIMES
      THEN                                   IF not-facing-east
         turnleft                              THEN
    IF not-facing-east                            turnleft
      THEN                             END
         turnleft
    IF not-facing-east
      THEN
         turnleft
 END
```

98

```
DEFINE-NEW-INSTRUCTION make-square-of-length-6
AS
 BEGIN
    ITERATE 4 TIMES
       BEGIN
         ITERATE 6 TIMES
            move;
         turnleft
       END
 END
```

99

The inner loop is performed completely for every single execution of the outer loop.

100

```
DEFINE-NEW-INSTRUCTION make-square-of-length-6
AS
 BEGIN
    ITERATE 4 TIMES
         BEGIN
            ITERATE 6 TIMES
               move ;
            turnleft
         END
 END
```

101

TASK: Move forward until a beeper is found.

102

```
IF not-next-to-a-beeper              ITERATE ? TIMES
  THEN                                  IF not-next-to-a-beeper
    move;                                 THEN
IF not-next-to-a-beeper                     move
  THEN
    move;
         .
         .
         .
IF not-next-to-a-beeper
  THEN
    move
```

103

```
WHILE <test> DO
   <instruction>
```

104

How the WHILE is executed:

1. Check <test> in the current situation.

2. If the <test> is false, the WHILE is finished and execution starts with the first instruction after the WHILE loop.

3. If the <test> is true, the WHILE loop is executed and then the entire WHILE instruction is re-executed.

105

```
WHILE not-next-to-a-beeper DO
   move
```

106

Formal property of the WHILE instruction:

When the WHILE instruction has finished executing, the test is guaranteed to be false.

107

```
WHILE <not-test> DO
  <instruction>
```

108

```
DEFINE-NEW-INSTRUCTION face-east
AS
 BEGIN
   WHILE not-facing-east DO
      turnleft
 END
```

109

```
WHILE next-to-a-beeper DO
   turnleft
```

110

An infinite loop occurs when the instructions in the body of a WHILE instruction do not cause Karel to progress toward his goal.

111

```
DEFINE-NEW-INSTRUCTION pickup-line
AS
 BEGIN
    WHILE next-to-a-beeper DO
       BEGIN
         pickbeeper;
         move
       END
 END
```

112

Correct rule:

Check <test> each time before entering loop body.

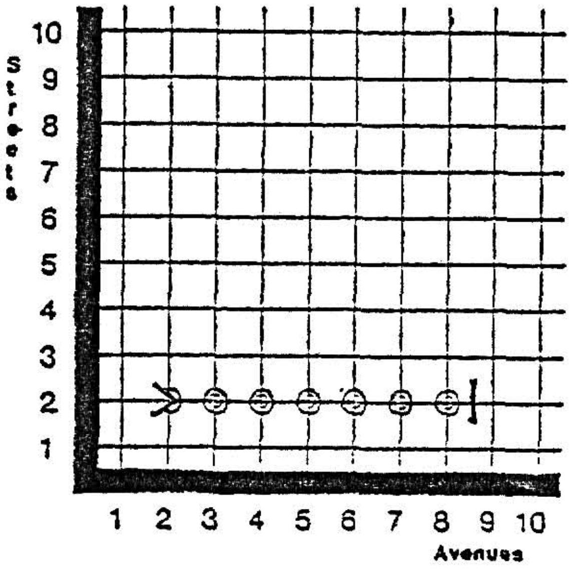
Once in the loop, execute all the instructions.

Recheck the <test> after the body has been completely executed.

113

TASK: Harvest a line of beepers whose end is marked by a wall.

114



```
DEFINE-NEW-INSTRUCTION harvest-to-wall
AS
 BEGIN
    WHILE front-is-clear DO
       pickbeeper;
       move
 END
```

115

```
DEFINE-NEW-INSTRUCTION harvest-to-wall
AS
 BEGIN
    pickbeeper;
    WHILE front-is-clear DO
      BEGIN
        move;
        pickbeeper
      END
 END
```

116

```
DEFINE-NEW-INSTRUCTION harvest-to-wall
AS
 BEGIN
    pickbeeper ;
    WHILE front-is-clear DO
      move ;
    pickbeeper
 END
```

117

```
WHILE <test1> DO
  BEGIN
    IF <test2>
      THEN
        <instruction1>
   <instruction2>
  END
```

118

```
DEFINE-NEW-INSTRUCTION sparse-harvest-to-wall
AS
 BEGIN
   pickbeeper;
   WHILE front-is-clear DO
       BEGIN
         move;
         IF next-to-a-beeper
           THEN
             pickbeeper
       END
 END
```

119

```
IF next-to-a-beeper
  THEN
    pickbeeper
  ELSE DO
    turnleft
```

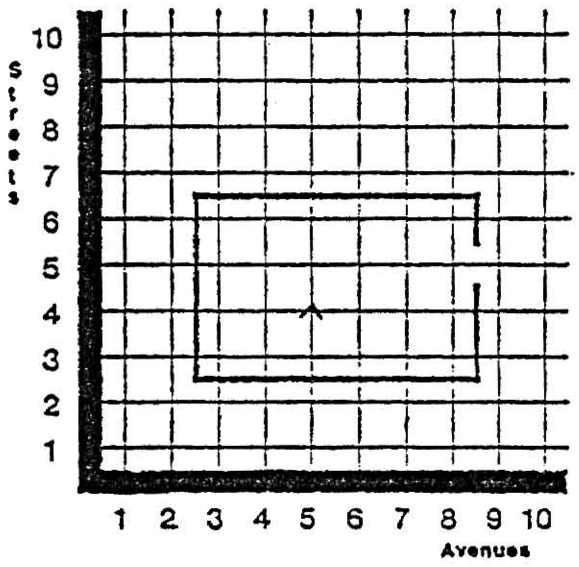
120

Writing a program is not always an easy task, even when you know all the rules.

121

TASK: Escape from any rectangular room with an open doorway exactly one block wide.

122



Initial Situation

123

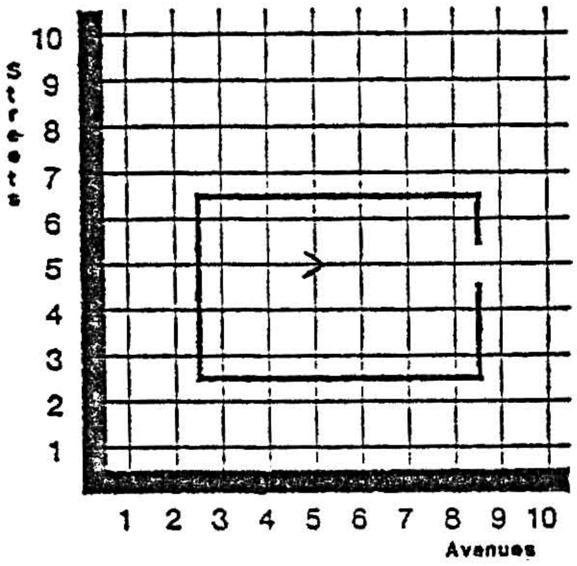
```
BEGINEXECUTION
    go-to-wall;
    turnleft;
    follow-until-door-is-on-right;
    exit-door;
    stop
ENDEXECUTION
```

124

```
DEFINE-NEW-INSTRUCTION go-to-wall
AS
 BEGIN
    WHILE front-is-clear DO
       move
 END
```

125

