DRLOGO.WS4

- "Digital Research's Dr. Logo"
  Gary Kildall & David Thornburg
  "BYTE", June 1983, p.208

(Retyped by Emmanuel ROCHE.)

Logo for personal computers has been heralded by some as the beginning of a revolution in computer languages that promises to be as far reaching as the introduction of the personal computer itself. Yet, many people think that Logo is not much more than a graphics language for children. Adding to this confusion is the fact that some commercial implementations of Logo are weak (somewhat akin to a version of English that contained no adjectives). Because of the confusion surrounding Logo itself, the appearance of a sophisticated version of this language on a professional microcomputer such as the IBM Personal Computer might be expected to raise some eyebrows. The development of a powerful Logo for 16-bit computers such as the IBM PC can change our way of thinking about programming.

In this article, we will show what makes Logo truly powerful, what it can be used for, and how Digital Research's Dr. Logo, with its powerful language, large workspace, and complete program-development environment, sets a new benchmark by which to measure the properties of useful computer languages.

To help you understand the power of Logo, we will give you some background about the earlier language LISP. LISP, developed more than 20 years ago by John McCarthy, is overwhelmingly the language of choice for researchers in the field of artificial intelligence. Unlike many other languages, LISP lets users perform operations on several data types, including numbers, words, and lists. A list can consist of a collection of words, numbers, or lists themselves. Because the names of LISP primitives or procedures are also words, one can write LISP programs that automatically generate other LISP programs. It is the ability to manipulate this type of data that gives LISP its name (LISt Processing).

LISP has been used to explore topics as diverse as image processing, the analysis of natural language, the computer solution of certain types of "intelligence" tests, and theorem proving. More mundane programs in LISP (such as word processors) have also been created. Viewed from any angle, it is a powerhouse of a language.

Dr. Logo incorporates the list-processing capabilities of LISP with a syntax that can be learned by children. More than the utility (and beauty and simplicity) of turtle graphics, it is this list-processing capacity that gives it so much power.

Other important characteristics are shared by Logo and LISP. Among these is the ability to extend the language through the creation of procedures that are treated just as if they were part of the language itself. As with some FORTH devotees, many Logo enthusiasts see themselves as not writing programs, but as

creating new "words" in Logo tailored to the solution of their particular programming task. While this may appear to be a subtle distinction, it has a tremendous effect on programming style. This style affected the design of Digital Research's Dr. Logo in several ways, especially in the debugging and procedure-management tools.

The power of Dr. Logo

Before showing what Logo procedures look like, we will list a few of the characteristics of Dr. Logo. To provide maximum power to the user, we designed the first implementation of Dr. Logo for the 16-bit IBM Personal Computer. The use of a 16-bit processor greatly increased the amount of workspace available to the user, and also yielded a modest speed improvement over 8-bit versions of the language. A Dr. Logo user with 192K bytes of RAM has about 10,000 nodes available for use. For comparison, an Apple II user running Apple Logo has only about 2800 free nodes to work with. It goes without saying that sophisticated applications require comparably more workspace than simple ones, and it was important to its designers that Dr. Logo be able to handle sophisticated applications.

In addition to list processing and turtle graphics primitives, Dr. Logo can work with integers (30-bits long, plus a sign) and both single-precision and double-precision floating-point numbers. A full set of transcendental functions (log, square root, etc.) allows this language to be used for scientific programs as well.

Dr. Logo is a superset of Apple Logo and more than just a language. A complete programming environment, it includes its own operating system, program editor, debugger, and a set of workspace-management tools designed to speed the successful implementation of even the most convoluted artificial-intelligence program.

The graphics system is designed to use either the color monitor alone, or to use the color monitor for turtle graphics or mixed text/graphics applications and the monochrome monitor for procedure editing, debugging, and pure text programs. The color display uses the 320- by 200-pixel medium-resolution mode, and supports 16 background colors (8 colors that are either bright or dim). It also supports 2 foreground color sets of 4 colors each.

A brief glimpse at Logo procedures

Before describing the editor and workspace-management tools, we will examine what a Logo procedure looks like by illustrating the creation and manipulation of a list. A list in Logo is a collection of words, numbers, or lists that are enclosed in square brackets ("[" and "]"). Each item in the list is separated by a space. For example, [cow horse sheep snake] is a list; so is [1 1 2 3 5 8]. The first list consists of the words cow, horse, sheep, and snake; the second list consists of the first 6 numbers of the Fibonacci series. A more complex list would be [car [dump truck] airplane [railroad engine]], in which 2 of the elements are words (car and airplane) and 2 elements are lists of 2

words each ([dump truck] and [railroad engine]). Also, a list can have one word in it ([yellow]) or even be empty ([]).

In common with other computer languages, Logo allows values to be assigned to names. For example, you can assign a list to a name with the MAKE command, e.g.:

```
make "friends [Pam Roy Pat George]
```

The quotation mark (") is used by Logo to indicate that FRIENDS is a word (a variable name in this case) and not a command. If we tell Logo to

```
print :friends
```

we will see

```
Pam Roy Pat George
```

on the screen. The colon (:) in front of FRIENDS lets Logo know that we want to see what is bound to the variable, rather than the variable name itself. If we had entered

```
print "friends
```

we would have seen

```
friends
```

on the screen instead.

You can take lists apart in Logo with commands such as FIRST, BUTFIRST, LAST, and BUTLAST. For example, if we enter

```
print first :friends
```

the screen will show

```
Pam
```

The command

```
print butfirst :friends
```

prints

```
Roy Pat George
```

Now that we know a little about lists, let us explore Logo's extensibility by creating a new command in the language. Suppose you did a lot of work with lists and you found that you would like to rotate a list by moving its first element to the rear end and pushing everything else up front. We can create a word (e.g., rotate) to do this for us. If we had such a procedure, we could make a rotated version of friends by entering

```
make "neworder rotate :friends
```

Because Logo does not have a primitive called ROTATE, we can create a procedure with this name that looks like the following:

```
to rotate :list
output sentence butfirst :list first :list
end
```

This procedure accepts a list (denoted by the local variable name :LIST) and makes a new list starting with all but the first word, and then appending the first word to the end of the list. The SENTENCE primitive (or native instruction) is used to assemble a list from two parts. The OUTPUT command passes the new list back out of the procedure to any procedure that used ROTATE, or to the command level.

Once defined, Logo procedures are treated just as if they were part of the language's native vocabulary. For example, if you were to enter

```
print rotate :friends
```

the list

```
Roy Pat George Pam
```

would appear on the screen.

Logo's ability to manipulate lists by taking them apart, adding to them, examining their contents, and altering their order is central to the use of Logo in the creation of knowledge-based programs. For an excellent introduction to the use of lists in the creation of a knowledge "tree" that "sprouts" new nodes as the program gets "smarter," you should read Harold Abelson's discussion of the program Animals in his book "Apple Logo".

In addition to the ability to perform list processing and arithmetic, Dr. Logo also supports an excellent turtle graphics environment. While much has been written about turtle graphics, especially on its use with children, it is important to understand that turtle graphics is of tremendous value to expert programmers as well. The power of this graphics environment comes through its description of the shape of an object as a series of incremental steps that create it. Once a procedure describing an object has been written, the object can be displayed at any screen location, orientation, and size without having to tamper with the basic description. For example, the procedure

```
to square :size
repeat 4 [forward :size right 90]
end
```

can be used to create a square at any screen position, angular orientation, or size. To draw a square at a given place, you first instruct the turtle (a cursor that has both position and orientation) to move to a specific x-y coordinate and heading (angle). Next, you type SQUARE 50, for instance, to create a square with sides 50 units long. This property of turtle graphics procedures, coupled with Logo's capacity to run recursive programs, has

allowed the easy exploration of geometrical shapes and their properties.

Logo, turtles, and kids

Anyone who has watched the personal computer industry for the past few years has probably seen the evolution of certains myths regarding computer languages. Many devotees of BASIC, for example, claim that it is the optimal choice for the home user because of its nearly universal adoption as the default language for personal computers. The fact that BASIC was the only high-level language that was readily available in compact form in the late 1970s is not considered to be relevant by many users. Fortunately, the recent availability of other languages on personal computers (Logo, Pascal, Forth, and Pilot, to name but a few) has afforded programmers other choices. But some of these languages have myths of their own.

In the case of Logo, the common myth is that it is a turtle graphics language designed to be used exclusively by children. As evidence in support of this myth, one is pointed to Seymour Papert's book "Mindstorms." It is true that Papert devotes the bulk of his book to the use of turtle graphics as a powerful programming and discovery tool for children, and that he stresses the accessibility of Logo to the young and inexperienced.

The problem with the Logo myth is that it suggests that Logo is exclusively for children's use. As with many myths, the reality of the situation is quite different. First, it is true that Logo supports turtle graphics. In this regard, it is similar to some versions of Pascal, Pilot, and Forth. Note also that, while turtle graphics is accessible to children, it also has applications of value to advanced programmers as well. Anyone who doubts this would benefit from reading "Turtle Geometry" by Abelson and diSessa, or "Discovering Apple Logo" by Thornburg.

The point is that Logo is no more a "kid's" language than is English. Yes, English is the language of "Mary Had a Little Lamb," but it is also the language of "Moby Dick" and Shakespeare's sonnets.

At its base, Logo is a symbol-manipulation language in the finest sense of the word. Rooted in the artificial-intelligence language LISP, Logo allows the user to extend its vocabulary, to use recursion, and to manipulate various types of data in ways that are nearly impossible with languages like BASIC.

It would be a shame if the myth of Logo kept serious programmers from exploring a language whose foundation goes to the heart of computer science itself.

Programming tools

Dr. Logo provides many tools to assist the programmer. While smaller Logo systems can adequately survive with a rudimentary procedure editor, larger Logo environments benefit from some of the extra tools that make program analysis and debugging less tedious. Dr. Logo's procedure editor allows the

use of both uppercase and lowercase letters for programs and data. Two primitives, UPPERCASE and LOWERCASE, allow the conversion of a word from one case to the other. Also, procedure listings can be indented to make decision branches and nesting easier to see. While not essential to the creation of good programs, such formatted listings are easier to read.

While Logo's syntax generally makes procedures easy to read, it is valuable to have comments appended to certain program lines. This ability is provided in Dr. Logo, along with the ability to strip these comments from procedures with the NOFORMAT primitive if more workspace is needed. If the name or syntax of a Logo primitive or editing command is forgotten, online help is available.

Once procedures are created, Dr. Logo has several primitives that help show how procedures interact with each other. This is especially important for those Logo enthusiasts who experiment with several coexisting versions of procedures before settling on the final choices. Most versions of Logo will print the names of resident procedures on receiving the POTS commands (Print Out TitleS). If, in Dr. Logo, you enter POTL, the workspace will be examined for all top-level procedures (those not called by other procedures) and their names will be displayed on the screen. If you enter POCALL followed by the name of a procedure, Dr. Logo will examine the calling structure of the named procedure, and print the names of the procedures used by the one mentioned, as well as the names of the procedures used by these secondary procedures, and so on until the calling sequence is complete. This gives a great deal of information on the internal organization of the Logo workspace. If, on the other hand, you enter POREF followed by a procedure name, all the procedures that reference this name will be found and displayed.

Many Logo programmers create procedures in a haphazard sequence. Because a listing of multiple procedures follows the sequence in which they were entered, large listings can be hard to assimilate. By using the Dr. Logo FOLLOW command, procedures can be resequenced in any order, thus allowing large listings to be more easily scanned.

Once you are ready to try a Logo program, Dr. Logo provides additional tools to assist in debugging. One of these tools allows the text screen to be split into windows corresponding to the command level, a user I/O port, and the debugger. The TRACE command traces the procedure, and displays what is happening and at what level the procedure is, relative to the top (command) level. Because a single recursive procedure (that calls a copy of itself) may oscillate through many levels, knowing the level at which an error occurs is helpful when fixing the fault. The command WATCH allows single-step execution of a procedure, with the ability to change values and see the effect of each statement.

The use of multiple text windows in debugging is only one application for this powerful tool. The development of good window-management tools can, by itself, increase the simplicity, flexibility, and power of this programming environment.

Applying Dr. Logo in Education

Perhaps because of its historic use as a discovery tool for children (and because of the typically small workspace found with most implementations), Logo is not generally perceived as an application language. It is anticipated that Dr. Logo will prove to be an exception in this regard.

The educational applications for Logo have typically focused on the use of turtle graphics. The beauty of turtle graphics is that children simultaneously acquire skills in programming, geometry, and art. Many children who are "turned off" by math have discovered it to be an exciting field through their exploration with turtle graphics. Furthermore, it has been found that, once a child uses Logo to discover new ways of thinking about mathematics, this new way of thinking continues to produce beneficial results -- even if the child is no longer exposed to Logo.

In the physical sciences, Logo can be used to construct "microworlds" in which bodies obey different natural laws, such as gravitation. By exploring these artificial microworlds, children can develop better intuitions about the properties of their own corner of the universe. (See "Designing Computer-Based Microworlds" by R.W. Lawler on page 138 of the August 1982 issue of "BYTE" magazine devoted to Logo.)

Given Logo's powerful list-processing capability, one would expect it to be of value in the language arts as well. To pick one simple example, suppose a child created several lists called nouns, verbs, adjectives, articles, etc., and assigned appropriate words to each list. The word order in each list can be randomized with the SHUFFLE command, and a random sentence can be constructed by assembling words from each list in a syntactically valid order. Legitimate nonsense sentences can be automatically generated in this fashion (e.g., No yellow toad smells tall people.) while bringing the child to look at and solve the structure of English.

The educational value of this program can be seen on several levels. First, if the child creates the lists of words, a misplaced word will show up as a misplaced part of speech. Having a verb appear when a noun is expected results in an obviously invalid sentence structure. The result is a self-reinforcing mechanism for learning the parts of speech. Second, the student can learn to identify valid sentence forms without sample words (sort of the reversal of the traditional parsing process). This helps to cement sentence structure concepts as well. Finally, the student learns some of the challenges awaiting those who want to create natural-language interfaces between people and computers.

Dr. Logo in Business

While Logo is not usually thought of as a language for business applications, Dr. Logo has several characteristics that may change this perception. The creation of an interactive illustration generator using an inexpensive graphics tablet is quite easy in Dr. Logo.

In addition to business graphics, the list processing capability of Dr. Logo makes it suitable for database management. In fact, one might envision incorporating some of the results of research in natural-language

understanding to generate a query system that responds to questions such as: "If we increase our salaries by 10 percent this year and increase our sales by 20 percent next month, what will our profit be in the fourth quarter?"

There is no question that many business applications will be found for Dr. Logo, but it is premature to set limits on the scope of these applications.

Dr. Logo in Artificial Intelligence

There has been much talk lately about knowledge-based or "expert" systems. The noble efforts of personal computer software experts notwithstanding, sophisticated microcomputer programs that can adapt to various queries are few and far between. The major reason for this is the inadequacy of most computer languages for dealing with the types of data and operations natural to adaptive systems. Because of Dr. Logo's close connection with LISP, we expect to see artificial-intelligence techniques appearing in personal computer software, rather than being limited to university and large industrial research centers as they have been in the past.

This movement is valuable for several reasons. First, it will help to demystify artificial-intelligence research. Second, it will result in the application of advances in artificial intelligence to the development of practical programs. To pick one example, suppose you had a computer program (called car repair) that allowed the following dialogue:

User: I hear noises when I steer the car.
Computer: Do you think the problem is in your steering mechanism?
User: Yes, I think so.
Computer: Do you have power steering?
User: Yes.
Computer: Is the noise loudest when you turn the steering wheel?
User: Yes, but I hear it when the car is idling, too.
Computer: You should check the level of your steering fluid before proceeding.
Do you know how to do that?
User: Yes.
Computer: Fine. Check the fluid level. If it is low, fill the reservoir and
see if the problem is fixed; otherwise, we will continue to explore
other causes.

Programs that allows this type of interaction can be used for many diagnostic applications, and might be far more valuable applications for home computers than checkbook balancers or recipe files.

Domestic applications for artificial intelligence represent a sleeping giant. The list-processing capability and large workspace of Dr. Logo will allow this giant to be awakened, and will enable the creation of a whole new class of applications software.

Dr. Logo is the first of a new family of languages that promises not only to change our programming style, but to alter the way we think about computing itself.

References

Logo:

1. "Apple Logo"
   Abelson, Harold
   BYTE/McGraw-Hill, 1982

2. "Turtle Geometry: The Computer as a Medium for Exploring Mathematics"
   Abelson, Harold and Andrea diSessa
   MIT Press, 1981

3. Special Logo Issue, BYTE, August 1982

4. "Mindstorms: Children, Computers, and Powerful Ideas"
   Papert, Seymour
   Basic Books, 1980

5. "Discovering Apple Logo: An Invitation to the Art and Pattern of Nature"
   Thornburg, David
   Addison-Wesley, 1983

Artificial Intelligence:

1. "Artificial Intelligence: An Introductory Course"
   Bundy, A., ed.
   North Holland, 1978

2. "Artificial Intelligence"
   Winston, Patrick
   Addison-Wesley, 1977

LISP:

1. Special LISP Issue, BYTE, August 1979

2. "LISP 1.5 Programmer's Manual"
   McCarthy, John et al.
   MIT Press, 1965

3. "LISP"
   Winston, Patrick and Berthold Horn
   Addison-Wesley, 1981

Listing
---

to graphics

```
;
; A sample business graphics program for bar graphs.
;
make "screen.height 190
make "screen.width 310
make "yfactor .25
make "zfactor .575
make "zdeg 22.5
make "xmin -139
make "xmax 139
make "ymin -79
make "ymax 119
make "return char 13
get.request
end

to get.request
(local "reply "h.or.v "s.or.o "2.or.3)
cleartext
make "reply prompt [Horizontal or vertical bars (h or v)] "char
if :reply = "h
  [make "h.or.v "h]
  [make "h.or.v "v]
if :reply = :return
  [stop]
make "reply prompt [Solid or open bars (s or o)] "char
if :reply = "s
  [make "s.or.o "s]
  [make "s.or.o "o]
if :reply = :return
  [stop]
make "reply prompt [2 or 3 dimensional (2 or 3)] "char
if :reply = 2
  [make "2.or.3 2]
  [make "2.or.3 3]
if :reply = :return
  [stop]
make :reply prompt [Values to be graphed] "list
if "reply = []
  [stop]
bar.graph :h.or.v :s.or.o :2.or.3 :reply
get.request
end

to prompt :text :type
local "reply
(type :text ": char 32)
if :type = "char
  [make "reply readchar print :reply output :reply]
  [output readlist]
end

to bar.graph :h.or.v :s.or.o :2.or.3 :values
cleartext
```

```
(local "max.value "min.value "origin "width "depth "axis "reply "graph.width
 "graph.height "proc "spacing)
if emptyp :values
  [stop]
make "max.value 0
make "min.value 999999999
if :h.or.v = "h
  [make "origin list :xmin :ymax
   make "graph.height :screen.width
   make "graph.width :screen.height
   make "axis 90]
if :h.or.v = "v
  [make "origin list :xmin :ymin
   make "graph.height :screen.height
   make "graph.width :screen.width
   make "axis 0]
if :2.or.3 = 2
  [make "spacing (:graph.width / count :values) * :yfactor]
  [make "spacing (:graph.width / count :values) * :zfactor]
if :2.or.3 = 2
  [make "width (:graph.width / count :values) * (1 - :yfactor)]
  [make "width (:graph.width / count :values) * (1 - :zfactor)]
make "depth :width * :zfactor
minmax :values
make "values scale :values :graph.height * .8 / :max.value
cleanup
penup setpos :origin pendown
if :h.or.v = "h
  [line [] list :screen.width ycor]
  [line [] list xcor :screen.height]
penup setpos :origin pendown
draw.bars :axis :width :spacing :2.or.3 :values
splitscreen
setcursor [0 23]
type [Press ENTER to Continue...]
make "reply readchar
end

to minmax :list
if emptyp :list
  [stop]
if first :list > :max.value
  [make "max.value first :list]
if first :list < :min.value
  [make "min.value first :list]
minmax butfirst :list
end

to scale :list :factor
if emptyp :list
  [output []]
output sentence (:factor * first :list)
scale butfirst :list :factor
end
```

```
to cleanup
hideturtle
setbg 0
penup
home
clean
pendown
end

to draw.bars :axis :width :spacing :2.or.3 :values
if emptyp :values
  [stop]
setheading :axis
draw.1.bar :s.or.o :2.or.3 first :values :width :depth :zdeg
setheading :axis + 90
forward :spacing + :width
draw.bars :axis :width :spacing :2.or.3 butfirst :values
end

to draw.1.bar :s.or.o :2.or.3 :height :width :depth :zdeg
(local "origin "direction)
make "origin pos
make "direction heading
if :s.or.o = "o
  [make "proc "open.bar]
  [make "proc "solid.bar]
run (list :proc :height :width)
if :2.or.3 = 2
  [stop]
forward :height
right 90 - :zdeg
forward :depth
right :zdeg
forward :width
right 180 - :zdeg
forward :depth
back :depth
left 90 - :zdeg
forward :height
right 90 - :zdeg
forward :depth
penup setpos :origin pendown
setheading :direction
end

to open.bar :height :width
repeat 2
 [forward :height right 90 forward :width right 90]
end

to line :pos1 :pos2
if not emptyp :pos1
  [penup setpos :pos1 pendown]
```

```
make "pos1 pos
setheading towards :pos2
forward sqrt sum
  sq ((first :pos1) - (first :pos2))
  sq ((last  :pos1) - (last  :pos2))
end

to sq :num
output :num * :num
end

to solid.bar :height :width
(local "course "origin)
make "course heading
make "origin pos
repeat :width / 2
  [forward :height right 90 forward 1 right 90
   forward :height left 90 penup forward 1 pendown left 90]
if remainder :width / 2 = 1
   [forward :height]
penup setpos :origin pendown
setheading :course
end


EOF
```

DRLRM0.WS4 (= "Dr. Logo Reference Manual", Foreword)

Dr. Logo
Language
Reference Manual
for the IBM Personal Computer

First Edition: August 1983

(Retyped by Emmanuel ROCHE.)

Foreword

Welcome to the world of Logo programming. Logo can help you grow as a programmer, whether or not you have ever programmed before, even if you are experienced with several programming languages. Digital Research has designed this version of Logo especially for your IBM Personal Computer.

What is Logo?

Logo is a powerful programming language that is rapidly gaining popularity because it is easy to learn and use. You use procedures as building blocks to create Logo programs. Logo itself is a collection of procedures, called "primitives", that you use to build your own programs.

During the 1970s, a team of computer scientists and educators at MIT, under the direction of Seymour Papert, developed Logo with turtle graphics to allow very young children to program and use a computer. They developed the turtle so that young learners could have, as Papert says, "an object to think with", a tool to help them learn in new ways.

The original Logo evolved in part from the LISP programming language. LISP dominates artificial intelligence programming because of its powerful symbolic and list processing capabilities. Logo too is a symbolic language, as opposed to an algebraic language like BASIC, COBOL, or FORTRAN. Because of its symbolic powers, Logo has strong potential to be used for sophisticated application and artificial intelligence programming.

