

A LANGUAGE TO LEARN  
PROGRAMMING  
CONCEPTS

By

VIRGINIE COCHARD

Software Engineer Diploma

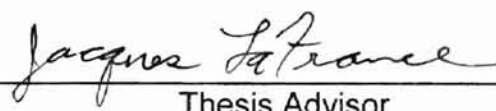
ENSEEIH T Toulouse, France

1990

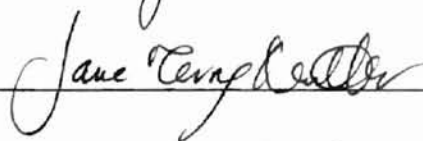
Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
MASTER OF SCIENCE  
December 1999

A LANGUAGE TO LEARN  
PROGRAMMING  
CONCEPTS

Thesis approved:

  
Thesis Advisor





  
Dean of the Graduate College

## TABLE OF CONTENTS

Chapter	Page
<b>I. INTRODUCTION.....</b>	<b>1</b>
1. Growing need for skilled software professionals.....	1
2. Problems when learning a high-level programming language .....	1
3. Concerns for Computer Science Education .....	2
4. New approach .....	2
5. Outline of thesis .....	3
<b>II. BACKGROUND.....</b>	<b>4</b>
1. Computer Science Education concerns.....	4
2. Existing approaches .....	12
<b>III. THE NEW ANTFARM .....</b>	<b>26</b>
1. Concerns addressed.....	26
2. The New Antfarm environment.....	33
3. Details of new contribution and examples.....	34
<b>IV. CONCLUSION.....</b>	<b>39</b>
<b>V. BIBLIOGRAPHY .....</b>	<b>42</b>
<b>VI. APPENDICES .....</b>	<b>44</b>
A1. Karel syntax description .....	45
A2. Karel++ syntax description.....	47
A3. Joseph syntax description .....	49
A4. Antfarm syntax description.....	52
A5. New Antfarm syntax description .....	54
A6. Logo syntax description.....	55
A7. New Antfarm user manual .....	57

# **I. INTRODUCTION**

## **1. Growing need for skilled software professionals**

With the fast pace the computer and software industry has taken on, with the cost of the software going up and with the increasing complexity of the systems, there is and undoubtedly will continue to be a need for adequately trained software programmers. Programs need to be well designed and written to limit the 'bugs' in them and to lower the cost of maintenance. Programs nowadays rarely involve only writing lines of code in a specific language. Programs are now becoming complex systems that link several applications together such as networking, databases, tying to industrial machines, etc. So the programmers have to spend more time mastering those applications, and should thus be well versed in programming into a high-level language.

## **2. Problems when learning a high-level programming language**

Learning a programming language first as an adult can be an intimidating task. Children usually do not have this 'fear' of the computer, but high-level languages such as Basic, Pascal, Fortran or C are not appropriate languages to teach to children, because they are too complicated and not very exciting.

Mastering a high-level language involves learning many concepts such as variable declaration, assignment operation, data types, procedures, functions, input/output parameters, top-down design, etc., as well as dealing with syntax errors, in order to write even fairly easy programs. Teaching programming languages at the Community College, I see the difficulty some students experience in grasping so many new abstract concepts at once. And, by today's standards with the Windows-like interfaces, the results of the



execution of such programs are not very spectacular nor thrilling, displayed usually in a plain DOS window.

### **3. Concerns for Computer Science Education**

The goal of this thesis is to provide an easily assimilable introduction to programming languages containing key concepts common to most standard programming languages as well as good programming techniques, accessible to children, college students and adults. Basic concepts a future programmer ought to know are problem solving, structure programming constructs, modularity, top-down design and basic data concepts. In order not to put a burden on the user, the learning process ought to be facilitated, for example by creating an efficient user-interface and an entertaining tool, and by using as simple a syntax as possible.

### **4. New approach**

The project of this thesis is to address the above concerns by creating a programming language which children and adults can learn and use easily. The idea of a language designed to teach programming is not new. The study of such existing languages shows however that they do not address all the concerns mentioned. Data manipulation is taught in the first lessons of a programming language class. However, none of the existing languages I found introduce the concepts of data in a satisfying way, if at all. Part of the new approach is to use a good metaphor for teaching data concepts. It is necessary to learn the syntax of a language to program in that language. In the new language, I seek to use a syntax which is as close as possible to English. This moves the

user's attention away from learning a formal language to focus on problem solving, basic control issues and data manipulation.

## **5. Outline of thesis**

Chapter II lists the concepts that a student should learn in order to be an effective programmer, and problems encountered in the learning process. It then reviews the existing tools that teach programming. Chapter III presents the new language created, and explains how it addresses the concepts we seek to teach. The appendices provide a detailed description of the languages studied, and include the New Antfarm manual.

## II. BACKGROUND

### 1. Computer Science Education concerns

According to Jacques LaFrance [10], in 1982 many people were becoming “programmers” without any awareness of the modern concept of proper design or of program quality level, having no conception of programming style, documentation or maintainability. Still now, at the horizon of the year 2000, it is essential that we focus on teaching good programming practices. Learning how to program is more than just learning the syntax of a language. It is learning a methodology and acquiring good practices to create programs that are well designed, easy to read, and to maintain.

A huge literature exists on software engineering concepts and effective programming techniques. Research has also been conducted on the difficulties that student programmers encounter (see for example [5, 6, 15, 17]).

#### *Problem solving concepts*

A key to programming is to be able to conceive a solution for a problem. Programming requires thinking, solving, planning, and organizing. Pseudo-code and flow charts are often used to help students in formulating the solution to the problem, without worrying about the targeted programming language.

Research confirms that problem solving skills are critical. Bonar and Cunningham state that “... success with programming seems to be tied to a novice’s ability to recognize general goals in the description of a task, and to translate those goals into actual program code” [4]. In a study by Spohrer and Soloway [20] it appears that “...many bugs arise as a result of *plan composition problems* – difficulties in putting the

“pieces” of a problem together – and not as a result of *construct-based problems*, which are misconceptions about language constructs.” An example of plan composition problem is the tax program. Depending on the marital status entered by the user, the tax computation for single or married will be called and the results are then printed. The bug occurs when the results are printed even when the value entered is not ‘m’ (for married), or ‘s’ (for single). The buggy program looks like (in Pascal):

```
IF (Marital_Status = 'm')
THEN MARRIED(Income, Tax)
ELSE IF (Marital_Status = 's')
    THEN SINGLE(Income, Tax)
    ELSE WRITELN('Bad input: Try again!');
WRITELN('Status =', Marital_Status);
WRITELN('Income =', Income:0:2);
WRITELN('Tax =', Tax:0:2);
```

Examples of construct-based problems are the “natural-language problem” where “novices become confused about the semantics of the constructs”, and the “human interpreter problem” where novices assume that the computer will interpret the construct the way they intend it to be interpreted [20].

In her paper on Computer Programming in High School vs College [16] Schollmeyer concludes: “To best prepare high school students for CS courses in college, the emphasis on the high school level should not be the instruction of the syntax of a programming language, as commonly observed. Instead, students’ problem solving skills should be addressed. This includes the emphasis on program specification and design, in particular the solving of subproblems rather than attacking the problem as a whole. A structured programming language usually provides students with better problem skills, which in addition to the programming concepts, are the skills deemed essential for success at the college level.”

Problem solving can be taught independently of a language and can then be applied to any language. And if the students do not end up being programmers, programming can help transfer problem solving and thinking skills to other areas such as writing. Dennies made three studies on the usefulness of teaching programming. “In two of them, ninth graders who learned structured programming methods using the Karel the Robot teaching language performed considerably better on a series of expository writing tasks than did students in the studies’ control groups. In the third study, students who began their introductory programming methods course with Karel performed substantially better on difficult structured programming tasks using Pascal“ [8].

### ***Structured programming constructs***

“Bohm and Jacopini [3] (...) proved that any logical system can be reduced to a combination of three basic logic forms. These forms are called sequence, selection, and iteration. A sequence is one operation after another; a selection is the making of a choice between two or more alternatives; and an iteration is the repetition of something until some terminating condition is met. Limiting ourselves to those basic structures is called structured programming.” [11]. The trend has been to omit the GoTo statement (unconditional branching) to prevent the students from writing unstructured programs.

Even though it has been stated that bugs arise more due to poor problem solving skills [4, 16, 20], students encounter many problems when using control structures, even the most simple construct: the sequential control structure. “What sometimes gets forgotten is that each instruction operates in the environment created by the previous instructions. Some beginners seem to imagine that the effects of each instruction are somehow saved up until the end of the program, at which point they all happen.” Some

students think "the system will of itself jump around and ignore sections of code which are not wanted under some circumstances" [5].

Several errors with the iteration control structure involve incorrectly determining when loops begin and end, and which statements are repeated. As an example given in a study of errors by high school students using Pascal [17], a WRITELN statement adjacent to the loop was included into the loop by some students. Also some students thought that a BEGIN/END block defined a loop despite the absence of a FOR or WHILE statement. "FOR loops are troublesome because beginners often fail to understand that behind the scenes the loop control variable is being incremented on each cycle of the loop".

### ***Modularity and top-down design***

Another major consideration in good programming is the modularity of programs, that is, to divide the initial problem into subproblems, which themselves can be divided into subproblems. Programs made out of such modules are more readable, easier to test and to maintain. Schollmeyer states [16] that "if students keep approaching problems by immediately attacking them rather than breaking them down into manageable chunks, they may never be able to write good, readable, and correct programs." Perkins et al. also state that "students do not always recognize that breaking problems down will aid in the solution of programming problems. (...) Students face trouble in breaking problems down because they often try to deal with decomposition issues in the middle of coding, instead of planning deliberately in advance." [15]

Top-down design is about breaking problems down into subproblems. Tomek [14] thinks it is important for the student to be able to write a program in the order it is designed, that is. starting with the main program, and then progressing to the more and

more detailed subprograms. Thus the student must be able to call subprograms that have not yet been defined. A program written this way will be easier to read, understand, and modify.

Modularity is created through the use of procedures. In a study of Pascal and high school students [17], the authors found the following problems when students used procedures: “All statements including those in procedure bodies were executed in the order they appeared (a frequent error); and procedures are executed when they are encountered, in a top-to-bottom scan of the program text (...) and again when they are called (a fairly frequent error).”

### ***Basic data concepts***

Computer science is about manipulating information. In fact the French word for ‘computer science’ is ‘informatique’. The students need to be aware of basic data concepts, such as data storage, data access, and input/output.

Problems with data are numerous. Problems with assignment statements include: “A:=B was interpreted as switching the values in the variables”, “the assignment statement had *no* effect”, and “A:=B was interpreted as A=B” (comparison operator) [17].

The INPUT statement is often not understood. “Some people find it very hard to grasp the idea of what goes on when a program reads in data, say from the keyboard (i.e., INPUT in Basic). In most languages the syntax disguises that a kind of assignment is involved and that the variable mentioned in the statement has its value changed (or initialized) by the statement.” Also students “do not appreciate that execution of the

READ stops further execution of the program at that point and that it cannot continue until something is typed in by the user".[5]

Fleury discusses parameter passing and explains how students construct their own rules, leading to misconceptions in the use of parameters and global variables [6]. An example of student-constructed rule is: "When the value of a VAR parameter is changed, that new value is available throughout the program." The student does not encounter any problem with this rule when all the procedures are called from the main program. The students derive it from the following correct rule: "When a procedure makes an assignment to one of its VAR parameters, that new value is also known outside the procedure that changes the value." Fleury notes that "parameter passing is not likely to be the only topic for which student-constructed rules are prevalent."

### *Language learning*

The concern of this thesis is not to teach a specific programming language. We do not know what the languages of the future will look like. Some studies, and personal experience as well, show that once one knows a language, it is much easier to learn another one [8]. Although students need to be aware that the rules of the syntax of the target language have to be respected in order for their programs to run, our goal is to minimize the energy the user will have to spend on learning the syntax of the language.

In Pascal many errors are made due to the semicolon. For example a FOR loop with a misplaced semicolon after the DO keyword can make "novices despair of ever learning to communicate with the computer" [5].

Syntax set aside, many problems arise because students attribute to the computer a certain intelligence. A study by Sleeman et al. shows that "the students attribute to the



computer the reasoning power of an average person” [17]. They found out that “several students had difficulty understanding how a READLN statement assigned values to a variable.” For example some students believed that a READLN statement used with a meaningful variable name caused the program to select a value based on the name’s meaning. If the program encounters a READLN(Odd, Event) statement and the user input is ‘2 3’, then the students would say that 2 is stored into the Even variable and 3 into the Odd variable. A program finding the biggest value of a set of numbers, but using a variable named “smallest” was declared as finding the smallest value instead [17]. Students also had problem with the WRITELN statement. They thought for example that WRITELN(‘Enter a number:’) caused a number to be read.