```
DEFINE-NEW-INSTRUCTION go-to-wall
AS
 BEGIN
    WHILE front-is-clear DO
       move
 END
```



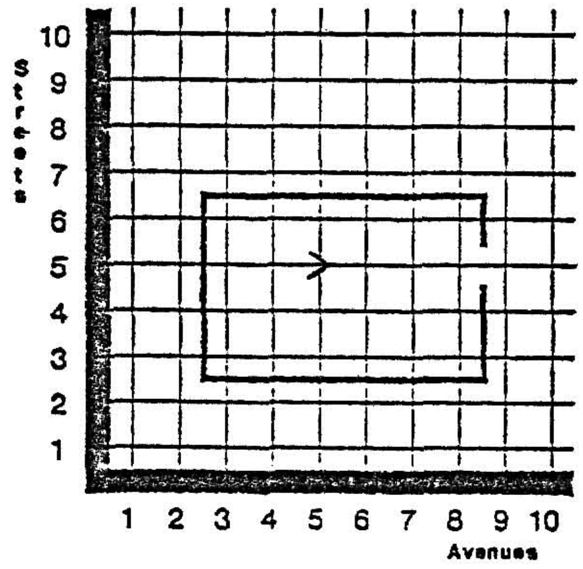
126

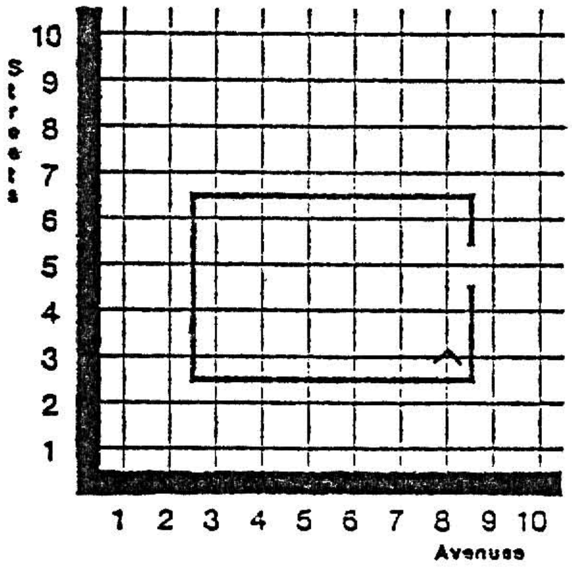
```
DEFINE-NEW-INSTRUCTION go-to-wall
AS
 BEGIN
    WHILE front-is-clear DO
       move
 END
```

```
BEGINEXECUTION
   go-to-wall;
   turnleft;
   follow-until-door-is-on-right;
   exit-door;
   stop
ENDEXECUTION
```

127

```
DEFINE-NEW-INSTRUCTION go-to-wall
AS
 BEGIN
   WHILE front-is-clear DO
      BEGIN
        sidestep-right
        IF front-is-clear
           THEN
             BEGIN
               sidestep-back-left;
               move
             END
      END
 END
```




Streets
10
9
8
7
6
5
4
3
2
1
1
2
3
4
5
6
7
8
9
10
Avenues

```
DEFINE-NEW-INSTRUCTION go-to-wall
AS
 BEGIN
    IF right-is-blocked
      THEN
        turnright
      ELSE
        shuffle-to-wall
 END;

DEFINE-NEW-INSTRUCTION shuffle-to-wall
AS
 BEGIN
    WHILE front-is-clear DO
         BEGIN
            sidestep-right;
            IF front-is-clear
              THEN
                BEGIN
                  sidestep-back-left;
                  move
                END
         END
 END;

DEFINE-NEW-INSTRUCTION sidestep-right
AS
 BEGIN
    turnright;
    move;
    turnleft
 END;

DEFINE-NEW-INSTRUCTION sidestep-back-left
AS
 BEGIN
    turnleft;
    move;
    turnright
 END;
```

130

```
BEGINEXECUTION
  go-to-wall;
  turnleft;
  follow-until-door-is-on-right;
  exit-door;
  stop
ENDEXECUTION
```

131

The follow-until-door-is-on-right instruction must:

1. finish when Karel senses a door on his righthand side.
2. keep Karel's righthand side adjacent to a wall while he follows the perimeter of the room.

132

```
DEFINE-NEW-INSTRUCTION follow-until-door-is-on-right
AS
 BEGIN
   WHILE right-is-blocked DO
     follow-perimeter
 END
```

133

```
DEFINE-NEW-INSTRUCTION follow-perimeter
AS
 BEGIN
   IF front-clear
     THEN
       move
     ELSE
       turnleft
 END
```

134

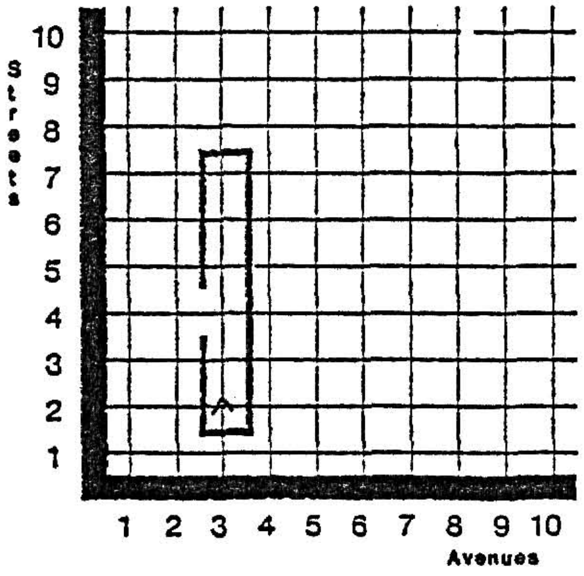
```
BEGINEXECUTION
   go-to-wall;
   turnleft;
   follow-until-door-is-on-right;
   exit-door;
   stop
ENDEXECUTION
```

135

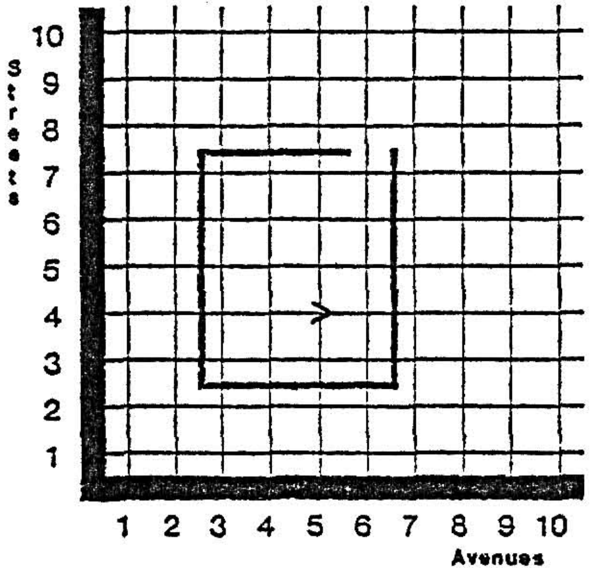
```
DEFINE-NEW-INSTRUCTION exit-door
AS
 BEGIN
    turnright;
    move;
    move
 END
```

```
BEGINPROGRAM

     DEFINE-NEW-INSTRUCTION turnright
     AS
      BEGIN
          ITERATE 3 TIMES
             turnleft
      END;

     DEFINE-NEW-INSTRUCTION sidestep-right
     AS
      BEGIN
          turnright;
          move;
          turnleft
      END;

     DEFINE-NEW-INSTRUCTION sidestep-back-left
     AS
      BEGIN
          turnleft;
          move;
          turnright
      END;

     DEFINE-NEW-INSTRUCTION shuffle-to-wall
     AS
      BEGIN
          WHILE front-is-clear DO
             BEGIN
               sidestep-right;
               IF front-is-clear
                 THEN
                   BEGIN
                      sidestep-back-left;
                      move
                   END
             END
      END;

     DEFINE-NEW-INSTRUCTION go-to-wall
     AS
      BEGIN
          IF right-is-blocked
            THEN
              turnright
            ELSE
              shuffle-to-wall
      END;

     DEFINE-NEW-INSTRUCTION follow-until-door-is-on-right
     AS
      BEGIN
          WHILE right-is-blocked DO
             follow-perimeter
      END;

     DEFINE-NEW-INSTRUCTION exit-door
     AS
      BEGIN
          turnright;
          move;
          move
      END;


     BEGINEXECUTION
          go-to-wall;
          turnleft;
          follow-until-door-is-on-right;
          exit-door;
          stop
     ENDEXECUTION

ENDPROGRAM
```

137



A very skinny room



A door in an unexpected place

138

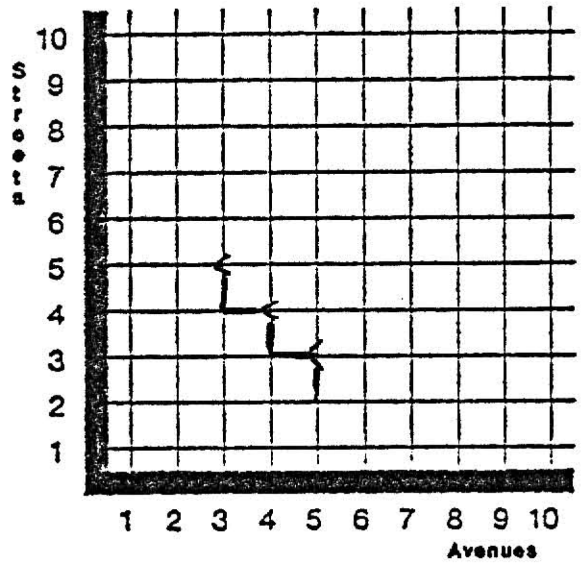
```
zig-nw
zag-se
```

139