Logo has evolved into a philosophy of education and a family of programming languages that supports the Logo philosophy on many different kinds of computers. However, implementations of Logo for the first generation of personal computers were limited by the amount of memory that 8-bit microprocessors can address. These implementations focused on the educational aspects of Logo, mainly turtle graphics, and were not suitable for commercial or academic programming. Dr. Logo greatly expands Logo's application potential.

What is special about Dr. Logo?

Dr. Logo is the first implementation of Logo designed for the second generation of personal computers, such as your IBM Personal Computer. The second generation, 16-bit microprocessors can address more than one hundred times more memory than their 8-bit predecessors. Dr. Logo helps you take advantage of the additional memory by providing greater workspace for your procedures, and additional workspace management, cross-reference, and debugging commands.

What do I need to run Dr. Logo?

You must have at least an IBM Personal Computer with 192K of memory, one disk drive, a color/graphics monitor adapter, and a color monitor. Of course, Dr. Logo's performance in enhanced if you have 256K or more of memory, a second disk drive, and a second monitor. The two-display system requires a monochrome adapter card and a monochrome monitor. When you have two displays available, Dr. Logo displays text on the monochrome monitor and graphics on the color monitor. Your IBM Personal Computer "Guide to Operations" tells how to install and set switches for the additional adapter card and monitor. Dr. Logo can print your graphic displays on any Epson printer that has the graphics option.

How should I use this book?

This book is a reference manual for Dr. Logo. Its organization follows traditions established by the manuals for many computer languages. If Dr. Logo is your first computer language, the organization might not be what you expect!

You might expect, if you asked someone for directions to the local supermarket, an answer like: "Go three blocks to the next stoplight, turn right, go another two blocks, and it is on the left side of the street." You would be very surprised if all you received was a map of the town and the store's address!

This book is like the map in that it contains more information than you will probably ever use, including information that you can find nowhere else. It can help you explore all of Dr. Logo, but it is not a guided tour of the highlights! If you want a guided tour, read a book that is designed to introduce you to Logo. This book is designed to define Dr. Logo's primitives and programming environment in detail.

It is possible, of course, to explore Dr. Logo with this book as your only guide. To do this, begin by turning to the END of the book and reading Appendix E, "Getting Started". In just a few pages, it tells you how to turn on your computer, insert your Dr. Logo system disk, start Dr. Logo, create a procedure, save it, erase it, restore it, and turn off your computer. Like many programming language manuals, this book begins by describing the smallest element of Dr. Logo, a character, and develops in following sections how to

build words, expressions, and procedures from these elements.

Information in this book is arranged both functionally and alphabetically. If you know what you want to do (for example, use the printer or draw a picture), you can find a list of all the related primitives in Appendix C, "Functional Command List", and background information on how they work in one of the first five sections. Section 6, "References to Primitives", gives a detailed description of each primitive in alphabetical order.

If you are new to computers, you might find some of the details confusing. Hang in there! As you spend time programming with Dr. Logo, you will find not only that the details make sense, but that many of them help simplify your programming tasks.

Table of Contents

(To be done by WS4...)

Appendixes

(Idem)

Tables and Figures

Tables

(Idem)

Figures

(Idem)

EOF

DRLRM1.WS4 (= "Dr. Logo Reference Manual", Section 1)

(Retyped by Emmanuel ROCHE.)

Section 1: Components of Dr. Logo

A procedure is an action that Dr. Logo can do. A Logo program is made up of procedures. You will probably start by writing one-procedure programs, but you will build your first procedures out of procedures that come with Dr. Logo, called "Dr. Logo primitives". Later, you will use the procedures that you have written, as well as Dr. Logo primitives, to build complex programs. Procedures are the basic building blocks for programming with Dr. Logo.

A procedure is made up of expressions. An expression has two parts: a procedure name and inputs to the procedure. That is why you need Dr. Logo primitives to build your first procedure. To Dr. Logo, everything you type is either a procedure name or an input to a procedure.

Although you can build a procedure out of many expressions, you can also ask Dr. Logo to evaluate expressions one at a time by entering them individually to the ? prompt, which is sometimes called the interpreter or toplevel. For example,

```
?print "Salutations!
Salutations!
```

In this example, print "Salutations! is an expression. The procedure name "print" identifies a Logo primitive that displays its input, Salutations!, on the screen. As in all examples in this book, the line following "?" shows what the user types at the keyboard. After you type a line at toplevel, you must press the Enter key, <--+, to tell Dr. Logo that you want the expressions on the line evaluated.

A procedure can require a certain number and kind of inputs. Input to a procedure can be words, numbers, or lists. This section formally defines how to put characters together to form procedure names, words, numbers, and lists, so that you can combine them into expressions. Then, it tells you how Dr. Logo evaluates a line when you put more than one expression on it.

1.1 Dr. Logo character set

To build a procedure name or word, you can use almost any character on your keyboard, including upper- and lowercase letters, numerals, and symbols. For example

```
box1
box2
RobinNest
```

```
draw&go
```

However, Logo gives special meaning to certain characters. For example, blank spaces delimit words or numbers; a blank space before and a blank space after a word set the word off from the rest of the line. The following special characters are also Logo delimiters:

Table 1-1. Dr. Logo special characters

Character	Type	Action/use
[		Begins a list
]		Ends a list
(		Begins a grouped expression
)		Ends a grouped expression
;		Begins comments
=	logical	Equal infix operation, outputs TRUE or FALSE
<	logical	Less-than infix operation, outputs TRUE or FALSE
>	logical	Greater-than infix operation, outputs TRUE or FALSE
+	arithmetic	Addition prefix operation, outputs sum of inputs.
-	arithmetic	Subtraction infix operation, outputs difference of two inputs.
*	arithmetic	Multiplication infix operation, outputs product of inputs.
/	arithmetic	Division infix operation, outputs quotient of inputs.
^	arithmetic	Exponent infix operation, outputs first input number raised to the second input power.

When you use a delimiter in an expression, you do not need to precede it with a blank to set it off from the rest of the line. Whenever one of these characters appears, Dr. Logo assumes that it starts a new word or number separate from anything else on the line. This simplifies typing expressions in many cases, such as

```
256+1026
```

but complicates using these characters in situations other than those that Dr. Logo expects. For example, if you try to use a dash as a hyphen (instead of a minus sign), you might see Dr. Logo add unwanted spaces:

```
?print [high-resolution turtle graphics]
high - resolution turtle graphics
```

To tell Logo that you want it to treat a delimiter as a part of a word, precede the special character with a backslash ("\") called "the quoting character" by Dr. Logo, which generates it when you type Ctrl-Q.

```
?print [high\-resolution turtle graphics]
high-resolution turtle graphics
```

Dr. Logo recognizes other special characters called "control characters" as

commands. Control characters can edit a procedure, interrupt and terminate a procedure, and change between full text, full graphics, and splitscreens. Section 3, "Editing Commands", and Section 4, "Text and Graphic Screens", describe Dr. Logo control characters.

1.2 Logo objects

A Logo procedure can require a specific number and kind of inputs to perform its task. The required input can be one or more words, lists, or numbers. When a procedure requires any one of these, we say that it can accept a "Logo object", meaning that it needs a word, a list, or a number to complete its operation. To make your procedures more flexible, you can represent objects with names called "variables". This section describes words, lists, numbers, and variables.

1.2.1 Words

A Logo word is a group of one or more consecutive characters separated from other characters on the line by delimiters. A blank space separates a word from the rest of the characters that follow it on the line. You can use periods (".") or underbars ("_") as connectors to make Dr. Logo treat several words as one word, as in the following example:

```
?print "choc.chip
choc.chip
?print "oatmeal_raisin
oatmeal_raisin
```

In English, a sentence is composed of words called "verbs" and "nouns". In Logo, an expression is composed of words called "procedure names" and "inputs", which can be other names. Dr. Logo needs a way to distinguish an input name (noun) from a procedure name (verb). So, when you type a Logo expression, you must precede an input name with quotes ("). If you do not, Logo complains. For example,

```
?print "hi_there
hi_there
?print hi_there
I don't know how to hi_there
```

Although quotes have this special meaning at the beginning of a word, they are not a delimiter, and can be use as a normal character within a word. Do not use quotes at the beginning of a word, except for this reason. Dr. Logo has a special primitive, quote, that has the same effect as quotes.

```
?print quote hi_there
hi_there
```

Other Dr. Logo primitives can examine a word character by character, take a word apart, and put a word together. When manipulating a word in one of these

ways, Logo treats a character in a word as a one-letter word. For example,

```
?first "zebra
z
?first first "zebra
z
?first "z
z
?butfirst "atypical
typical
?last "rough
h
?butlast "dates
date
?word "R2 "D2
R2D2
```

As mentioned in Section 1.1, "Dr. Logo character set", you can tell Dr. Logo that you want to use a delimiter character in a word by preceding the delimiter with a backslash ("\"). However, if a delimiter character is the first character in the word, and the word is preceded by quotes ("), in most cases you do not have to put a backslash between the quotes and the delimiter character.

```
?"+more
+more
?ascii "*
43
```

However, blank spaces and square brackets ("[" and "]") are treated as delimiters, even when preceded by quotes ("). Use a backslash ("\") after quotes (") to tell Logo to treat a space or square brackets as a normal character.

```
?ascii "[
ascii doesn't like an empty word as input
?ascii "\[
93
```

The Dr. Logo primitive "readquote" (or "rq") creates a word that contains all the characters in a line typed at the keyboard. If the line contains delimiters, readquote inserts backslant characters in front of each one, to make the line one word. You can use readquote to create a command line for a function key to recall; fkey requires a word as input. See the following example.

```
?fkey 2 rq
setd "b: resetd
?setd "b: resetd <Ctrl-G> <Enter>
```

1.2.2 Lists

A literal list is a series of Logo objects enclosed in square brackets ("[" and "]"). You can create a list without square brackets by using the "list" or "sentence" primitive. Each element of a list can be a word, a number, or another list. Within the list, objects are delimited from one another by spaces. Dr. Logo treats every object in a literal list as a literal object, so you do not need to put quotes in front of a word included within a literal list. The following are valid literal lists:

```
[Salutations!]
[L M N]
[[bread butter] [soup sandwich] [cheese crackers]]
[10 20 [22 25] 30]
```

The following examples show how to create a list using "sentence" and "list":

```
?list "L "M "N
[L M N]
?sentence [I like] "hamburgers
[I like hamburgers]
```

The same Logo primitives that manipulate elements of words manipulate elements of lists.

```
?butfirst [not really]
[really]
?butlast [not really]
[not]
?fput 5 [10 20 [22 25] 30]
[5 10 20 [22 25] 30]
```

After the fput expression is executed, the list above has five elements, the fourth of which is a list. The following example shows how to count elements of lists within lists:

```
?count [5 10 20 [22 25] 30]
5
?count item 4 [5 10 20 [22 25] 30]
2
```

The following table compares the four primitives that can take two objects as input and output a word or list.

Table 1-2. Comparison of list primitives

Primitive	Input 1	Input 2	Output
list	"yellow	"green	[yellow green]
sentence	"yellow	"green	[yellow green]
fput	"yellow	"green	"yellowgreen
lput	"yellow	"green	"greenyellow
list	"sky	[is blue]	[sky [is blue]]
sentence	"sky	[is blue]	[sky is blue]
fput	"sky	[is blue]	[sky is blue]

```
lput          "sky          [is blue]      [is blue sky]

list          [say hello]    [to me]        [[say hello] [to me]]
sentence      [say hello]    [to me]        [say hello to me]
fput          [say hello]    [to me]        [[say hello] to me]
lput          [say hello]    [to me]        [to me [say hello]]

list          [hello]        []             [[hello] []]
sentence      [hello]        []             [hello]
fput          [hello]        []             [[hello]]
lput          [hello]        []             [[hello]]
```

1.2.3 Numbers

A number is one or more numerals separated from other characters on the line by spaces or other delimiters. A number is a kind of word, but you do not have to put quotes (") in front of a number, unless you want to use it as a variable name. Dr. Logo does not try to treat a word that starts with a numeral as a procedure name. In fact, Dr. Logo will not let you define a procedure name that starts with a numeral.

A number can contain a + or - to indicate sign. But, because these special characters are delimiters and arithmetic operators as well as signs, you might accidentally combine two numbers into one expression. To prevent this, Dr. Logo interprets spaces and numbers combined with arithmetic operators as follows:

5-2 is interpreted as 3.

5 - 2 is also interpreted as 3. However,

5 -2 is interpreted as two numbers: 5 and -2.

Dr. Logo uses two kinds of numbers: integers and decimal numbers. You can input negative or positive decimal numbers with up to 15 significant digits, and any integer between 2147483647 and -2147483648.

Most arithmetic operations can accept either an integer or a decimal number as input. Some take integers as input, but output a decimal number.

```
?13 / 7
1.85714285714286
```

Dr. Logo supports primitives that convert decimal numbers to integers, create exponential and random numbers, and perform trigonometric and logarithmic functions. Internally, Dr. Logo uses both single- and double-precision integers and floating-point numbers. Dr. Logo always uses double-precision for calculations, but converts numbers to single-precision for display or storage when accuracy to 15 places is not compromised. Internally, numbers are stored in IEEE standard 64-bit format, in the dynamic range 10 to the power of -308 to 10 to the power of +308.

In the world of mathematics, there are numbers that cannot be represented by numerals. Dr. Logo outputs special words that represent these numbers: +INF represents the positive infinite number, -INF represents the negative infinite number, and NAN represents "not a number" for any mysterious entity that is not a number at all.

```
?1/0
+INF
?-1/0
-INF
?0/0
NAN
```

1.2.4 Variables

A variable is a "container" that holds a Logo object. A container in your kitchen labeled "Cookie Jar" might contain chocolate chip, peanut butter, some other kind of cookies, or no cookies at all. In Dr. Logo, a variable can have any name you give it, and contain any object -- word, number, or list. Here is the simplest way to create a variable:

```
?make "favorites [choc.chip peanut.but !
oatmeal.raisin cream.fill brownie pinwh!
eel shortbread snickerdoodle]
```

After the make expression is executed, the long list of cookie types has a name: "favorites.

Referring to an object by a variable name lets you write an expression that is independent of the object it manipulates. For example, the person responsible for keeping the cookie jar filled could enter:

```
?if emptyp :cookie.jar [make "cookie.ja!
r first shuffle :favorites]
```

which means, if the cookie jar is empty, put the first kind of cookie from the shuffled list of favorites into the cookie jar.

Because Logo programmers use variables frequently, they use several terms to describe the relationship between a variable's name and its contents. For example, if the make expression puts snickerdoodle in cookie.jar, a Logo programmer might describe the relationship in any or all of these ways:

- cookie.jar is a variable, snickerdoodle is its value
- cookie.jar is the name of snickerdoodle
- cookie.jar contains snickerdoodle
- cookie.jar is bound to snickerdoodle

- snickerdoodle is the "thing" of cookie.jar
- snickerdoodle is the contents of cookie.jar
- snickerdoodle is the value of cookie.jar

There are two ways to reference the contents of a variable:

```
?thing "cookie.jar
snickerdoodle
?:cookie.jar
snickerdoodle
```

A colon (":") before a variable name makes Dr. Logo reference the contents of the variable, instead of treating the variable name as a word.

No matter how you think of variables, remember that you can use a variable in an expression without being concerned about the variable's actual value. Variables help you write expressions that are independent of the data they manipulate. Section 2, "Working with Procedures", tells how to use variables within procedures.

1.3 Lines and expressions

You can put as many expressions on a line as you wish, but a line at toplevel cannot contain more than 132 characters.

In general, Dr. Logo evaluates expressions on a line from left to right. It treats everything, including other expressions, to the right of a primitive name or identifier as input to that primitive if the primitive requires an input.

```
?random 10 > 5
random doesn't like TRUE as input
?5 > random 10
FALSE
```

However, Dr. Logo does not evaluate arithmetic expressions in strict left-to-right order. It evaluates / and * expressions first, from left to right, then goes back and evaluates + or - expressions.

```
?2 * 3 + 7 / 5
7.4
```

To make Dr. Logo evaluate expressions in a different order, you can group expressions in parentheses ["(" and ")"]. Dr. Logo evaluates the expression in the innermost parentheses first. The order in which Dr. Logo evaluates your expression can make a big difference in the output!

```
?(random 10) > 5
FALSE
?2*(3+7)/5
4
```

EOF

DRLRM2.WS4 (= "Dr. Logo Reference Manual", Section 2)

(Retyped by Emmanuel ROCHE.)

Section 2: Working with procedures

This section discusses several aspects of working with Dr. Logo procedures. It tells how to construct a procedure and give it multiple inputs. It also describes how Dr. Logo keeps track of executing procedures.

2.1 Constructing procedures

The following sections describe how to put a procedure together: how to define it, how to make it readable, and how to use a variable to pass information between procedures and make a procedure require an input.

2.1.1 Naming and defining procedures

To define a procedure is to teach Dr. Logo a new verb, that is to say, to tell Dr. Logo how to do a new thing. You teach Dr. Logo a new verb by describing the new activity with primitives and other words Dr. Logo already know or will know before you execute the procedure. For example, Dr. Logo knows primitives that make the turtle go forward, repeat, and turn right. Using these primitives, you can describe how to draw a square, and use that description to define a new procedure.

The simplest way to define a new procedure is to begin a line with the special word "to", which makes Dr. Logo remember the next word you type as a new procedure name. Your procedure name should tell what your procedure does, and can start with any character, except a numeral. Dr. Logo will complain if you try to use the name of a primitive as a procedure name, unless you have set the system variable REDEFP to TRUE. When you start a line with "to", Dr. Logo gives you a new prompt character, >, to tell you it will not immediately execute the instructions that you enter to define the new procedure.

```
?to square
>repeat 4 [forward 60 right 90]
>end
square defined
?
```

The special word "end" tells Dr. Logo that you have finished defining your procedure, and returns you to the ? prompt. You must enter "end" by itself as the last line of a procedure. Now that "square" is defined, you can use square as a procedure name, as follows. Also see Colorplate 23 at the beginning of Section 6, "References to Primitives".

?square

Logo programmers use terms similar to those they use for variables to describe the relationship of a procedure and its definition. Here is how these terms apply to the procedure just defined:

- "to square" is the title line of square
- square is the name of [repeat 4 [forward 60 right 90]]
- [repeat 4 [forward 60 right 90]] is the definition of square
- [repeat 4 [forward 60 right 90]] is the body of square

The body of a procedure is a special kind of list: a list of expressions. This list can contain as many expressions as you want, or it can be an empty list. Because Dr. Logo treats the definition of a procedure as a list, you can write a procedure that first combines expressions into a list, and then defines this list as the body of a new procedure. The description of the "define" primitive in Section 6, "References to Primitives", tells how one procedure can define another.

You can use the Dr. Logo screen editor to modify your procedures' definitions. When you use the editor, you can change the body of a procedure, add a new procedure, or change the title line of an existing procedure. When you exit the editor, the new or changed procedures are defined, replacing any previous definitions for those names. The descriptions of the ed, edall, and edps primitives in Section 6 tells how to load procedures into the Dr. Logo editor. Section 3, "Editing Commands", tells how to use the editor.

2.1.2 Writing readable procedures

All programmers want to make their procedures as readable as possible. A readable procedure is a distinct advantage when you try to share your work with friends, or try to use a procedure several months after writing it.

Two traditional tools that make a program readable are short procedures and long names for procedures and variables. In general, short procedures are more readable than long ones. Short procedures are easier to understand, test, and combine with other procedures. Long procedure and variable names can help describe the purpose and function of procedures, or the kind of data represented by variables. However, long names take up valuable memory space, so you might have to shorten your names if you write a large program that needs all your workspace.

Dr. Logo gives you special help in writing readable procedures. For example, Dr. Logo lets you break long expressions into two or more lines. Generally, a line in a procedure is a single expression: a procedure name and its inputs. However, some procedures need complex inputs, such as lists of instructions or predicate expressions. This kind of line can grow so long that it becomes difficult to read. For example, an "if" command with a long predicate expression and two instructions lists can easily exceed the width of your display. When this happens, Dr. Logo prints an exclamation point ("!") and displays the remainder of the command on the next line.

```
?to check.for.favorites
>if memberp :cookie.jar :favorites (pr !
[edible cookies available] (pr [forget !
it!])
>end
check.for.favorites defined
```

You can clean up your procedure's appearance by breaking the long expression into several lines. If you press the Enter key and begin the next line with a space or a tab, Dr. Logo treats the new line as a continuation of the expression on the previous line.

```
?to check.for.favorites
>if memberp :cookie.jar :favorites
  [pr [edible cookies available]]
  [pr [forget it!]]
>end
check.for.favorites defined
```

Dr. Logo also lets you put comments in your procedures. A comment is text that Dr. Logo ignores and does not try to evaluate or execute. You can use comments to describe the function or purpose of a procedure or expression. You can start a comment at the beginning of a line or after the last expression on the line, but a comment must be the last object on the line. Start a comment with a semicolon (";"), as shown in the following example. Also see Colorplate 3.

```
?to triangle  ; Draw an equilateral tri!
angle
>repeat 3 [forward 20 right 120]
>end
triangle defined
?to flag
>fd 40
>triangle
>back 40  ; Return to original position
>end
flag defined
```

Comments take up space, but if space becomes a problem, you can use the "noformat" primitive to remove comments from your workspace.

2.1.3 Using variables in procedures

Variables have special capabilities within a procedure. You can use variables to define inputs to a procedure, and to pass information between procedures.

In general, it is not good programming practice to bury constants in your procedures. In the following procedure, 40 is a constant:

```
?to flag
>forward 40
```

```
>triangle
>back 40
>end
flag defined
```

Constants do not mean much to someone who reads your procedure later. And because they might appear on more than one line, constants are difficult to change systematically. Using variables instead of constants simplifies your procedures. A variable name can tell a person reading your procedure something about the data object. And to change the data, you need change only the make expression.

```
?to flag
>make "pole 40
>forward :pole
>triangle
>back :pole
>end
flag defined
```

To simplify your procedure even further, you can use a variable on a procedure's title line to define an input.

```
?to flag :pole
>forward :pole
>triangle
>back :pole
>end
flag defined
```

A variable on the title line makes your procedure require an input. When your procedure is called, Dr. Logo defines a variable that has the name given on the title line. The variable's value is the input object. If no object is input, Dr. Logo complains.

```
?flag
Not enough inputs to flag
?flag 80
?flag 40
```

When you give your procedure a variable name as input, Dr. Logo gives the contents of the variable to your procedure.

```
?make "big 80
?make "small 40
?flag :big
?flag :small
```

The next few paragraphs tell how Dr. Logo keeps track of variable definitions. At toplevel, Dr. Logo assigns values to variable names in your workspace. You define variables with make and name expressions directly to the interpreter's ? prompt. These are called "global variables".