Benedict DuBoulay notes that “the use of English words in programming languages can mislead in further ways than just suggesting more intelligence than the system possesses.” She cites the example of “THEN” that is sometimes used “in the sense of what next: “I went to the shop and then bought a paper” ”. Another example is the “REPEAT” which “misleads beginners who expect that there must be something already in existence to be repeated.” [5]

### ***Facilitating the learning process***

Perkins et al. [15] distinguish between two kinds of students: the movers who will go on trying programs and the stoppers who will stop at the first problem they encounter. The latter “appear to view bugs more as reflecting on the value of their performance. For them, programming mistakes can be devastating because the mistakes are so obvious.” Computer work “can become a threat to self-esteem and one’s standing with peers and

teachers.” So it is crucial to make the learning process as easy as possible. An entertaining language will be more accessible for children as well as adults.

An interpreted language instead of a compiled language facilitates the understanding of the execution of a program. With a compiled language, a full program has to be written and compiled before it can be run. As DuBoulay [5] notes, “much technical detail has to be mastered, both to do with the language itself and the system for managing programs before even a simple first program can be run”. A compiled language does not allow the user to experiment easily with single commands. “In general, only complete programs can be run, making it impossible to find out easily what a single command, procedure or function will do on its own. This makes it hard for the novice to learn how to build programs out of smaller parts that he has become familiar with.” [5]

A visualization of the effects of the program will help students in understanding the program. According to Perkins et al. [15], “a vital skill for any programmer is what we call “close tracking.”(...) As the student proceeds through the code, the student must map its effects onto changes in what might be called a “status representation”, specific to the problem. For instance, in LOGO graphics problems, the turtle signals the status in large part; the student must read with precision how each piece of code alters the position and orientation of the turtle, and whether the pen is currently up or down.”

Another way of facilitating the learning process is to use an English-based language [g]. We can also help the learning process further by creating language constructs that are “closer to how people “naturally” specify problem solutions.” [19]. Soloway et al. [19] found that students overwhelmingly preferred a READ/PROCESS strategy inside a loop over a PROCESS/READ strategy as the former is closer to the way

one thinks. They also found out that “students write programs correctly more often using a construct that permits them to exit from the middle of the loop”, even though it is considered bad programming technique!

## **2. Existing approaches**

### ***The Turtle of Logo***

The Logo language was developed in the late 1970s by Seymour Papert, Daniel Bobrow and Wallace Feurzeig at MIT. Their aim was to produce a graphical programming environment that was both powerful and simple to use. The Logo language is based on the Lisp language and is quite complex. A subset of Logo, called the Turtle of Logo provides children a way of playing and at the same time learning mathematical and geometry concepts, and developing problem solving skills. The Turtle of Logo is a very popular language, and many different versions of Logo have been developed and are still being developed. It has mainly been used for the purpose of teaching programming languages to children. The version of Logo described in this document is MSWLogo, developed by George Mills [7]. A free version of MSWLogo can be downloaded from the Internet [13]. The description of the Turtle of Logo language can be found in Appendix A6.

In Logo, a turtle moves around under the control of a computer, drawing as it goes. The user can command the turtle to move forward or backward and turn a certain angle to the left or right, drawing shapes such as lines, rectangles, and triangles as it moves.

### *Structured programming constructs*

Logo includes the three basic constructs: sequence, selection, and iteration, and allows the nesting of any of those constructs. A feature of the selection construct which is not present in the other languages is the TEST instruction. It evaluates and stores the Boolean value of the tested expression. The ISTRUE and ISFALSE instructions can then be used and will be executed depending on the result of the last TEST instruction. It is to be noted that this is close to the way one may reason: 'I first check if this is true. If it is true then I will...'.

Logo provides the user with a 'For loop' instruction, and with the 'Repeat...Until' and 'Do...While' instructions with pre- or post-condition (the test on the condition is made before or after the set of instructions is executed). In the 'For loop' instruction the user can specify the increment. One can for example loop for index equal to 2 up to 10 by increment of 1.5.

Logo has a branching instruction (GOTO) that is used in conjunction with the TAG instruction. None of the other languages studied here provide a GOTO instruction for the reason that using GOTO instructions is considered a bad programming practice, leading to unstructured programs.

Logo supports arithmetic predicates (such as less than, equal to, etc.), the AND, OR and NOT logical operators, and a number of predicates for lists and arrays.

### *Modularity and top-down design*

Logo allows the user to define subprograms to create modular programs. An undefined instruction can be used to define another instruction. However, when the latter instruction is run, the former instruction must have been defined. This allows the user to

write his program just as he designs it, that is from the top down with the main goal first and then the subgoals.

### *Basic data concepts*

Logo supports the concept of variables, but it does not teach about variable type concept. Also, the syntax for creating and using variables is awkward. The MAKE instruction is used to create and initialize a variable. When referenced, the variable name must be preceded by a double-quote ("myvar for example). The value of a variable is accessed by preceding its name with a colon (:myvar for example). Subprograms can have input parameters.

Logo also supports input/output functionalities. In MSWLogo, the user can output to the drawing area or to the communication area, using LABEL or PRINT. The user can use Windows graphical interface objects such as dialog boxes, check boxes, radio buttons, lists, etc., in his program to get user input.

### *Facilitating the learning process*

The user can program the turtle by typing one command at a time and running it. The effect of the command, if any, is visible on the screen, providing immediate feedback. The drawing of geometrical patterns on the screen gives an entertaining purpose to the programming task.

### *Language Learning*

The major drawback of Logo is its awkward syntax, originating from the Lisp language. For example, it uses square brackets as block constructs. The syntax of its control structures looks like mathematical formulas, and the declaration and use of variables require the user to memorize when to use a colon or a double-quote character.

## *Josef*

In the early 1980s, Ivan Tomek created a language in which Josef, a robot, can understand and execute simple commands. Josef can travel on the screen following a map of streets. He can move forward and turn left or right. He can communicate; he can say things and listen or more precisely get messages. Marks are symbols that Josef can place on the map to leave a trace, and he can erase a previously set mark. Josef is also able to pick up or get objects, carry those objects, and leave them in a different place. Josef can see objects if they are in his present location, and can tell whether he has an object in his possession. A complete description of syntax of the language can be found in Appendix A3.

### *Structured programming constructs*

Josef supports the three basic control structures, and allows the user to nest those structures. When using a repetition instruction, the user has the choice between either a 'Repeat...Until' instruction using a post-condition, or a 'While...' instruction using a pre-condition. A conditional expression can be the Boolean constants TRUE or FALSE, predicates on the state of the robot, such as BLOCKED, CORNER, HAVE(object), etc., a Boolean variable, a Boolean function call, or an arithmetic or string relational expression. More complex conditional expressions can be built with the AND, OR, and NOT operators as well as the use of parentheses to prioritize the evaluation of the expressions.

### *Modularity and top-down design*

Joseph supports subprograms. At the difference of the other languages, it supports functions. As in Logo, the fact that one can use an undefined instruction to define another instruction allows the user to write his program at the same time he designs it from the top down.

### *Basic data concepts*

Josef's language supports variables. This concept is explained in Josef's manual as follows [21]: a variable is like a paper card with a label on it (the variable name), and something can be written on the card (the variable value). But this meaning is not carried by the syntax. Variables do not need to be declared. Josef does not introduce the user to the concept of data type. String, integer and Boolean values can be stored in a variable. Variables can be used in arithmetic expressions, and assigned values using the assignment operator (`:=`), the same way as in Pascal.

Josef introduces the concept of input/output. Text can be output to the communication area on the screen with the SAY instruction (for example `SAY('I can communicate')`). Text can be read from the keyboard and stored into a variable with the LISTEN instruction (for example `LISTEN(GREETING)`).

Subprograms in Josef can have input and/or output parameters, unlike the other languages which support no parameter or input parameter only.

### *Facilitating the learning process*

Like Logo, it is easy to 'play' with Josef, because the user can type one instruction at a time, execute it, and watch its effect on the screen.

### *Language Learning*

Although Josef's language is friendlier than the Logo language, it uses keywords such as BEGIN and END for block constructs that are used in selection and repetition control structures, and in the definition of new instructions. It uses the assignment (`:=`) and the equality comparison (`=`) operators of Pascal. Incrementing an integer is done by using the following instruction: `COUNT := COUNT + 1`, which is far from obvious and requires some explanation from a teacher.

## ***Karel***

Richard E. Pattis designed in the early 1980s another robot named Karel, very similar to Josef. Karel is a robot that can travel along a grid of streets displayed on the screen. He can move forward and turn in place. There are wall sections on the map, and Karel must navigate around them. Karel has some capabilities to see, hear, touch, and he knows about direction. Beepers are small objects that emit a quiet beeping noise. Karel can pick them up, carry them to another corner, and put them down. A complete description of syntax of the language can be found in Appendix A1.

### *Structured programming constructs*

As in Logo and Josef, Karel supports the three basic control structures, as well as the nesting of those structures. Apart from the 'For loop' instruction, the only repetition instruction allowed with Karel is a 'While loop' with pre-condition.

Only a well-defined set of test conditions is available. Each condition is available in both its positive and negative forms (for example front-is-clear and front-is-blocked), removing the need for a negation operator. There are no Boolean operators such as AND and OR to build conditions such as "front-is-clear AND facing-east".

### *Modularity and top-down design*

Karel supports the concept of subprograms. However, as in Pascal, an undefined instruction cannot be used in the definition of another instruction. Thus the user must write the subprograms from the bottom up, creating a program harder to read.

### *Basic data concepts*

Karel does not support the concept of variables, subprogram parameters nor input/output functions.



### *Facilitating the learning process*

The user must write a full program and then run it, as opposed to running an instruction at a time. After the program has been run, the environment of Karel is reset and Karel's new instructions are erased from the memory. The feedback process is not as good as in the other languages.

### *Language Learning*

Although Karel is the simplest language in terms of the number of instructions, I find Karel's language syntax awkward to learn. It makes use of many keywords to be used when defining new instructions or block constructs. Instructions have to be separated with a semi-colon as in Pascal. And a program must end with the 'turnoff' instruction to avoid an error shutoff.

### ***Karel++***

In the late 1990s Pattis added a new dimension to Karel's language by creating Karel++ in order to teach object-oriented programming. The environment of Karel++ is similar to Karel's except that one can have several robots. Karel++ teaches the user how to create a new class of robots based on an already existing class, and how to define new instructions for this class of robots. Object-oriented programming concepts taught are class hierarchy, inheritance, polymorphism and definition of an instance (a robot) of a class. A class of robots can have local robots. The base class `ur_Robot` has a predefined constructor. However the user cannot define his own constructors. Other concepts taught are Boolean functions to allow the user to define his own predicates, and also pointers (or aliases) to robots.

Karel++ syntax is more complex than Karel's. The data concept is introduced in Karel++ by the definition of an instance of a class of robots. However Karel++ does not

teach the basic concepts of variables. A complete description of syntax of the language can be found in Appendix A2.

### ***Antfarm***

Jacques LaFrance created the original Antfarm language in 1981. In Antfarm, an ant can move in a field, plant seeds, eat plants, smell and see things ahead. Each ant starts with a certain amount of energy and this energy decreases as time goes by. An ant will starve if not fed enough and explode if fed too much, and it is the responsibility of the user to keep his ants alive. A complete description of syntax of the language can be found in Appendices A4.

### ***Structured programming constructs***

As in the other languages, Antfarm supports the three basic control structures and the nesting of those structures. It is to be noted that only one instruction can be specified in the IF and IF...ELSE selection instructions. This was done to avoid the use of block constructs to simplify the language syntax. However if one wishes to specify a set of instructions, one can define a new instruction for this set of instructions and use this newly defined instruction in the selection instruction.

Apart from the 'For loop' instruction, Antfarm supports only a 'Do...Until' instruction with post-condition. Again here no block structure is needed since the keywords Do and Until delimit the set of instructions to be repeated.

Antfarm provides a set of predicates such as SEE object direction, STARVED, BEYOND row, to be used as conditional expressions. The AND, OR and NOT operators allow the user to build more complex conditions. The user must define a new test (using

the same syntax as the one to define a new instruction) to prioritize the evaluation of the conditions.

#### *Modularity and top-down design*

Antfarm supports subprograms, and just as Josef and Logo, allows the user to type his instructions from the top down. Antfarm limits, on purpose, the length of an instruction in order to force the user to create subprograms.

#### *Basic data concepts*

As in Karel, Antfarm does not support the concept of data, input/output functions, nor subprogram parameters.

#### *Facilitating the learning process*

Antfarm facilitates the learning process by allowing the user to type one command at a time, run it, and see the effect of this command on the screen.