```
DEFINE-NEW-INSTRUCTION zig-nw
AS
 BEGIN
    move;
    turnright;
    move;
    turnleft
 END
```

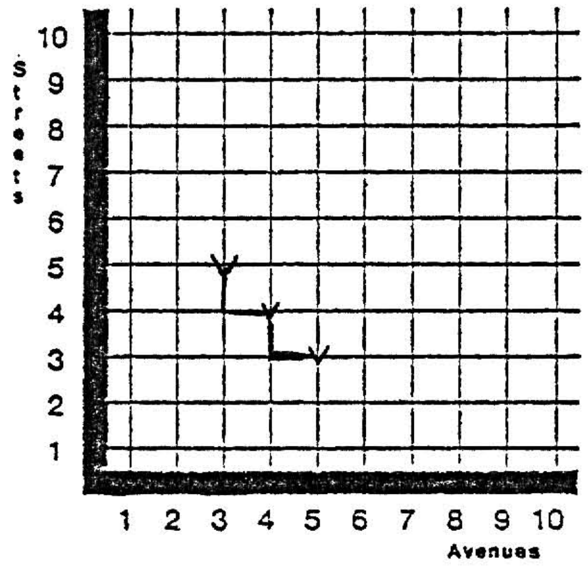
```
DEFINE-NEW-INSTRUCTION zag-se
AS
 BEGIN
   move;
   turnleft;
   move;
   turnright
 END
```

140



zig-nw

141

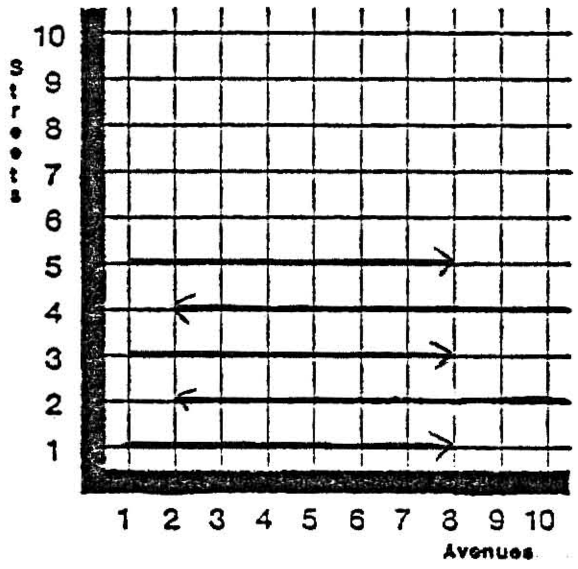


zag-se

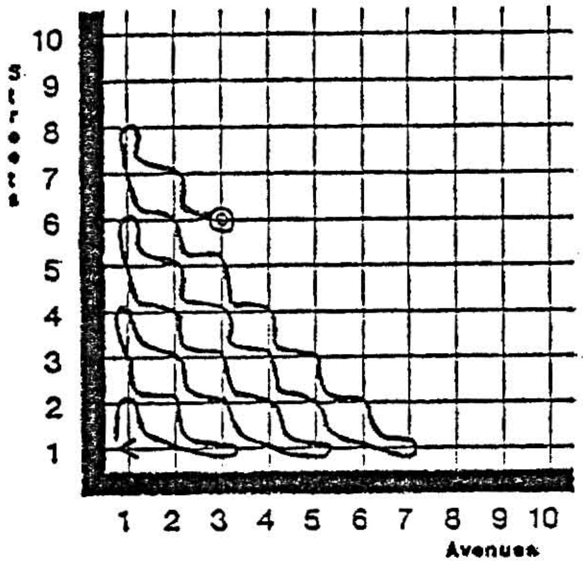
142

TASK: Find a beeper in an unenclosed area.

143



144



145

```
DEFINE-NEW-INSTRUCTION find-beeper
AS
 BEGIN
   go-to-origin;
   face-west;
   WHILE not-next-to-a-beeper DO
     IF facing-west
       THEN
         zig-move
       ELSE
         zag-move
 END
```

146

The zig-nw instructions:

1. Karel will eventually be restrained by the western boundary wall.
2. Precondition - Karel must be facing west.
3. Precondition is invariant over instruction.

The zag-se instruction:

1. Karel will eventually be restrained by the southern boundary wall.
2. Precondition - Karel must be facing south.
3. Precondition is invariant over instruction.

147

```
DEFINE-NEW-INSTRUCTION advance-to-next-diagonal
AS
 BEGIN
   IF facing-west
     THEN
       turnright
     ELSE
       turnleft;
   move;
   turnaround
 END
```

148

Karel can add!

149

Question corner - The corner where Karel picks up the beeper (represented by S Street and A Avenue).

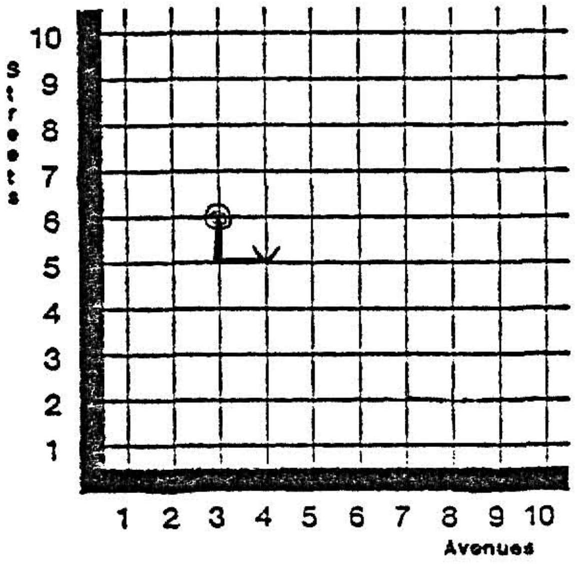
Answer corner - The corner where Karel deposits the beeper.

The answer corner will always be on First Street and A + S Avenue.

150

```
BEGINEXECUTION
    find-beeper;
    pickbeeper;
    compute-sum;
    putbeeper;
    stop
ENDEXECUTION
```

151



152

$$6 + 3 = 5 + 4$$

$$S + A = (S - 1) + (A + 1)$$

153

$$S + A = (S - 1) + (A + 1) = (S - 2) + (A + 2) = . . . = 1 + (A + S - 1)$$

154