Dr. Logo keeps track of variables defined by each procedure as it is

executing. These are the variables your procedure defines in the title line and with make expressions. Different procedures can have variables that have the same name. Logo programmers sometimes say that variables defined by a procedure are "bound" to their values by that procedure. See the following example.

```
?make "cookie.jar "snickerdoodle
?to look.in :cookie.jar
>print :cookie.jar
>end
look.in defined
?:cookie.jar
snickerdoodle
?look.in "brownie
brownie
?:cookie.jar
snickerdoodle
```

You can use a variable in your procedure that is not defined by your procedure. At execution time, if the procedure has been called by another procedure, Dr. Logo looks to see if the calling procedure defined the variable. If it finds a definition, it does not look any further. If it does not find a definition, it searches up the levels of calling procedures to toplevel before it complains that the variable is undefined.

Because Dr. Logo searches variable bindings this way, you can use variables to pass information between procedures. Usually, the exchange happens this way: the called procedure accesses a variable defined by the calling procedure. It changes the definition of the variable, then returns control to the calling procedure. The calling procedure can then use the information the called procedure stored in the changed variable.

The "local" primitive can hide a variable from Dr. Logo's search. If calling and called procedures are using variables of the same name, as is unavoidably the case when a procedure calls itself, a local expression in the called procedure prevents it from altering the calling procedure's value of the variable. An input to a procedure is always local to the procedure.

```
?to peek.in :cookie.jar
>(print "cookie.jar "contains :cookie.j!
ar
>look.again
>end
peek.in defined
?to look.again
>; cookie.jar not defined here,
>; just used!
>(print [cookie.jar still contains] :co!
okie.jar
>end
look.again defined
?peek.in "sugar
cookie.jar contains sugar
cookie.jar still contains sugar
```

```
?to peer.in
>local "cookie.jar
>make "cookie.jar "choc.chip
>(print [cookie.jar contains] :cookie.j!
ar
>look.again
>end
peer.in defined
?ern "cookie.jar
?:cookie.jar
cookie.jar has no value
?peer.in
cookie.jar contains choc.chip
cookie.jar still contains choc.chip
?look.again
cookie.jar still contains            !
cookie.jar has no value in look.again: !
(print [cookie.jar still contains] :co !
okie.jar
```

2.2 Giving inputs to procedures

You can use one or more variables in the title line of a procedure to define inputs to the procedure. Because of this, procedures you define always have a fixed number of inputs.

Most primitives also require a fixed number of input objects. However, some primitives can accept a variable number of inputs when the expression is enclosed in parentheses ["(" and ")"]. Without parentheses, these primitives normally require two inputs. With parentheses, they accept more or fewer inputs than are normally required, as shown in the following example.

```
?(word "sum "mer "sun "shine)
summersunshine
?list : favorites
Not enough inputs to list
?(list :favorites)
[[choc.chip peanut.but oatmeal.raisin !
cream.fill brownie pinwheel shortbread!
snickerdoodle]]
```

Most procedures expect a certain kind of input. When you input something to a procedure, it is simply an object, and the variable name is its container. What the procedure does with the object determines what kind of object is required. In fact, a procedure executes normally until it reaches an expression that requires a different kind of input than was supplied. For example, if a procedure requires a number as input, it will not know what to do with a word, and Dr. Logo complains. See the following example and Colorplate 36.

```
?to pentagon :size
>pr [The five angles of a pentagon]
```

```
>pr [equal 360 / 5 or 72 degrees.]
>repeat 5 [fd :size rt 72]
>end
pentagon defined
?pentagon 40
The five angles of a pentagon
equal 360 / 5 or 72 degrees.
?pentagon "forty
The five angles of a pentagon
equal 360 / 5 or 72 degrees.
fd doesn't like forty as input in pent!
agon: repeat 5 [fd :size rt 72]
```

As you write procedures that take words and numbers apart and then put them back together, you might create a quoted number such as "123. In general, Dr. Logo accepts a quoted number anywhere an unquoted number is expected. It also accepts an unquoted number where a quoted word is expected. For example, a primitive that expects numbers as input accepts quoted numbers.

```
?sum "53 "42
95
```

A primitive that expects a word or list as input accepts unquoted numbers.

```
?first 9876
9
```

One exception is the "ascii" primitive. It requires a quoted character as input. "ascii" does not distinguish among numerals, letters, and symbols.

```
?ascii "2
50
?ascii 2
ascii doesn't like 2 as input
```

2.3 Classifying procedures

All procedures operate the same way. You use them by following a procedure name with inputs in an expression. However, Logo programmers sometimes find it convenient to classify procedures, that is to say, to group procedures that have something in common together and give the group a name. For example, the procedures that come with Dr. Logo and make up Dr. Logo are called "primitives".

Most procedures can be classified as either a command or an operation. A command initiates an action. For example, commands move the turtle, draw pictures, and display text. A command procedure generally ends with "end" or "stop". An operation returns an object: TRUE, FALSE, a word, number, or list. The last expression to be evaluated in an operation procedure is always an output expression. Of course, you can write a procedure that initiates an action before it outputs an object. However, most primitives are simple commands or simple operations. The descriptions in Section 6, "References to

Primitives", refer to primitives as operations or commands.

Note that the po primitives, which display things on the text screen, are commands, not operations. Commands such as pocall and popkg format lists with spaces and tabs for display on the screen, making the lists inappropriate for input to another procedure.

Logo programmers also use classification names to describe kinds of operations. Logical operations such as "and", "not", and "or" return either TRUE or FALSE. An expression that contains a logical operator and, therefore, evaluates to either TRUE or FALSE is called a "predicate expression". Because of this, other logical operation names end with p. For example,

```
?numberp "two
FALSE
?numberp 2
TRUE
```

Other logical operators you can use to form predicate expressions are equalp, emptyp, memberp, listp, and wordp. When you write a procedure that outputs TRUE or FALSE, give the procedure a name that ends with p to indicate that it is a logical operation.

Arithmetic operations output numbers. Most arithmetic expressions are formed normally, with the procedure name followed by its inputs.

```
?product 2 3
6
```

However, some arithmetic operations are defined by symbols: + - * / ^, instead of names. Some logical operations, called "relational operators", also use symbols: < > =, instead of procedure names. Procedures that use symbols instead of procedure names can be infix operators, which means you can put the symbol between the inputs in the expression. These symbols also work as prefix operators, where the primitive symbol precedes its inputs.

```
?2 * 3
6
?= 2 2
TRUE
```

2.4 Evaluating procedures

The Dr. Logo interpreter evaluates one line at a time as you type lines at your keyboard. Dr. Logo also evaluates your procedures one line at a time. When an expression within a procedure begins with the name of a different procedure, Dr. Logo must execute the called procedure before it can return to evaluate the next line of the calling procedure. It must also keep track of where it stopped executing the calling procedure, in case the called procedure calls yet another procedure.

To do this, Dr. Logo uses a stack. At toplevel, there is nothing on the stack.

During the execution of a procedure, there is one procedure on the stack until it calls another; then, there are two. If the second procedure calls a third, there are three on the stack, and so on. The number of procedures on the stack is sometimes called the level number. The debugging facilities "trace" and "watch" display the level number as procedures execute. Toplevel is level number 0.

Dr. Logo assigns part of memory to the stack. Dr. Logo also uses this part of memory to store the values of local variables. If a procedure calls itself and the level number becomes very large, or if the procedure defines a great many local variables, the stack space might fill up. When this occurs, Dr. Logo displays a message and stops executing the procedure.

When a procedure is never called by any other procedure, but does call on other procedures itself, Logo programmers classify it as a superprocedure and the procedures it calls as subprocedures. Dr. Logo's potl primitive displays the names of the superprocedures in your workspace. A pocall expression displays the subprocedures called by the input-named procedure. A poref expression displays the procedures that call the input-named procedure.

EOF

DRLRM3.WS4 (= "Dr. Logo Reference Manual", Section 3)

(Retyped by Emmanuel ROCHE.)

Section 3: Editing commands

To edit with Dr. Logo means to enter new text, or to change text you have already entered. Editing can be as simple as fixing a typing error in a line you have typed to the interpreter's ? prompt, or as complex as defining several long procedures at once in Dr. Logo's screen editor.

Dr. Logo gives you three ways to edit: line editing, procedure editing, and screen editing. You use line editing commands in both procedure and screen editing. Procedure editing is a simple extension of line editing, and screen editing builds on procedure editing. This section tells when, why, and how to use Dr. Logo's line editing, procedure editing, and screen editing commands.

You give the commands described in this section to Dr. Logo with control characters, not with expressions. To enter a control character, hold down the control key (marked Ctrl on your keyboard) and press the required letter key. Not all of Dr. Logo's control character commands edit text; some interrupt and terminate procedure execution. The last part of this section introduces Dr. Logo miscellaneous control character commands.

3.1 Line editing

You can use line editing commands to correct any text you are entering to Dr. Logo. When you write a program that asks the user to type something at the keyboard, the user can also use line editing commands to change his input. The following table summarizes Dr. Logo's line editing control characters.

Table 3-1. Line editing control characters

Format: Character
Effect

Ctrl-A
Moves the cursor to the beginning of the line.

Ctrl-B
Moves the cursor [B]ack one character; that is to say, it moves the cursor one position to the left.

Ctrl-D
[D]eletes the character indicated by the cursor.

Ctrl-E
Moves the cursor to the [E]nd of the line.

Ctrl-F
Moves the cursor [F]orward one character; that is to say, it moves the cursor one position to the right.

Ctrl-H
Deletes the character to the left of the cursor.

Ctrl-I
[I]nserts a tab (three spaces).

Ctrl-K
[K]ills the remaining line; that is to say, it deletes all characters right of the cursor to the end of the line. Deleted characters are stored in buffer.

Ctrl-Y
[Y]anks text from the buffer; that is to say, redisplays line most recently stored in the buffer by an Enter or Ctrl-K keystroke.

The following examples show how you can use these control characters when you are entering expressions to the Dr. Logo interpreter. In these examples, the underbar ("_") represents the cursor.

Once you have typed a line, you can use control characters to move the cursor left and right over the text. You can make corrections anywhere in the command line.

```
?repaet 36 [fd 8 lf 10]_       (repeat and lt mistyped)
?r_epaet 36 [fd 8 lf 10]        (Ctrl-A to beginning of line)
?repaet_ 36 [fd 8 lf 10]        (Ctrl-F to move cursor right)
?rept_ 36 [fd 8 lf 10]         (Ctrl-H deletes chars to the left)
?repeat_ 36 [fd 8 lf 10]        (ea corrects repeat)
?repeat 36 [fd 8 lf 10]_       (Ctrl-E to end of line)
?repeat 36 [fd 8 lf_ 10]        (Ctrl-B moves cursor left)
?repeat 36 [fd 8 l_ 10]         (Ctrl-D deletes cursor position)
?repeat 36 [fd 8 lt_ 10]        (t corrects lt)
```

You can press the Enter key to send your command to Dr. Logo no matter which character the cursor is indicating in the command line. When you press Enter to send a line to the interpreter, Dr. Logo stores the line in a buffer. You can recall the stored line with Ctrl-Y. This is handy if you want to execute the command again, execute it again with a minor modification, or if you made a typing error and pressed Enter before you corrected it.

```
?save figures
I don't know how to figures
?save figures_          (Ctrl-Y recalls line)
?save "f_igures          (Ctrl-B moves cursor left;
                    (" corrects line.)
?save "figures_         (Save file on default drive; Enter
                     then Ctrl-Y recalls line.)
?save "f_igures          (Ctrl-B moves cursor left)
?save "b:f_igures        (b: to make copy on disk b: Enter)
```

You can delete all or part of a line with a Ctrl-K command. If you have second thoughts, you can recall the deleted characters with Ctrl-Y.

```
?erns "figure.pack_     (erase names? maybe not...)
?e_rns "figure.pack      (Ctrl-A to beginning of line)
?_                  (Ctrl-K erases line)
?erns "figure.pack_     (Yes, erase names; Ctrl-Y recalls line)
```

You can also use Ctrl-K and Ctrl-Y to repeat a portion of a command line.

```
?(pr "I char 3 "N "Y
I o N Y
?(pr "I char 3 "N "Y    (Ctrl-Y recalls line)
?(pr "_I char 3 "N "Y    (Ctrl-B moves cursor left)
?(pr _               (Ctrl-K erases part of line)
?(pr "I char 3 "N "Y_   (Ctrl-Y recalls partial line)
?(pr "I char 3 "N "Y "  "I char 3 "N "Y "  "I char 3 "N "Y "
                   (Quoted blank spaces and Ctrl-Y
                    keystrokes extend command.)
I o N Y  I o N Y  I o N Y
```

3.2 Procedure editing

When you are interacting with the Dr. Logo interpreter, during a pause, or while watching a procedure's executon, you can use the special words "to" and "end" to enter and exit Dr. Logo's procedure editor. For example, when you type a line that begins with "to" to the interpreter's ? prompt, Dr. Logo enters the procedure editor.

```
?to circle :size
>repeat 36 [fd :size rt 10]
>end
circle defined
?
```

The procedure editor's prompt, >, tells you that Dr. Logo will not immediatelly evaluate the expressions you enter to define the new procedure. Within the procedure editor, you can use any of the line editing control characters commands to move the cursor, correct errors, delete characters and lines, and recall lines.

To exit the procedure editor, you must enter the special word "end" by itself as the last line of the procedure. After it reads an end line, Dr. Logo defines the procedure and returns you to the situation from which you entered the procedure editor.

3.3 Screen editing

To make changes to a defined procedure without reentering the complete

procedure definition to the procedure editor, you must use Dr. Logo's screen editor. Using the screen editor, you can move from line to line in a procedure and make changes in each line. You can also load any number of procedures into the screen editor and make changes to each one. In addition, you can use the screen editor to change the contents of variables.

The screen editor is smart; that is to say, it makes certain assumptions about your objectives for an editing session. For example, if you define a procedure in the screen editor and forget to enter "end", the screen editor adds an end line to your procedure. When the execution of one of your procedures produces an error message and you call the screen editor, it assumes that you want to edit that procedure, automatically loads that procedure into itself, and positions the cursor at the line in which the error occurred.

As with the procedure editor, you can enter the screen editor while interacting with the interpreter, during a pause, or while you are WATCHing the execution of a procedure. You cannot write a procedure that uses the screen editor.

Before you can use the screen editor to make changes to your procedures or variables, you must load the ones you want to change into the screen editor's buffer. The screen editor's buffer is its own private workspace. Within the buffer, you can make changes, add text, and delete text. However, the changes you make do not become a part of Dr. Logo's workspace until you exit the screen editor. The changes must become a part of Dr. Logo's workspace before you can save them on disk.

There are four primitives that load procedures and variables into the screen editor's buffer and enter the screen editor. All four start with "ed". Each one enters the screen editor; the only difference between these primitives is what they load into the screen editor's buffer. The descriptions of these primitives in Section 6, "References to Primitives", tell what input names these primitives require to load selective groups of procedures and variables into the screen editor's buffer. The following table describes these four primitives.

Table 3-2. Load primitives

Format: Primitive
Purpose

edit (ed)
Use edit, or its abbreviation ed, to load a procedure or a list of procedures into the screen editor's buffer. If the execution of a procedure has ended with an error message and you immediately enter edit without an input procedure name, the screen editor automatically loads the erroneous procedure into the buffer, and positions the cursor at the offending line.

edall
Use edall to load both variables and procedures into the screen editor's buffer. "edall" without an input name loads all procedures and variables in Dr. Logo's workspace into the screen editor's buffer.

edns

Use edns to load a group of variables, names into the screen editor's buffer. "edns" without an input name loads all variables in Dr. Logo's workspace into the screen editor's buffer.

edps
Use edps to load a group of procedures into the screen editor's buffer. "edps" without an input name loads all procedures in Dr. Logo's workspace into the screen editor's buffer.

When you use one of these primitives and enter the screen editor, it clears all normal text from the screen, and displays only the contents of its buffer. The screen editor does not display a prompt character, but the cursor indicates the location that an insertion or control character command will affect.

While in the screen editor, you can use all the line editing control characters described in Section 3.1, "Line Editing", to make corrections within lines. The table below summarizes the screen editing control characters you can use to move the cursor from line to line, to page through the buffer, and exit the screen editor.

Table 3-3. Screen editing control characters

Format: Character
Effect

Ctrl-C
Exits screen editor; updates Dr. Logo's workspace with definitions of all procedures and variables from screen editor's buffer.

Ctrl-G
Exits screen editor but does not update Dr. Logo's workspace. Any changes made during the screen editing session are discarded.

Ctrl-L
Readjusts display so that line currently indicated by the cursor is positioned at the center of the screen. If the cursor is less than 12 lines from the beginning of the buffer, the screen editor simply beeps when Ctrl-L is pressed.

Ctrl-N
Moves the cursor to the [N]ext line; the cursor moves down one line towards the end of the buffer.

Ctrl-O
[O]pens a new line. A Ctrl-O keystroke is equivalent to pressing Enter followed by Ctrl-B.

Ctrl-P
Moves cursor to the [P]revious line; the cursor moves up one line towards the beginning of the buffer.

Ctrl-V

Displays the next screen full of text in the screen editor's buffer, the next 24 lines towards the bottom of the buffer.

ESC-V
Displays the previous screen full of text in the screen editor's buffer, the previous 24 lines towards the beginning of the buffer.

ESC-<
Positions the cursor at the beginning of the screen editor's buffer.

ESC->
Positions the cursor at the end of the screen editor's buffer.

The following examples show how you can combine these screen editing control characters with line editing control characters for editing shortcuts. Several combinations of control character commands are convenient and worth remembering. Ctrl-E Ctrl-D deletes the Carriage Return between two lines and makes them into one line. Ctrl-A Ctrl-K deletes the line currently indicated by the cursor. You can combine Ctrl-O with Ctrl-Y and Ctrl-K to quickly move a line within the screen editor. The line moved below makes the countdown procedure print a 0 before it stops. The edit is performed with only eight control character commands.

```
?countdown 4
4
3
2
1
?ed "countdown
                (editor clear screen)
to countdown :number_
if :number = 0 [stop]
pr :number
countdown :number - 1
end
                (Ctrl-N Ctrl-A positions cursor
                 at line to be moved)
to countdown :number
i_f :number = 0 [stop]
pr :number
countdown :number - 1
end
                (Ctrl-K deletes line)
to countdown :number
_
pr :number
countdown :number - 1
end
                (Ctrl-N positions cursor at new location)
to countdown :number

pr :number
c_ountdown :number - 1
```

```
end
                (INS Ctrl-Y inserts line)
to countdown :number

pr :number
i_f :number = 0 [stop]
countdown :number - 1
end
                (Ctrl-C defines procedure.)
countdown defined
?countdown 4
4
3
2
1
0
?
```

The paging control character commands become useful when you have more than 24 lines of text in the screen editor's buffer. You can enter edall to load everything from Dr. Logo's workspace into the screen editor's buffer, then experiment with Ctrl-L, Ctrl-V, ESC-V, ESC-<, and ESC->.

To exit the screen editor normally, press Ctrl-C. In this case, Dr. Logo updates the definitions of procedures and variables in its workspace with the definitions from the screen editor's buffer. If you have modified the title line of a procedure, Dr. Logo defines a new procedure and does not change the original definition of the procedure currently in the workspace. If you omitted an end line at the end of a procedure, Dr. Logo adds one as it updates its workspace.

To discard the changes you have made during an editing session, exit the screen editor with Ctrl-G. This leaves Dr. Logo's workspace in the same condition you found it when you entered the screen editor.

When you exit the screen editor, Dr. Logo returns you to the same situation from which you entered. For example, if you entered the screen editor during a pause in a procedure, Dr. Logo returns to the appropriate pause prompt and waits for your next command.

In a certain situation, you can use a few screen editing commands outside the screen editor. This situation occurs when you are entering a long expression outside the screen editor and Dr. Logo prints a "!" before starting on the next line. In this case, you can use Ctrl-P and Ctrl-N to move up and down between lines. You can also use Ctrl-O to open a new line, but all characters right of the Ctrl-O keystroke are ignored; Dr. Logo treats the Ctrl-O as the end of the expression. However, the characters are not lost. You can recall them with Ctrl-Y.

3.4 Other control character commands

Dr. Logo recognizes several control character commands for actions other than

line editing. In fact, the only control characters that have no meaning for Dr. Logo are Ctrl-U, Ctrl-J, Ctrl-R, and Ctrl-X. The remaining control characters terminate and interrupt procedure execution and scrolling screen displays, and switch between a full graphics screen, a full text screen, and a split screen. Section 4, "Text and Graphic Screens", describes screen control character commands in detail. The following table summarizes the miscellaneous control character commands.

Table 3-4. Additional control character commands

Format: Character
Effect

Ctrl-G
Immediately terminates the currently executing procedure.

Ctrl-L
Displays a full graphics screen; devotes the monitor to turtle graphics.

Ctrl-M
Carriage Return; same as pressing the Enter key.

Ctrl-Q
Generates the [Q]uoting character ("\") that makes Dr. Logo treat a delimiter character as a literal character.

Ctrl-S
Displays a [S]plitscreen; divides the monitor between a partial graphic screen and a text window.

Ctrl-T
Displays a full [T]ext screen; devotes the monitor to text.

Ctrl-W
Interrupt the scrolling of a text display; [W]aits until the next keystroke to continue the display.

Ctrl-Z
Interrupts the currently executing procedure, displays pause prompt to allow interactive debugging. Enter "co" to continue execution of interrupted procedure.

EOF

DRLRM4.WS4 (= "Dr. Logo Reference Manual", Section 4)

(Retyped by Emmanuel ROCHE.)

Section 4: Text and graphic screens

One of Dr. Logo's most exciting capabilities on your IBM Personal Computer is to create graphic designs and display text in many beautiful colors. Dr. Logo shows you a graphics cursor called "the turtle", which dutifully draws designs according to your instructions. However, this section does not tell you how to control the turtle; it tells you how to control the physical properties of your display. It tells what combinations of text and graphics are possible, how to switch between text and graphic screens, and how to control the colors of drawings and text.

4.1 Monitors, screens, and windows

In memory, Dr. Logo maintains two screens: a text screen and a graphic screen. A text screen contains only text, with up to 80 characters in one line. The text screen never contains more than 25 lines. A graphic screen can contain drawings and text, but can have only 40 characters in one line. The graphic screen potentially extends beyond the range of your monitor. The portion of the graphic screen your monitor can display is called "the visual field".

Dr. Logo maintains these complete screens in memory, so that it can display combinations of parts of them on your monitor and restore the full display of either screen. You control what combination of text and graphics appears on your monitor with Dr. Logo primitives and control character commands. Your color monitor can display both text and graphic screens. A monochrome monitor can display only the text screen.

To let you see the text of your procedures, or your interaction with the interpreter, without devoting your entire monitor to screen, Dr. Logo can direct part of the text screen into a text window. When a window of text appears on the graphic screen, the combination of text and graphics is called "a splitscreen". When you enter "debug", Dr. Logo divides the monitor into two text windows, then directs text displayed by a procedure to the lower PROGRAM window, and text displayed by debugging facilities to the upper DEBUG window.

4.2 Displaying text and graphic screens

Dr. Logo supports both primitives and control character commands

that make your monitor display the text screen and the graphic screen in various combinations, as described in the following table.