#### *Language Learning*

One of the objectives of Antfarm has been to minimize any effort due to the language syntax. Among all the languages studied here, Antfarm is the language closest to English. It minimizes the use of keywords and any keyword used has a meaning, such as the DO and UNTIL keywords in the 'Do...Until' instruction.

#### ***Another approach: ToonTalk***

I did not find any papers indicating that other languages such as the ones studied above had been created recently. With the advance in computer technology, in particular in graphical interfaces, new ideas have emerged such as ToonTalk [9]. The idea was that since the programming language itself is a problem, we could simply get rid of it [18]. This language is like a video game where the user "writes" a program by manipulating objects on the screen with the mouse.

Ken Kahn developed ToonTalk in 1995. It is an animated programming environment designed for children. In ToonTalk, programs are not made up of text, and they are more than pictures. They are animated programs, just as video games. "Writing" a program is made by manipulating numbers, robots, birds, toolboxes, hand-held vacuum cleaners... on the screen using a mouse, like you would use a video game. Conventional programs such as factorial and parallel quick sort or games can be constructed without writing a line of text. A great deal of effort has been spent to find easily understandable concrete metaphors for abstract concepts. For example, communication is illustrated by birds, who, when given something, fly to their nest, leave the item there and return. In ToonTalk, several objects can 'run' at the same time. According to its author, concurrency is appropriate to children because they "expect that each object is running all the time".

In ToonTalk the data types available are numbers, characters, and boxes. A data can be concatenated with another one by clicking and dragging one next to the other. Boxes can contain numbers, characters and boxes: simply click the data on the screen and drag and drop it onto the box. Arithmetic can be performed with ToonTalk. For example, to add 2 numbers, click and drag one number on top of the other. A hammer will appear and hit the numbers and replace the target number with the sum. Operations on strings can also be made. Dragging a 1 on top of the letter A will produce a B. A data can be duplicated by using the magician's wand. Data can be erased and a box can be emptied by using the hand-held vacuum cleaner. One can swap the values of 2 locations by moving one from its location to a temporary place on the screen, then move the second

value into the first location, and then dragging the first value into the freed second location.

ToonTalk helps the student understand data manipulation in a concrete and visual way. It also introduces him to the concept of compound data, similar to the Pascal 'records' and the C 'structures'. Procedures are represented by robots. We first have to train a robot to do something by giving him a box (parameter). We are led into its thought bubble and we teach him what to do with the data. After we are done training the robot, we press the escape key and return to the real world. To have the robot perform what he has just learned (that is call the procedure) we simply give him a box similar to the one we gave him for the training. Comparison tests are made through the scale object. A line of robots provides something like an "if then else" capability. First In – First Out queues can be created by using birds and nests. A loop with a counter can be created by using a scale. Notebooks are stored for permanent storage of everything the user builds.

"Programming" with ToonTalk is more like using a calculator than programming with a high-level language. The concept of giving instructions to the computer gets lost with the removal of the language. The difference between ToonTalk and a high-level language is similar to the difference between a graphical user interface and a command interface: learning how to use the former does not provide knowledge on how to use the latter. Also ToonTalk does not teach concurrency, the concurrency mechanism being buried in the interpreter. So, although ToonTalk approaches the concepts of structured programming, modularity, data, and obviously solved the problem of learning the syntax of a language, I find it is too far from the high-level languages. And it is to be proven that

children and adults will be able to transfer this knowledge to a programming task with a high-level language.

### ***Examples***

Following are examples of syntax for each language. Refer to the appendixes A1 through A6 for a more complete comparison.

#### *General form of a program*

These examples have the robots or the animals learn how to turn to face the opposite direction, and then have them move one step forward and then turn around.

Note: An ant turns 45 degrees, Karel and Josef turn 90 degrees at a time. In Logo, you specify the angle of rotation.

```
Karel:  BEGINNING-OF-PROGRAM
        DEFINE-NEW-INSTRUCTION turnaround AS
        BEGIN
            turnleft;
            turnleft
        END
        BEGINNING-OF-EXECUTION
            move;
            turnaround;
            turnoff
        END-OF-EXECUTION
    END-OF-PROGRAM
```

```
Josef:  NEW TURNAROUND
        BEGIN
            LEFT LEFT
        END

        MOVE
        TURNAROUND
```

```
Antfarm: LEARN TURNAROUND DO TURN LEFT 4 TIMES

        MOVE TURNAROUND
```

Logo: TO TURNAROUND  
RT 180  
END  
  
FD 1 TURNAROUND

### *Block constructs*

All the languages, except Antfarm, use keywords to delimit a set of instructions.

Antfarm uses the LEARN instruction instead.

Karel: IF front-is-blocked  
THEN  
BEGIN  
turnleft;  
turnleft  
END

Josef: IF BLOCKED THEN  
BEGIN  
LEFT LEFT  
END

Antfarm: LEARN TURNAROUND DO TURN LEFT 4 TIMES  
IF ROW 22 TURNAROUND

Logo: IF :myvalue<3 [ RT 90 FD 10]

### *Subprograms:*

Here are examples of subprogram definitions and subprogram calls

Karel: DEFINE-NEW-INSTRUCTION stepback AS  
BEGIN  
turnleft;  
turnleft;  
move;  
turnleft;  
turnleft  
END  
  
stepback;

Josef:       NEW STEPBACK  
               BEGIN  
               LEFT LEFT MOVE LEFT LEFT  
               END

              STEPBACK

Antfarm:     LEARN STEPBACK DO TURN LEFT 4 TIMES MOVE TURN LEFT 4  
               TIMES

              STEPBACK

Logo:        TO STEPBACK  
               RT 180 FD 1 RT 180  
               END

              STEPBACK

#### *Data concepts*

Only Josef and Logo support the concept of variables.

Josef:     REPORT := 'I told them'  
            SAW\_TRUCK := TRUE  
            COUNT := 5  
            DO COUNT + 3 TIMES MOVE  
            COUNT := COUNT + 1

Logo:     MAKE "MYVAR "HELLO  
            LABEL :MYVAR  
            MAKE "MYVAR 10  
            LABEL :MYVAR\*2



### III. THE NEW ANTFARM

#### 1. Concerns addressed

The original Antfarm language addresses many of the concerns listed in the second chapter. It includes the three structured programming constructs. It allows the user (and actually forces him) to build modular programs. The syntax is very close to English, making it easier for children and adults to use. Its topic is also entertaining: caring for an ant, not letting it starve to death, nor overfeeding it, brought much enthusiasm among the children [10]. However, the original Antfarm has several drawbacks: its interface is not very user-friendly; it is not implemented to run on Windows, the system of choice for teaching novices how to use computers; and it does not teach data concepts, procedure parameters, or input/output functions. The following lists the features of the New Antfarm that address the concerns seen in chapter two.

#### *Problem-solving concepts*

The New Antfarm stimulates the thinking process. Having the ant move and plant to create patterns on the field, making sure the ant will not die of starvation or of overfeeding, checking how much energy left it has, are all ways of improving the user's problem-solving skills. There are just enough features to stimulate the imagination of children, and at the same time keeping the problems simple to solve. As a comparison, the Turtle of Logo provides only graphics to enhance those skills, which, I think, is not as stimulating. And although one can create complex graphics, there is not much variety offered to enhance problem-solving skills. Josef and Karel provide, as in Antfarm, a more complex and interesting setting. The vocabulary and syntax of the New Antfarm being

close to English allows a user to think in his native language, further helping the thinking, planning and solving process. The drawback of Josef, Karel and Logo is the syntax, which is an obstacle to learning problem-solving skills.

### ***Structured programming constructs***

Being able to run one command at a time as in the New Antfarm enables the user to apprehend the sequential aspect of a program. By comparison, in Karel, the user has to type in the whole program before running it. Seeing the immediate effect of the command on the New Antfarm screen helps the user understand that each instruction operates in the environment created by the previous instructions.

Unless there is only one command in an iteration or selection command, there is a need to delimit the set of commands included. A problem which students encounter is in determining when a loop begins or ends. The syntax of the New Antfarm seeks to prevent this problem. Only one command is allowed in the selection command, removing the need for delimiting keywords. There is no “THEN” keyword following the condition. This simplifies the language, and also removes any ambiguity about the meaning of the word “THEN”. Indeed we have seen in chapter II that some students understand “THEN” in the sense “I do this, and THEN I do that”. The form of the selection command is very simple in the New Antfarm. Examples of IF commands in the New Antfarm are:

IF SMELL FOOD EAT, and

IF SMELL FOOD EAT ELSE MOVE.

Several commands can be included in a repetition command. A repetition command starts with the keyword DO. The commands repeated are delimited at the end by the condition, again removing the need for block-construct keywords. For example,

DO MOVE TURN LEFT UNTIL FACING EAST

is not likely to create confusion about which commands are repeated. In comparison, Josef, Karel and Logo use block-construct keywords to delimit the instructions. A Karel example is:

```
WHILE not_facing_east DO
  BEGIN
    move;
    turnleft
  END
```

In the Josef example, Josef moves until he is blocked, leaving a mark at each place.

```
REPEAT
  BEGIN
    MARK('*')
    MOVE
  END
UNTIL BLOCKED
```

The Logo example prints the numbers 1, 2 and 3 as the turtle draws a line of 90 steps:

```
MAKE "I 0
DO.WHILE [MAKE "I :I+1 LABEL :I FD 30 ] [ :I<3]
```

The QUIT instruction has been included because exiting from the middle of a loop is an intuitive strategy, as we have seen in chapter II [19]. Also the QUIT instruction can be used in the following example where the ant turns in place, no more than one full turn, until it finds food:

```
DO IF SMELL FOOD QUIT TURN LEFT 8 TIMES.
```

Without the QUIT instruction the user could use the following instruction:

```
DO IF NOT SMELL FOOD TURN LEFT 8 TIMES,
```

but it is not as clear and efficient as the previous one.

### *Modularity and top-down design*

Neither Karel, Josef, nor Logo forces the user to create modular programs. In Antfarm and the New Antfarm, the length of a list of commands that can be entered at one time is intentionally limited to 80 characters. The user must define new commands to write more complex programs.

The terminology 'LEARN' used to define a new command prevents the mistake where students think the command is executed at the same time as it is defined. And the fact that no commands can precede or follow the definition of a new command also will remove the above ambiguity. An example of the definition of a new command in the New Antfarm is:

```
LEARN STEPBACK IS DO TURN LEFT 4 TIMES MOVE DO TURN LEFT 4
TIMES
```

The new command is run as follows:

```
STEPBACK
```

As a comparison, following are examples for respectively Karel, Josef and Logo.

```
DEFINE-NEW-INSTRUCTION stepback AS
  BEGIN
    turnleft;
    turnleft;
    move;
    turnleft;
    turnleft
  END
stepback;

NEW STEPBACK
  BEGIN
    LEFT LEFT MOVE LEFT LEFT
  END
STEPBACK
```

```
TO STEPBACK
  RT 180 FD 1 RT 180
END
STEPBACK
```

Although those programs appear to be very similar to the eyes of a programmer, the New Antfarm has fewer syntax-specific complexities to confuse a novice programmer than the other languages.

The student can write his modules in a top-down fashion because he can use undefined commands in the definition of other commands, defining those undefined commands later, something the Karel language does not allow. In the following example the problem of drawing a square is using a command drawing a side.

```
LEARN DRAW_SQUARE IS DO DRAW_SIDE TURN LEFT 4 TIMES
LEARN DRAW_SIDE IS DO MOVE 10 TIMES
DRAW_SQUARE
```

### ***Basic data concepts***

Although Josef and Logo's languages include variables, nothing is done in their syntax to help the user visualize concretely what a variable represents. The new Antfarm uses the metaphor of baskets in which items can be stored, just like a space in memory where numbers are stored. The data concepts introduced through this metaphor are the concepts of variable, variable name, variable type, variable declaration and initialization, and variable manipulation and deallocation. No "weird" symbol such as ':= ' (in Josef) or ' ' (in Logo) is used to perform an assignment operation. The English words PUT and PICK are used instead. The assignment operation defined in the New Antfarm is a cumulative operation: the value is added to the previous value contained in the variable. An example of assignment operation in the New Antfarm is:

```
PUT 10 SPROUT FROM THE BLUE BASKET INTO THE RED BASKET
```

Examples in Josef are:

```
REPORT := 'I told them'  
COUNT := 100
```

Examples in Logo are:

```
MAKE "MYVAR "HELLO  
MAKE "MYNUMBER 10
```

The New Antfarm seeks to promote good programming practices. Declaring variables before using them is mandatory in the New Antfarm. The user has to use the GET A BASKET command and the type of the variable has to be specified at that time, thus promoting good programming practices. None of the other languages contain those features. Also, in the New Antfarm, variables can be initialized when declared.