```
DEFINE-NEW-INSTRUCTION compute-sum
AS
 BEGIN
   face-south;
   WHILE front-is-clear DO
       zag-se;
       face-east;
       move
 END
```

155

REVIEW

I. Solve the task

1. Understand the task.
2. Break the large task into small, independent subtasks.
3. Solve the subtasks within constraints of language.
4. Combine subtask solutions to produce large task solution.

156

II. Correct the program

1. Eliminate lexical errors.
2. Eliminate syntax errors.

157

III. Simulate the program

1. Program should execute.
2. Program should perform intended task.

158

GOOD LUCK AND GOOD PROGRAMMING!

Appendix C

SCRIPT

1 This is Karel.

2 This is Karel's world. It isn't too exciting according to our standards--no tornadoes, no Aggieville, etc.--but it does have enough variety to allow Karel to perform some interesting tasks. Karel's world is a great flat plane with the standard north, south, east, and west compass points. On the south and west, the plane is bounded by infinitely long walls which restrain Karel from falling over the edge. Horizontal streets and vertical avenues criss-cross the plane at regular one block intervals. The intersection of First Street and First Avenue is called the origin. Besides Karel, two other types of objects occupy Karel's world.

3 There are wall sections which can be manufactured in any desired length. These wall sections may be placed between adjacent street corners to block Karel's path. The other type of object is a beeper. Beepers are small plastic balls that emit a quiet beeping noise. They may be situated on street corners, but they do not block Karel's path.

Now, let's take a closer look at Karel.

4 Karel is a mobile robot that can move forward in the direction he is facing. He can also turn in place. Karel also possesses the senses of sight, sound, direction, and touch.

5 Although Karel can see, he is very nearsighted and can only see wall sections when they are closer than one-half block away. His hearing isn't too good either, unfortunately, because he can only hear a beeper if he and the beeper are on the same corner. Because Karel can't see where he is going, he has been equipped with an internal compass which allows him to determine the direction he is facing. Finally, Karel has arms that can be used to pick up and put down beepers. Notice the soundproof beeper-bag around Karel's waist. This is used to carry the beepers Karel picks up. Karel can also determine if he has any beepers by poking around in the bag. Karel has one other interesting feature; that is, he can receive a set of instructions, memorize them, and then perform the actions associated with them.

6 However, before we can instruct Karel, we must know Karel's situation; that is, we must know Karel's position, the location and size of each wall section, and the location of each beeper, including those in Karel's beeper bag.

7 The situation before Karel starts is called the initial situation. The situation after Karel turns himself off is called the final situation.

8 Now we know who Karel is and what his world is like. We also know what it takes to describe his initial and final situation.

9 We are now almost ready to learn how to perform a task. But first, we should define some of the terminology we will be using. A task is something we want done. The detailed set of instructions which explains how to perform the task is called a program, and the language in which the program is written is called the programming language.

10 We will be using Karel his limited world, and a limited set of instructions to perform tasks.

11 There are five primitive instructions which Karel understands. The stop instruction informs Karel that he has completed his task. Karel performs a stop instruction by turning himself off, and he will not resume action until restarted by another task. The stop instruction must be the last instruction in every program.

12 The move instruction causes Karel to go forward one block in the direction he is facing. A move will not change the direction Karel is facing. Also, Karel will not move if he sees a wall section or a boundary between himself and the corner he is facing.

13 To change Karel's direction, the turnleft instruction is used. It causes Karel to turn 90 degrees to the left. A turnleft will not cause Karel to change his position, only his direction. Karel can always perform a turnleft.

14 The pickbeeper instruction causes Karel to pick up a beeper located at the same corner where he is standing and put the beeper in his beeper bag. When Karel picks up a beeper, he does not change his direction or location. If several beepers are present, Karel will pick up one and only one. Of course, if there is no beeper present, Karel will not be able to pick one up.

15 If we want Karel to take a beeper out of his beeper bag and place it at the corner where he is standing, we issue a putbeeper instruction. The putbeeper instruction will not cause Karel to move or change direction. It causes him to deposit only one beeper if he has any. If he has none, he will not leave one.

16 We now know Karel's five primitive instructions: stop, move, turnleft, pickbeeper, and putbeeper.

17 With the five instructions, we are ready to make Karel work. Suppose we want Karel to move a beeper from Second Street and Fourth Avenue to Fourth Street and Fifth Avenue.

18 First, we set up the initial situation. Then, we press Karel's start button.

19 Next, we read Karel the program, making sure that we include every word and all punctuation. Karel starts executing the instructions only after hearing ENDPROGRAM and continues until he turns himself off.

20 Of course, we don't really have a robot named Karel; therefore, we must simulate Karel's actions to verify the program. Let's see how this program works.

(Read and follow program instructions.)

Notice that the instructions all lie between the words BEGINEXECUTION and ENDEXECUTION. The instructions were executed sequentially; in other words, from top to bottom. All the instructions were executed, and Karel continued to perform until he turned himself off at the stop instruction. This program did not cause Karel to shut himself off prematurely; however, if the program contained an instruction which could not be executed, Karel would have performed an error shutoff.

Let's look at the program carefully and pay particular attention to the grammar and punctuation. The actions we wanted Karel to perform are instructions. They are always written in lower case letters. The other words (BEGINPROGRAM, BEGINEXECUTION, ENDEXECUTION and ENDPROGRAM) are called reserved words. They delimit different portions of the program and are always written in uppercase letters. The only punctuation in the program is the semicolon. Notice that it separates the instructions. For readability, the program has been indented; however, Karel is not affected by indentation.

21 Remember, reserved words delimit the different portions of the program. The rules for using reserved words are: every program must start with BEGINPROGRAM and end with ENDPROGRAM; every program must contain a BEGINEXECUTION and an ENDEXECUTION; and every BEGIN must have a matching END. Reserved words will always be written in uppercase letters.

Also, the only punctuation allowed in a program is the semicolon. It is used to separate instructions. Semicolons do not separate reserved words nor do they separate reserved words from instructions.

Indentation of program segments is stressed because it makes the program easier to read and aids in the detection of errors. However, Karel cannot hear indentation; thus, it does not affect him.

22 Don't tell Karel to perform the impossible because he can't. Instead, he will perform an error shutoff. That is, he will turn himself off. The three instructions which could cause an error shutoff are: MOVE - if he is blocked by a wall section or boundary; PICKBEEPER - when Karel is not next to one; and PUTBEEPER - when his beeper bag is empty.

23 Of course, no one expects to write errors in their program. Unfortunately, we do because programming requires an inhuman amount of precision. In principle, there should never be any errors since we know all the rules. In practice, there is an abundance of errors; therefore, we need to know about them.

24 There are four categories of errors.

Lexical errors occur when Karel hears a word not in his vocabulary and then turns himself off. For example, SIT is a word unknown to Karel and would cause an error shutoff.

Syntactic errors occur when Karel hears incorrect grammar or inaccurate punctuation and then performs an error shutoff. Examples would be putting a semicolon after a reserved word or not having an END for a BEGIN.

Execution errors occur when Karel cannot perform an instruction, so he turns himself off. Asking Karel to PUTBEEPER when he has none will cause an execution error.

An intent error is when Karel successfully executes the program but fails to successfully complete his task. For example, suppose we wrote a program where Karel was supposed to pick up all the beepers in his world. Then, after Karel performed all the instructions and turned himself off normally, there was one or more beepers left. An intent error would have occurred because the program executed correctly but it did not perform the task correctly.

25 Remember, Karel does not know _what_ you want him to do; all he knows is what you _tell_ him to do.

26 Suppose we want Karel to turn right. To perform this act, Karel must receive three turnleft instructions. This is simple to do but rather awkward.

27 Look at another clumsy aspect of robot programming. What if we need a program that will move Karel over vast distances? For example, assume Karel must move ten miles, pick up a beeper, and then move another ten miles. Because Karel understands about moving "blocks" but not "miles," we are forced to supply him a program which contains 160 move instructions. This may be a simple solution, but it produces a very long program.

28 The problem is that people think in one language and must program in another. Because shorter programs are easier to write and understand, it would be better to have Karel learn "new" instructions. Also, it is more desirable for Karel to learn new definitions rather than have us be slaves of the machine.

29 In order for Karel to learn new instructions, a dictionary of useful instruction names and their definitions must be provided. Each definition is built from simpler instructions which Karel already understands. The first definitions would be built using primitive instructions, and then these new definitions could be used to build more definitions.

30 Going back to the previous examples, we can now tell Karel that the definition of a turnright instruction is three turnleft instructions and that move mile is defined as eight move instructions. Whenever Karel needs to execute either of these instructions, he can simply refer to his dictionary and execute the instruction definition.