Table 4-1. Displaying screens

Format: Control character
Primitive
Display

Ctrl-L
fullscreen
Displays a full graphic screen; devotes the monitor to graphics.

Ctrl-S
splitscreen
Displays a splitscreen; divides the monitor between a partial graphic screen and a text window.

Ctrl-T
textscreen
Displays a full text screen; devotes the monitor to text.

(no control character)
debug
Displays two text windows on the monitor.

(no control character)
nodebug
Closes text windows, returns monitor to a cleared full text screen.

The control character commands are the quickest way to switch displays while you are interacting with the interpreter. Use the fullscreen, splitscreen, and textscreen primitives when you want to switch displays within a procedure. You must always use debug and nodebug to control the two debugging text windows. "debug" disables Ctrl-S and splitscreen; you cannot display a splitscreen while you are using the debugging text windows. "nodebug" restores Ctrl-S and splitscreen.

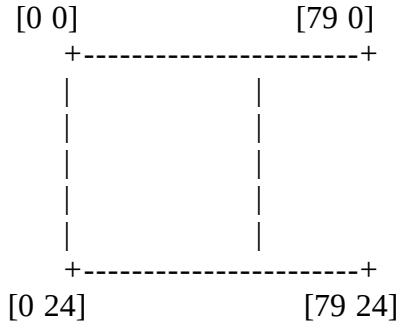
When you are using one color monitor, Dr. Logo automatically displays a splitscreen when you enter a command that moves the turtle. However, Dr. Logo can support both a color and a monochrome monitor. When you have two monitors, Dr. Logo automatically displays the text screen on the monochrome monitor, and the graphic screen on the color monitor.

Having two monitors changes the way some of the display control primitives work. When you have two monitors, Dr. Logo does not automatically display a splitscreen when you enter a graphics command because full graphic and text screens are always displayed. However, you can use Ctrl-S and splitscreen to open a text window on the graphic screen. To remove the splitscreen

text window and display the full text screen on the monochrome monitor, you can use fullscreen, Ctrl-L, or Ctrl-T. If you use textscreen at any time on a two monitor system, Dr. Logo displays the text screen on the color monitor.

4.3 Text screen

The text screen has positions for 2000 text characters: 25 lines of text with 80 characterrs in each line. The coordinate system shown below references these characters. A text coordinate list contains two numbers: the first is the row or character number, and the second is the line number.



You use this coordinate system with "setcursor" to position the cursor within the text screen.

Dr. Logo lets you control the foreground and background color of each character cell. "textbg" sets the background of the text screen to the color represented by the input number. There are eight possible colors, so normally you input a number in the range 0 to 7. However, textbg also controls whether or not the foreground characters blink. To display blinking characters, input a number between 8 and 15 to textbg, as shown in the following table.

Table 4-2. Background colors

Normal	Blinking	Background color
0	8	Black
1	9	Blue
2	10	Green
3	11	Cyan
4	12	Red
5	13	Magenta
6	14	Brown
7	15	Grey

"textfg" sets the foreground of the text screen to the color represented by the input number. The IBM Personal Computer supports 16 foreground colors, but if your color monitor does not support two levels of intensity, you can see only the first eight colors listed in the following table.

Table 4-3. Foreground colors

Input number	Foreground colors	Input number	Foreground colors
0	Black	8	Black
1	Blue	9	Bright blue
2	Green	10	Bright green
3	Cyan	11	Bright cyan
4	Red	12	Bright red
5	Magenta	13	Bright magenta
6	Brown	14	Yellow
7	Grey	15	White

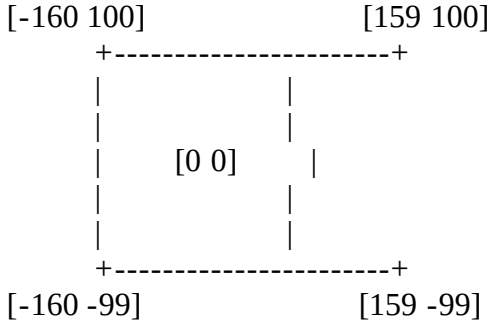
4.4 Graphic screen

Dr. Logo gives you complete control of the 60,000 individual dots that make up the visual field of your graphic screen; there are 300 dots horizontally and 200 dots vertically. Technically, these dots are called "pixels". A turtle step is equivalent to one dot. For example, when you enter "forward 1", the turtle moves forward one pixel. The dots make any diagonal lines the turtle draws seem jagged. Although Dr. Logo calculates angles and headings with great precision, the turtle rounds to the nearest degree before drawing.

4.4.1 Graphic coordinates and the visual field

You can reference each dot on the graphic screen individually by its coordinates. Using coordinates, you can quickly move the turtle to any location on the screen (setpos), or change any single dot to the turtle's current pencolor (dot).

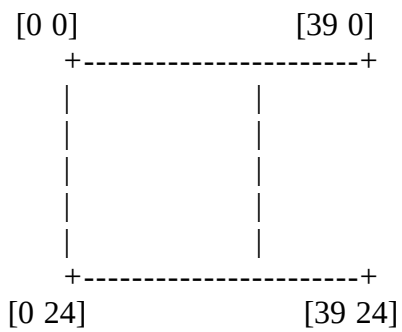
A graphic screen coordinate list contains two numbers: an x coordinate indicates the horizontal position, and a y coordinate indicates the vertical position. For example, the coordinates of the turtle's home position are [0 0], the center of the screen. To reference a dot in the visual graphic field, the x coordinate can be in the range -160 to +159, and the y coordinate can be in the range -99 to +100.



When you first start Dr. Logo, the visual field is just a window on a greater graphics plane. You can use numbers outside the visual coordinate ranges to reference positions outside the visual field; the turtle can draw a part of a design off the monitor and return. You can limit the turtle to the visual field with two primitives: fence and wrap. "fence" sets a boundary around the edge of the visual field. When the turtle encounters the boundary, Dr. Logo complains "Turtle out of bounds" and stops any executing procedure. "wrap" makes the turtle reappear on the opposite side of the monitor when it exceeds the boundary. Use "window" to remove the boundary.

4.4.2 Graphic text coordinates

On the graphic screen, there are positions of 1000 text characters: 25 lines of text with 40 characters per line. The coordinate system shown below references these character cells. In a text coordinate list, the first number is the row or character number, and the second number is the line number.



You can use this coordinate system to position the cursor within the splitscreen text window (setcursor), or to take input from a lightpen (lpen).

When you use "turtletext" to display text on the graphic screen, the first character of the input object appears in the first character cell to the right of the turtle's center line. The character might not appear directly under the turtle if the turtle is not on a character cell boundary.

4.4.3 Graphic colors

Although your color monitor can display many different colors, the graphic screen can contain only four colors at any one time. In Dr. Logo, the number you input to setbg specifies with set of four colors you want your monitor to display.

First, you have to select a background color. The IBM Personal Computer supports eight background colors in two levels of intensity, although your color monitor might not display different intensities. The numbers setbg accepts select background colors as described in the following table.

Table 4-4. Background color intensity

Low intensity	High intensity
0 16 32 48 Black	8 24 40 56 Black
1 17 33 49 Blue	9 25 41 57 Blue
2 18 34 50 Green	10 26 42 58 Green
3 19 35 51 Cyan	11 27 42 59 Cyan
4 20 36 52 Red	12 28 43 60 Red
5 21 37 53 Magenta	13 29 45 61 Magenta
6 22 38 54 Yellow	14 30 46 62 Yellow
7 23 40 55 White	15 31 47 63 White

Each of the four numbers for a background color specifies a different pen for the turtle to use. The turtle has four pens. Each pen has four unique colors of ink, one of which is the background color that the turtle uses for erasing. Use setpc to select the pen's ink color.

When the background color is in the range 0 to 15,

- setpc 1 selects dark green ink
- setpc 2 selects dark red ink
- setpc 3 selects dark yellow ink

When the background color is in the range 16 to 31,

- setpc 1 selects bright green ink
- setpc 2 selects bright red ink
- setpc 3 selects bright yellow ink

When the background color is in the range 32 to 47,

- setpc 1 selects dark cyan ink
- setpc 2 selects dark magenta ink
- setpc 3 selects dark grey ink

When the background color is in the range 48 to 63,

- setpc 1 selects bright cyan ink
- setpc 2 selects bright magenta ink
- setpc 3 selects bright white ink

For all background color numbers, setpc 0 selects background color (erasing) ink. When Dr. Logo first starts up, the background color is 1, and the pencolor number is 2.

4.4.4 Graphic text color

"textbg" controls the background color of the text on the graphic screen, both the splitscreen text window and characters typed by turtletext. The background color of graphic text can be

any one of the four colors of the turtle's current pen. For example, the following table shows how a textbg command affects the graphic text when the background of the graphic screen is set to 1 (blue) and the four pencolors are blue, green, red, and yellow.

Table 4-5. Background color of graphic text

textbg number	Graphic text background
0, 4, 8, 12	Blue
1, 5, 9, 13	Green
2, 6, 10, 14	Red
3, 7, 11, 15	Yellow

Unlike the text screen, when you input a number greater than seven to textbg, graphic screen text characters are not affected. Characters on the graphic screen cannot blink.

"textfg" controls the foreground color of graphic text. The characters in the splitscreen text window and the turtletext characters can be one of the four colors of the turtle's current pen. For example, the following table shows how a textfg command colors the graphic screen characters when the background of the graphic screen is set to 1 (blue) and the four pencolors are blue, green, red, and yellow.

Table 4-6. Foreground color of graphic text

textfg number	Graphic text color
0, 4, 8, 12	Blue
1, 5, 9, 13	Green
2, 6, 10, 14	Red
3, 7, 11, 15	Yellow

EOF

DRLRM5.WS4 (= "Dr. Logo Reference Manual", Section 5)

(Retyped by Emmanuel ROCHE.)

Section 5: Property lists, workspace, and disks

Storage is one of Dr. Logo's most important resources. There are two kinds of storage: temporary storage and permanent storage. When you enter procedure and variable definitions at your keyboard, they are temporarily stored in a part of your computer's memory called "the workspace". To record your procedures permanently, you must save them on disk.

To help organize your workspace and disk files, Dr. Logo lets you bundle procedures and variables into packages. Dr. Logo does this by adding system properties to an object's property list. This section tells how to use property lists and packages, how to organize your workspace for best performance, and how to create, copy, rename, and erase disk files.

5.1 Property lists

Any object in the workspace can have a property list. In fact, Dr. Logo uses property lists to create variables, procedures, and packages.

A property list is made up of property pairs. The first element of a property pair is the property name; the second element is its value. You can assign your own properties to an object. Property lists make it simple to store and retrieve information, as shown in the following example.

```
?pprop "Kathy "ext 42
?pprop "Meryle "ext 58
?pprop "Ellen "ext 66
?pps
Kathy's ext is 42
Meryle's ext is 58
Ellen's ext is 66
```

You can create whatever property pairs are appropriate to your application. Dr. Logo adds or removes property pairs when you use primitives identified in the following table. Do not use system property names for purposes other than those listed in the table.

Table 5-1. Property pairs

Property name	Primitive	Property value
.APV	make, name, ern, erall, erns	The value of a global variable

.BUR	bury, unbury	When TRUE, the package is buried
.CAT	catch, throw	Catch descriptor
.DEF	to, define, er, erall, erps	The definition of a procedure
.ENL	to, ed	End of a procedure line that is broken by a Carriage Return and spaces or tabs.
.FMT	to, ed	Beginning of a procedure line that is broken by a Carriage Return and spaces or tabs.
.FUN		Identifies an active function, a function in the process of being evaluated.
.PAK	package	The name of the package to which this object belongs.
.PAR		The parameters of an active function
.PAU		Pause
.PKG	package	When TRUE, object is a package name
.PRM		Memory location of the primitive
.REM		Remark or comment that follows a semicolon
.SPC		Number of spaces that follow a Carriage Return in a broken procedure line.

When you use plist to display property lists, you will always see an .APV pair in a variable name's property list, and a .DEF pair in a procedure's property lists. For example,

```
?make "flavor "chocolate
?plist "flavor
[.APV chocolate]
?to eat.cookie
>pr [yum yum!]
>end
eat.cookie defined
?plist "eat.cookie
[.DEF [[] [pr [yum yum!]]]]
```

However, you might never see some of the pairs listed above in a property list. This is because Dr. Logo puts properties such as .PAU, .FUN, and .PAR into a procedure's property list only during the procedure's execution.

5.2 Managing your workspace

Your workspace is the part of your computer's memory that Dr. Logo allocates for the temporary storage of your procedures and variables. This section tells how Dr. Logo measures your workspace, and how you can use packages to organize your workspace.

5.2.1 Measuring your workspace

Dr. Logo measures your workspace in nodes. A node is equivalent to five bytes, and can hold five characters. In general, the more workspace you have, the better your procedures will perform.

You can check how many free nodes there are in your workspace at any time with the "nodes" primitive. Try nodes immediately after you start Dr. Logo, to see the maximum size of your workspace. If you have enough memory in your IBM Personal Computer, you can have over 10,000 nodes at start-up. If you use "noprim" to remove the poprim information from the workspace, you add approximately 600 nodes to the maximum size of your workspace.

You tie up nodes in your workspace by entering procedures and other objects at the keyboard, or reading them in from disk with "load". Dr. Logo also adds to the contents of the workspace by making copies of local variables and recursive procedures during a procedure's execution.

5.2.2 Garbage collection

Dr. Logo does not automatically throw out these copies of local variables and recursively-called procedures after a procedure finishes execution. However, when there are fewer than 200 free nodes in your workspace, a part of Dr. Logo called "the garbage collector" sorts through the workspace and erases any copies of variables or procedures that are no longer needed. You might occasionally see an executing procedure hesitate for a few moments while the garbage collector does its work.

You can call the garbage collector with the "recycle" primitive. If you use recycle to clean up the workspace before initiating a time-critical procedure, you minimize the chance that the garbage collector will interrupt the execution of the procedure. Try using "nodes" before and after "recycle" to learn how many nodes are taken up by the temporary copies of variables and procedures.

The garbage collector uses the stack as it sorts objects in your workspace. Although Dr. Logo allocates stack space dynamically, if garbage collection occurs during the execution of a recursive procedure or other situation where the stack is heavily used, the garbage collector can run out of stack space. When this occurs, Dr. Logo displays a message and stops executing the procedure.

When workspace becomes critical because there are fewer than 400 free nodes,

Dr. Logo displays an exclamation point prompt ("!"). In this situation, enter "recycle" immediately. If, after "recycle", you still have fewer than 600 nodes, it is time to reorganize your procedures, save some of them on disk, and remove them from the workspace, to give the remaining procedures room to execute. You can use "load" within a procedure to restore saved procedures to the workspace when they are needed.

5.2.3 Packages

Packages help you organize your workspace. When you put related procedures and variables in a package, you can display, edit, save, and erase those procedures and variables as a group separate from other procedures and variables in the workspace.

Use "package" to create a package or add items to a package. "package" requires two inputs. The first is the name of the package. You can give "package" either a name or a list of names of items to be placed in the package as the second input. The list can contain a mixture of variable and procedure names. For example,

```
?package "cookies "flavor
?package "cookies [eat.cookies flavor]
```

"popkg" displays the name and contents of each package defined in the workspace, as shown in the following example.

```
?popkg
cookies
  "flavor
  eat.cookies
```

When you input a package name to one of the following primitives, it takes action only on the procedures and variables contained in the package.

```
edall   erall   glist   pops    save
edns    erns    poall   pots
edps    erps    pons    pps
```

If you do not specify a package name to these primitives, they act on all procedures and variables in the workspace, except for those in buried packages. A buried package is hidden from the primitives listed above. To bury a package, use "bury", as shown in the following example.

```
?bury "cookies
?popkg
cookies is buried
  "flavor
  eat.cookies
```

Dr. Logo creates and buries packages by adding system properties to property lists. It adds the .PAK property with the name of the package to the packaged procedure's or variable's property list. It adds the .PAK property with the

value TRUE to the package name's property list. When you bury a package, Dr. Logo adds the .BUR property with the value TRUE to the package name's property list. The primitives listed above that can take a package name as input check the property list of the package name for the .BUR property before taking action.

5.3 Drives, disks, and files

When you turn off your computer, all the procedures and variables in your workspace are lost. So, before you turn your computer off, you must save them on a more permanent medium, a disk. Dr. Logo stores information on disk in files, and gives each file the name you specify. The part of your computer that reads and writes file on a disk is called a "drive". This section gives some background information on drives, disks, and files.

Many of the primitives that copy and erase disks and files display a message that asks you to confirm that you do, indeed, want to copy or erase before the primitive proceeds. This is because it is sometimes too easy to erase or copy over important files by mistake. However, such messages make it inconvenient to use these primitives from within a procedure. The system variable NOACK controls whether or not these primitives display a message. Make NOACK TRUE to suppress the messages.

5.3.1 Drives

Your IBM Personal Computer has one or two drives. A drive can be either single- or double-sided, which means the drive can write on one or both sides of the disk. If your drive can write on both sides of the disk, you can store twice as much information on a disk.

When you first start Dr. Logo, it makes drive A: the default drive. This means that, until you tell Dr. Logo to do otherwise, it looks for information on the disk in drive A:. If you have a single-drive system, drive A: will always be your default drive. If you have more than one drive, you can tell Dr. Logo to change the default drive with a "setd" command. "defaultd" outputs the name of the default drive in uppercase.

```
?defaultd
A:
?setd "b:
?defaultd
B:
```

Before you can save anything on disk, you must put a formatted disk in a drive. You cannot save information on your Dr. Logo system disk, so if you have a one-drive system, you must remove the Dr. Logo system disk from the drive and insert a data disk. To tell Dr. Logo that you have inserted a new disk, enter "resetd". If you try to save information on a newly-inserted disk without entering "resetd", Dr. Logo will complain.

5.3.2 Disks

Dr. Logo cannot save information on a disk that is fresh from the box. The disk must be initialized or formatted to single- or double-sided. During the initialization process, Dr. Logo tests and prepares one or both surfaces of the disk for future load and save operations. Appendix E, "Getting Started", and the description of "initd" in Section 6, "References to Primitives", both tell how to initialize a disk.

Dr. Logo measures the space on your disk in bytes, not nodes. "spaced" outputs the number of free bytes on the disk in the specified drive. A single-sided disk can hold 150,000 bytes. A double-sided disk can hold 300,000 bytes.

5.3.3 Files

A file is a set of related information stored on disk. A Dr. Logo file contains objects such as procedures and variables with their property lists. You can create a Dr. Logo file with "save". Dr. Logo saves either everything in the workspace, or just the objects you specify with packages.

Dr. Logo gives the file the name you specify by writing the name in the disk's directory. The name you specify cannot contain more than eight characters. If you specify a name longer than eight characters, Dr. Logo uses the first eight characters as the name.

You can change a file's name with "changef", copy a file with "copyf", and erase a file with "erf". "getfs" outputs a list of the Dr. Logo file names on the disk in the default or specified drive. Like drive names, Dr. Logo outputs file names in uppercase.

"erf" and "getfs" can accept an ambiguous file name as input. An ambiguous file name can reference more than one file, because it contains a wildcard character and gives Dr. Logo a pattern to match. The wildcard character is the question mark ("?"), which must be the last character in the file name.

```
?getfs
[SHAPES PIGLATIN PLAID]
?getfs "p?
[PIGLATIN PLAID]
```

EOF

DRLRM6.WS4 (= "Dr. Logo Reference Manual", Section 6)

(Retyped by Emmanuel ROCHE.)

ROCHE> Since only the CP/M-86 version of "Dr. Logo for the ACT Apricot F1" computer has been found, I have retyped only the documentation of the primitives contained in this version, not the explanations of all the primitives of the original copy-protected Dr. Logo, which can format floppy disks, copy full disks, etc (like computers with BASIC in ROM, which have no separate Operating Systems).

Section 6: References to primitives

This section defines the Dr. Logo primitives. The description of each primitive is presented in the following form:

Action

Quickly tells what the primitive does.

Syntax

Shows the correct way to enter an expression using the primitive. If you can use an abbreviation for the primitive name, the abbreviation appears on the second line.

Do not enter a command that is identical to the syntax line! The syntax line usually contains symbols that Dr. Logo does not recognize. By using these symbols, the syntax line can represent the wide variety of expressions that you can enter using the primitive.

The syntax line uses two kinds of symbols to show you how to enter an expression. Typographical symbols tell you how many times you can enter an input. Input symbols tell what kind of object the primitive can accept as input. The typographical symbols are

ellipsis: (...)

Means that the primitive can accept a variable number of inputs; that is to say, you can use parentheses to make the primitive accept more or fewer inputs than are ordinarily required.

If the primitive name is preceded by a left parenthesis ["("], the primitive processes as many inputs as it finds before the closing right parenthesis [")"] without complaint. If no expression follows the parenthesized expression on the line, you do not need to type the closing parenthesis.

angle brackets: < >

Enclose optional inputs. (Inputs shown in angle brackets are not required.)

"or" bar: |

Separates alternative inputs. Enter one of the two inputs separated by |. For example, name | name_list means you can enter either a name or a list of names.

The following symbols represent the objects you can input to a primitive:

object

Means the primitive can accept any object as input: a word, a name, a number, a list, or an expression that outputs a word, a name, a number, or a list. You can also use a variable to represent any object input to a procedure.

name

Represents a special word that identifies either a procedure, a variable, or a "package" of procedures and variables. When a primitive can accept only a certain kind of name as input, one of the following symbols appears:

- varname a variable name
- procname a procedure name
- pkgname a package name
- d: a drive name
- fname a file name that can begin with an optional drive name to specify a drive other than the default
- prop a property name; the first member of a property pair in a property list

list

Means a list is required as input. A literal list is a series of Logo words, numbers, or lists enclosed in square brackets ("[" and "]"). You can construct a "not literal" list that contains the value of a variable by using the LIST or SENTENCE primitive. Some of the special kinds of lists are:

- instr_list contains instructions to be executed
- coord_list contains a pair of numbers that define a location on the screen
- name_list contains names of variables, procedures, and/or packages

numbers: n, _n

Represent input numbers. N represents any number. You can enter an expression that outputs a number anywhere N appears. _N is appended to a descriptive term when a special number is required. For example, DEGREES_N means the input number is interpreted as a number of degrees.

infix primitive inputs: a, b

Represent inputs to an infix primitive, where the primitive identifier is a symbol embedded between the inputs. A and B can represent both numbers and objects. In arithmetic expressions, A and B represent numbers, as in 1 + 2 and 12 / 6. In logical expressions, A and B can represent any object, as in [a b c] = [a b c], 3 > 1, or :son = "greg.

pred_exp

Represents a predicate expression, an expression that, when evaluated, outputs either TRUE or FALSE.

Explanation
===========

Describes the action or result of an expression in detail. Discusses optional inputs, punctuation, and how the primitive works with other primitives, if appropriate.

In these discussions, "you" means you, the Dr. Logo programmer. "Your user" refers to a person who runs your Dr. Logo procedure, and might type something at the keyboard to interact with your procedure.

Examples
========

Shows expressions and procedures that demonstrate the capabilities of the primitive. This section sometimes includes discussions of the example expressions and procedures.