Additional features include variable reset and deallocation. Following is an example of variable declaration and assignment in the New Antfarm.

```
GET A BLUE SPROUT BASKET WITH 30 SPROUTS  
GET A RED SPROUT BASKET  
PUT 10 SPROUTS FROM THE BLUE BASKET INTO THE RED BASKET
```

When teaching a high-level language, I have noticed students being confused about the relationship between an input/output function call in a program, and the user interaction when the program is run. Thus I added input/output functions to Antfarm. The SAY command displays a message. The LISTEN FOR INPUT gets input from the user. The terminology "LISTEN" should help make it clear to the user that the program will stop the execution and wait for something to be entered (or listen for something to be said).

The last concept added to make this introduction to programming languages complete is the concept of subprogram parameters, presented as a substitution. Josef and

Logo include this concept too. The syntax in those languages makes use of parentheses or colon to specify the parameters. The more natural way of the New Antfarm does not use any parentheses, and also allows the user to use “template” words to make the commands more like English. In the following example of the New Antfarm, NUMBER is a parameter and BIG, MOVE and OF are template words:

```
LEARN MAKE BIG MOVE OF NUMBER IS DO MOVE NUMBER TIMES  
MAKE BIG MOVE OF 10
```

An example of Josef's use of parameters is:

```
NEW MAKE_BIG_MOVE_OF(NUMBER)  
BEGIN  
  DO NUMBER TIMES MOVE  
END  
MAKE_BIG_MOVE_OF(10)
```

An example of Logo is:

```
TO MAKE_BIG_MOVE_OF :NUMBER  
  REPEAT NUMBER [FD 1]  
END  
MAKE_BIG_MOVE_OF 10
```

### ***Language learning***

As we have seen all throughout this thesis, the syntax in Antfarm and New Antfarm is much more straightforward than in any of the other languages. The use of keywords has been reduced to a bare minimum. The predefined command names have been chosen so that they have a concrete meaning, avoiding computer jargon. Because they do not fully apprehend the task of programming, some students assume that the computer will act differently than what it has been instructed to do. The New Antfarm language is straightforward enough that it will hopefully prevent students from attributing

a certain intelligence to the computer. A language like that used in the New Antfarm may help demystify programming languages.

### *Facilitating the learning process*

To prevent the user from having feelings of failure, friendly messages are displayed when an error occurs. Those error messages also seek to help the user “fix” his problem by reminding him of the syntax of the command used. The fact that the New Antfarm is an interpreted language also helps the learning process. It allows the user to experiment with single commands. Karel is not an interpreted language, which makes it hard to distinguish what each procedure does. The New Antfarm gives immediate visual feedback, as do any of the other languages. The New Antfarm should help students specify problem solutions naturally because it is not about manipulating abstract data but about giving instructions to an ant-robot to do certain things in a simplified English language. I implemented the new Antfarm to run under Windows NT and Windows 95, thus greatly improving the interface compared to the original Antfarm, and making it easier and more enjoyable to use.

## **2. The New Antfarm environment**

The New Antfarm program draws a field on the screen with an ant on it as well as two rows of food in the upper left part of the field. The ant is initially well fed, but every action the ant performs consumes one unit of its energy. The ant can plant seeds. Those seeds grow, becoming sprouts, stalks, branching plants, flowers, and eventually grow fruits or “atples”. The ant can eat the products of its garden to gain energy back. If the ant loses all its energy or if it eats too much, it dies, becoming a skeleton. The remains of a dead ant have some food value. New ants can be obtained by the NEW command. The



ANT command will direct commands to a different ant. The ant can perform basic actions such as move, turn left or right (basic commands), repeat given commands several times (repeating commands) or perform a certain action only if some given condition is met (selection commands). One can teach new commands to the ant colony (subprograms). The ant can possess baskets (variables) of different colors to carry items such as seeds, sprouts or fruits. It can store items into these baskets, and later eat or plant items from these baskets.

### **3. Details of new contribution and examples**

#### ***The interface***

The interface consists of the field window where the ants live and the command window where the user types his instructions to the ant. The status bar at the bottom of the command window displays the active ant number, its energy level and its position on the field. Double-clicking on an ant will bring up a window with the following information: ant number, energy, position on the field, and possessed baskets and their contents. The TELL command brings up a window containing all the new commands the ant colony has learned. In order to help the user in the task of programming, messages are displayed when a problem occurs, such as executing an unknown or misspelled command or moving an ant outside of the field.

#### ***Data Concepts***

A variable is represented by a basket. A color, acting as a variable name, identifies a basket. Those baskets can contain items such as seeds, sprouts, or flowers. A basket has a type, for example SEED, SPROUT, or FLOWER. A basket of a certain type can contain only items of that same type. Before a variable can be used, it has to be

declared. The metaphor for declaring a variable is to get a basket. The syntax for the GET command is:

GET [THE | A] <color> <type> BASKET

For example GET BLUE SPROUT BASKET declares the variable BLUE of type SPROUT. A variable can be initialized at the time it is declared. The syntax of the modified GET command is:

GET [THE | A] <color> <type> BASKET WITH <number> <type>[S]

For example GET BLUE SPROUT BASKET WITH 30 SPROUTS declares the variable BLUE of type SPROUT and initializes it with the value 30.

Four operations can alter the content of a variable. The PICK command places the item (if any) under the ant's head into a basket. The syntax of the PICK command is:

PICK INTO [THE] <color> BASKET

The basket of the specified color must exist, and the item type must match the basket type. The PUT command transfers items from one basket into another basket. The syntax of the PUT command is:

PUT <number> <type>[S] FROM [THE] <color1> BASKET INTO [THE]  
<color2> BASKET

The two specified baskets must exist and have the same type. The specified number is subtracted from the first basket and added to the second basket. If the first basket has less than the specified number of items, its whole content is transferred to the second basket. For example PUT 10 SEEDS FROM THE BLUE BASKET INTO THE RED BASKET. The EAT command removes one item from a basket and uses it to feed the ant. The syntax of the EAT command is:

EAT FROM [THE] <color> BASKET

For example, EAT FROM THE BLUE BASKET. The PLANT command is similar to the EAT command except that the item is planted under the ant's head instead of fed to the ant. The syntax of the PLANT command is:

PLANT FROM [THE] <color> BASKET

A typical assignment operation replaces the old value of a variable with a new one. Those four operations are special assignment operations. The PICK command adds the constant 1 to the variable, the EAT and PLANT commands subtract the constant 1 from the variable. The PUT command is similar to an assignment of one variable to another variable with one exception: the specified number is subtracted from the content of the first variable, and it is added to the content of the second variable.

A variable can be reset with the EMPTY command. The syntax of the EMPTY command is

EMPTY [THE] <color> BASKET

This is equivalent to assigning a null value to a variable. When a variable is not needed anymore, it can be deallocated with the THROW AWAY command. The syntax of this command is:

THROW AWAY [THE] <color> BASKET

### ***Input/Output Functions***

I added input/output functions to the original Antfarm language to bring the awareness that a program can, and usually does, interact with its user. The SAY command outputs the data on the screen. The LISTEN command will store the data entered by the user into a special variable called the INPUT basket. The INPUT basket is

not a 'typed' variable. It can contain either strings or integers. The previous value of the INPUT basket is replaced by the new value. The assignment operation on the INPUT basket works like the standard assignment operation of most programming languages.

The syntax of the SAY and LISTEN commands are:

SAY <string>

LISTEN FOR INPUT

where <string> is any sequence of characters enclosed in double-quotes. Here are some examples using the SAY and LISTEN commands.

SAY "Please, give me a number of seeds."

LISTEN FOR INPUT

PUT INPUT FRUIT FROM RED BASKET INTO BLUE BASKET

SAY "Which basket shall I use? (give me a color)"

LISTEN FOR INPUT

DO PICK INTO INPUT BASKET MOVE 10 TIMES

### ***Subprograms and subprogram parameters***

The syntax for defining subprograms differs from the original Antfarm. First I removed the END keyword, present in the original Antfarm, that was necessary at the end of the definition of a new instruction when it was followed by another instruction. In the New Antfarm, no instruction can follow the definition of a new instruction. Then I introduced the keyword IS to separate the new instruction definition heading from the body to determine the list of parameters, if any.

The New Antfarm also introduces subprogram parameters. Josef and Logo make use of special characters to specify the parameters in the subprogram heading, and the body is enclosed in their block construct. The main issue being to keep the syntax close to English, no such keywords are used to isolate parameters. The first word after the LEARN keyword is the subprogram name. If a word is used in the subprogram heading

(after the subprogram name and before the IS keyword), and in the body of the subprogram (after the IS keyword), and if this word is not a keyword, then it is a parameter. When the subprogram is called, the value entered in place of the parameter is substituted in the body of the subprogram. If this word is not a parameter, then it is a template word. The purpose of template words is to let the user create meaningful command names. The syntax for defining a new command is:

```
LEARN <name> { <parameter or template word> } IS <list of commands>
```

The following is an example of a subprogram definition and call.

```
LEARN STEPBACK IS DO TURN LEFT 4 TIMES MOVE DO TURN LEFT 4  
TIMES
```

```
STEPBACK
```

The next example shows the definition of a subprogram with parameters and how it is called.

```
LEARN MAKE NUMBER STEPS IS DO MOVE NUMBER TIMES
```

```
MAKE 10 STEPS
```

Note that NUMBER is the parameter because it is not a keyword and it appears in the body of the definition. STEPS is a template word because it does not appear in the body of the definition.

#### IV. CONCLUSION

The goal of this thesis was to create a better tool to teach good programming practices to adults and children. The concepts considered were structured programming and basic data concepts. This tool had also to facilitate the learning process. Ultimately, this demanded a comparative analysis of the existing programming literature and tools for education. The original Antfarm language addresses part of those concerns, and its syntax is close to English. Thus I chose to improve and extend the original Antfarm. It now runs under Windows 95 and Windows NT with an interface more user-friendly than the original one. I extended the language to include the missing concepts of data, input/output, and subprogram parameters.

There are several advantages of the New Antfarm over the other languages. It helps users enhance their problem-solving skills by reducing the effort spent in translating the solution into the New Antfarm language. The structured programming constructs are introduced in a natural language with as few keywords as possible. It forces the user to create modular programs as he solves more complicated problems. It introduces the user to the concepts of data in an intuitive way. Josef and Logo include variables, but the syntax is not self-explanatory about the nature of variables. Neither the old Antfarm nor Karel support data concepts. Variable declaration, initialization and manipulation concepts are introduced in the New Antfarm. The New Antfarm includes input/output functions, which are not present in the old Antfarm or Karel. It also includes subprogram parameters, which neither the old Antfarm nor Josef do. The last but not least advantage is that the syntax of Antfarm is very close to English in order to facilitate the

learning process. In contrast, Josef and Karel are based on Pascal, Logo on Lisp, and their syntaxes reflect their less friendly origins.

There are several ways the New Antfarm can be improved or extended. The usefulness of the following improvements and extensions will depend on the type of population using the language, adults, high-school students, children, future programmers or people using the program just for the fun of it. First of all, the user interface can be made more attractive, for example by drawing a more real-looking ant. Sound could be added to the program. For example, the ant could make a sound when it eats, tries to get out of the field, or is about to die. Debugging tools could be provided. Or simply the program could highlight in the command line the words where an error occurred. The concept of data is introduced, but not the concept of data structure. For example, arrays, records and enumerations could be introduced. A metaphor for arrays could be baskets with several compartments. Assignment operations are demonstrated, but we have seen that those assignment commands differ from the usual assignment operation. They add or subtract the assigned value from the content of the variable, instead of replacing the content of the variable with the new value. A metaphor would have to be found to introduce the concept of real assignment. Test conditions for variables could be introduced. For example, arithmetic comparisons based on the content of a basket and comparisons based on the type of a basket could be added to the language. Comments are not essential in the New Antfarm since the language is already self-explanatory. However the user could make use of comments at the beginning of his programs to explain what they are doing. A tutorial and a list of exercises might help certain students, most likely adults, to start working with the New Antfarm.

Anyone who wishes to add features to the New Antfarm should keep in mind that a tool that seeks to introduce a subject should not be complex. Thus, users should be able to use only the basic features of the New Antfarm. More advanced users can then make use of more features as they get more familiar with the product and with the task of programming. I believe that after learning the New Antfarm, the user will be more confident and ready to learn a high-level programming language.