31 "Karel's definition mechanism defines a new instruction to have the same meaning as one other instruction." The form is: DEFINE-NEW-INSTRUCTION, <new name>, AS, <instruction>. We now have two new reserved words: DEFINE-NEW-INSTRUCTION and AS. DEFINE-NEW-INSTRUCTION signals Karel that a new instruction is being defined. AS separates the new instruction name from its definition.

32 To use the mechanism, replace <new name> with any word in lowercase letters or numbers. The two exceptions to this are the word cannot already be an instruction name, nor can it be a reserved word. The word may, however, be hyphenated to permit multiple word names.

33 <instruction> is replaced with the definition of <new name>. This includes any single instruction that Karel understands, primitive or previously defined.

34 Although the single instruction restriction is severe, the definition mechanism is still very useful. If Karel is ever sent to France, the French programmers could employ DEFINE-NEW-INSTRUCTION to translate their French instructions into English.

35 To build complex commands, we need to replace <instruction> with a sequence of instructions. This is accomplished through the use of block structuring. By placing a sequence of instructions between the reserved words BEGIN and END, one big instruction is produced. Indentation should be used to reinforce the idea that the BEGIN-END block represents one large aggregate instruction. Look more closely at this new grammar rule. Observe that the instructions inside the BEGIN-END block are separated by semicolons, while the first and last instructions are not separated from the reserved words BEGIN and END. BEGIN-END blocks and the BEGINEXECUTION-ENDEXECUTION block have equivalent internal punctuation. There may be as many instructions inside a BEGIN-END block as needed. The block is not necessary for a single instruction; however, it is grammatically correct.

The BEGIN-END block is executed by executing the instruction within the block sequentially, and once the block is being executed all the instructions inside that block will be executed unless Karel turns himself off.

36 "The fundamental property of the BEGIN-END block is that Karel understands the entire block to represent one instruction."

37 With block structuring, we can now define the new instructions of turnright and movemile.

38 "If you call a thumb a finger, how many fingers do you have?" "The correct answer is eight; calling a thumb a finger does not make it one." Similarly, "just because you define an instruction named turnright, it doesn't mean that the instruction really turns Karel to the right."

39 This definition is perfectly legal. Karel would execute a turnright instruction by executing two turnleft instructions. Karel doesn't understand what the instruction name is supposed to accomplish. He only understands and executes the definition. "Consequently, any new instruction we define may contain an intent error," as this one does.

40 "The rule is that Karel executes a newly defined instruction by executing its definition. Do not try to shortcut this process by doing what the instruction 'means.' Learn to interpret Karel's programs as literally as he does."

41 Another rule to follow is the instruction name should specify "what" the instruction does, while the definition should specify "how." If the two do not match, one of them should be changed.

42 Here is an example of a complete robot program. All the definitions are between BEGINPROGRAM and BEGINEXECUTION. Every instruction is defined before it is used. Also note that the new instruction definitions are separated by semicolons and that the last definition is separated from the reserved word BEGINEXECUTION by a semicolon.

43 The important things to remember about instruction definitions are:

1. They all go between the reserved words BEGINPROGRAM and BEGINEXECUTION.

2. Only instruction definitions may be located between BEGINPROGRAM and BEGINEXECUTION.

3. This portion of the program is called Karel's dictionary.

4. Each instruction must be defined before it is used or a lexical error will occur.

5. The definitions must be separated by semicolons.

6. A semicolon must separate the last definition from the reserved word BEGINEXECUTION.

7. Every program must contain a complete set of definitions for all new instruction names that it uses because Karel does not remember new instruction definitions forever. Each time he is started, his vocabulary reverts back to his original set of primitive instructions and reserved words.

44 Remember block structuring (making one large instruction out of a sequence of instructions by enclosing them in a BEGIN-END block)? What happens if the words BEGIN and END are omitted? To answer that question, we must understand how Karel breaks a program into its constituent components.

45 Define a unit as either an instruction, any type of BEGIN-END block, or an entry in Karel's dictionary.

Define boxing as the operation of drawing boxes around every unit in a program. Karel boxes every program read to him in order to determine what it means.

46 In this program, boxes have been drawn around every unit. The entire program is also considered a unit; however, a box has not been drawn around it.

"There are three important observations about boxing. First, the definition of a new instruction is the first box after the AS. Notice that the first unit in each of these definitions is a BEGIN-END block. Second, recognize that semicolons are placed between every pair of adjacent units. Finally, notice that units are frequently <u>nested</u> inside of other units."

47 The main geometric property of boxing is that boxes may be one inside the other, nested, or they may be adjacent; boxes may <u>never</u> overlap.

48 The way to box a program is to box the primitive instructions first and work outward. Start boxing at the beginning of the program and build the biggest boxes possible before starting boxes further down.

49 Now we can answer the question of what happens when the words BEGIN and END are omitted. Look how the turnright definition is boxed. Karel understands turnright to be defined as one turnleft instruction. The remaining turnleft instructions are not part of the definition and would cause a syntax error because they are not valid dictionary entries.

"Remember, Karel only hears the program read and does not know the indentation." As Karel is read the program, he checks for lexical errors and boxes the units in order to check for syntactic errors.

50 Review what we know (primitive instructions, definitions of new instructions, block structuring, types of errors, and boxing). Do you think you know enough to construct a robot program? Maybe, but first let's see how it should be done.

51 The method to use is called stepwise refinement. This method will lead to concise, simple to read, and easy to understand programs.

52 The steps to follow are:

1. Write the sequence of instructions in the BEGINEXECUTION-ENDEXECUTION block using any instruction names desired.

2. Write the definitions of the new instructions used in the BEGINEXECUTION-ENDEXECUTION block.

3. Write the definitions of the new instructions used in the BEGIN-END blocks of new instruction definitions.

4. Repeat step 3 until all instructions are defined.

53 Suppose we want Karel to harvest a field of beepers.

54 First, write the BEGINEXECUTION-ENDEXECUTION block. This is only one of many possible ways to solve this task.

55 Next, write the definitions for the new instructions. Notice that there are still new, undefined instructions within these definitions.

56 Write the definitions for the new instructions used in the definitions. Now everything is defined, so we are ready to assemble all the instructions into one robot program.

57 Remember to put the definitions in the proper order; each instruction must be defined before it is used.

58 By using stepwise refinement, the "harvest a field of beepers" task seemed to solve itself. Let's review what happened. First, the task had to be understood. Then, the task was broken into smaller, easier to understand, independent subtasks. Finally, the subtasks were solved thus producing the solution for the main task.

59 After a program has been written, it should be verified because stepwise refinement does not guarantee a correct program. In fact, programs should be considered guilty of being wrong until they are proven correct.

60 A correct program is important, but an understandable program is equally important. Understandable programs are easier to correct, and good programmers are separated from bad ones by their ability to write understandable programs.

61 What makes an understandable program? "A good program is the simple composition of easily understandable parts." This is accomplished through stepwise refinement. But breaking a program into small pieces is not enough. New instruction names should provide a description of how the program accomplishes the task. Then, after the instruction has been proven correct, the name will help us to remember what the instruction does.

62 Why bother writing understandable programs? The answer is because the programs will be easier to read, easier to verify, easier to correct, and easier to change in the event that the task is slightly changed. Also, "complete programs are often so useful that we want to include them as instructions in even bigger programs." "This conversion is relatively straightforward because all we must do is define a new instruction whose definition is the sequence of instructions within the BEGINEXECUTION-ENDEXECUTION block." By doing this, we will save time because we won't be solving the same task many times.

63 Now that we know the prerequisite for writing a good program, let's extend our programming vocabulary. "Karel is built to understand two similar IF instructions, the IF-THEN and the IF-THEN-ELSE instruction." Both of these instructions command Karel to test his present situation and execute an appropriate instruction depending on the result of the test.

64 The IF-THEN instruction introduces two new reserved words: IF and THEN. "The word IF signals Karel that an IF instruction is present. The <instruction>, which is known as the THEN clause, is separated from <test> by the word THEN. The THEN clause is an instruction nested inside the IF instruction."

65 When Karel executes the IF-THEN instruction, he first checks to see if <test> is true or false in the current situation. If <test> is true, he executes the THEN clause. If <test> is false, he does not execute the THEN clause.