Most of the examples show you exactly what to type to get the response shown in the text. However, some examples, such as those for erasing, must assume there is already something in the workspace to erase. A brief description of assumptions about the workspace precedes these examples. You might not be able to reproduce these examples exactly, but, by studying them, you will learn the

capabilities of the primitives they demonstrate.

The examples assume that you are using a graphic screen, which has a line length of 80 characters ("SETRES 1"). (For legacy reasons, Dr. Logo starts in SETRES 0, providing 320x200 pixels. If you want a VGA screen (640x480), read the explanations of the SETRES primitive.) When you have typed a 80-character line, Dr. Logo enters an exclamation point ("!") which indicates that your text continues on the next line. You do not type the exclamation point yourself.

```
 *         (see product)
 +         (see sum)
 .contents (see contents)
 .deposit  (see deposit)
 .examine  (see examine)
 .in       (see in)
 .out      (see out)
 .replace  (see replace)
 .reptail  (see reptail)
 .setseg   (see setseg)
 \  "Quoting character" = Ctrl-Q
```

There is a handful of commands which are not primitives. They can be classified in 3 groups:

1) Non-standard I/O devices: PADDLE, BUTTONP, LPEN, LPENP
2) Enter/leave the editor/interpreter: TO, END, BYE
3) System variables: ERRACT, REDEFP

abs

Action:
Outputs the absolute value of the input number.

Syntax:
abs n

Explanation:
ABS outputs the absolute value of the input number. You can use ABS to show the distance from the turtle's position to home excluding any + or - sign that indicates its location in the coordinate scale. For example, if you move the turtle back 50 steps from home, its position if [0 -50]. However, its distance from home is the absolute value of -50 (sometimes shown as |-50|), or 50 turtle steps.

Examples:
```
?cs
?back 50
?pos
[0 -50]
?abs last pos
50
```

allopen

Action:
Outputs a list of all data files currently open.

Syntax:
allopen

Explanation:
Outputs a list of all data files currently open.

(ROCHE> There is a system message, saying that "Only 4 files can be open".)

Examples:
```
?allopen
```

and

Action:
Outputs TRUE if all input predicate expressions output TRUE.

Syntax:

```
  and pred_exp pred_exp (...)

  Explanation:
  AND  outputs TRUE if all input predicate expressions output  TRUE.  Otherwise,
  AND outputs FALSE.

  Without  punctuation,  AND  requires and accepts two input  objects.  AND  can
  accept more or fewer inputs when you enclose the AND expression in parentheses
  ["(" and ")"]. If no other expressions follow the AND expression on the  line,
  you do not need to type the closing right parenthesis [")"].

  You can use an AND expression to test different conditions, or build your  own
  predicate procedure.

  Examples:
  ?and "TRUE "TRUE
  TRUE

  ?and "TRUE "FALSE
  FALSE

  ?and "FALSE "FALSE
  FALSE

  ?and (3<4) (7>3)
  TRUE

  ?and (3=4) (7>3)
  FALSE

  ?(and (3<4) (7>3) (9=9) (6<5))
  FALSE

  ?to tub.right? :temperature
  >if and (:temperature > 88) (:temperature < 102)
  >   [print [Just Right!]]
  >   [print [Not Right.]]
  >end
  tub.right? defined
  ?tub.right? 90
  Just Right!

  ?to decimalp :object
  >output and (numberp :object) (pointp :object)
  >end
  decimalp defined
  ?to pointp :object
  >if emptyp :object
  >   [output "FALSE]
  >if (first :object) = ".
  >   [output "TRUE]
  >output pointp butfirst :object
  >end
  pointp defined
  ?decimalp 1995
  FALSE
  ?decimalp 19.95
  TRUE
  ?decimalp [nineteen.ninety-five]
  FALSE


  arctan
  ------

  Action:
  Outputs the arc tangent of the input number.

  Syntax:
  arctan n

  Explanation:
  ARCTAN outputs in degrees the angle whose tangent is the input number.

  Examples:
  ?arctan 0
  0

  ?arctan 1
  45
```

```
?arctan 10
84.2894068625003

?arctan 100
89.4270613023165

?to plot.arctan
>make "val -pi
>make "inc pi / 37.5
>make "x -150
>setx 150
>setx :x
>plot.a :val
>end
plot.arctan defined
?to plot.a :val
>if :x > 150
>   [stop]
>forward arctan :val
>sety 0
>setx :x + 4
>make "x :x + 4
>make "val :val + :inc
>plot.a :val
>end
plot.a defined
?plot.arctan
```

ascii

Action:
Outputs the ASCII value of the first character in the input word.

Syntax:
ascii word

Explanation:
ASCII outputs an integer between 0 and 255 that is the American Standard representation for the first character in the input word. The input word must contain at least one character. The first character can be a letter, number, or special character.

The American Standard Code for Information Interchange (ASCII) is a standard code for representing numbers, letters, and symbols. The IBM Personal Computer has many unique characters that are also represented by ASCII codes. The "Dr. Logo Command Summary" contains a list of characters and their ASCII values.

Examples:

```
?ascii "g
103

?ascii "good
103

?ascii "2
50

?to encode :word
>if emptyp :word
>   [output "]
>output word secret first :word encode butfirst :word
>end
encode defined
?to secret :character
>make "secret.code 5 + ascii :character
>if :secret.code > ascii "z
>   [make "secret.code :secret.code - 26]
>output char :secret.code
>end
secret defined
?to decode :word
>if emptyp :word
>   [output "]
>output word crack first :word decode butfirst :word
>end
decode defined
?to crack :character
>make "cracked.code (ascii :character= -5
>if :cracked.code < ascii "a
```

```
>   [make "cracked.code :cracked.code + 26]
>output char :cracked.code
>end
crack defined
?make "password encode "plastics
?:password
uqfxynhx
?decode :password
plastics
```

back bk

Action:
Moves the turtle the input number of steps in the opposite direction of its heading.

Syntax:
back distance_n
bk distance_n

Explanation:
BACK moves the turtle the specified number of steps in the opposite direction of its current heading. The turtle's heading and pen do not change. If the turtle's pen is down, the turtle leaves a trace of its path. On the IBM Personal Computer, a turtle step is equivalent to one dot (pixel).

BACK can help you write a procedure that leaves the turtle in the same position when the procedure ends as it was when the procedure started. Leaving the turtle in the same position makes it easy to call the procedure from another procedure.

Examples:

```
?cs
?back 50
```

```
?cs
?to flag
>forward 50
>repeat 3
>  [right 120 forward 25]
>back 50
>end
flag defined
?to wheel
>repeat 12
>  [flag left 30]
>end
wheel defined
?flag
?wheel
```

bury

Action:
Hides the specified package from subsequent workspace management commands.

Syntax:
bury pkgname | pkgname_list

Explanation:
BURY hides the specified package or packages from workspace management commands. BURY works by setting the bury property (.BUR) in the package's property list to TRUE. The following primitives check property lists before taking action, and ignore any buried procedures or variables:

```
        edall   erall   glist   pops    save
        edns    erns    poall   pots
        edps    erps    pons    pps
```

All of these primitives optionally accept a package name as input. If no package name is specified, these commands address the entire contents of the workspace, except for buried packages. All other procedures access buried procedures and variables normally. For example, a procedure that receives a buried variable name as input accesses the buried variable normally. POTL and POPKG display the names of buried procedures. PROCLIST includes all defined procedure names, buried or unburied, in the list it outputs.

Examples:
These examples assume you have the following items in your workspace: two packages named FIGURES and TITLES, two variables named BIG and SMALL, and four procedures named PRAUTHOR, PRDATE, SQUARE, and TRIANGLE.

```
?popkg
figures
  "big (VAL)
  "small (VAL)
  to square
  to triangle
titles
  to prauthor
  to prdate
?bury "titles
?popkg
figures
  "big (VAL)
  "small (VAL)
  to square
  to triangle
titles is buried
  to prauthor
  to prdate
?pots
to square
to triangle
```

butfirst bf

Action:
Outputs all but the first element in the input object.

Syntax:
butfirst object
bf object

Explanation:
BUTFIRST outputs all but the first element of the input object. If the input object is a list, BUTFIRST outputs a list containing every element of the input list, except the first element. If the input object is a word, BUTFIRST outputs a word containing all but the first character of the input word. If the input object is an empty word or empty list, BUTFIRST returns an error.

Examples:

```
?butfirst "abalone
balone

?butfirst [semi sweet chocolate]
[sweet chocolate]

?butfirst [[chocolate chip] [walnut date] [oatmeal raisin]]
[[walnut date] [oatmeal raisin]]

?butfirst "y

?butfirst [brownie]
[]

?to vanish :object
>if emptyp :object
>   [stop]
>print :object
>vanish butfirst :object
>end
vanish defined
?vanish "abracadabra
abracadabra
bracadabra
racadabra
acadabra
cadabra
adabra
dabra
abra
bra
ra
a
```

butlast bl

Action:
Outputs all but the last element in the input object.

Syntax:
butlast object
bl object

Explanation:
BUTLAST outputs all but the last element of the input object. If the input object is a list, BUTLAST outputs a list containing every element of the input list, except the last element. If the input object is a word, BUTLAST outputs a word containing all but the last character of the input word. If the input object is an empty word or empty list, BUTLAST returns an error.

Examples:
```
?butlast "drawn
draw

?butlast [fudge walnut]
[fudge]

?butlast "y

?butlast [snickerdoodle]
[]

?to vanish :object
>if emptyp :object
>   [stop]
>print :object
>vanish butlast :object
>end
vanish defined
?vanish "turkey
turkey
turke
turk
tur
tu
t
```

buttonp (= BUTTON Predicate) (Not a primitive)

Action:
Outputs TRUE if the button on the specified paddle (joystick) is down.

Syntax:
buttonp paddle_n

Explanation:
BUTTONP outputs TRUE if the button on the specified paddle or joystick is down. Dr. Logo can accept input from two paddle. Each paddle can have two buttons. You use BUTTONP to determine whether or not a button in pressed. If you do not have a paddle or joystick, BUTTONP always output FALSE.

BUTTONP requires an input number to identify one of the four paddle buttons. Numbers in the range 0 to 3 identify the paddle buttons as follows:

0 identifies button 1, paddle 1
1 identifies button 2, paddle 1

2 identifies button 1, paddle 2
3 identifies button 2, paddle 2

Examples:
DRAW allows the user to guide the turtle with the joystick. The two BUTTONP commands allow the user to stop drawing or erase the drawing by pressing the paddle buttons. DRAW is more fully described under PADDLE.

```
?to draw
>repeat 10000
>  [make "xin paddle 0
>   make "yin paddle 1
>   make "xin int ((:xin * (300 / 190)) - 150)
>   make "yin int ((:yin * (-200 / 144)) + 90)
```

```
>   setheading towards list :xin :yin
>   forward [amount * 0.1]
>   if buttonp 0 [stop]
>   if buttonp 1 [clean]
>end
draw defined
?to amount
>output int sqrt
>  ((abs :xin) * (abs :xin) +
>  ((abs :yin) * (abs :yin)))
>end
amount defined
?draw
```

bye   (Not a primitive)

Action:
Exits Dr. Logo and returns to the operating system.

Syntax:
bye

Explanation:
Exits current session of Dr. Logo and returns to the operating system. You can enter BYE to Dr. Logo's ? or ! prompt; BYE is not valid when you are in the editor or while you are executing a procedure. When you enter BYE, any procedures or variables you have not saved on disk are lost.

Examples:
?bye

catch

Action:
Traps errors and special conditions that occur during the execution of the input instruction list.

Syntax:
catch name instr_list

Explanation:
CATCH works with the THROW primitive to let your procedure handle special conditions. For example, by using CATCH and THROW, your procedure can display a special message if your user types something incorrectly. CATCH and THROW can also intercept an error that would normally make Dr. Logo display a message on the screen.

CATCH and THROW each require a name as input. To pair a CATCH expression with a THROW expression, you must give the CATCH and THROW expressions the same input name.

When a CATCH command is executed, Dr. Logo simply executes the input instruction list. Execution proceeds normally until a THROW expression, usually in a called procedure, identifies a special condition. Then, Dr. Logo returns to the procedure that contains the CATCH command identified by the THROWn name. Dr. Logo then executes the line that follows the CATCH command.

There are two special names you can input to catch: TRUE and ERROR. TRUE matches any THROW name, so CATCH "TRUE catches any THROW. Dr. Logo automatically executes a THROW "ERROR command when an error occurs. Therefore, a CATCH "ERROR expression catches any error that occurs. Without CATCH "ERROR, an error makes Dr. Logo print a message on the screen, terminate your procedure's execution, and return to toplevel (the ? prompt). The description of the ERROR primitive tells how to find out what the error was.

Examples:
The COIL procedure asks the user to enter increasingly larger numbers as the turtle draws a coil on the screen. If the user types a number that is not bigger than the last one entered, COIL reminds the user what to type and continues working.

```
?to coil
>print [Enter a small number.]
>make "previous 0
>forward grow.number
>right 30
>trap
```

```
>end
coil defined
?to grow.number
>make "growth first readlist
>if :growth < :previous
>   [throw "not.bigger]
>make "previous :growth
>output :growth
>end
grow.number defined
?to trap
>catch "not.bigger [draw.coil]
>(print [Enter a number bigger than] :previous)
>trap
>end
trap defined
?to draw.coil
>print [Enter a bigger number.]
>forward grow.number
>right 30
>draw.coil
>end
draw.coil defined
?coil
```

The THROW "NOTBIGGER instruction in the GROW.NUMBER procedure always returns Dr. Logo to the TRAP procedure. If a STOP instruction had been used instead of THROW, Dr. Logo would return to the procedure that called GROW.NUMBER, which might be either COIL or DRAWCOIL.

The following procedures allow the user to type commands just as if typing to the Dr. Logo interpreter. However, if the user enters a command incorrectly, the MY.MESSAGE procedure traps the normal Dr. Logo error message, and prints a custom message.

```
?to my.message
>catch "error [interpret]
>(print "Oops! first butfirst error [!!!])
>print [What do you want to do about that?]
>run readlist
>my.message
>end
my.message defined
?to interpret
>print [What next, boss?]
>run readlist
>interpret
>end
interpret defined
?interpret
Hello!
```

changef (= CHANGE Filename)

Action:
Changes the name of a file in the disk directory.

Syntax:
changef < d: > new_fname old_fname

Explanation:
CHANGEF changes the name of a file in a disk directory. Enter the name you want to give the file, followed by the file's current name. You can put a disk specifier in front of the old name if the file is not on the default disk drive.

Examples:
The following examples assume you have three Dr. Logo files on the disk in the default drive: PIGLATIN, FLY, and SHAPES.

```
?dir
[PIGLATIN.LOG FLY.LOG SHAPES.LOG]
?changef "pigl "piglatin
?dir
[PIGL.LOG FLY.LOG SHAPES.LOG]
```

(ROCHE> I found the following procedure useful...)

```
to ren :new fname :old fname
```

```
changef :new_fname :old_fname
end
```

char

```
Action:
Outputs the character whose ASCII value is the input number.

Syntax:
char n

Explanation:
CHAR outputs the character whose ASCII value is the input number. CHAR
requires an integer between 0 and 255 as input.

The American Standard Code for Information Interchange (ASCII) is a standard
code for representing numbers, letters, and symbols. The IBM Personal Computer
has many unique characters that are also represented by ASCII codes. The "Dr.
Logo Command Summary" contains a list of characters and their ASCII codes.

Examples:
?char 103
g

?char 50
2

?repeat 20 [(type random 10 char 9)]
3   0   6   1   3   5   1   8   0   1  8   6   7   1   1   9   3   8   4   5

?to encode :word
>if emptyp :word
>   [output "]
>output word secret first :word encode butfirst :word
>end
encode defined
?to secret :character
>make "secret.code (ascii :character) + 5
>if :secret.code > (ascii "z)
>   [make "secret.code :secret.code - 26]
>output char :secret.code
>end
secret defined
?to decode :word
>if emptyp :word
>   [output "]
>output word crack first :word decode butfirst :word
>end
decode defined
?to crack :character
>make "cracked.code (ascii :character) - 5
>if :cracked.code < (ascii "a)
>   [make "cracked.code :cracked.code + 26]
>output char :cracked.code
>end
crack defined
?make "password encode "elephant
?:password
jqjumfsy
?decode :password
elephant
```

clean

```
Action:
Erases the graphic screen without affecting the turtle.

Syntax:
clean

Explanation:
CLEAN erases everything the turtle has drawn on the graphic screen, but leaves
the turtle in its current heading and location.

Examples:
?to tri.spi :side
```

```
>if :side > 80
>   [stop]
>forward :side right 120
>tri.spi (:side + 3)
>end
tri.spi defined
?tri.spi 5
?clean
```

clearscreen cs

Action:
Erases the graphics screen and puts the turtle in the home position.

Syntax:
clearscreen
cs

Explanation:
CLEARSCREEN erases everything the turtle has drawn on the graphic screen, and returns the turtle "home" to location [0 0] heading 0 (North).

Examples:

```
?to tri.spi :side
>if :side > 80
>   [stop]
>forward :side right 120
>tri.spi (:side + 3)
>end
tri.spi defined
?tri.spi 5
?clearscreen
```

cleartext ct

Action:
Erases all text in the window that currently contains the cursor, then positions the cursor in the upper left corner of the window.

Syntax:
cleartext
ct

Explanation:
CLEARTEXT erases all text displayed in the window currently containing the cursor, then positions the cursor at the upper left corner of the window. For example, if your entire screen is devoted to text, CLEARTEXT erases the entire screen, and places the cursor in the upper left corner.

Examples:
This example assumes you have the following items in your worskpace: two packages named FIGURES and TITLES, two variables named BIG and SMALL, and four procedures named PRAUTHOR, PRDATE, SQUARE, and TRIANGLE.

```
?popkg
figures
  "big (VAL)
  "small (VAL)
  to square
  to triangle
titles
  to prauthor
  to prdate
?cleartext
```

close

Action:
Closes the named data file.

Syntax:
close fname

Explanation:
Closes the named data file.

Examples:
?close "Ketchum

closeall

Action:
Closes all the data files currently open.

Syntax:
closeall

Explanation:
Closes all the data files currently open.

(ROCHE> There is a system message, saying that "Only 4 files can be open".)

Examples:
?closeall

co
--

Action:
Ends a pause that is caused by a PAUSE expression or a Ctrl-Z keystroke.

Syntax:
co

Explanation:
CO continues the execution of a procedure interrupted by a PAUSE. During a pause, you can interact with the interpreter to debug your procedure. Enter CO when you have finished with the interpreter and want to continue the execution of the procedure.

There are three ways to cause a pause during the execution of a procedure:

1) the execution of a PAUSE expression within the procedure
2) a Ctrl-Z keystroke
3) any error, if the system variable ERRACT is TRUE.

When a PAUSE interrupts the execution of a procedure, Dr. Logo displays a "Pausing..." message and shows the name of the interrupted procedure before the interpreter's ? prompt. No matter which way you begin a pause, you must use CO to end it and continue your procedure's execution.

Examples:
```
?to box.spi :side
>if and (:side > 80) (:side < 90)
>   [pause]
>forward :side right 90
>box.spi (:side + 5)
>end
box.spi defined
?box.spi 5
Ctrl-Z
Pausing... in box.spi: fd
box.spi ?:side
60
box.spi ?co
Pausing... in box.spi: [if and :side > 80 :side <  90 [pause]]
box.spi ?:side
85
box.spi ?co
```

.contents

Action:
Displays the contents of the Dr. Logo symbol space.

Syntax:
.contents

Explanation:
Displays the contents of the Dr. Logo symbol space.

```
Examples:
?sort .contents
[()  *  + - .APV .BUR .DEF .ENL .FIL .FMT .PAK .PKG .PRM .RDR .REM  .SPC  .WTR
.contents .deposit .examine .in .out .replace .reptail .setseg / < <= <> =  =<
=> > >< >= FALSE TRUE ^ abs allopen an empty word and arctan ascii back bf  bk
bl bury butfirst butlast catch changef char clean clearscreen cleartext  close
closeall  co  copydef copyoff copyon cos count cs ct  cursor  defaultd  define
definedp degrees dir dirpic dot dotc ed edall edf edit edns edps eform  emptyp
equalp  er  erall erase erasefile erasepic ern erns erps erract error  exp  fd
fence fill first follow form forward fput fs fullscreen glist go gprop heading
help hideturtle home ht if iff iffalse ift iftrue int item keyp label last  lc
left  list listp load loadpic local log log10 lowercase lput lt  make  memberp
mouse name namep nodes noformat not notrace nowatch numberp op open or  output
package  pal  pause pd pe pendown penerase penreverse penup  pi  piece  pkgall
plist  po poall pocall pons popkg pops poref pos potl pots pprop pps  pr  prec
primitivep  print proclist product pu px quotient radians random  rc  readchar
readeofp  reader  readlist readquote recycle redefp remainder  remprop  repeat
rerandom  right rl round rq rt run save savepic screenfacts se sentence  setbg
setcursor  setd  seth  setheading setpal setpan setpc  setpen  setpos  setprec
setread  setres  setscrunch  setsplit  setwrite  setx  sety  setzoom  sf  show
showturtle  shuffle  sin sort splitscreen sqrt ss st stop sum  tan  test  text
textscreen  tf  thing  throw tones toplevel towards trace  ts  tt  turtlefacts
turtletext  type uc unbury uppercase wait watch where window word  wordp  wrap
writer xcor ycor]

(ROCHE>
 .APV = Associated Property Value
 .BUR = BURied
 .CAT = CATch
 .DEF = DEFinition of a procedure
 .ENL = ENd of Line
 .FIL = FILe specification
 .FMT = ForMaT (indentation)
 .PAK = PAcKaged procedure
 .PKG = PacKaGe name
 .PRM = PRiMitive
 .RDR = ReaDeR
 .REM = REMark (;)
 .SPC = SPaCe
 .WTR = WriTeR
)


copydef
-------

Action:
Makes a copy of a procedure definition, and gives it a new name.

Syntax:
copydef new_procname old_procname

Explanation:
COPYDEF  makes  a  copy of a procedure definition, and gives it  a  new  name.
COPYDEF  creates  a  new  procedure by making an exact  copy  of  an  existing
procedure  definition,  and  giving it a new title  line.  The  new  procedure
becomes a part of your workspace, and can be referenced, edited, and saved  on
disk.

You cannot associate a second name with a Dr. Logo primitive, unless you  have
made the system variable REDEFP TRUE. If the old_procname is a primitive,  the
new_procname  takes  all  the characteristics of a  primitive;  it  cannot  be
printed out or edited.

Examples:
?to box :side
>repeat 4
>  [forward :side right 90]
>end
box defined
?copydef "square "box
?po "square
to square :side
repeat 4 [forward :side right 90]
end


copyoff
-------

Action:
```

Stops echoing text at the printer.

Syntax:
copyoff

Explanation:
COPYOFF stops the echoing of text at the printer. If you have a printer, you can start printer echo with COPYON.