## V. BIBLIOGRAPHY

- [1] Appleby D., VandeKopple J., *Programming Languages Paradigm and Practice*, McGraw-Hills, 1997, pp.16-17
- [2] Bergin J., Stehlik M., Roberts J., Pattis R., *Karel++ - A gentle introduction to the art of object-oriented programming*, John Wiley & Son, 1997
- [3] Bohm C., Jacopini G., "Flow diagrams, Turing machines and languages with only two formation rules", *Communications of the ACM*, vol. 9, n.5 (May 1966), pp. 366-371.
- [4] Bonar J., Cunningham R., ridge: Tutoring the Programming Process, in *Intelligent Tutoring Systems, Lessons Learned*, Lawrence Erlbaum Associates Publishers, 1988, p. 410
- [5] Du Boulay B., Some Difficulties of Learning to Program, *Journal of Educational Computing Research*, Vol 2(1), pp. 57-73, 1986
- [6] Fleury A., Parameter Passing: The Rules the Students Construct, *SIGCSE '91* pp 283-286, 1991
- [7] Fuller J., *An Introduction to MSWLogo*,  
<http://www.southwest.com.au/~jfuller/logotut/menu.htm>, 1999
- [8] Goldenson D., Why Teach Computer Programming? Some Evidence about Generalization and Transfer, Call of the North, NECC 1996, *Proceedings of the Annual National Educational Computing Conference*, pp.144-158
- [9] Kahn K., ToonTalk – An animated Programming Environment for Children, *Journal of Visual Languages and Computing*, Vol 7 Nb 2, 1996, pp.197-217. Also under <http://www.toontalk.com/english/papers.htm>
- [10] LaFrance J., "Reorienting students to structured programming with Antfarm", *Proceedings of the 1982 Western Educational Computing Conference*, San Diego, California, pp. 35-42
- [11] LaFrance J., "Crisis in programming or history repeats itself", *Proceedings of the 1983 National Educational Computing Conference*, Baltimore, Maryland, pp. 126-130
- [12] LaFrance J., *Antfarm, a language for learning programming concepts*, Wims Computer Consulting, 1981
- [13] Mills G., *MSWLogo Kits*, <http://www.softronix.com/logo.html>.

- [14] Pattis R., *Karel the Robot*, John Wiley & Son, 1981
- [15] Perkins D., Hancock ., Hobbs R., Martin F., Simmons R., Conditions of Learning in Novice Programmers, *Journal of Educational Computing Research*, Vol 2(1), pp. 37-55, 1986
- [16] Schollmeyer M., Computer Programming in High School vs College, *SIGCSE '96* 2/96 pp 378-382, 1996
- [17] Sleeman D., Putnam R., Baxter J., Kuspa L., Pascal and High School Students: A Study of Errors, *Journal of Educational Computing Research*, Vol 2(1), pp. 5-23, 1986
- [18] Smith D., Cypher A. & Spohrer J., *KIDSIM: Programming Agents Without a Programming Language*, In *Communications of the ACM*, 37(7), July 1994, pp. 54 - 67.
- [19] Soloway E., Bonar J., Ehrlich K., Cognitive Strategies and Looping Constructs: An Empirical Study, *Communications of the ACM*, Vol 26, Nb 11, pp. 853-860, 1983
- [20] Spohrer J., Soloway E., Novice Mistakes: Are the Folk Wisdoms Correct?, *Communications of the ACM*, Vol 29, Number 7 pp. 624-632, 1986
- [21] Tomek I., *The first book of Josef*, Prentice-Hall, 1983

## VI. APPENDICES

Appendices A1 to A6 describe the following languages: Karel, Karel++, Josef, Antfarm, New Antfarm and Logo. The Extended Backus Naur Form, or EBNF, is used to describe the syntax of the languages. Following is the list of EBNF symbols used and their meaning [1].

Symbol	Meaning
::=	is defined to be
	alternatively, or
<something>	<something> is to be replaced by its definition
<b>something</b>	a word in boldface is called a terminal, indicating an indivisible language element allowing no further replacements
[something]	0 or 1 occurrence of something, i.e. optional
{something}	0 or more occurrences of something
(this   that)	grouping; either of this or that

The languages are described here in the goal of comparing them. The descriptions might not be complete. For a full description, refer to the following references: [14] for Karel, [2] for Karel++, [21] for Josef, [12] for Antfarm, and [7] for Logo.

## A1. Karel syntax description

```
<program> ::=  BEGINNING-OF-PROGRAM
                { <new instruction definition> ; }
                BEGINNING-OF-EXECUTION
                { <instruction>; }
                turnoff
                END-OF-EXECUTION
                END-OF-PROGRAM
```

The semi-colon is used as in the PASCAL language, i.e. it is used to separate instructions. The turnoff instruction is required at the end of the program execution to avoid an error shutoff.

```
<instruction> ::= <basic instruction> |
                  <block structure instruction> |
                  <selection instruction> |
                  <repetition instruction> |
                  <subprogram call>
```

```
<basic instruction> ::=
    move |           Karel moves one block forward
    turnleft |        Karel pivots 90 degrees to the left
    pickbeeper |      Karel puts a beeper in his beeper bag
    putbeeper |       Karel places a beeper on the corner
    turnoff           Karel turns himself off
```

```
<block structure instruction> ::= BEGIN
                                   {<instruction> ;}
                                   <instruction>
                                   END
```

```
<selection instruction> ::= IF <condition>
                             THEN <instruction>
                             [ ELSE <instruction> ]
```

```
<repetition instruction> ::= <iterate instruction> | <while instruction>
<iterate instruction> ::= ITERATE <positive number> TIMES
<while instruction> ::= WHILE <condition> DO
                        <instruction>
```

```
<condition> ::= front-is-clear | front-is-blocked |
                 left-is-clear | left-is-blocked |
                 right-is-clear | right-is-blocked |
                 facing-north | not-facing-north |
                 facing-south | not-facing-south |
                 facing-east | not-facing-east |
                 facing-west | not-facing-west |
                 next-to-a-beeper | not-next-to-a-beeper |
                 any-beepers-in-beeper-bag | no-beepers-in-beeper-bag
```



## A2. Karel++ syntax description

This description is taken out of the reference [2].

```
<program> ::= { <newClassDefinition> }  
            { <newInstructionDefinition> }  
            task  
            { { <robotInitialization> }  
              { <instruction> }  
            }
```

```
<newClassDefinition> ::= class <newClassName> : <oldClassName>  
                        { { <robotInitialization> } // Local robots  
                          // New instructions  
                          { <returns> <newInstructionName> ( ); }  
                        }
```

```
<newInstructionDefinition> ::= <returns> <className> :: <instructionName> ( )  
                               { { <instruction> }  
                               }
```

If <returns> is Boolean, then one or more of the <instruction> should be **return** <BooleanValue>

```
<robotInitialization> ::= <className> <robotName> (street, avenue, facing, beepers);
```

```
<instruction> ::= <instructionName> ( );  
                 <robotName>.<instructionName> ( );  
                 <aliasName> -> <instructionName> ( );  
                 <conditionalInstruction>  
                 <repetitionInstruction>
```

<primitiveInstruction> of the ur\_Robot and Robot classes:

```
class ur_Robot  
{  
    void move();  
    void turnOff();  
    void turnLeft();  
    void pickBeeper();  
    void putBeeper();  
};  
class Robot: ur_Robot  
{  
    Boolean frontIsClear();  
    Boolean nextToABeeper();  
    Boolean nextToARobot();  
    Boolean facingNorth();  
    Boolean facingSouth();  
    Boolean facingEast();  
    Boolean facingWest();  
    Boolean anyBeepersInBeeperBag();  
};
```

<conditionalInstruction> ::=

```
    if ( <test> )
    {      { <instruction> }
    }
    [ else
    {      { <instruction> }
    }]
```

<repetitionInstruction> ::=

```
    loop ( <positiveNumber> )
    {      <instruction> }
    }

    while ( <test> )
    {      <instruction> }
    }
```

<test> ::= <BoolFunctionName> ( ) |

```
    <robotName>. <BoolFunctionName> ( ) |
    <aliasName> -> <BoolFunctionName> ( ) |
    ! <BoolFunctionName> ( ) |
    ! <robotName>. <BoolFunctionName> ( ) |
    ! <aliasName> -> <BoolFunctionName> ( )
```

<...Name> ::= Any new word in letters, numbers and "-" that begins with a letter.

<BooleanValue> ::= **true** | **false**

<positiveNumber> ::= Any positive integer

<returns> ::= **void** | **Boolean**

### A3. Josef syntax description

Josef is an interpreted language, that is, each instruction is read, syntactically checked, and run one by one. The environment is not reset until one exits Josef's program completely, and starts it again. The user programs Josef by typing and running a list of instructions or by defining a new instruction. The instructions are separated by a blank space or a carriage return.

**<program> ::= { <instruction> } | <new instruction definition>**

**<instruction> ::=**  
    <basic instruction> |  
    <block structure instruction> |  
    <selection instruction> |  
    <repetition instruction> |  
    <subprogram call>

**<basic instruction> ::= <primitive instruction> | <non-primitive instruction>**

A primitive instruction takes one unit of time to execute, while a non-primitive instruction takes zero time to execute.

**<primitive instruction> ::=**

**MOVE** |

**LEFT** |

**RIGHT** |

**ERASE** |

**PAUSE** [( <arithmetic expression> )] |

**GET** ( <string expression> ) |

**LEAVE** ( <string expression> ) |

**MARK** ( <string expression> ) |

**CONSUME** ( <string expression> )

Josef moves to the next location in the direction he is facing.

Josef turns left by 90 degrees.

Josef turns right by 90 degrees.

Replaces the mark placed by Josef by the original map symbol.

With a parameter, Josef pauses for the specified number of time units. Without parameter, Josef pauses until the user presses the Return key.

Josef gets the object specified.

Josef leaves the object specified.

Josef marks its present location with the first character of the specified string.

One object of the type specified disappears.

**<non-primitive instruction> ::=**

**<identifier> ::= <expression> |**

**MAP** ( <string expression> ) |

**ENABLE** ( <string expression> ) |

**DISABLE** ( <string expression> ) |

**LOCATION** ( <identifier> , <identifier> ) |

Assigns <expression> to the variable <identifier>.

Changes the map where Josef is.

Enables the specified interrupt procedure.

Disables the specified interrupt procedure.

Returns the street name and number of Josef's current location in the two specified parameters.



<b>XY</b> ( <identifier>, <identifier> )	Returns the column and row numbers of Josef's current location in the two specified parameters.
<b>LISTEN</b> ( <identifier> { , <identifier> } )	Josef reads a list of inputs (separated by semicolons or carriage returns) from the keyboard and save it in the specified variable(s).
<b>SAY</b> ( <expression> { , <expression> } )	Josef displays the specified text parameters in the communication area.
<b>SPEED</b> ( <arithmetic expression> )	Sets the speed at which Josef execute MOVE instructions.
<b>TIME</b> ( <arithmetic expression> )	Starts Josef's clock at the specified value.
<b>IMAGE</b> ( <string expression>, <string expression>, <string expression>, <string expression> )	Changes the representation of Josef on the map. Can be used to make Josef transparent, and thus see what's in his location.
<b>EXIT</b> [ ( <string expression> ) ]	Without parameter, Josef quits the execution of the present instruction. Meaningful only inside a repetition or block structure instruction. With a parameter Josef exits the specified subprogram.
<b>QUIT</b>	Quits execution of the current program and returns to direct command mode. Useful in interrupt programming.

<block structure instruction> ::= **BEGIN**  
   { <instruction> }  
**END**

<selection instruction> ::= **IF** <condition> **THEN** <instruction>  
   [ **ELSE** <instruction> ]

<repetition instruction> ::= **REPEAT** <instruction> **UNTIL** <condition> |  
**WHILE** <condition> **DO** <instruction> |  
**DO** <arithmetic expression> **TIMES** <instruction>

<condition> ::=  
**TRUE** |  
**FALSE** |  
<identifier> |  
<function call> |  
**BLOCKED** | Returns TRUE if Josef cannot move.  
**CORNER** | Returns TRUE if Josef's current location is a corner of 2 or more streets.  
**ENTRY** ( <string expression> ) | Returns TRUE if the specified map can be entered from Josef's current location.  
**SEE** ( <string expression> ) | Returns TRUE if the specified object is in Josef's current location.  
**HAVE** ( <string expression> ) | Returns TRUE if Josef has the specified object.  
( <condition> ) |  
**NOT** <condition> |  
<condition> (**AND** | **OR**) <condition> |  
<arithmetic expression> (= | <> | <= | >= | > | <) <arithmetic expression> |  
<string expression> (= | <>) <string expression>

<new instruction definition> ::=  
**NEW** <identifier> [ ( <identifier> { , <identifier> } ) ]  
[ **USE INTERRUPT** <subprogram call> [ **WHEN** <string expression> ] ]  
<block structure instruction>

If the block structure instruction in the new instruction definition contains a  
 <return instruction> then a function is defined, else a subprogram is defined.

<subprogram call> ::= <identifier> [ ( <expression> {, <expression> } ) ]  
 <function call> ::= <identifier> [ ( <expression> {, <expression> } ) ]  
 where <identifier> is the name of a new instruction.