66 For example, let's simulate this program segment. First, Karel checks whether or not he is next to a beeper. If he is, he will execute the pickbeeper instruction. The IF-THEN instruction being completed, Karel now continues executing the program at the move instruction. But what happens if there are no beepers? Karel would check if he was next to a beeper and determine that the <test> was false. Karel would not execute the THEN clause but would continue executing the program starting at the first instruction following the IF-THEN instruction: move. "An error shutoff cannot occur in either case because Karel executes the pickbeeper instruction only when he confirms the presence of at least one beeper on the corner."

67 Here is the complete list of the vocabulary words that can be substituted for <test>. "Notice that each condition is represented in both its positive and negative form."

68 Karel has TV cameras for eyes and can turn his head to the left and to the right. Karel's front, left, and right are respectively clear if he does not see a wall between himself and the next corner in the direction he is commanded to look. "The next-to-a-beeper test is true when Karel is on the same corner as one or more beepers. He cannot hear beepers any further away, and he cannot hear beepers in his soundproof beeper bag. Karel has an internal compass, and he consults it to determine which direction he is facing." "Finally, Karel can test whether he has any beepers left in his beeper bag by reaching in with his arm."

69 Remember the harvest-a-field task we solved earlier? For that task, there was a field of beepers with one beeper located on every corner. Now, suppose we have the same task, but the initial situation has changed. Instead of having one beeper at every corner, there are now barren corners. The original program would cause Karel to perform an error *shutoff at the first corner without a beeper.*

70 Changing the original program is relatively easy. In fact, we can reuse almost all of the previous solution; that's one of the advantages of good programming. The only thing that must be modified is the harvest-a-furrow instruction.

71 It now looks like this. Of course, the pick-beeper-if-present instruction must be written, but this is easily done using the IF-THEN instruction.

72 The BEGIN-END block around the definition is unnecessary in this example, because the definition contains only one instruction, the IF-THEN. "We shall adopt the convention of always defining a new instruction by putting its definition in a BEGIN-END block." It is recommended that you do this also but, more importantly, you should know whether the block is necessary or redundant.

73 Here is an example of how to use a BEGIN-END block inside the IF-THEN instruction. In this example, we are defining an *instruction* which will turn Karel around only if his front is blocked. The BEGIN-END block enclosing the instruction definition is redundant; however, the BEGIN-END block nested in the THEN clause is necessary.

74 Look at what happens when this block is removed. It still looks correct, but is it? Do not let the *indentation fool you.* Karel doesn't know how the program is indented; therefore, he does exactly what he is told. If his front is blocked, he turns around. If his front is not blocked, he turns left.

75 This is how the instruction is boxed by Karel. Without a BEGIN-END block in the *THEN* clause, Karel will always execute at least one turnleft instruction. "This illustrates a subtle intent error." "It should start to dawn on us that programming errors are indeed possible, and it requires effort to understand the meaning of an instruction." Therefore, do not passively test an instruction, but try to think of special situations where the instruction might fail.

76 To box an IF-THEN instruction, first box the instructions in the THEN clause, which may be a single instruction or a BEGIN-END block. Then, box the entire IF-THEN instruction. This includes the reserved words IF and THEN, the test, and the previously boxed THEN clause. Note the punctuation. The IF-THEN instruction is separated from the instructions by a semicolon, and the last instruction in each of the BEGIN-END blocks does not have a semicolon.

77 The second kind of IF instruction is the IF-THEN-ELSE. It is similar to the IF-THEN instruction except for the ELSE clause. The reserved word ELSE separates the ELSE clause from the THEN clause. The ELSE clause is nested inside the IF instruction after the THEN clause. The THEN and the ELSE are indented identically. There should not be a semicolon separating the THEN clause from the ELSE clause.

78 To execute the IF-THEN-ELSE instruction, Karel first checks to see if <test> is true or false. If <test> is true, Karel executes the THEN clause. If <test> is false, he executes the ELSE clause. Karel will always execute either the THEN clause or the ELSE clause, but he will not execute both.

79 Suppose we wanted Karel to run a mile hurdle race with wall sections representing the hurdles.

80 One possible solution could be to have him perform eight race-stride instructions if the definition of race-stride looked like this. Of course, the jump-hurdle instruction would have to be defined.

81 Observe these two instructions. While they are different in form, they execute equivalently. In fact, we can always derive an equivalent IF-THEN-ELSE instruction from another one by negating the test and switching the THEN and ELSE clauses. "This gives us the extra freedom to arrange IF-THEN-ELSE instructions to read naturally."

82 "When there is no preference for which test is best, one suggested rule is to always use the test that makes the THEN clause smaller." This keeps the THEN and ELSE clauses visually close to the IF so that we can easily see that the instruction is an IF-THEN-ELSE. "If both clauses are large, indent them further to help show where the IF-THEN-ELSE ends."

83 To box the IF-THEN-ELSE instruction, first box the THEN clause, then box the ELSE clause. Finally, box the entire IF-THEN-ELSE instruction.

84 Do not forget to use BEGIN-END blocks inside the THEN and ELSE clauses. Look at the definition of race-stride without the BEGIN-END block. After Karel boxes the instructions, he encounters an ELSE. But because he is not boxing within an IF instruction, Karel reports a syntax error. The reserved word ELSE cannot appear outside of an IF instruction. "A similar error occurs if we accidentally put a semicolon between the THEN clause and the reserved word ELSE."

85 If the BEGIN-END block is mistakenly omitted from an ELSE clause, a subtle intent error will occur.

86 By now, you should have learned two very important lessons. The tactical lesson is never forget BEGIN-END blocks. The strategic lesson is the bigger the instruction, the more complicated it becomes.

87 There is one last class of IF instructions. "These are known as nested IF instructions because they involve an IF instruction nested inside a THEN or ELSE clause of another IF." No new evaluation rules are necessary to execute nested IF's, but we will have to pay closer attention to the established rules. Because it is easy to lose track of where we are in the instruction, simulating a nested IF instruction is difficult.

88 This is an example of a nested IF instruction. Notice how the inner IF instruction is boxed within the ELSE clause of the outer IF. Also note that in this example, the inner IF instruction is the last instruction in a BEGIN-END block; therefore, there is no semicolon separating it from the reserved word END. Look how the two END instructions are placed back to back. This will cause all the instructions to finish at the same time.

Let's simulate Karel's actions for the three possible corner situation. If Karel is at a corner where there are no beepers, he executes the outside IF and determines that he is not next to a beeper. "Therefore, he executes the putbeeper instruction in the THEN clause of the outside IF instruction, leaving one beeper on the corner." "Now assume there is one beeper on the corner." Karel executes the outside IF, finds the test is false, and he executes the ELSE clause. This is a BEGIN-END block composed of two instructions; he executes pickbeeper first, picking up the only beeper on the corner. Now, he executes the nested IF instruction and finds there are no more beepers on the corner. Therefore, he executes the THEN clause of this instruction, which puts a beeper back on the now empty corner. Karel is done with the block, the ELSE clause, the outer IF, and the entire replant-exactly-one instruction. "Finally, assume Karel is on a corner with two beepers. Karel executes the outside IF, finds the test is false, and therefore executes the ELSE clause. He starts the BEGIN-END block by executing pickbeeper first, picking up one of the two beepers on the corner." "He executes the nested IF instruction and finds there is still a beeper on the corner, so he skips the THEN clause. Once again, Karel is finished with the nested IF, the outside IF, and the entire instruction, and he has left one beeper on the corner." If leaving one beeper on every corner that Karel visits was our purpose, we have now verified that this instruction is correct. As we have just seen, nesting IF instructions can get complicated. Therefore, it is advisable to avoid nesting more than two levels deep. This example contains one level of nesting.

89 Look at these two nested IF instructions and carefully study how each instruction is boxed. The only difference between these two instructions involves the boxing of the ELSE clause. The first instruction has the ELSE boxed with the nested IF, while the second instruction has the ELSE boxed with the outside IF. Obviously, these are two different instructions, but Karel cannot tell them apart because they both contain the same words arranged in the same order. This is known as the dangling ELSE problem. To solve the dangling ELSE problem, we must know how Karel would box this instruction.

90 The rule is: whenever Karel reads an ELSE clause, he boxes it with the most recently read IF instruction that it can be a part of. This means that the ELSE clause is boxed with the first possible preceding IF instruction.

91 In the event that we want the ELSE clause to be boxed with the outer IF, a BEGIN-END block would be used. The reserved word END forces Karel to conclude the nested IF instruction. "Thus, when the ELSE is finally read to Karel, there is only one IF instruction he can match it with."

92 Let's review the IF instruction. First, Karel decides if the <test> is true or false. If the <test> is true, Karel performs the THEN clause and then continues with the first statement following the IF instruction. If the <test> is false, Karel performs the ELSE clause if there is one present and then continues with the first statement following the IF instruction.

93 Here are the five forms of the IF instruction that we have seen. The first two are not nested, and the last three are.

94 There are only two more instructions built into Karel's vocabulary: the ITERATE and WHILE instructions. Both of these instructions give Karel the ability to repeatedly execute any instruction he understands.