Examples:
These examples assume you have three files named PIGL, FLY, and SHAPES on the disk in the default drive.

```
?copyon
?dir                          \
[PIGL.LOG FLY.LOG SHAPES.LOG] > This is echoed on the printer.
?copyoff                      /
?dir
[PIGL.LOG FLY.LOG SHAPES.LOG]
```

copyon

Action:
Starts echoing text at the printer.

Syntax:
copyon

Explanation:
COPYON starts echoing text at the printer, if you have one. After COPYON, everything Dr. Logo displays on your text screen is also printed at the printer. You can stop printer echo with COPYOFF.

Examples:
The following exampes assume you have three files named PIGL, FLY, and SHAPES on the disk in the default drive.

```
?copyon
?dir                          \
[PIGL.LOG FLY.LOG SHAPES.LOG] > This is echoed on the printer.
?copyoff                      /
?dir
[PIGL.LOG FLY.LOG SHAPES.LOG]
```

cos

Action:
Outputs the cosine of the input number of degrees.

Syntax:
cos degrees_n

Explanation:
COS outputs the trigonometric cosine of the input number of degrees. COS outputs a decimal number between 0 and 1.

Examples:

```
?cos 0
1

?to plot.cosine
>setpc 2
>make "val 0
>make "x -150
>make "inc (300 / 60)
>setx 150 setx -150
>penup setpos list :x 90 pd
>setpc 1
>plot.c :val
>end
plot.cosine defined
?to plot.c :val
>if :x > 150
>   [stop]
>make "y (90 * (cos :val))  ; 90 makes plot visible
>setheading towards list :x :y
>setpos list :x :y
```

```
>make "x :x + :inc
>make "val :val + 6
>plot.c :val
>end
plot.c defined
?plot.cosine
```

count

Action:
Outputs the number of elements in the input object.

Syntax:
count object

Explanation:
COUNT outputs the number of elements in the input object, which can be a word, number, or list. To count the items in a list within a list, use an ITEM expression as input to COUNT.

Examples:

```
?count "chocolate
9

?count [chocolate]
1

?count [vanilla strawberry [mocha unsweetened milk german]]
3

?count item 3 [vanilla strawberry [mocha unsweetened milk german]]
4
```

cursor

Action:
Outputs a list that contains the column and line numbers of the cursor's position within the text window.

Syntax:
cursor

Explanation:
CURSOR outputs a coordinate list that contains the column and line numbers of the cursor's position within the text window. The first element of the list is the column number; the second, the line number. The line number ranges from 0 to 24. The column number ranges from 0 to 79.

Examples:

```
?cleartext
?cursor
[0 1]

?(type  [The current cursor position is\ ] show cursor
The current cursor position is [32 23]

?print sentence [The current cursor position is:] cursor
The current cursor position is: 0 24
```

defaultd (= DEFAULT Drive name)

Action:
Outputs the name of the current default drive.

Syntax:
defaultd

Explanation:
DEFAULTD outputs the name of the current default drive. Dr. Logo looks in the directory of the disk in the default drive when you do not specify a drive name in a disk command such as SAVE, LOAD, ERASEFILE, CHANGEF, or DIR.

Examples:

```
?to saved
>make "disk.name defaultd
```

```
>end
saved defined
?to restored
>setd :disk.name
>end
restored defined
?defaultd
A:
?saved
?setd b:
?defaultd
B:
?restored
?defaultd
A:
```

define

Action:
Makes the input definition list the definition of the specified procedure name.

Syntax:
define procname defin_list

Explanation:
DEFINE allows you to write a procedure that can, in turn, define other procedures. A DEFINE expression defines a procedure without using an editor (such as EDIT or ED), TO, or END.

DEFINE requires two inputs: a name and a definition list. Note that the name you input to DEFINE cannot be the name of a Dr. Logo primitive, unless REDEFP is TRUE.

A definition list is a special kind of list with a special format. The first element of the input list must be a list of names for inputs to the procedure. Do not put colons (":") before names in this list! If the procedure is to require no inputs, the first element in DEFINE's input list must be an empty list. DEFINE uses the input name and the first element of the definition list to compose the title line of the procedure.

Each remaining element of DEFINE's input list must be a list containing one line of the procedure definition. Do not put END in this list; END is not a part of a procedure's definition.

The TEXT primitive also uses this format when it outputs a definition list. In fact, you can use a TEXT expression to input a definition list to DEFINE.

Examples:

```
?define "say.hello [[] [print [Hello world!]]
?po "say.hello
to say.hello
print [Hello world!]
end
?to learn
>make "definition [[]]
>print [Enter expressions you would like saved in a procedure.]
>print [Enter ERASE to delete your last instruction.]
>read.lines
>print [Type Y to define procedure, any other key to abandon.]
>test lowercase readchar = "y
>iftrue [type [Name for procedure?]
>        make "title first readlist
>        define :title :definition]
>end
learn defined
?to read.lines
>make "newline readlist
>if lowercase :newline = [end]
>   [stop]
>if lowercase :newline = [erase]
>   [delete]
>   [run :newline make "definition lput :newline :definition]
>read.lines
>end
read.lines defined
?to delete
>print sentence "Deleting last :definition
>make "definition butlast :definition
```

```
>end
delete defined
?learn
Enter expressions you would like saved in a procedure.
Enter erase to delete your last instruction.
cs
erase
Deleting cs
setpc 1
forward 40 right 90
setpc 2
repeat 36 [forward 5 left 10]
left 90 setpc 1 back 40
print "balloon!
balloon!
end
Type Y to define procedure, any other key to abandon. y
Name for procedure?
balloon
?balloon
balloon!
```

definedp (= DEFINED Predicate)

Action:
Outputs TRUE if the input name identifies a defined procedure.

Syntax:
definedp object

Explanation:
DEFINEDP outputs TRUE if the input name identifies a procedure currently defined in the workspace. DEFINEDP returns FALSE if the input name identifies a primitive name, a variable name, or anything but a defined procedure name.

Examples:

```
?to balloon
>setpc 1
>forward 40 right 90
>setpc 2
>repeat 36
>  [forward 5 left 10]
>left 90 setpc 1 back 40
>print "balloon!
>end
balloon defined
?definedp "balloon
TRUE
?definedp "definedp
FALSE
```

degrees

Action:
Outputs the number of degrees in the input number of radians.

Syntax:
degrees radian_n

Explanation:
DEGREES outputs the number of degrees in the input number of radians, where degrees = radians * (180 / pi).

Examples:

```
?degrees 1
57.2957795130823

?degrees 2
114.591559026165

?degrees 3
171.887338539247

?to degrees.cycle :vals
>if emptyp :vals
>   [stop]
>(print  [There  are]  degrees  (run first :vals)  [degrees  in]  first  :vals
```

```
"radians.)
>make "vals butfirst :vals
>degrees.cycle :vals
>end
degrees.cycle defined
?make "vals [[pi] [(pi / 2)] [(pi /4)] [(pi / 8)]]
?degrees.cycle :vals
There are 180 degrees in pi radians.
There are 90 degrees in (pi / 2) radians.
There are 45 degrees in (pi / 4) radians.
There are 22.5 degrees in (pi / 8) radians.
```

.deposit

Action:
Puts a number into a memory location.

Syntax:
.deposit n n

Explanation:
Puts the second input number into the memory location specified by the first input number. This location is relative to the absolute location established by .SETSEG. THIS PRIMITIVE SHOULD BE USED WITH CAUTION!

Examples:
?.deposit 2 3

dir

Action:
Outputs a list of the Dr. Logo file names on the default or specified disk.

Syntax:
dir < d: >

Explanation:
DIR outputs a list of Dr. Logo file names on the default or specified disk. If you do not specify a disk drive, DIR looks in the directory of the default disk.

DIR accepts an ambiguous file name as input. An ambiguous file name can refer to more than one file because it contains a wildcard character and gives Dr. Logo a pattern to match. Dr. Logo can then display the file names that match the pattern.

The wildcard character is a question mark ("?"). When the last character in your input file name is a question mark, DIR displays the file names that begin with the characters that precede the question mark. (ROCHE> There seems to be a bug in the version available, as ? does not work. To get ambiguous file names, fill the name (and/or typ) field(s) with ?. Example: DIR F???????.T??)

The list DIR outputs contains the names of only those files that are identified by the LOG file type in the disk's directory. SAVE automatically gives the file type LOG to the files it stores on disk.

Examples:
These examples assume you have three files named SHAPES, PLAID, and PIGLATIN on the disk in the default drive, and two files named COIL and FLY on the disk in drive B.

```
?dir
[SHAPES.LOG PLAID.LOG PIGLATIN.LOG]
?dir "b:
[COIL.LOG FLY.LOG]
?dir "p???????
[PLAID.LOG PIGLATIN.LOG]
```

dirpic

Action:
Outputs a list of the Dr. Logo picture files on the default or specified disk.

Syntax:

dirpic < d: >

Explanation:
DIRPIC outputs a list of Dr. Logo picture files on the default or specified disk. If you do not specify a disk drive, DIRPIC looks in the directory of the default disk.

DIRPIC accepts an ambiguous picture filename as input. An ambiguous picture filename can refer to more than one picture file because it contains a wildcard character and gives Dr. Logo a pattern to match. Dr. Logo can then display the picture filenames that match the pattern.

The wildcard character is a question mark ("?"). When the last character in your input picture filename is a question mark, DIRPIC displays the picture filenames that begin with the characters that precede the question mark. (ROCHE> There seems to be a bug in the version available, as ? does not work. To get ambiguous file names, fill the name (and/or typ) field(s) with ?. Example: DIR F???????.T??)

The list DIRPIC outputs contains the names of only those picture files that are identified by the PC0 file type in the disk's directory. SAVEPIC automatically gives the picture filetype PC0 to the files it stores on disk. (ROCHE> for SETRES 0 images. But images saved under SETRES 1 are given the PC1 file type... and DIRPIC is then unable to find them! This is clearly a bug. The solution is to use an ambiguous filespec, with .PC? for the file type.)

Examples:
These examples assume you have three picture files named SHAPES, PLAID, and PIGLATIN on the disk in the default drive.

```
?dirpic
[SHAPES.PC0 PLAID.PC0 PIGLATIN.PC0]
?dirpic "p???????.pc?
[PLAID.PC0 PIGLATIN.PC0 PLAID.PC1 PIGLATIN.PC1]
```

dot

Action:
Plots a dot at the position specified by the input coordinate list.

Syntax:
dot coord_list

Explanation:
DOT plots a dot at the position specified by the input graphic screen coordinate list. The turtle is not affected in any way.

Examples:

```
?dot [50 50]

?to snow
>make "x random 150 * (first shuffle [1 -1])
>make "y random 100 * (first shuffle [1 -1])
>dot list :x :y
>snow
>end
snow defined
?snow
```

dotc (= DOT Color)

Action:
Outputs the color number of a given dot.

Syntax:
dotc coord_list

Explanation:
Outputs the color number of the dot at the coordinates specified, or -1 if the location is not on the graphic viewport.

Examples:

```
?dotc [50 10]
2
```

edall (= EDit ALL)

Action:
Loads all the variables and procedures in the workspace or specified package(s) into the screen editor's buffer, and enters the screen editor.

Syntax:
edall < pkgname | pkgname_list >

Explanation:
With or without an input, EDALL loads variables and procedures into the screen editor's buffer, and enters the screen editor. EDALL without an input loads all procedures and variables from Dr. Logo's workspace into the screen editor's buffer. If there is nothing in the workspace, the screen editor displays an empty buffer into which you can type either procedures or variables.

You can input a package name or a list of package names to specify a group of procedures and variables to edit. EDALL accepts only defined package names as input.

Within the screen editor, you can use line editing and screen editing control character commands to move the cursor, make changes to text, and exit the screen editor. The control character commands are described in Section 3, "Editing Commands", and summarized in Appendix B, "Dr. Logo control and escape character commands".

Examples:
These examples assume you have two packages named DRAW.PACK and MOVE.PACK in your workspace.

?edall

EDALL loads all the variables and procedures in Dr. Logo's workspace into the screen editor's buffer.

?edall "draw.pack

EDALL loads all the variables and procedures in the package DRAW.PACK into the screen editor's buffer.

?edall [draw.pack move.pack]

EDALL loads all the variables and procedures in the packages DRAW.PACK and MOVE.PACK into the screen editor's buffer.

edf (= EDit File)

Action:
Loads a disk file into the text editor.

Syntax:
edf fname

Explanation:
EDF takes a file name as input. It can load the specified disk file directly into the screen editor's buffer, or create a new disk file with the specified name and enter the screen editor with an empty buffer.

When you start the screen editor using EDF, Dr. Logo updates the disk rather than the workspace when you press Ctrl-C to exit the screen editor. If you press Ctrl-G, Dr. Logo stops and does not update the disk.

One of the advantages of saving information directly from the screen editor's buffer to disk is that you can store individual command lines or expressions in the disk file. When you end a workspace edit with Ctrl-C, Dr. Logo updates only the definitions of procedures and variables in the workspace. Any stand-alone expressions within the edit buffer are lost. However, when Dr. Logo updates the disk, it saves stand-alone expressions. When you LOAD a file that contains stand-alone expressions, Dr. Logo executes them in the same sequence it reads them from disk. Use EDF to create a STARTUP file that automatically executes the procedures it contains.

(ROCHE> When you type Ctrl-C, Dr. Logo saves the file on the disk. If you type Ctrl-G, it is erased from memory. Note that EDF is a full-screen file editor, and can edit files up to 4KB. It does not word-warp but, if you need word-wrap, then you should use a real word-processor, in which case WordStar is, of course, the standard under CP/M.)

Examples:
?edf "startup

edit ed

Action:
Loads the specified procedure(s) and/or variable(s) into the screen editor's buffer, and enters the screen editor.

Syntax:
edit < name | name_list >
ed < name | name_list >

Explanation:
An EDIT command enters the screen editor. EDIT can load procedures, variables, or both into the screen editor's buffer. Input a procedure or variable name, or a list of procedure and variable names to specify what you want to edit.

EDIT is "smart" and assumes certain things about your objectives for an editing session. First, it knows that you frequently use the screen editor to correct errors in procedures. So, when the execution of a procedure ends with an error message and you immediately enter an edit command without specifying a procedure name, EDIT automatically loads the erroneous procedure into the screen editor, and positions the cursor at the line in which the error occurred.

If no error has occurred, EDIT without an input procedure name displays an empty screen editor buffer. When you start to edit an empty buffer, the display is completely blank. You can type variable and procedure definitions into the empty buffer.

EDIT's second assumption is that, when you are defining a new procedure, you need title and ending lines. When you input an undefined procedure name in an edit command, EDIT creates title and ending lines in the screen editor's buffer.

Within the screen editor, you can use line editing and screen editing control character commands to move the cursor, make changes to text, and exit the screen editor. The control character commands are described in Section 3, "Editing Commands", and summarized in Appendix B, "Dr. Logo control and escape character commands".

Examples:
The following examples assume that you have the following in your workspace: a variable named PASSWORD and four procedures named SQUARE, VANISH, ENCODE, and SECRET.

I don't know how to repaet in square: repeat 4 [forward 25 left 90]
?ed

EDIT automatically loads SQUARE into the screen editor's buffer and positions the cursor at the misspelling "repaet".

?edit "vanish

EDIT loads the procedure VANISH into the screen editor's buffer, and positions the cursor at the end of the title line.

?ed [encode secret password

ED loads the ENCODE and SECRET procedures and the variable PASSWORD into the screen editor's buffer, and positions the cursor after ENCODE's title line.

?ed "new.idea

ED with an undefined procedure name automatically creates title and end lines in the screen editor's buffer.

edns (= EDit NameS)

Action:
Loads all the variables in the workspace or specified package(s) into the screen editor's buffer, and enters the screen editor.

Syntax:
edns < pkgname | pkgname_list >

Explanation:
With or without an input, EDNS loads variables into the screen editor's buffer, and enters the screen editor. EDNS without an input loads all variables from Dr. Logo's workspace into the screen editor's buffer. If there is nothing in the workspace, the screen editor displays an empty buffer into which you can type variable and procedure definitions.

You can input a package name or a list of package names to specify a group of variables to edit. EDNS accepts only defined package names as input.

Within the screen editor, you can use line editing and screen editing control character commands to move the cursor, make changes to text, and exit the screen editor. The control character commands are described in Section 3, "Editing Commands", and summarized in Appendix B, "Dr. Logo control and escape character commands".

Examples:
The following examples assume that you have two packages named DRAW.PACK and MOVE.PACK in your workspace.

?edns

EDNS loads all the variables in Dr. Logo's workspace into the screen editor's buffer.

?edns "draw.pack

EDNS loads all the variables in the package DRAW.PACK into the screen editor's buffer.

?edns [draw.pack move.pack]

EDNS loads all the variables in the packages DRAW.PACK and MOVE.PACK into the screen editor's buffer.

edps (= EDit ProcedureS)

Action:
Loads all the procedures in the workspace or specified package(s) into the screen editor's buffer, and enters the screen editor.

Syntax:
edps < pkgname | pkgname_list >

Explanation:
With or without an input, EDPS loads procedures into the screen editor's buffer, and enters the screen editor. EDPS without an input loads all procedures from Dr. Logo's workspace into the screen editor's buffer. If there is nothing in the workspace, the screen editor displays an empty buffer into which you can type either procedures or variables.

You can input a package name or a list of package names to specify a group of procedures to edit. EDPS accepts only defined package names as input.

Within the screen editor, you can use line editing and screen editing control character commands to move the cursor, make changes to text, and exit the screen editor. The control character commands are described in Section 3, "Editing Commands", and summarized in Appendix B, "Dr. Logo control and escape character commands".

Examples:
The following examples assume that you have two packages named DRAW.PACK and MOVE.PACK in your workspace.

?edps

EDPS loads all the procedures in Dr. Logo's workspace into the screen editor's buffer.

?edps "draw.pack

EDPS loads all the procedures in the package DRAW.PACK into the screen editor's buffer.

?edps [draw.pack move.pack]

EDPS loads all the procedures in the packages DRAW.PACK and MOVE.PACK into the screen editor's buffer.

eform (= Exponential FORMat)

Action:
Outputs a number in scientific notation.

Syntax:
eform n1 n2

Explanation:
Outputs N1 in scientific notation, using N2 digits. N2 must be a positive, real number from 1 through 15.

Examples:
```
?eform 1.2 1
 1.E+00
```

emptyp (= EMPTY Predicate)

Action:
Outputs TRUE if the input object is an empty word or an empty list.

Syntax:
emptyp object

Explanation:
An EMPTYP expression returns TRUE only if the input object is an empty word or empty list. Use EMPTYP as the predicate expression in an IF command to determine if all elements of an object input to the procedure have been processed.

Examples:
```
?to nest.circles :sizes
>if emptyp :sizes
>   [stop]
>repeat 36
>  [forward first :sizes left 10]
>nest.circles butfirst :sizes
>end
nest.circles defined
?make "sizes [1 2 3 4 5 6 7 8]
?nest.circles :sizes
```

end (Not a primitive)

Action:
Indicates the end of a procedure definition.

Syntax:
end

Explanation:
END is a special word that signals the end of a procedure definition. When you are using TO to define a procedure, you must put END by itself as the last line of the procedure.

END signals to the procedure editor (the > prompt) that you have finished defining a procedure, and returns you to the interpreter's ? prompt. The screen editor automatically inserts END, if you press Ctrl-C and exit without entering END. Dr. Logo also adds an END line to a procedure defined with a DEFINE expression.

END is not part of a procedure's definition list, and is not a primitive. You can use END as a procedure or variable name, if you are confident that the name will not cause undue confusion.

Examples:
```
?to pent
>repeat 5
>  [forward 25 left 72]
>end
pent defined
```

END signals the procedure editor that you have finished defining a procedure, and returns you to the interpreter's ? prompt.

```
?define "pent [[] [repeat 5 [forward 25 left 72]
?po "pent
to pent
repeat 5 [forward 25 left 72]
end
```

The DEFINE expression automatically adds an END line to the procedure definition.

equalp (= EQUAL Predicate)

Action:
Outputs TRUE if the input objects are equal numbers, identical words, or identical lists.

Syntax:
equalp object object

Explanation:
EQUALP outputs TRUE only if the input objects are equal numbers, identical words, or identical lists; otherwise, EQUALP outputs FALSE.

Examples:

```
?equalp " []
FALSE

?equalp "leaf first [leaf stem flower]
TRUE

?equalp 72 (360 / 5)
TRUE

?equalp [1 2 3] [2 3 4]
FALSE
```

erall (= ERase ALL)

Action:
Erases all the procedures and variables from the workspace or specified package(s).

Syntax:
erall < pkgname | pkgname_list >

Explanation:
ERALL erases the definitions of procedures and variables. ERALL without an input name erases the definitions of all procedures and variables from Dr. Logo's workspace, except for any procedures and variables in buried packages. ERALL without an input name also erases the package names of any unburied packages.

You can give ERALL a package name or a list of package names to specify a group of procedures and variables to be erased. ERALL can accept a buried package name as input; with this request, ERALL can erase a buried package.

ERALL works by removing property pairs from the property lists associated with package, variable, and procedure names. It removes the .PKG pair from a package's property list, the .APV and .PAK pairs from a variable's property lists, and the .DEF and .PAK pairs from a procedure's property list. ERALL does not remove any other property pairs from a property list. So, after you input a buried package name to ERALL, GLIST ".BUR still outputs the name of the erased package. To completely remove the name from your workspace, you must use a REMPROP command to eliminate all pairs from its property list.

Examples:
These examples assume you have two packages named DRAW.PACK and MOVE.PACK in your workspace.

```
?erall
```

This command erases the definitions of all unburied procedures, variables, and packages from Dr. Logo's workspace.

```
?erall "draw.pack
```

This command erases the definitions of procedures and variables in the DRAW.PACK package, as well as the definition of DRAW.PACK as a package name.

?erall [draw.pack move.pack]

This command erases the definitions of the procedures, variables, and package names associated with DRAW.PACK and MOVE.PACK.

erase er

Action:
Erases the specified procedure(s) from the workspace.

Syntax:
erase procname | procname_list
er procname | procname_list

Explanation:
ERASE erases the specified procedure or procedures from Dr. Logo's workspace. ERASE works by removing the .DEF pair from a procedure's property list. If a procedure has other property pairs in its list besides .DEF, you must remove them with REMPROP to completely erase the procedure name from the workspace.

Examples:
Each of these examples assume you have two procedures named WHEEL and FLAG in a package named MANDALA in your workspace.

?erase "wheel

This command erases the definition of the WHEEL procedure from Dr. Logo's workspace.

?erase [wheel flag]

This command erases the definitions of the procedures WHEEL and FLAG from Dr. Logo's workspace.

```
?plist "wheel
[.PAK mandala [[] [repeat 12 [flag left 30]]]]
?erase "wheel
?plist "wheel
[.PAK mandala]
```

ERASE removes only the definition (.DEF) pair from a procedure name's property list. Use REMPROP to delete .PAK.

erasefile

Action:
Erases the specified Dr. Logo file.

Syntax:
erasefile fname

Explanation:
ERASEFILE erases a file name from a disk directory. You can enter a drive name before the file name, to erase a file not on the default drive.