<return instruction> ::= **RETURN** <expression>

<expression> ::= <string expression> | <arithmetic expression> | <condition>

<string expression> ::=

<string constant> |

<identifier> |

<function call> |

**DIRECTION** | Returns one of the letters U, D, R, L depending on Josef's direction.

**ENTRY** | Returns the list of maps that can be entered from this location. To be used repeatedly to obtain each map one by one. The last name returned is NONE.

**HAVE** | Returns the list of objects in Josef's possession. Works like ENTRY.

**KEY** | Returns the character of the currently depressed key, null character if no key is depressed.

**MARK** | Returns the character by which Josef marked the current location, null character if the mark was not created by Josef.

**SEE** | Returns the list of objects in Josef's current location. Works like ENTRY.

<arithmetic expression> ::=

<integer> |

<identifier> |

<function call> |

**RANDOM** ( <arithmetic expression> ) | Returns a random number between 0 and the specified value.

**SPEED** | Returns the current speed of Josef.

**TIME** | Returns the time elapsed since time zero.

**VOC** ( <identifier> ) | Returns the decimal equivalent of the VOC code of the specified map.

- <arithmetic expression> |

{ <arithmetic expression> } |

<arithmetic expression> ( + | - | \* | / ) <arithmetic expression>

<identifier> ::= string that starts with a letter, can include letters, digits and the underscore and has up to 15 characters.

<string constant> ::= string delimited by single quotes, such as 'Hello'.

<integer> ::= a positive integer.

#### A4. Antfarm syntax description

The user programs an ant by typing a line of up to 80 characters, containing either a list of instructions, or the definition of a new instruction. The instructions are separated by one or several blank spaces or tabs.

`<program> ::= { <instruction> } | <new instruction definition>`

`<instruction> ::=`  
    `<basic instruction> |`  
    `<simulator instruction> |`  
    `<selection instruction> |`  
    `<repetition instruction> |`  
    `<subprogram call>`

A basic instruction, executed by the ant, takes one unit of time to execute. Any other instruction takes zero time to execute.

`<basic instruction> ::=`

<b>MOVE</b>	Take one step (column, row or both) forward
<b>BACKUP</b>	Take one step (column, row or both) backward
<b>TURN LEFT</b>	Turn 45 degrees (1/8 <sup>th</sup> turn) to the left
<b>TURN RIGHT</b>	Turn 45 degrees (1/8 <sup>th</sup> turn) to the right
<b>EAT</b>	Eat whatever is under its head,
<b>PLANT</b>	Plant under its head a seed
<b>REST</b>	Do nothing
<b>WAIT</b>	Do nothing
<b>NOTHING</b>	Do nothing

`<simulator instruction> ::=`

<b>NAME</b> <string>	Change the ant colony name to a new name
<b>ANT</b> <nb>	Change the ant responding to the commands to another
<b>NEW</b> <nb>	Start a new ant for that number
<b>HELP</b>	List all the built-in commands
<b>QUIT</b>	End a loop or the program
<b>DONE</b>	End a loop
<b>STOP</b>	End a loop
<b>OFF</b>	Turn off the consuming of energy
<b>ON</b>	Turn on the consuming of energy
<b>TELL</b>	Print all the new commands the ant colony has learned
<b>FORGET</b> (<identifier>  <b>ALL</b> )	Forget the specified command or all the commands the colony has learned
<b>RENAME</b> <identifier> <b>TO</b> <identifier>	Renames a learned command
<b>CHANGE</b> <identifier>	Changes the content of the specified learned command

`<selection instruction> ::=`

**IF** <condition> <instruction> [**ELSE** | **OTHERWISE** <instruction>]

<iteration instruction> ::=  
**DO** <instruction> <nb> **TIMES** |  
**DO** <instruction> (**TO** | **UNTIL**) <condition>

<condition> ::=  
**ROW** <nb> |  
 {**COLUMN** | **COL**} <nb> |  
 (**BEYOND**|**PAST**) (**ROW**|**COLUMN**|**COL**) <nb>|  
**FACING** <facing direction> |  
**SEE** <object> <direction> |  
**SMELL** <object> |  
**STARVED** | **HUNGRY** | **FED** | **FULL** | **STUFFED** |  
**NOT** <condition> |  
 <condition> **AND** <condition> |  
 <condition> **OR** <condition> |  
 <identifier>

where <identifier> must be the name of a new condition defined with the **LEARN** instruction.

<object> ::= **DIRT** | **JUNK** | **PLANT** | **FOOD** | **SPROUT** | **FLOWER** | **MARKER**  
 <direction> ::= **AHEAD** | **LEFT** | **RIGHT**  
 <facing direction> ::= **N** | **NE** | **E** | **SE** | **S** | **SW** | **W** | **NW**

<new instruction definition> ::= **LEARN** <identifier> {<instruction>}  
 <subprogram call> ::= <identifier>

<identifier> ::= any string made up of characters, excluding blanks, tabs and carriage returns.  
 Letters, digits, hyphen, underline, dash... are allowed.

<nb> ::= any positive integer.

<string> ::= string with no tabs or blank spaces

<string expression> ::= string in between double quotes such as "Hello, world!"

## A5. New Antfarm syntax description

This is what has been added to the original Antfarm language in this thesis:

<basic instruction> ::=

All the basic instructions of the original Antfarm |

**EAT [FROM [THE] <color> BASKET] |** Eat whatever is under its head, or one item from the specified basket

**PLANT [FROM [THE] <color> BASKET] |** Plant under its head a seed, or one item from the specified basket

**GET [THE|A] <color> <basket type> BASKET**

**[WITH <nb> SEEDS|SEED|SPROUTS|SPROUT|FRUITS|FRUIT] |**

Receive a basket of the specified type, empty or with a certain number of items in it

**PICK INTO [THE] <color> BASKET |**

Pick what is under its head and put it into the specified basket

**PUT [<nb> FROM] [THE] <color> BASKET INTO [THE] <color> BASKET |**

Put all or a certain number of items from the first basket into the second basket

**EMPTY [THE] <color> BASKET |**

Empty the specified basket

**THROW AWAY [THE] <color> BASKET |**

Get rid of the specified basket

**SAY <string expression> |**

Print the message on the screen

**LISTEN FOR INPUT |**

Get a message from the keyboard and puts into the INPUT basket

**WHAT IS IN { [THE] <color> BASKET | ALL BASKETS}**

Print on the screen the list of the contents of the specified basket or of all the baskets

<basket type> ::= **SEED | SPROUT | FLOWER | FRUIT | JUNK**

<color> ::= **BLUE | GREEN | YELLOW | RED**

<new instruction definition> ::=

**LEARN <identifier> [ { <identifier> } ] IS {<instruction>}**

In the Extended Antfarm language, the 'IS' keyword is required when defining a new instruction.

If an identifier in the list of identifiers is not in any instruction listed after IS, then it is a template word. This is to allow the user to write in a more natural language.

<subprogram call> ::= <identifier> [ { <arguments> } ]

<argument> ::= <string> | <nb>

## A6. Logo syntax description

To program the turtle, the user types instructions in the Input box, or defines a new instruction.

<program> ::= { <instruction> } | <new instruction definition>

<instruction> ::=     <basic instruction> |  
                      <selection instruction> |  
                      <repetition instruction> |  
                      <subprogram call>

<basic instruction> ::=

<b>FORWARD</b> <nb>	Turtle moves forward the specified number of steps (or <b>FD</b> )
<b>BACK</b> <nb>	Turtle moves backward the specified number of steps (or <b>BK</b> )
<b>RIGHT</b> <nb>	Turtle turns clockwise the number of degrees specified (or <b>RT</b> )
<b>LEFT</b> <nb>	Turtle turn counterclockwise the number of degrees specified (or <b>LT</b> )
<b>PENUP</b>	Turtle's pen is up (doesn't draw as it moves) (or <b>PU</b> )
<b>PENDOWN</b>	Turtle's pen is in the down position (or <b>PD</b> )
<b>PENERASE</b>	Turtle erases as it moves (or <b>PE</b> )
<b>HIDETURTLE</b>	Removes the turtle 'triangle' from the screen (or <b>HT</b> )
<b>SHOWTURTLE</b>	Makes the turtle visible again (or <b>ST</b> )
<b>CLEARSCREEN</b>	Erases the screen and returns the turtle to its home position (or <b>CS</b> )
<b>HOME</b>	Returns the turtle to home position in the center of the drawing screen
<b>SETPENCOLOR</b> [<nb> <nb> <nb>]	Determines pen color, each number for red, green and blue being between 0 and 255
<b>SETPENSIZE</b> [ width height]	Sets width and height of the drawing pen
<b>BYE</b>	Exits MSWLogo
<b>LABEL</b> <value>	Takes a word (preceded by double-quote ("Hello for example), or a list ([Hello to the world] for example), and prints the input on the graphics window, starting at the turtle's position, and with the same the same orientation as the turtle's
<b>MAKE</b> "<string> <value>	Assigns the specified value to the variable varname, which must be a word. If a variable with the same name exists, the value of that variable is changed. If not, a new global variable is created. Value can be a string, an integer, a real, or a list
<b>NAME</b> <value> "<string>	Same as MAKE but with the inputs in reverse order
<b>LOCAL</b> "<string>	Accepts as input one or more words, or a list of words. A variable is created for each of the inputs. Those variables are local to the command in which they are created.
<b>TAG</b> tag	Defines a tag in a procedure. Use with GOTO
<b>GOTO</b> tag	Jumps to the corresponding tag inside the current procedure

<repetition instruction> ::= <repeat instruction> | <for instruction> | <condition instruction>

<repeat instruction> ::= **REPEAT** <nb> [ { <instruction> } ]

Will repeat the instruction list the specified number of times

```

<for instruction> ::= FOR [ <nb> <nb> <nb> ] [ { <instruction> } ]
                                Will repeat the instruction list for the index (first number) going from
                                second number to third number by increment of fourth number

<condition instruction> ::=
    DO.WHILE [ { <instruction> } ] <condition> |           While loop with post-condition
    WHILE <condition> [ { <instruction> } ] |             While loop with pre-condition
    DO.UNTIL [ { <instruction> } ] <condition> |          Until loop with post-condition
    UNTIL <condition> [ { <instruction> } ]                Until loop with pre-condition

<selection instruction> ::= <if instruction> | <ifelse instruction> |
                                <test instruction> | <iftrue instruction> | <iffalse instruction>

<if instruction> ::= IF <condition> [ { <instruction> } ]
                                It the condition is true, execute the instruction list
<ifelse instruction> ::= IFELSE <condition> [ { <instruction> } ] [ { <instruction> } ]
                                If the condition is true, execute the first instruction list, else execute the
                                second one
<test instruction> ::= TEST <condition>
                                Evaluate the condition to true or false and memorize the result for the
                                next IFTRUE or IFFALSE commands
<iftrue instruction> ::= IFTRUE [ { <instruction> } ]
                                Execute the instruction list if the last TEST command evaluated to true
<iffalse instruction> ::= IFFALSE [ { <instruction> } ]
                                Execute the instruction list if the last TEST command evaluated to false

<variable value> ::=
    THING "<string>" |           Returns the value stored in the variable varname
    :<string>                    Same as THING

<value> ::= "<string>" | <string list> | <nb> | <variable value>
<string> ::= string made up of characters, digits, dash, underline,... with no blank spaces
<string list> ::= [ { <string> } ]
<nb> ::= integer | real | - <nb> | <nb> ( + | - | * | / ) <nb>

<condition> ::= <comparison condition> | <logical condition>
<comparison condition> ::= <nb> ( < | > | = ) <nb> | <value> = <value>
<logical condition> ::= ( AND | OR ) <condition> <condition> | NOT <condition>

<new instruction definition> ::=
    TO procname [ { :<string> } ]
        { <instruction> }
    END

<subprogram call> ::= <string> [ { :<value> } ]

```

## **A7. New Antfarm user manual**

### **1. ANTIFARM PROGRAM PHILOSOPHY**

The Antfarm program is a tool for teaching fundamental programming concepts to children. It is designed to lead the learner naturally into the commonly accepted best programming style. The author hopes that by making this the learner's first experience with programming he or she will carry these principles over into other languages such as BASIC which do not naturally encourage good programming style, or PASCAL or C. The principles which Antfarm promotes are structured programming and top-down design with small subprogram units. Antfarm also introduces the concepts of variables, input/output procedures and subprogram parameters. These principles are currently used by the larger professional program development organizations and are known to make programming more straightforward and less time-consuming. Following these programming principles yields programs that are easier to understand, document, change, and maintain. These principles are based on the following ideas:

- that programming consists of many small steps put together to make a working program that accomplishes a larger task,
- that a program is sequential, with the steps proceeding orderly from one to the next,
- that programming consists of sequencing steps that are themselves sequences of smaller steps, that in turn are sequences of even smaller steps, and so on until the last steps are the basic commands understood by the machine,
- that complex programs can be made up by this method (called "top-down design") in such a way that no part is ever logically complex itself,



- that in addition to the basic steps there are the control steps of selection, (choosing to do something or not, or choosing between alternatives), and iteration (repeating a set of steps until a certain condition is met).