95 This is the general form of the ITERATE instruction. It is a shorthand notation used to tell Karel to repeat another instruction a specified number of times. ITERATE and TIMES are two new reserved words, and <positive-integer> represents the number of times we want Karel to repeat the <instruction>. "We refer to <instruction> as the body of the ITERATE instruction, and we shall also use the term ITERATE loop to verbally suggest that this instruction loops back and executes itself . . ."

96 Remember the turnright instruction? By using the ITERATE instruction, we can accomplish the same task with less effort.

97 Suppose we would like an instruction which makes Karel face east. Obviously, the definition on the left will work, but using the ITERATE will again accomplish the same task with less effort.

98 As with IF instructions, ITERATE instructions may also be nested. Look at this instruction. "Assuming no blocking walls, this instruction moves Karel in a square whose sides are six blocks long." To understand what happens, let's simulate Karel's actions. Karel first hears the words ITERATE 4 TIMES and understands that he execute the following instruction four times. He enters the BEGIN-END block and receives the words ITERATE 6 TIMES. Karel understands that he must execute the following instructions six times, so he performs six moves. Being done with the nested ITERATE instruction, KAREL turns left. Now, Karel has finished the BEGIN-END block or the body of the outer ITERATE loop. But wait, Karel remembers that

he is supposed to perform this loop four times and he has only done it once. So again, Karel enters the BEGIN-END block of the outer ITERATE instruction. He receives the words ITERATE 6 TIMES, and once again he performs six move instructions. After moving six blocks, Karel then turns left. Once again, Karel has completed the outer loop body. Having gone through the outer ITERATE loop twice, Karel will now proceed to go through it two more times. When he has finished, Karel will have executed twenty-four moves and four left turns.

99 In other words, the inner loop will be performed completely for every single execution of the outer loop.

100 Observe how the ITERATE instruction is boxed. First, the instructions in the body are boxed. Then, box the complete ITERATE instruction. If there are nested ITERATE instructions, box the innermost one first and work outward.

101 Until now, we have been limited to programs where we knew how many times an instruction had to be executed. But what happens when Karel's task is to move forward until he finds a beeper?

102 Here are two possible solutions. Unfortunately, Karel doesn't understand either of them because he doesn't know how to interpret the dots or the question mark. What we need is an instruction which has Karel perform an action until a condition becomes false.

103 "The WHILE instruction allows Karel to continually repeat any instruction until a condition becomes false." "The new reserved word WHILE starts the instruction, and DO separates <test> from the body of the WHILE". The conditions which can replace <test> are the same as those for the IF instruction."

104 Karel executes the WHILE instruction according to the following rules. "If <test> is false, Karel is done with the WHILE instruction, and he continues executing instructions following the entire WHILE loop. If <test> is true, Karel executes <instruction> and then re-executes the entire WHILE instruction."

105 Now we can solve the task of having Karel move forward until he finds a beeper.

106 There is a very important property of the WHILE instruction: when the WHILE instruction is finished executing, the <test> is known to be false. "We can use this property when writing programs that require a certain condition to be true at some time."

107 "Whenever <test> must be true, we write something similar to this WHILE instruction into our program" The <not-test> is the negative form of the condition we are currently testing. As long as the negative form remains true, the WHILE loop will be executed; however, as soon as the negative form becomes false, in other words when the condition becomes positive, the loop will be finished.

108 Look at an example of how the <not-test> condition is executed. The definition of the face-east instruction can now be written with the WHILE instruction. The not-test condition is not-facing-east. When this loop is finished, the not-facing-east condition is guaranteed to be false; thus, Karel is facing east.

109 What would happen if Karel executed this WHILE instruction? Note that the body of the WHILE loop is a turnleft instruction and not a pickbeeper. Because turning left will never make the condition false, Karel will remain at one corner and continue turning indefinitely. "We say he is stuck in an infinite loop."

110 An infinite loop occurs when the instructions in the body of a WHILE instruction do not cause Karel to progress toward his goal. Fortunately, the WHILE instruction is the only instruction which has the ability to cause an infinite loop. Unfortunately, infinite looping is a kind of intent error, because Karel cannot detect when he is stuck in one.

111 Carefully read this example of a WHILE instruction. Notice that it does not get Karel stuck in an infinite loop. "This instruction has Karel pick up a line of beepers, finishing when he picks up the last beeper in the line or when there is a gap in the line. The instruction is interesting because it is the first WHILE loop body we have seen that is a sequence of instructions and therefore requires a BEGIN-END block." This poses a very interesting question: when does Karel test the condition and exit the loop? "A common misconception is that Karel checks test after each instruction he executes in the loop body." However, this is true only when there is one instruction in the body.

112 The correct rule is: check the condition each time before entering the loop body. Once in the loop, execute all the instructions. Then recheck the <test> after the body has been completely executed.

113 Given the task of harvesting a line of beepers whose end is marked by a wall, let's see how we should use the WHILE instruction.

114 Here we see Karel's initial situation and an instruction that looks like it will solve the task. Let's simulate Karel's actions and see if they would really work. While Karel's front is clear, he will pickbeeper, move, check the test, pickbeeper, move, check the test, pickbeeper, move, and stop. Observe that the last beeper did not get harvested. Of course, the situation can be easily remedied by inserting another pickbeeper after the WHILE instruction; however, it is preferred to have the special case first to make it more visible.

115 This would be a correct and more understandable definition. Notice that the extra pickbeeper is before the WHILE instruction and that the instructions in the loop body have been switched. The WHILE instruction was used in this example, but the ITERATE instruction could have been used because they are both structurally identical.

116 "What would Karel do if we read him the previous instruction but forgot to put in the BEGIN and END delimiters in the body of the WHILE instruction?" Karel would box the instruction into three instructions. He would then pick the first beeper, move until he saw the wall, and then pick the last beeper. This is an intent error, because Karel understood and executed all the instructions but he did not perform the required task. Remember, multiple instruction WHILE and ITERATE loop bodies need BEGIN-END blocks.

117 Another cause of trouble and misunderstanding is the use of an IF instruction inside a WHILE loop. "Confusion arises because both of these instructions perform tests and have similar execution rules. But by keeping a level head and applying the rules we already know, this type of instruction is easily simulated."

118 Give Karel the task of harvesting all the beepers between his starting corner and the wall. Here is an example of one possible initial situation and a refinement of the original harvest-to-wall instruction.

"The execution of this instruction in the initial situation starts by having Karel pick up the first beeper. Karel's front is clear so he moves ahead and executes the IF instruction, checking for a beeper. Because he is next to one, he executes the THEN clause and picks it up. The body is now completely executed; therefore, Karel re-executes the WHILE loop. Once more, he checks whether his front is clear. It is, so he executes the loop body by moving ahead and executing the IF. But this time Karel is not next to a beeper, so he does not execute pickbeeper. Again, he is done with the IF and done with the body of the loop. Therefore, he re-executes the WHILE loop. He continues in this manner until his task is successfully completed."

119 "Here is a type of syntax error that is difficult to spot." Do you see what is syntactically wrong? "You either see it or you do not, so studying the instruction is probably futile. The problem is the reserved word DO does not belong in the ELSE clause." "This is an example of another error that is hard for the human mind to see but trivial for Karel's small but precise intellect to spot. Many times, programming errors are hard to find, but once spotted they are obvious."

120 Writing a program is not always an easy job, even when you know all the rules. Therefore, let's develop a program in a logical manner, commit the mistakes, find the mistakes, and rewrite the program until it is correct.

121 Karel's task is to escape from any rectangular room with an open doorway exactly one block wide.

122 This illustrates one possible initial situation. Using this situation, we will obtain a general plan for escaping from rooms. As we develop the program, be on guard for errors and special situations which would make the program incorrect.

123 This program accomplishes the task. "Initially, Karel starts somewhere in the room facing some arbitrary direction. He starts the task by moving to the wall that he is initially facing. Karel then follows the walls of the room in a counter-clockwise direction until he senses the door, keeping his right side to the wall. He exits through the door and finally stops." We must now define the instructions go-to-wall, follow-until-door-is-on-right, and exit-door.

124 The go-to-wall "instruction moves Karel forward until he senses a wall directly in front of him. The test we eventually wish to be true is front-is-blocked, so using the formal WHILE property, we should be able to write this instruction."