ERASEFILE accepts an ambiguous file name as input. An ambiguous file name can refer to more than one file because it contains a wildcard character and gives Dr. Logo a pattern to match. Dr. Logo can then erase any files whose names match the pattern.

The wildcard character is a question mark ("?"). When the last character in your input file name is a question mark, ERASEFILE erases any files whose names begin with the characters that precede the question mark. (ROCHE> There seems to be a bug in the version available, as ? does not work. To get ambiguous file names, fill the name (and/or typ) field(s) with ?. Example: DIR F???????.T??)

Examples:
Each of these examples assume you have two files named PIGLATIN and PLAID on the disk in drive B. The first command erases only PIGLATIN.LOG. The second command erases both files.

?erasefile "b:piglatin

?erasefile "b:p???????

(ROCHE> I found the following procedure useful...)

```
to era :fname
erasefile :fname
end
```

erasepic

Action:
Erases the specified Dr. Logo image file.

Syntax:
erasepic fname

Explanation:
ERASEPIC erases a Dr. Logo image filename from a disk directory. You can enter a drive name before the image filename to erase a image file not on the default drive.

ERASEPIC accepts an ambiguous image filename as input. An ambiguous image filename can refer to more than one image file because it contains a wildcard character and gives Dr. Logo a pattern to match. Dr. Logo can then erase any image files whose names match the pattern.

The wildcard character is a question mark ("?"). When the last character in your input image filename is a question mark, ERASEPIC erases any image files whose names begin with the characters that precede the question mark. (ROCHE> There seems to be a bug in the version available, as ? does not work. To get ambiguous file names, fill the name (and/or typ) field(s) with ?. Example: DIR F???????.T??)

Examples:
Each of these examples assume you have two image files named PIGLATIN and PLAID on the disk in drive B. The first command erases only PIGLATIN.LOG. The second command erases both files.

```
?erasepic "b:piglatin

?erasepic "b:p???????
```

ern (= ERase one Name)

Action:
Erases the specified variable(s) from the workspace.

Syntax:
ern varname | varname_list

Explanation:
ERN erases a specified variable name or names from Dr. Logo's workspace. ERN works by removing the .APV pair from the property list associated with the variable name. If a variable has other property pairs in its list, besides .APV, you must remove them with REMPROP to completely erase the variable name from the workspace.

Examples:
Each of these examples assume you have two variables named BIG and SMALL in a package named SIZE in your workspace.

```
?ern "big
```

This command erases the variable name BIG from Dr. Logo's workspace.

```
?ern [big small]
```

This command erases the variable names BIG and SMALL from Dr. Logo's workspace.

```
?plist "big
[.PAK size .APV 80]
?ern "big
?plist "big
[.PAK size]
```

ERN removes only the .APV pair from a variable name's property list. Use REMPROP to remove .PAK.

erns (= ERase several NameS)

Action:
Erases all the variables from the workspace or specified package(s).

Syntax:
erns < pkgname | pkgname_list >

Explanation:
ERNS erases a group of variable names from Dr. Logo's workspace. ERNS without an input name erases all variable names from Dr. Logo's workspace, except for any variable names in buried packages. You can give ERNS a package name or a list of package names to specify a group of variables names to be erased. ERNS can accept a buried package name as input; with this request, ERNS can remove the value from a variable in a buried package.

ERNS works by removing the .APV pair from a variable name's property list. It does not remove any other pair, such as the .PAK pair that indicates the name is packaged. Therefore, after you use ERNS with a package name as input and list the contents of the package with GLIST ".PAK, you will still see your variable names listed under the package name, even though they have no values.

Examples:
These examples assume you have two packages named DRAW.PACK and MOVE.PACK in your workspace.

?erns "draw.pack

This command erases all variable names from the DRAW.PACK package.

?erns [draw.pack move.pack]

This command erases all variable names from the DRAW.PACK and MOVE.PACK packages.

erps (= ERase ProcedureS)

Action:
Erases all the procedures from the workspace or specified package(s).

Syntax:
erps < pkgname | pkgname_list >

Explanation:
ERPS erases a group of procedures from Dr. Logo's workspace. ERPS without an input name erases all procedures from Dr. Logo's workspace, except for any procedures in buried packages. You can give ERPS a package name or a list of package names to specify a group of procedures to be erased. ERPS can accept a buried package name as input; with this request, ERNS can remove the definition from a procedure name in a buried package.

ERPS works by removing the .DEF pair from a procedure name's property list. It does not remove any other property. Therefore, after you use ERPS with a package name as input and list the contents of the package with GLIST ".PAK, you will still see your procedure names listed as regular names under the package name.

Examples:
These examples assume you have two packages named DRAW.PACK and MOVE.PACK in your workspace.

?erps

This command erases all procedures, and leaves variables.

?erps "draw.pack

This command erases all procedures from the DRAW.PACK package.

?erps [draw.pack move.pack]

This command erases all procedure names from the DRAW.PACK and MOVE.PACK packages.

erract (= ERRor ACTion) (Not a primitive)

Not a procedure, but a system variable.

When ERRACT is TRUE, any error causes a PAUSE, during which you can find the origin of the error. Default = FALSE.

error

Action:
Outputs a list whose elements describe the most recent error.

Syntax:
error

Explanation:
ERROR can output a list that describes the most recent error. If the most recent error has already displayed a message on the screen, or if ERROR has already output a list describing the most recent error, ERROR outputs an empty list.

A non-empty ERROR output list contains six elements to describe an error:

1) A number that identifies the error. Dr. Logo error numbers are listed in Appendix A, "Dr. Logo Error Messages".

2) A message that explains the error. This is the message that is usually displayed on the screen.

3) The name of the procedure that contains the erroneous expression. If the error occurred at toplevel, this element is an empty list.

4) The complete line that contains the erroneous expression.

5) The procedure name part of the erroneous expression, if any.

6) The input object part of the erroneous expression, if any.

Dr. Logo can take one of two actions when an error occurs. Which action Dr. Logo takes depends on a system variable called ERRACT. The default state of ERRACT is FALSE. You can set ERRACT to TRUE with a make statement.

While ERRACT is FALSE, Dr. Logo runs a THROW "ERROR command when an error occurs. If no CATCH "ERROR command has been run, Dr. Logo terminates execution of the procedure, prints an error message, and returns to toplevel. If the error occurred within the scope of a CATCH "ERROR command, Dr. Logo returns to the line following the CATCH "ERROR without displaying an error message on the screen. This allows you to handle an error in your own way. The description of the CATCH command gives an example of how to display a custom error message.

While ERRACT is TRUE, Dr. Logo does a PAUSE when an error occurs. During the pause, you can execute an ERROR command to discover the cause of the error, then change variables with MAKE or procedures with EDIT before entering CO to continue execution of the procedure.

Examples:

```
?to safety.circle :size
>catch "error [repeat 180 [forward :size right 2]
>show error
>end
safety.circle defined
?safety.circle 2
[]
?safety.circle "two
[41  [forward doesn't like two as input] safety.circle [catch  "error  [repeat
180 forward :size right 2]]] forward rt]
```

.examine

Action:
Displays the contents of a memory location.

Syntax:
.examine n

Explanation:
Displays the contents (a byte value) of the memory location specified by the input number. This location is relative to the absolute location established

by .SETSEG. THIS PRIMITIVE SHOULD BE USED WITH CAUTION!

Examples:
```
?.examine 22
0
```

exp

Action:
Outputs the natural exponent of the input number.

Syntax:
exp n

Explanation:
EXP outputs the natural exponent of the input number.

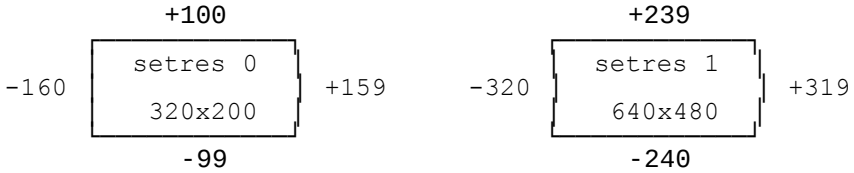
Examples:
```
?exp 1
2.71828182845905

?log (exp 1)
1
```

fence

Action:
Establishes a boundary that limits the turtle to onscreen plotting.

Syntax:
fence

Explanation:
FENCE establishes a boundary around the edge of the graphic screen. The fence limits the turtle to onscreen plotting. When the turtle encounters the fence, Dr. Logo displays a "Turtle out of bounds" message. The number of turtle steps from home to the fence are as follows:



To remove the fence, enter WINDOW.

Examples:
```
?to frame0
>setres 0
>fs clean home
>pu setpos [-160 99] pd
>setx 159
>sety -99
>setx -160
>sety 100
>pu home pd
>end
frame0 defined
?frame0
Ctrl-G
```

```
?to frame1
>setres 1
>setscrunch 1
>setzoom 1
>fs clean home
>pu setpos [-320 239] pd
>setx 319
>sety -240
>setx -320
>sety 239
>pu home pd
>end
frame1 defined
?frame1
Ctrl-G
```

fill

Action:
Fills an area.

Syntax:
fill

Explanation:
Fills an area with the current pen color, by changing the dot under the turtle

(and all horizontally and vertically contiguous dots of the same color) to the current pen color, using the current pen state. (ROCHE> To change the color of the pen, use SETPC.)

Examples:
?fill

first

Action:
Outpus the first element of the input object.

Syntax:
first object

Explanation:
FIRST outputs the first element of the input object. The first of a word or number is a single character. The first of a list is either a word or a list. FIRST of an empty word or empty list outputs an error.

Examples:
```
?first []
first doesn't like [] as input

?first "
first doesn't like an empty word as input

?first "rose
r

?first [weight 165]
weight

?to begin.vowelp :wrd
>output memberp first :wrd [a e i o u]
>end
begin.vowelp defined
?to pig :wrd
>if begin.vowelp :wrd
>   [output word :wrd "ay]
>   [output pig word (butfirst :wrd) (first :wrd)]
>end
pig defined
?to pig.latin :phrase
>if emptyp :phrase
>   [output []]
>output sentence (pig first :phrase) (pig.latin butfirst :phrase)
>end
pig.latin defined
?pig.latin [a ball of string]
[aay allbay ofay ingstray]
```

follow

Action:
Reorganizes workspace, so that the first input-named procedure is followed by the second.

Syntax:
follow procname procname

Explanation:
FOLLOW reorganizes the workspace, so that the first input-named procedure is followed by the second. You can use FOLLOW to specify the order in which Dr. Logo displays procedure titles and definitions (POALL, POPS, POTS, PROCLIST), saves procedures on disk (SAVE), and loads procedures into the screen editor's buffer (EDALL, EDPS). FOLLOW does not change the order of procedures in a package definition.

Examples:
These examples assume you have three procedures named ZOOM, BUZZ, and FLY in a package named FLY in your workspace.

```
?pots
zoom
fly
buzz
```

```
?popkg
fly
  to zoom
  to fly
  to buzz
?follow "buzz "zoom
?pots "fly
fly
buzz
zoom
?popkg
fly
  to zoom
  to fly
  to buzz
```

form (= real FORMat)

Action:
Outputs a number in real notation.

Syntax:
form n1 n2 < n3 >

Explanation:
Outputs a number, N1, with N2 digits before the decimal point, and N3 after it. N2 must be a positive, real number from 1 through 15. If N2 is not an integer, it is rounded to the nearest integer. If N3 is omitted, it is assumed to be zero and the decimal point, and any digits after it, are not printed.

Examples:

```
?form 1.234 2
 1
```

forward fd

Action:
Moves the turtle the input number of steps in the direction of its current heading.

Syntax:
forward distance_n
fd distance_n

Explanation:
FORWARD moves the turtle the input number of steps in the direction of its current heading. If the turtle's pen is down, the turtle leaves a trace of its path. On the IBM Personal Computer, a turtle step is equivalent to one dot (pixel).

Examples:

```
?forward 100
?cs

?to plaid
>wrap
>setpc 3 right 40 forward 10965
>setpc 2 right 90 forward 5000
>setpc 1 penup right 90 forward 6 pendown ht
>repeat 625
>  [forward 3 left 90 forward 1 right 90 back 3 left 90 forward 1 right 90]
>end
plaid defined
?plaid
```

fput (= First PUT)

Action:
Outputs a new object formed by making the first input object the first element in the second input object.

Syntax:
fput object object

Explanation:

FPUT outputs a new object formed by making the first input object the first element in the second input object. Generally, the new object is the same kind of object as the second input object: a word, number, or list. However, when you FPUT a word into a number, FPUT outputs a word.

Examples:
```
?fput "s "miles
smiles

?fput "banana [grape strawberry melon]
[banana grape strawberry melon]

?fput 20 20
2020

?fput [corn dog] [hamburger taco pizza]
[[corn dog] hamburger taco pizza]

?to exchange :from :to :in
>if emptyp :in
>   [output []]
>if (first :in) = :from
>   [output fput :to exchange :from :to butfirst :in]
>   [output sentence first :in exchange :from :to butfirst :in]
>end
exchange defined
?exchange "cat "dog [My cat has fleas.]
[My dog has fleas.]
```

fullscreen fs

Action:
Selects full graphic screen.

Syntax:
fullscreen
fs

Explanation:
FULLSCREEN removes the SPLITSCREEN or TEXTSCREEN from the display, and dedicates the monitor to graphics. FULLSCREEN is equivalent to a Ctrl-L keystroke. Generally, it is easiest to use Ctrl-L when you are typing commands at the keyboard, and FULLSCREEN (FS) from within a procedure.

Examples:
```
?repeat 12 [repeat 4 [forward 60 right 90] right 30]
?splitscreen
?fullscreen
```

glist (= Get property LIST)

Action:
Outputs a list of all objects in the workspace or specified package(s) that have the input property in their property lists.

Syntax:
glist prop < pkgname | pkgname_list >

Explanation:
With or without an input, GLIST outputs a list of objects that have the input property in their property lists. Without an input, GLIST checks the property lists of all objects in the workspace. If you specify a package name or a list of packages in the GLIST expression, GLIST checks only the objects in the specified package(s).

Property lists and Dr. Logo's system properties are described in Section 5, "Property Lists, Workspace, and Disks". To understand the examples below, remember that .DEF is the system property Dr. Logo gives to defined procedures.

Examples:
These examples assume that, in your workspace, you have packages named FLY and PIGLATIN.

```
?glist ".DEF
[countdown pig.latin begin.vowelp fly buzz triangle.text zoom pig]
```

```
?glist ".DEF "fly
[fly buzz zoom]

?glist ".DEF [fly pig.latin]
[pig.latin begin.vowelp fly buzz zoom pig]

?sort glist ".PRM
[()  * + - .contents .deposit .examine .in .out .replace .reptail .setseg /  <
<=  <>  =  =< => > >< >= ^ abs allopen and arctan ascii back  bf  bk  bl  bury
butfirst butlast catch changef char clean clearscreen cleartext close closeall
co  copydef  copyoff copyon cos count cs ct cursor  defaultd  define  definedp
degrees dir dirpic dot dotc ed edall edf edit edns edps eform emptyp equalp er
erall  erase  erasefile erasepic ern erns erps error exp fd fence  fill  first
follow form forward fput fs fullscreen glist go gprop heading help  hideturtle
home ht if iff iffalse ift iftrue int item keyp label last lc left list  listp
load  loadpic local log log10 lowercase lput lt make memberp mouse name  namep
nodes noformat not notrace nowatch numberp op open or output package pal pause
pd pe pendown penerase penreverse penup pi piece pkgall plist po poall  pocall
pons  popkg  pops  poref  pos potl pots pprop pps  pr  prec  primitivep  print
proclist  product  pu px quotient radians random rc readchar  readeofp  reader
readlist readquote recycle remainder remprop repeat rerandom right rl round rq
rt  run  save  savepic  screenfacts se  sentence  setbg  setcursor  setd  seth
setheading setpal setpan setpc setpen setpos setprec setread setres setscrunch
setsplit  setwrite  setx  sety setzoom sf show  showturtle  shuffle  sin  sort
splitscreen sqrt ss st stop sum tan test text textscreen tf thing throw  tones
towards trace ts tt turtlefacts turtletext type uc unbury uppercase wait watch
where window word wordp wrap writer xcor ycor]


go
--

Action:
Executes  the  line within the current procedure identified by LABEL  and  the
input word.

Syntax:
go word

Explanation:
GO executes the line within the current procedure identified by LABEL and  the
input  word.  This means that, after GO, Dr. Logo will next execute  the  line
following  the  LABEL expression. The GO and LABEL expressions must  have  the
same input word, and be in the same procedure.

GO and LABEL let you change the sequence in which Dr. Logo executes the  lines
in a procedure. Therefore, GO and LABEL cannot be on the same line. This means
you  cannot use GO and LABEL within an instruction list input to REPEAT,  RUN,
or the IFs.

Examples:
?to triangle.text :string
>label "loop
>if emptyp :string
>   [stop]
>print :string
>make "string butfirst :string
>go "loop
>end
triangle.text defined
?triangle.text [Get along little dogie.]
Get along little dogie.
along little dogie.
little dogie.
dogie.

?to countdown :n
>label "loop
>if :n < 0
>   [stop]
>type :n
>make "n (:n - 1)
>go "loop
>end
countdown defined
?countdown 9
9876543210


gprop   (= Get PROPerty)
-----
```

Action:
Outputs the value of the input-named property of the input-named object.

Syntax:
gprop name prop

Explanation:
GPROP outputs the value of the input-named property of the input-named object. The input name identifies the object that has the property. If the object does not have the input-named property, GPROP outputs an empty list.

Section 5, "Property Lists, Workspace, and Disks", describes property lists, and defines the properties Dr. Logo gives to objects in the workspace. To understand the examples below, remember that a property list is made up of property pairs. The first element of the pair is the property; the second, its value. Dr. Logo gives defined procedures the .DEF property, and defined variables the .APV property.

Examples:
This example assumes you have a procedure named pig in your workspace.

```
?gprop "pig ".DEF
[[wrd] if begin.vowelp :wrd [output word :wrd "ay]
[output ! pig word (butfirst :wrd) (first :wrd)]]

?make "height "72"
?gprop "height ".APV
72"

?gprop "height ".DEF
[]
```

heading

Action:
Outputs a number that indicates the turtle's current heading.

Syntax:
heading

Explanation:
HEADING outputs the turtle's current heading as a real number between 0 and 359 inclusive. The turtle's heading corresponds to traditional compass headings:

Degrees	Direction	Pointing
0	North	up
90	East	right
180	South	down
270	West	left

Examples:

```
?home show heading
0

?right 1 show heading
1

?right 1 show heading
2

?left random 360 heading
126

?to compass :n
>if memberp :n
>   [0 90 180 270] [points]
>if (:n > 360)
>   [stop]
>(type heading char 9)
>right 5 penup forward 25 pendown forward 5 penup back 30
>compass :n + 5
>end
compass defined
?to points
>pu
>forward 40
```

```
>if :n = 0
>   [turtletext "N]
>if :n = 90
>   [turtletext "E]
>if :n = 180
>   [turtletext "S]
>if :n = 270
>   [turtletext "W]
>setpos [0 0]
>end
points defined
?compass 0
```

help

Action:
Displays the names and abbreviations of all Dr. Logo primitives.

Syntax:
help < primitive >

Explanation:
HELP displays the names and abbreviations of all Dr. Logo primitives.

HELP displays only 25 lines on the screen at a time. It waits for you to press any key before displaying the next 25 lines.

Examples:

```
?help
```

HELP displays the list of all Dr. Logo primitives. Press any key to view next screenful of names.

hideturtle ht

Action:
Makes the turtle invisible.

Syntax:
hideturtle
ht

Explanation:
HIDETURTLE makes the turtle invisible. When invisible, the turtle draws faster and does not distract visually from the drawing. To make the turtle visible again, enter SHOWTURTLE.

Examples:

```
?hideturtle
?showturtle
?to star :size
>repeat 5
>  [forward :size left 217 forward :size left 71]
>end
star defined
?star 30
?ht
```

home

Action:
Returns the turtle to position [0 0] (the center of the graphic screen) heading 0 (North).

Syntax:
home

Explanation:
HOME positions the turtle at "home", position [0 0] (the center of the screen), heading 0 (North). HOME is the position in which the turtle first appears when you start Dr. Logo.

HOME does not change the turtle's pen state. For example, if the turtle's pen is down when HOME is executed, the turtle draws a line from its current loation to the center of the screen.

Examples:
```
?forward 80 left 120 forward 80
?home
```

if
--

Action:
Executes one of two instructions lists, depending on the value of the input predicate expression.

Syntax:
```
if pred_exp instr_list < instr_list >
```

Explanation:
IF transfers execution to one of two instruction lists, depending on the TRUE or FALSE value of the input predicate expression. You can use an IF expression to make a decision regarding the flow of control within your procedure.

The first input to IF must be a predicate expression, one that outputs TRUE or FALSE. If the predicate outputs TRUE, Dr. Logo executes the first instruction list. If the predicate outputs FALSE, Dr. Logo executes either the second instruction list (if any), or the next line in the procedure.

IF requires literal instruction lists as input; that is to say, an instruction list input to IF must be enclosed in square brackets ("[" and "]"). Dr. Logo allows nested IF expressions. This means you can use an IF statement within an instruction list you input to IF.

Examples:
```
?setheading random 360
?if heading < 180 [print "East] [print "West]
West
```

The COIN procedures that follow demonstrate several ways to use IF. COIN1 shows IF making a simple decision between two input instruction lists.

```
?to coin1
>if 1 = random 2
>   [print "heads]
>   [print "tails]
>end
coin1 defined
```

COIN2 shows IF providing input to another procedure.

```
?to coin2
>print if 1 = random 2 ["heads] ["tails]
>end
coin2 defined
```

COIN3 shows how IF executes the next line in a procedure if no second instruction list is input.

```
?to coin3
>if 1 = random 2
>   [output "heads]
>output "tails
>end
coin3 defined
```

COIN4 shows how the second instruction list makes the line following the IF expression independent of the results of the IF test.

```
?to coin4
>if 1 = random 2
>   [type "heads]
>   [type "tails]
>print [\ side up]
>end
coin4 defined
```

iffalse iff

Action:
Executes the input instruction list if the most recent test was FALSE.

Syntax:
iffalse instr_list
iff instr_list

Explanation:
IFFALSE executes its input instruction list if the result of the most recent TEST expression was FALSE. If the result of test was TRUE, IFFALSE does nothing.

You can use TEST, IFFALSE, and IFTRUE as an alternate to IF to control the flow of execution within your procedure. These primitives are particularly useful when you need Dr. Logo to evaluate expressions after it evaluates a predicate expression, but before it executes the chosen instruction list.

IFFALSE requires a literal instruction list as input; that is to say, any instruction list you input to IFFALSE must be enclosed in square brackets ("[" and "]"). Dr. Logo allows nested IF expressions. This means you can use TEST, IFFALSE, and IFTRUE within the instruction list you input to IFFALSE.