Learning these principles from the beginning of a study of programming concepts will enable the students to learn good habits as their skills improve. These good habits will enable them to progress in programming more easily and become better programmers than students not well grounded in these basic concepts.

## **2. PROGRAM OPERATION**

This program is designed to run under Windows NT and Windows 95. Double-click the Antfarm icon to start the program. A message box will ask for the name of the Ant Colony. This name will be used when the user saves or restores the new commands that the ant has learned as well as the field the colony lives on. It can be changed with the RENAME command. When the Colony Name is typed, and the OK button is clicked, the ant farm's field is displayed with an ant on it (Ant number 1). The field also contains in the upper left corner two rows of 10 mature plants.

To instruct the ant to execute commands, the user types the commands in the command line window, and then clicks the RUN button or presses the ENTER key. The size of a command is limited to 80 characters. Error and warning messages will be displayed in the message window. The communication window is used for passing messages from and to the ant colony.

### 3. OVERVIEW OF THE ANTFARM PROGRAM

The program draws a “field” on the screen with an ant on it as well as two rows of food in the upper left part of the field. The ant is initially well fed, but every action the ant performs consumes one unit of its energy. The ant can plant seeds. Those seeds grow, becoming sprouts, stalks, branching plants, flower, and eventually become mature plants, growing a fruit or “atple”. The ant can then eat the products of its garden to gain energy back. If the ant loses all its energy or if it eats too much, it dies, becoming a skeleton. The remains of an old ant have some food value. The ant can perform basic actions such as move, turn left or right (basic commands), repeat given commands several times (repeating commands) or perform a certain action only if some given condition is met (selection commands). The ant can possess baskets of different colors to carry items such as seeds, sprouts, fruits, or junk. The ant colony has the ability to learn new commands. New ants can be obtained by the NEW command. The ANT command will direct commands to a different ant.

#### 4. BASIC COMMANDS

The basic actions the ant can perform are moving, turning, eating and planting new seeds. Each of these actions consumes one unit of the ant's energy and takes one unit of time to execute. The command words to be typed to instruct the ant to perform these actions are the following:

MOVE	The ant takes one step (column, row or both) forward.
BACKUP	The ant takes one step (column, row or both) backward.
TURN LEFT	The ant turns 45 degrees ( $1/8^{\text{th}}$ turn) to the left.
TURN RIGHT	The ant turns 45 degrees ( $1/8^{\text{th}}$ turn) to the right.
EAT	The ant eats whatever is under its head. The energy of the ant will increase by the value of the food eaten, after one unit of energy is consumed due to the action of eating.
PLANT	The ant plants a seed under its head.
REST	Do nothing.
WAIT	Do nothing.
NOTHING	Do nothing.

#### 5. TIME UNITS

A time unit is defined as the time it takes for the ant to interpret and perform a single basic command. Simulator commands do not take any time, as well as the selection and iteration commands, although any basic command performed inside a selection or iteration command does. For example, if the ant repeats 4 times the command MOVE, 4 units of time will be spent (and 4 units of its energy will be consumed).

## 6. ENERGY LEVELS

The ant is initially fully fed with 400 units of energy. Every action consumes one unit of its energy. The possible energy levels for an ant are from 0 to 400. 0 means the ant died of starvation, 1 that it is almost dead: one more action would kill it. If fed past 400, the ant explodes. When it dies, it turns into a skeleton. The remains of a dead ant stay on the screen and have some food value. The student must tell the ant what to do to plant a garden and to eat food to stay alive. Here are some keywords we will be using to qualify the state of energy the ant is in:

STARVED	The ant has 100 units of energy or less.
HUNGRY	The ant has 200 units of energy or less.
FED	The ant has strictly more than 200 units.
FULL	The ant has strictly more that 300 units.
STUFFED	The ant has strictly more that 375 units.

0	100	200	300	375	400
STARVED					STUFFED
HUNGRY			FULL		
		FED			

The consuming of energy can be turned off with the OFF command and back on with the ON command. The OFF mode can be used for such things as programming the ant to follow a large maze.

## 7. GROWTH OF PLANTS

Seeds and sprouts planted by the ant on the field will grow with time as follows:

- Seeds grow into germinated seeds in 60 time units.
- Germinated seeds grow into sprouts in 40 time units.
- Sprouts will grow into stalks in 40 time units.
- Stalks grow into branching plant in 140 time units
- Branching plants will grow a flower in 60 time units
- Plants with flower will become a mature plant and grow a fruit called “atple” in 60

time units.

The whole process from seed to mature plant takes 400 time units. Plants in baskets do not grow nor do they perish.

## 8. FOOD VALUES

The following table gives the food values for different things the ant can eat. This value gets added to the ant’s energy level after the one unit it takes to do the eating is subtracted off.

Mature food	25 units
Flowering plant	19
Branching plant	15
Stalk	11
Sprout	7
Germinated Seed	5
Seed	3
Skull	9
Bone	7
Head	13
Leg or Tail	11
Fat body	30
Field marker	two times the marker’s number (row or column)

## 9. REPEATING COMMANDS

One will soon discover that it takes a lot of repetitious typing to move the ant somewhere that is very far away. It would be desirable to tell the ant to do something over and over several times. The commands to control repetition, called iteration in computer science, are the DO commands. Iteration involves two main parts. One is that which is to be repeated, the “body” of the iteration or loop. The other is the termination condition, the test that determines when the loop is finished looping. The word DO is the signal of the beginning of an iteration. All the command words from there up to the specification of the ending test constitute the body of the loop, that which is repeated. The following are the variations of the DO commands:

### **DO commands number TIMES**

This form tells the ant to repeat the specified commands the specified number of times. For example, the command

**DO EAT PLANT MOVE 10 TIMES**

tells the ant to eat what is under his head, plant a new seed there, and then move one step forward. It is to do this over and over 10 times.

### **DO commands TO test**

### **DO commands UNTIL test**

where ‘test’ is one of the following forms:

**ROW number**  
**COLUMN number**  
**COL number**  
**SEE object direction**  
    (objects are **DIRT**, **JUNK**, **PLANT**, or **FOOD**)  
    (directions are **AHEAD**, **LEFT**, or **RIGHT**)  
**SMELL object**

**STARVED, HUNGRY, FED, FULL, or STUFFED**  
**BEYOND ROW number**  
**PAST ROW number**  
**BEYOND COLUMN number**  
**BEYOND COL number**  
**PAST COLUMN number**  
**PAST COL number**  
**FACING facing\_direction**  
    (facing\_directions are N, NE, E, SE, S, SW, W, or NW)  
**NOT test**  
**test AND test**  
**test OR test**

These two forms tell the ant to keep repeating the commands until the indicated test is satisfied or True.

The rows and columns are numbered along the left side and the top of the screen for convenience. The 'ROW number' or 'COLUMN number' condition is satisfied whenever the ant's head is on that row or column at the time the test condition is checked. If the ant is never going to reach the specified row or column it will repeat the commands until it reaches the end of the field.

Two tests are based on the ant's senses, 'SEE object direction' and 'SMELL object'. The ant can see any of the three squares in front of it, the one directly ahead, the one ahead and to the left, and the one ahead and to the right. In each case this is the square the ant would reach by either moving once or moving once and turning once. The ant can smell the square directly under its head.

The object 'DIRT' is a blank square; the object 'JUNK' is any part of a dead ant; the object 'PLANT' is any of the plant symbols above the ground (this does not include seeds and sprouts); and 'FOOD' is the mature plant.

The ant has five tests of its appetite. Refer to the chapter 6 on Energy Levels.

The 'BEYOND' and 'PAST' tests allow the ant to check whether it is located beyond a particular point. The ant is beyond column x if the ant's head is in a column with a number greater than x. For example, the ant is beyond column 10 if its head is in column 11, 12, 13, etc. 'NOT BEYOND ROW 7' would be a test for the ant's head being located at or before row 7.

The 'FACING direction' command allows the ant to check its orientation; for example, FACING E is true when the ant is horizontal, pointing to the right.

Tests can be combined to create more complex tests using the AND, OR, and NOT words. For example you can create tests such as:

ROW 5 OR COLUMN 10  
HUNGRY AND SMELL FOOD  
SEE FOOD AHEAD AND NOT STUFFED

The tests are evaluated from left to right. To prioritize the evaluations of a test, one can use the LEARN command (see DEFINING NEW COMMANDS).

Some examples of DO commands are:

DO MOVE TO ROW 2

This causes the ant to move forward until its head reaches row 2. If the ant is not headed in the direction of row 2, it will move a long way, probably off the field, before giving up. If it has not starved by then, it can be turned around and headed back onto the field.

DO MOVE EAT UNTIL NOT SEE FOOD AHEAD

This will cause the ant to eat a row of food if it is positioned heading into the row in front of the first plant. However, if the row is long or the ant is already reasonably well fed, it may eat too much and explode before it reaches the end of the row of food plants.



Suggested exercises:

1. Make the ant turn completely around.
2. Tell the ant to make a row of 20 plants.
3. Move the ant from its starting position to row 2, turn left twice and move to the first food.
4. Tell the ant to eat every other plant in the row of food.

## 10. SELECTION COMMANDS

There are three basic logic structures that make programs possible. The first two have already been covered. They are that of a sequence, which is progressing logically from one thing to another in order, and that of an iteration, which is repeating something until some condition is met. The third logic structure is the selection, which is choosing between two or more alternative courses of action. The ant can choose to do something or not to do something by the use of the IF command which has the following form:

### IF test command

The test can be any of the forms given above in the discussion of the iteration, the DO command. If the test specified is true, the command is performed; otherwise, the command is skipped over and not done. The command can be any single command, including the repetition command. For example,

IF SMELL DIRT PLANT

causes the ant to plant a seed in the space under its head only if that space is empty. Only the command word immediately after the test is affected. This means that if multiple commands are to be performed when the test is true, they must be made into a new command for the ant to LEARN so that a single command name can follow the test. How to define new commands is explained in the following chapter.

Using the selection command inside a DO command with the selected command being STOP or QUIT causes the DO command to be stopped. For example,

DO MOVE IF ROW 2 STOP IF COLUMN 10 STOP 50 TIMES

causes the ant to move forward 50 times or until reaching row 2 or column 10, whichever of these three conditions happens first.

Here are some more examples of the IF command:

DO MOVE IF SMELL FOOD EAT 20 TIMES

would cause the ant to go forward 20 times and at each spot if there is a mature food there, he eats it.

LEARN EATLEFT IS MOVELEFT EAT  
LEARN MOVELEFT IS TURN LEFT MOVE TURN RIGHT  
IF SEE FOOD LEFT EATLEFT

would cause the ant to eat any food it sees ahead on the left by moving over there and eating.

IF SEE FOOD LEFT EATLEFT IF NOT SEE FOOD LEFT MOVE

would cause the ant to move over to the next row (or column) to the left and eat if it saw food over there; but if it did not see food, it would simply move one step straight ahead.

Sometimes, we want an alternative for when a test is true, and a different one for when the test is false. Here are similar forms to help you do that:

**IF test command ELSE command**  
**IF test command OTHERWISE command**

If the test is true, then the first command is performed and the second command is skipped. If the test is false, then the first command is skipped and the second command is performed.

IF SMELL FOOD EAT ELSE MOVE

causes the ant to eat if there is any food under its head, else to move one step forward.

The commands may be any single command including repetition, but may not be another selection command. Sequences and selections can be used for the commands by defining them with LEARN and using their names as the single commands.

Suggested exercises:

1. Tell the ant to eat food if it is there but not eat anything else.
2. Tell the ant to plant a seed where there is dirt in the place where the seed would be planted, i.e. under the ant's nose.

## 11. DEFINING NEW COMMANDS

In order to write bigger programs, it is necessary to build them up out of smaller pieces. This is done by having the ant learn definitions for new command words. These new words all have to be defined in terms of other words, either other defined words or basic commands. The words used in the definition do not themselves have to be defined at the time they are used in a definition, only when the ant is given a program to perform involving them. When learning something, the ant is unable to do something else. So no command can follow a LEARN command in the command line. A command learned by an ant is also learned by any ant of the colony. The command to make the ant learn a new command is:

### **LEARN name IS commands**

The name given is remembered by the ant as the sequence of commands given. This name can then be used as a new command. For example

LEARN TURNAROUND IS DO TURN LEFT 4 TIMES

can be used to tell the ant how to turnaround so that TURNAROUND can subsequently be used as a command to the ant.