125 "Although this simple instruction works in the initial situation we saw earlier, it does not work correctly in all initial situations. Unfortunately there are some initial situations where the WHILE instruction never terminates, leaving Karel in an infinite loop. These situations are characterized by Karel initially facing the door instead of one of the walls. When Karel executes this go-to-wall instruction in such a situation, he would zoom out of the room without knowing he exited and would keep on moving."

"We call this type of situation a beyond-the-horizon situation. Normally programming is guided by a few sample situations that seem to cover all of the facets of the problem. Although we would like to prove that all other situations are not too different from these, frequently the best we can do is hope. But as we learn more about the task, we might uncover situations that are beyond our original horizons. These situations are legal, but special trouble-causing cases. Once we have discovered a beyond-the-horizon situation, we might have to change our program to account for it."

These situations are usually hard to find because they are not intuitively obvious. "Good programmers become skilled at extending their horizons and finding dangerous situations that interfere with programs accomplishing their tasks."

126 To correct the program, the go-to-wall instruction will have to be modified; however, the beginning decomposition is still valid. "The fact that the door is only one block wide is the key" to the problem.

127 "We shall program Karel to move forward in a right shuffling motion. He will check for walls directly in front of him and in front of the corner on his right. In this way, he is guaranteed to not pass through the door unnoticed." But even this correction has a hidden problem.

128 There is now a new beyond-the-horizon situation where Karel cannot shuffle right because of a wall.

129 This problem is easily corrected by placing a test in the go-to-wall instruction. "These instructions are now correct and will work in any room situation."

130 "We continue by writing the follow-until-door-is-on-right instruction. Recall that the initial decomposition has Karel execute a turnleft instruction after go-to-wall; therefore, we can assume that Karel's right is blocked by one of the room walls."

131 The follow-until-door-is-on-right instruction must satisfy two criteria: "It must finish when Karel senses a door on his right-hand side, which can be sensed when Karel's right becomes clear. If a door has not been found, the instruction must keep Karel's right side adjacent to a wall while following the perimeter of the room in a counter-clockwise direction."

"We call a condition that must be true during the execution of a portion of a program an invariant. The invariant for the second criterion is that Karel's righthand side be within one half block of the room wall as he follows the perimeter. To do this, Karel moves forward except when he reaches a corner. In this situation, he turns to the left, ready to parallel the next wall."

132 Again, we will use the formal property of the WHILE instruction. Now Karel will follow the perimeter of the room until the door is on his right.

133 The follow-perimeter instruction has Karel move forward until he comes to a corner of the room. Once at the corner, Karel turns left and again is ready to move forward. The invariant that his right side remain next to the wall is always true.

134 The last instruction we have to write is exit-door.

135 Because the door must be on Karel's right, here is how the instruction would be written.

136 The program is now written and must be assembled. The entire final program is now presented.

137 As one final note, here are two beyond-the-horizon situations. In both, Karel "successfully escapes the room and stops, but not quite in the way we might expect." And there might be other beyond-the-horizon situations that would force the program to fail.

138 The last new instructions we will look at are zig-northwest and zag-southeast. These instructions move Karel diagonally northwest and southeast.

139 "Both of these instructions are simply defined using Karel's primitive instructions." However, with the aid of these diagonal moving instructions, a novel set of beeper manipulation programming problems will be easily solved.

140 The zig-northwest instruction allows Karel to move northwest until he is restrained by the western boundary wall. Note that there is nothing forcing Karel to move northwest; thus, for this instruction to execute correctly, Karel must be facing west when the instruction starts. This is called a precondition. "A precondition of an instruction is some condition that must be true before the instruction can be correctly executed." Also, notice that Karel is facing west when the instruction has been completed. That means the precondition is invariant over the instruction. Finally, Karel's front must be clear for the zig-northwest instruction to execute correctly; otherwise, an error shutoff will occur.

141 The zag-southeast instruction allows Karel to move southeast until he is restrained by the southern boundary wall and has similar properties as zig-northwest. The precondition for zag-southeast is: Karel must be facing south. Again, the precondition is invariant over the instruction and Karel's front must be clear.

142 Give Karel the task of finding a beeper in an unenclosed area.

143 The obvious solution would be to put Karel at the origin, face him east, and have him move forward looking for a beeper. If he doesn't find one, have him go to Second Street and continue searching. Karel would continue this strategy until he found a beeper. But this solution won't work because First Street extends forever and Karel would be caught in an infinite loop.

144 What is needed is a search pattern that will take Karel to every corner within an area and then expand the area if necessary. This search pattern does just that and will solve the task.

145 The find-beeper instruction can be written to implement the search pattern. It "starts by moving Karel to the origin and facing him west." "This establishes the precondition for zig-northwest. The WHILE loop's purpose is to keep him moving until he finds a beeper, and by the formal WHILE property it is correct if the loop always terminates. The IF instruction nested in the body of the loop continues moving Karel in the direction he is facing."

146 "The moving instructions zig-move and zag-move operate similarly. They move Karel diagonally to the next corner as long as he is able, otherwise he is blocked by the boundary wall and must advance to the next diagonal."

147 "The advance-to-next-diagonal instruction starts its job by facing Karel away from the origin; he turns a different direction depending on whether he has been zigging or zagging. Karel then moves one corner farther away and turns around. If Karel was zigging when he executes advance-to-next-diagonal, he will be positioned to continue by zagging and vice versa."

148 The last and one of the most interesting aspects about Karel is that we can program him to perform addition.

149 We can represent an addition problem by placing a beeper on the question corner and have Karel deposit it on the answer corner. The street and avenue numbers for the question corner (called S and A, respectively) represent the numbers to be added. The answer corner will always be on First Street and A + S Avenue.

150 "This problem can be broken into two separate phases. In the first phase, Karel must locate the question corner and pick up the beeper. This can be accomplished by executing the find-beeper instruction followed by a pickbeeper instruction. During the second phase, Karel computes the sum of the two numbers and puts the beeper down on the answer corner."

"The second phase can be accomplished by having Karel zag down toward First Street."

151 Let's see why zagging helps solve the problem. Suppose Karel finds a beeper on Sixth Street and Third Avenue. By performing a zag southeast instruction, Karel will move to Fifth Street and Fourth Avenue. "This occurs because a zag-southeast instruction decreases the street number Karel is on by one and increases the avenue number he is on by one. The invariant for this problem is: the sum of the street number and avenue number that Karel is on is always S + A."

152 By executing a zag-southeast, Karel has preserved the invariant because 6 plus 3 equals 5 plus 4. In other words, S + A equals S - 1 + A + 1.

153 "If Karel continues performing zag-southeast instructions whenever his front is clear, he will continually move south until he arrives on First Street. Our invariant equations tell us that:" S + A will equal 1 + A + S - 1.

"Now that Karel's street position is 1, his avenue position will be A = S - 1. To complete the sum, all Karel has to do is to move one avenue east and put down the beeper. He will then be on the answer corner of First Street and (A + S)th Avenue."

154 The definition for the compute-sum instruction can now be written. The face-south instruction must be included before the WHILE loop to insure that Karel executes zag-southeast correctly. The face-east instruction is necessary because Karel will be facing south when the WHILE loop is completed, and he must advance one block east before he can deposit the beeper.

155 We now know how to write a robot program. The first step is to solve the task. To do this, first we must understand the task. Then break the task into small, independent subtasks. The subtasks are then solved within the constraints of the language. After all the subtasks have been solved, they are combined to form the solution to the original task.

156 The second step is to correct the program. This means eliminating all the lexical and syntax errors.

157 The third and final step is to simulate the program. By simulating the program, we should be able to determine if the program executes and if it performs the intended task.

158 You have now seen all the rules, so good luck and start programming!

A NEW PEDAGOGICAL APPROACH TO TEACHING PROBLEM SOLVING

by

SUSAN MARGARET CARROLL

B. A., Emporia State University, 1973

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

ABSTRACT

With the ever-increasing demand for computer programming knowledge, it has become necessary to define a method for the instruction of problem solving which will aid in the development of analytical thinking.

The Karel approach to the teaching of problem solving utilizes a mobile, programmable robot. Through the use of Karel, it is possible to teach problem solving, the primary computer language concepts, and introduce the rigorous demands of regimentation required for computer programming. Although the Karel language is not a true "computer language," it is similar to many high-level languages, especially PASCAL, in the areas of punctuation, block structuring, conditional testing, looping, and subroutines.

This paper examines the Karel method, explains the behavioral objectives to be obtained, provides the rudimentary means for implementation, and makes recommendations for future study.