Here is an example of using LEARN to create a large program. The program is named PLANT-A-FIELD and it has the ant plant four rows side by side of ten plants each. The program is designed top-down by thinking of the natural way of looking at the problem. Assuming the ant is facing right we can imagine it planting one row, turning the corner, and planting another row back to the left. It turns that corner and repeats these steps for the next two rows. This is:

LEARN PLANT-A-FIELD IS PAIR-OF-ROWS PAIR-OF-ROWS

After having expressed the logic for the whole problem, we can now concentrate on just the part PAIR-OF-ROWS. Following the discussion above, it would be:

LEARN PAIR-OF-ROWS IS PLANT-A-ROW RIGHT-END  
PLANT-A-ROW LEFT-END

Similarly, we would continue designing each part until all new words have ultimately been defined in terms of the basic commands. The rest of the defined commands for PLANT-A-FIELD would be:

LEARN PLANT-A-ROW IS DO PLANT MOVE 10 TIMES  
LEARN RIGHT-END IS TURN RIGHT MOVE DO TURN RIGHT 3  
TIMES  
LEARN LEFT-END IS TURN LEFT MOVE DO TURN LEFT 3 TIMES

Suggested exercises:

1. Enter TURNAROUND and PLANT-A-FIELD as commands and try them.
2. Teach the ant to understand BOX as meaning go around in a square. Do this by defining SIDE, TOP, BOTTOM, and CORNER.
3. Define a command EATJUSTFOOD that will eat food if it is there but not eat anything else.
4. Define a command PLANTSEED that will plant a seed when there is dirt in the place where the seed would be planted, i.e. under the ant's nose.
5. Define a set of commands to use to FINDFOOD, searching left, right and ahead. It should move over the food ready to eat and then stop.

Additional practice exercises:

5. Program the ant to plant a triangle.

6. Program the ant to do a dance.
7. Program the ant to go around a square eating (but not too much) and planting in the open spaces. Note that this program could be used to make the ant survive automatically for a long time.
8. Program the ant to plant a pyramid.
9. Teach the ant the command GARBAGE-TRUCK which will travel 10 spaces (rows or columns, depending on direction) picking up (eating) any junk that is ahead, left, or right of its path.

### *Test commands*

The ant can learn tests, just like it learns commands. For example we can teach the ant how to check if it has reached the edge of the field:

LEARN BORDER IS ROW 2 OR ROW 22 OR COLUMN 2 OR COLUMN 35

Or those two other examples:

LEARN READY-TO-EAT IS NOT STUFFED AND SMELL FOOD OR  
STARVED  
LEARN OK-TO-PLANT IS SMELL DIRT OR SMELL JUNK

could be used in programs such as:

DO IF READY-TO-EAT EAT IF OK-TO-PLANT PLANT MOVE 20 TIMES  
DO IF OK-TO-PLANT AND NOT BORDER PLANT MOVE 10 TIMES

### *Commands with parameters*

Sometimes, we want the ant to perform some commands like moving forward 20 steps, and sometimes moving 40 steps. We can teach the ant to move a certain number of steps that will be specified only when we will ask the ant to execute the new command.

The syntax of the LEARN command with parameters is:

## **LEARN name parameters-or-template-words IS commands**

The name given along with its parameters and template words is remembered by the ant as the sequence of commands given. A parameter cannot be an Antfarm keyword, or a new command word. A template word can be any keyword, or any word that does not appear in the sequence of commands after the IS keyword. When we ask the ant to execute the 'name' command, we tell it what parameters to use. Those parameters will be substituted in the commands before the ant starts the execution. For example:

```
LEARN MAKEBIGMOVEOF NUMBER IS DO MOVE NUMBER TIMES
```

tells the ant how to move NUMBER steps forward. To have the ant move 20 steps forward, we only need to type:

```
MAKEBIGMOVEOF 20
```

and to move 40 steps forward:

```
MAKEBIGMOVEOF 40
```

Template words can be used to make the program more readable.

```
LEARN MAKE MOVE OF NUMBER STEPS IS DO MOVE NUMBER  
TIMES  
MAKE MOVE OF 20 STEPS  
MAKE MOVE OF 40 STEPS
```

The noisy words 'THE' and 'A' can also be added. They will simply be ignored. For example, the last command could also be written: MAKE A MOVE OF 40 STEPS.

## ***TELL and FORGET commands***

The TELL and FORGET commands allow us to know what the ant colony has learned, and tell it which commands to forget. They have the following forms:



## **TELL**

This command lists all the new commands the ant colony has learned so far.

## **FORGET name**

## **FORGET ALL**

The first command erases the given name from the list of new commands the ant colony has learned. After this, that name will be an undefined command. Using the word 'ALL' in place of a name causes the ant colony to forget all the commands it has learned.

## 12. ANTS AND BASKETS

An ant can possess baskets in which it can store things to be used later. The ant colony is very picky about order so that a basket can hold only one type of items. The available types of basket are the SEED, SPROUT, FLOWER, FRUIT and JUNK basket types. The baskets are distinguished from one another by their color. The available colors are RED, GREEN, BLUE and YELLOW. Because the ants could (and would) be confused, the ant cannot possess 2 baskets of the same color, even if they are of different types. To get a basket, the ant has to request one using the GET command. The GET command takes one unit of time to execute and consumes one unit of the ant's energy.

**GET THE color type BASKET**

**GET A color type BASKET**

**GET color type BASKET**

The ant will receive an empty basket of the specified color that can contains only the specified type of items. 'Color' must be RED, GREEN, BLUE or YELLOW. 'Type' must be SEED, SPROUT, FLOWER, FRUIT, or JUNK.

**GET THE color type BASKET WITH number itemtype**

**GET A color type BASKET WITH number itemtype**

**GET color type BASKET WITH number itemtype**

This is the same as above, except that the basket will be filled with the specified number of items. 'Type' and 'itemtype' must be identical, and must be one of the following: SEED, SPROUT, FLOWER, or FRUIT. It is possible to add an 'S' at the end

of itemtype for better English. 'Number' must be a positive integer. For example:

GET A RED SPROUT BASKET WITH 10 SPROUTS

### **13. MORE BASIC COMMANDS USING BASKETS**

Items can be moved into baskets with the PUT and PICK commands. PUT moves items from one basket to another, while PICK moves items from the field into a basket.

The EMPTY command allows the ant to empty a basket, and the THROW AWAY command to get rid of a basket. We have seen how the ant can PLANT a seed that is stored in its mouth, and how it can EAT something on the field. The PLANT and EAT commands can also be used to plant and eat from a basket.

**PICK INTO THE color BASKET**

**PICK INTO color BASKET**

The ant picks what is under its head and puts it into the specified basket

**PUT THE color BASKET INTO THE color BASKET**

**PUT color BASKET INTO color BASKET**

The ant will transfer the whole content of the first basket into the second. The two baskets must be of the same type.

**PUT number itemtype FROM THE color BASKET INTO THE color BASKET**

**PUT number itemtype FROM color BASKET INTO color BASKET**

The ant will transfer the specified number of items from the first basket into the second. The two baskets must be of the same type. 'Itemtype' is defined as in the GET command. If the first basket has less than the specified number of items, then the entire first basket's content is transferred.

**EMPTY THE color BASKET**

**EMPTY color BASKET**

The ant will empty the specified basket.

**THROW AWAY THE color BASKET**

**THROW AWAY color BASKET**

The ant gets rid of the specified basket.

**EAT FROM THE color BASKET**

**EAT FROM color BASKET**

The ant will eat one item from the specified basket. There will be one less item in this basket, and the ant's energy will be increased by the food value of this item, after it has been decreased by one unit for the action of eating.

**PLANT FROM THE color BASKET**

**PLANT FROM color BASKET**

The ant will plant one item from the specified basket. The ant can plant only from SEED or SPROUT baskets.

#### **14. COMMUNICATING WITH THE ANT: INPUT/OUTPUT**

Ants have the capability to communicate with any specie that can type on a keyboard and decipher text on the screen, such as most human beings. They can say things by writing messages in the communication area of the screen, and listen to what is typed on the keyboard.

##### **SAY “some message”**

The ant will display the message “some message” in the communication area of the screen. This command takes one unit of time.

##### **LISTEN FOR INPUT**

The ant will wait for the user to enter a message in the communication area and to click the SEND MESSAGE button. The INPUT basket is then emptied, and the entered message stored in it. This command takes one unit of time.

In this next example, the ant asks the user for a number, listens to it, and stores the number into its input basket. It then transfers as many fruits as the user said from the red basket into the blue basket.

```
SAY “Please, give me a number of seeds”  
LISTEN FOR INPUT  
PUT INPUT FRUIT FROM RED BASKET INTO BLUE BASKET
```

In this example, the ant asks the user for the basket color, and then picks what is under its head, places it into that basket, and moves, and repeats the PICK and MOVE commands 10 times.

```
SAY “Which basket shall I use? (give me a color)”  
LISTEN FOR INPUT  
DO PICK INTO INPUT BASKET MOVE 10 TIMES
```

An ant can say what is in its baskets, including the INPUT basket.

**SAY WHAT IS IN THE color BASKET**

**SAY WHAT IS IN color BASKET**

**SAY WHAT IS IN THE INPUT BASKET**

**SAY WHAT IS IN INPUT BASKET**

**SAY WHAT IS IN ALL BASKETS**

This will cause the ant to list everything that is in the specified basket(s).

## 15. SIMULATOR COMMANDS

The following simulator commands take zero time to execute and thus do not have any effect on the ant's energy:

### **CHANGE oldname TO newname**

Change the name of a learned command (oldname) to newname.

**RENAME name** Change the Ant Colony name.

**NEW number** Create a new ant with the given number. The new ant becomes the active ant. The previous active ant becomes inactive. If an alive ant with this number already exists, it is destroyed.

**ON** Set the Energy Mode to ON. Ants' energy will be decreased with time, and increased as they eat.

**OFF** Set the Energy Mode of OFF. The ants' energy will not changed.

**ANT number** If there is an alive ant with that number, then deactivate the current active ant if any, and activate the ant with the number specified.

**QUIT** End a loop if inside a loop, end the command line else.

**STOP** Same as above.

**DONE** Same as above.

## 16. FILE HANDLING

### *Ant Commands File*

The commands an ant colony learns can be saved or retrieved as an ASCII file through the submenus: “Save Ant Commands”, “Save Ant Commands As...” and “Open Ant Commands...” provided under the File menu. The default name for an Ant Commands file is the colony name followed by the extension “ANT”.

### *Field File*

Storing and retrieving a field is achieved with the “Save Field”, “Save Field As...” and “Open Field...” submenus under the File menu. Saved fields include everything about the field except the ants that are alive. The default name for a Field file is the colony name followed by the extension “FLD”.

If you wish to change the default name for the Ant Commands and Field files, use the RENAME command to change the colony name.

To start a new session, (new colony on a new field), use the “New Field” submenu under the File menu.



## 17. List of Keywords and Noisy Words

### Keywords:

AHEAD  
ALL  
AND  
ANT  
AWAY  
BACKUP  
BASKET  
BEYOND  
CHANGE  
COL  
COLUMN  
DIRT  
DO  
E  
EAT  
ELSE  
EMPTY  
FACING  
FED  
FLOWER  
FOOD  
FOR  
FORGET  
FROM  
FRUIT  
FULL  
GET  
HELP  
HUNGRY  
IF  
IS  
INPUT  
INTO  
JUNK  
LEARN  
LEFT  
LISTEN  
MOVE  
N  
NE  
NEAT  
NEW  
NOT  
NW  
NOTHING  
OFF  
ON  
OR  
OTHERWISE  
PAST

PICK  
PLANT  
PUT  
QUIT  
RENAME  
REST  
RIGHT  
ROW  
S  
SAY  
SE  
SW  
SEE  
SEED  
SMELL  
SPROUT  
STARVED  
STUFFED  
TELL  
THROW  
TIMES  
TO  
TURN  
UNTIL  
W  
WAIT  
WITH

### Noisy Words:

A  
THE

VITA

Virginie Cochard

Candidate for the Degree of

Master of Science

Thesis: A LANGUAGE TO LEARN COMPUTER CONCEPTS

Major Field: Computer Science

Biographical:

Education: Graduated from the Superior and Special Mathematics, preparatory years for the Engineer Schools in 1987. Admitted to the National Superior Engineer School of Electronic, Electrotechnic, Computer Science and Hydraulic of Toulouse, France (ENSEEIH) in the Computer Science and Applied Mathematics Department and graduated in 1990. Completed the Requirements for the Master of Science degree with a Major in Computer Science at Oklahoma State University in December, 1999.