Examples:
The COIN5 procedure is similar to the COIN procedures shown as examples under if, but shows how to use TEST, IFFALSE, and IFTRUE. COIN5 evaluates a expression after it evaluates a predicate expression, but before it executes the chosen instruction list.

```
?to coin5
>test 1 = random 2
>if 1 = random 1000000
>   [print [Landed on edge!] stop]
>iftrue [type "heads]
>iffalse [type "tails]
>print [\ side up]
>end
coin5 defined
```

iftrue ift

Action:
Executes the input instruction list if the most recent test was TRUE.

Syntax:
iftrue instr_list
ift instr_list

Explanation:
IFTRUE executes its input instruction list if the result of the most recent TEST expression was TRUE. If the result of test was FALSE, IFTRUE does nothing.

You can use TEST, IFFALSE, and IFTRUE as an alternate to IF to control the flow of execution within your procedure. These primitives are particularly useful when you need Dr. Logo to evaluate expressions after it evaluates a predicate expression, but before it executes the chosen instruction list.

IFTRUE requires a literal instruction list as input; that is to say, any instruction list you input to IFTRUE must be enclosed in square brackets ("[" and "]"). Dr. Logo allows nested IF expressions. This means you can use TEST, IFFALSE, and IFTRUE within the instruction list you input to IFFALSE.

Examples:
The COIN5 procedure is similar to the COIN procedures shown as examples under IF, but shows how to use TEST, IFFALSE, and IFTRUE. COIN5 evaluates a expression after it evaluates a predicate expression, but before it executes the chosen instruction list.

```
?to coin5
>test 1 = random 2
>if 1 = random 1000000
>   [print [Landed on edge!] stop]
>iftrue [type "heads]
>iffalse [type "tails]
>print [\ side up]
>end
coin5 defined
```

.in

```
Action:
Displays the contents of a I/O port.

Syntax:
 .in port_n

Explanation:
Displays  the contents (a byte value) of the specified I/O port. Port  numbers
range from 0 to 65535.

Examples:
?.in 397
50


int
---

Action:
Outputs the integer portion of the input number.

Syntax:
int n

Explanation:
INT outputs the integer portion of the input number. INT discards any  decimal
point and subsequent numerals from the input number. To round a decimal number
to the nearest integer, use ROUND.

Examples:
?int 3.333333
3

?int 3
3

?int 28753 / 12
2396

?int -75.482
-75

?to integerp :n
>if numberp :n
>   [output :n = int :n]
>(print :n [is not a number.]
>end
integerp defined
?integerp 6.65
FALSE
?integerp 6
TRUE
?integerp "six
six is not a number.


item
----

Action:
Outputs the specified element of the input object.

Syntax:
item n object

Explanation:
ITEM outputs an element of the input object. The input number specifies  which
element of the input object ITEM outputs.

Examples:
?item 4 "dwarf
r

?item 2 [brownie snickerdoodle wafer]
snickerdoodle

?make "favorites [brownie snickerdoodle [oatmeal raisin]]
?count item 3 :favorites
2

?to make.sentence :n :v
```

```
>print (sentence item ((random (count :n)) + 1) :n
>                item ((random (count :v)) + 1) :v
>end
make_sentence defined
?make_sentence   [elephants   crocodiles  mambasnakes]   [pirouette   tapdance
rollerskate]
mambasnakes pirouette
```

keyp (= KEY Predicate)

Action:
Outputs TRUE is a character has been typed at the keyboard, and is waiting to be read.

Syntax:
keyp

Explanation:
KEYP outputs TRUE if a character has been typed at the keyboard, and is waiting to be read by READCHAR, READQUOTE, or READLIST.

When you type a character at toplevel, Dr. Logo displays the character on the screen. However, while a procedure is executing, Dr. Logo stores any characters that are typed at the keyboard in a buffer, and displays them on the screen when the procedure ends. By using KEYP in a procedure, you can discover whether or not your user has typed something while your procedure is executing.

What your user types into the buffer is critical if your procedure contains subsequent READQUOTE, READCHAR, or READLIST expressions that read the buffer and input its contents to the rest of your procedure. You can use KEYP to determine if your user has typed something, then check that the buffer contains the kind of input your procedure requires before passing the input to the procedure.

Examples:

```
?to sketch
>forward 2
>wait 10
>if keyp
>   [turn readchar]
>sketch
>end
sketch defined
?to turn :way
>if ascii :way = 6 ; Right Arrow key
>   [right 10]
>if ascii :way = 2 ; Left Arrow key
>   [left 10]
>end
turn defined
?sketch

?to delay
>print [Any keystroke delays next number by a second.]
>print.out
>end
delay defined
?to print.out
>if keyp
>   [wait 60 sink readchar]
>(type random 10 char 9)
>print.out
>end
print.out defined
?to sink :object
>end
sink defined
Any keystroke delays next number by a second.
6    9   2   5   8   0   3   6   1   9  1   0   6   1   3   5   1   8   0    1
7   Ctrl-G

?to count.stars
>make "number 1 + random 10
>draw.stars :number
>print [How many stars?]
>label "back
>if keyp
>   [make buffer readquote
```

```
>    (if not numberp first :buffer
>        [ern "buffer go "back]
>        [go on])]
>   [go "back]
>label "on
>if :number = :buffer
>   [print "Right!]
>   [print [Wrong! There are] :number]
>cs
>count.stars
>end
count.stars defined
?to draw.stars :number
>if :number = 0>
>   [stop]
>penup setpos list
>   (random 100 * first shuffle [1 -1])
>   (random 70 * first shuffle [1 -1])
>pendown star pu
>draw.stars :number - 1
>end
draw.stars defined
?to star2 ; Five pointed star
>repeat 5
>  [forward 30 left 127 forward 30 left 70]
>end
star2 defined
?count.stars
How many stars?
4
Right!
How many stars?
5
Right!
How many stars?
6
Wrong! There are 5
How many stars? Ctrl-G
Stopped! in count.stars: go
```

label

Action:
Identifies the line to be executed after a GO expression.

Syntax:
label word

Explanation:
LABEL identifies the line to be executed after a GO expression. The LABEL and GO expressions must be in the same procedure, and have the same input word.

GO and LABEL let you change the sequence in which Dr. Logo executes the lines in a procedure. Therefore, GO and LABEL cannot both be on the same line. This means you cannot use both GO and LABEL within an instruction list input to REPEAT, RUN, or the IFs.

Examples:

```
?to triangle.text :string
>label "loop
>if emptyp :string
>   [stop]
>print :string
>make "string butfirst :string
>go "loop
>end
triangle.text defined
?triangle.text [Wyoming will be your new home.]
Wyoming will be your new home.
will be your new home.
be your new home.
your new home.
new home.
home.

?to countdown :n
>label "loop
>if :n < 0
>   [stop]
```

```
>type :n
>make "n (:n - 1)
>go "loop
>end
countdown defined
?countdown 9
9876543210
```

last

Action:
Outputs the last element of the input object.

Syntax:
last object

Explanation:
LAST outputs the last element of the input object. The last of a word or number is a single character. The last of a list is either a word or a list. LAST of an empty word or empty list outputs an error.

Examples:

```
?last []
last doesn't like [] as input

?last "
last doesn't like an empty word as input

?last "skyline
e

?last [weight 165]
165

?to mirror :word
>(print reverse :word "| :word
>end
mirror defined
?to reverse :word
>if emptyp :word
>   [output "]
>output (word last :word reverse butlast :word)
>end
reverse defined
?mirror "mirror
rorrim | mirror
```

left lt

Action:
Rotates the turtle the input number of degrees to the left.

Syntax:
left degrees_n
lt degrees_n

Explanation:
LEFT rotates the turtle the input number of degrees to the left. Usually, you input a number of degrees between 0 and 359. If the input number is greater than 359, the turtle appears to move the input number minus 360 degrees to the left. If you input a negative number to LEFT, the turtle turns to the right.

Examples:

```
?repeat 36 [left 10]

?repeat 36 [left -10]

?to around.L :s
>repeat 360
>  [forward :s left 10 make "s :s + .01]
>end
around.L defined
?around.L 2
```

list

Action:
Outputs a list made up of the input objects.

Syntax:
list object object (...)

Explanation:
LIST outputs a list made up of the input objects. LIST is like SENTENCE, but does not remove the outermost brackets from an input object.

Without punctuation, LIST requires and accepts two input objects. LIST can accept more or fewer inputs when you enclose the LIST expression in parentheses ["(" and ")"]. If no other expressions follow the LIST expression on the line, you do not need to type the closing right parenthesis [")"].

There are two ways of creating lists in Dr. Logo:

1) using square brackets ("[" and "]")
2) using LIST or SENTENCE

Each way results in a different kind of list. When you create a list by enclosing elements in square brackets ("[" and "]"), you create a "literal" list. Dr. Logo treats the elements of the list literally; it does not evaluate expressions or look up the values of variables named in a literal list.

When you use LIST or SENTENCE to create a list, you can use variables and expressions to specify the elements Dr. Logo will put in the list. You can use list and sentence to create a list for input to most procedures. However, IF, IFFALSE, and IFTRUE require literal lists as input.

Examples:
```
?show list "big "feet
[big feet]

?list "potatoes [mashed baked fried]
[potatoes [mashed baked fried]]

?(list
[]

?(list 21 22 23 24
[21 22 23 24]

?to square
>repeat 4
>  [forward 60 right 90]
>end
square defined
?make "procname "square
?run (list :procname

?to zip.zap
>setpos list
>   random 150 * first shuffle
>   random 100 * first shuffle
>zip.zap
>end
zip.zap defined
?zip.zap
```

listp (= LIST Predicate)

Action:
Outputs TRUE if the input object is a list.

Syntax:
listp object

Explanation:
LISTP outputs TRUE if the input object is a list. Otherwise, LISTP outputs FALSE.

Examples:
```
?listp "force
FALSE

?listp [father brother sister]
TRUE
```

```
?listp "
FALSE

?listp []
TRUE

?to reflect :object
>if listp :object
>   [print reverse.list :object]
>   [print reverse :object]
>end
reflect defined
?to reverse.list :list
>if emptyp :list
>   [output "]
>output (sentence last :list reverse.list butlast :list
>end
reverse.list defined
?to reverse :word
>if emptyp :word
>   [output "]
>output (word last :word reverse butlast :word
>end
reverse defined
?reflect "skywalker
reklawyks
?reflect [wherever you go]
go you wherever


load
----

Action:
Reads the input-named Dr. Logo file from the disk into the workspace.

Syntax:
load fname < pkgname >

Explanation:
LOAD  reads the contents of the input-named Dr. Logo file from disk  into  the
workspace. You can precede the file name with a drive specifier if the file is
not on the default drive.

Dr.  Logo defines each procedure and variable as it is read in from the  disk.
If a procedure in the workspace has the same name as a procedure read in  from
the disk, the procedure in the workspace is overwritten.

LOAD can load only those files saved with a SAVE command (files with the  file
type  LOG in the disk directory). You can load only one file at a  time.  When
you interrupt LOAD with Ctrl-G or Ctrl-Z, LOAD stops reading from the Dr. Logo
file, and defines only those procedures displayed on your screen.

LOAD can accept a package name as an optional input after the file name.  LOAD
then puts all items read from the file in the input-named package.

Examples:
These  examples assume you have a Dr. Logo file named PIGLATIN on the disk  in
the default drive, and a file named FLY on the disk in drive B.

?load "piglatin
begin.vowelp defined
pig defined
pig.latin defined
?load "b:fly
fly defined
buzz defined
zoom defined


loadpic
-------

Action:
Reads the input-named image file from the disk into the screen.

Syntax:
loadpic fname

Explanation:
```

LOADPIC reads the contents of the input-named image file from disk into the screen. You can precede the file name with a drive specifier if the file is not on the default drive.

LOADPIC can load only those files saved with a SAVEPIC command (files with the file type PC0 in the disk directory). (ROCHE> for SETRES 0 images. But images saved under SETRES 1 are given the PC1 file type... and DIRPIC is then unable to find them! This is clearly a bug. The solution is to use an ambiguous filespec, with .PC? for the file type.) You can load only one file at a time.

Examples:
```
?loadpic "shapes
?dirpic
[SHAPES.PC0 PLAID.PC0 PIGLATIN.PC0]
?dirpic "p???????.pc?
[PLAID.PC0 PIGLATIN.PC0 PLAID.PC1 PIGLATIN.PC1]
```

local

Action:
Makes the input-named variable(s) accessible only to the current procedure and the procedures it calls.

Syntax:
```
local var_name (...)
```

Explanation:
LOCAL makes the input-named variables "local" to the procedure in which the LOCAL expression occurs. Local variables can be changed by the procedure in which the LOCAL expression occurs, or by any procedure it calls.

Without punctuation, LOCAL requires and accepts one input name. LOCAL can accept more inputs when you enclose the LOCAL expression in parentheses ["(" and ")"]. If no other expressions follow the LOCAL expression on the line, you do not need to type the closing right parenthesis [")"].

Most frequently, you will use variables of the same name within calling and called procedures, to pass information between the two procedures. However, you will also find applications where you do not want the called procedure to modify the calling procedure's value of the variable. If calling and called procedures are using variables of the same name, which is unavoidably the case when a procedure calls itself, a LOCAL expression in the called procedure prevents the called procedure from altering the calling procedure's value of the variable.

Examples:
DRAWING.REPORT and DRAW.CIRCLE both use X and Y as variable names. The LOCAL expression in DRAW.CIRCLE prevents DRAW.CIRCLE from modifying DRAWING.REPORT's values of X and Y. Without the LOCAL expression in DRAW.CIRCLE, DRAWING.REPORT would print CIRCLE.FINISHED at the point where the turtle began drawing the circle.

```
?to drawing.report
>make "x (-150)
>make "y 80
>penup setpos list :x :y
>turtletext "drawing.circle
>draw.circle
>setpos list :x :y
>turtletext "circle.finished setheading 0 forward 10
>end
drawing.report defined
?to draw.circle
>(local "x "y)
>make "x 10
>make "y 50
>penup setpos list :x :y setheading 90
>pendown repeat 36 [forward 10 right 10] pu
>end
draw.circle defined
?drawing.report
```

The LOCAL statement in DECIMAL.OF.FRACTION forces Dr. Logo to create a new variable named INVERSE each time DECIMAL.OF.FRACTION calls itself. (ROCHE> I don't know why, but all the sample Dr. Logo programs from Digital Research have opening and closing parentheses ["(" and ")"] surrounding LOCAL statements, even if there is only one variable name, like below.)

```
?to decimal.of.fraction :n
```

```
>(local "inverse)
>make "inverse 1 / :n
>if :n = 1
>   [stop]
>decimal.of.fraction :n - 1
>(print ["1/] :n char 9 :inverse
>end
decimal.of.fraction defined
?decimal.of.fraction 10
1/ 2    0.5
1/ 3    0.333333333333333
1/ 4    0.25
1/ 5    0.2
1/ 6    0.166666666666667
1/ 7    0.142857142857143
1/ 8    0.125
1/ 9    0.111111111111111
1/ 10   0.1
```

log

Action:
Outputs the natural logarithm of the input number.

Syntax:
log n

Explanation:
LOG outputs the natural logarithm of the input number. A natural logarithm is the logarithm to the base e, where e = 2.71828182845905.

Examples:

```
?log 1
0

?log (exp 1)
1

?log (180 / pi)
4.04822696504081
```

log10

Action:
Outputs the base 10 logarithm of the input number.

Syntax:
log10 n

Explanation:
LOG10 outputs the base 10 logarithm of the input number.

Examples:

```
?log10 100
2

?log10 734
2.86569605991607

?log10 3950
3.59659709562646
```

lowercase lc

Action:
Outputs the input word with all alphabetic characters in lowercase.

Syntax:
lowercase word
lc word

Explanation:
LOWERCASE outputs the input word with all alphabetic characters converted to lower case.

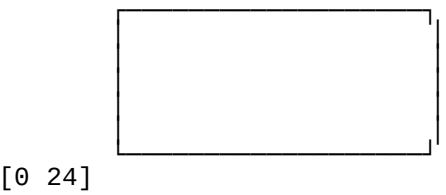
Examples:
```
?lowercase "SOUTH
south

?lowercase "MaryAnn
maryann

?to quiz
>print [Do you like chocolate chip cookies?]
>if "y = lowercase first readquote
>   [print [I do too!]]
>   [print [Wow, I have never met anyone who did not!]
>end
quiz defined
?quiz
Do you like chocolate chip cookies?
YES!!!
I do too!
```

lpen (= LightPEN) (Not a primitive)

Action:
Outputs a coordinate list that indicates the position of the lightpen.

Syntax:
lpen

Explanation:
LPEN outputs the position where the lightpen first touched the screen since the last LPEN expression. LPEN represents the lightpen's position in the same way cursor represents the cursor location on a 40 column text screen. Both LPEN and CURSOR use a coordinate system that originates at the upper left corner of the display.



Examples:
SKETCH shows how to use LPENP to determine whether or not the lightpen has been used before trying to read the lightpen's position with LPEN. If there is no lightpen input waiting to be read when Dr. Logo executes a LPEN expression, LPEN outputs [0 0].

CONVERTX and CONVERTY show a way to convert from lightpen coordinates into turtle graphics coordinates.

```
?to sketch
>forward 2
>wait 10
>if lpenp
>   [turn.to lpen]
>sketch
>end
sketch defined
?to turn.to :position
>convert.x convert.y
>(print :position "= list :x :y)
>setheading towards list :x :y
>sketch
>end
turn.to defined
?to convert.x
>make "x (first :position) * 7.5 - 150
>end
convert.x defined
?to convert.y
>make "y (last :position) * (-8) + 100
>end
convert.y defined
?sketch
```

lpenp (= LightPEN Predicate) (Not a primitive)

Action:
Outputs TRUE if lightpen input is waiting to be read.

Syntax:
lpenp

Explanation:
LPENP outputs TRUE if lightpen input is waiting to be read. Use LPENP in an IF statement to determine whether or not the lightpen has been used, before trying to read the lightpen's position with LPEN (see LPEN).

Examples:
```
?to sketch
>forward 2
>wait 10
>if lpenp
>   [turn.to lpen]
>sketch
>end
sketch defined
?to turn.to :position
>convert.x convert.y
>(print :position "= list :x :y)
>setheading towards list :x :y
>sketch
>end
turn.to defined
?to convert.x
>make "x (first :position) * 7.5 - 150
>end
convert.x defined
?to convert.y
>make "y (last :position) * (-8) + 100
>end
convert.y defined
?sketch
```

lput (= Last PUT)

Action:
Outputs a new object formed by making the first input object the last element in the second input object.

Syntax:
lput object object

Explanation:
LPUT outputs a new object formed by making the first input object the last element in the second input object. Generally, the new object is the same kind of object as the second input object: a word, number, or list. However, when you LPUT a word into a number, LPUT outputs a word.

Examples:
```
?lput "s "plural
plurals

?lput "s [plural]
[plural s]

?lput "cherries [apples bananas]
[apples bananas cherries]

?lput [San Jose] [[San Diego] [San Francisco]]
[[San Diego] [San Franscico] [San Jose]]

?make "extensions [[Hal 33] [John 85]]
?to add.entry :item
>make "extensions lput :item :extensions
>end
add.entry defined
?add.entry [Steve 96]
?:extensions
[[Hal 33] [John 85] [Steve 96]]
```

make

```
  ----

  Action:
  Makes the input object the value of the input-named variable.

  Syntax:
  make varname object

  Explanation:
  MAKE  makes  the  input  object the "contents" or  value  of  the  input-named
  variable. If the input-named variable does not exist, MAKE creates it. If  the
  input-named  variable already exists, MAKE discards its current  contents  and
  gives it the input object as a value.

  MAKE  works just like NAME, except that the order of the inputs are  reversed.
  MAKE  works  by adding the system property .APV with the value  of  the  input
  object to the input-named variable's property list.

  Examples:
  ?make "flavor "chocolate
  ?:flavor
  chocolate

  ?make "chocolate "semi\-sweet
  ?:chocolate
  semi-sweet

  ?thing "flavor
  chocolate

  ?thing :flavor
  semi-sweet

  ?to say.hello
  >print [Hello! What is your name?]
  >make "answer readquote
  >(print [Nice to meet you,] word :answer ".
  >end
  say.hello defined
  ?say.hello
  Hello! What is your name?
  Oscar
  Nice to meet you, Oscar.


  memberp         (= MEMBER Predicate)
  -------

  Action:
  Outputs  TRUE  if  the first input object is an element of  the  second  input
  object.

  Syntax:
  memberp object object

  Explanation:
  MEMBERP  outputs  TRUE if the first input object is an element of  the  second
  input object; otherwise, MEMBERP outputs FALSE.

  Examples:
  ?memberp "y "only
  TRUE

  ?memberp 7 734395
  TRUE

  ?memberp "or "chore
  TRUE

  ?memberp "chocolate [[vanilla] [chocolate] [strawberry]]
  FALSE

  ?memberp [chocolate] [[vanilla] [chocolate] [strawberry]]
  TRUE

  ?to vowelp :object
  >if memberp :object [a e i o u]
  >   [output "TRUE]
  >   [output "FALSE]
  >end
  vowelp defined
```

```
?vowelp "c
FALSE
?vowelp "i
TRUE
```

mouse

Action:
Outputs a list giving the state of the mouse.

Syntax:
mouse

Explanation:
Outputs a list giving the state of the mouse.

Examples:
```
?mouse
[-160 100 FALSE FALSE FALSE]
```

where:

- the first item (-160) is the mouse X-coordinate
- the second item (100) is the mouse Y-coordinate
- the third item (FALSE) is the mouse left button
- the fourth item (FALSE) is the mouse right button
- the fifth item (FALSE) is a predicate saying if the mouse is inside the graphic window

(The above mouse list was gotten from text screen.)

name

Action:
Makes the input object the value of the input-named variable.

Syntax:
name object var_name

Explanation:
NAME makes the input object the "contents" or value of the input-named variable. If the input-named variable does not exist, NAME creates it. If the input-named variable already exists, NAME discards its current contents and gives it the input object as a value.

NAME works just like MAKE, except that the order of the inputs are reversed. NAME works by adding the system property .APV with the value of the input object to the input-named variable's property list.

Examples:
```
?name "flavor "chocolate
?:flavor
chocolate

?name "chocolate "semi\-sweet
?:chocolate
semi-sweet

?thing "flavor
chocolate

?thing :flavor
semi-sweet
```

namep (= NAME Predicate)

Action:
Outputs TRUE if the input word identifies a defined variable.

Syntax:
namep word

Explanation:
NAMEP outputs TRUE if the input word identifies a defined variable. Otherwise, NAMEP outputs FALSE.

Examples:
```
?make "flavor "chocolate
?:flavor
chocolate
?namep "flavor
TRUE

?namep "raspberry
FALSE

?to decrement :name
>if not namep :name
>   [print [Not a variable.]]
>if numberp thing :name
>   [make :name (thing :name) - 1]
>end
decrement defined
?make "age 14
?decrement "age
?:age
13
```

nodes

Action:
Outputs the number of free nodes in the workspace.

Syntax:
nodes

Explanation:
NODES outputs the number of free nodes in the workspace. A node is equal to one byte. (ROCHE> In general, you have 59KB of memory to play with. Four times the size of the workspace of the 8-bits version of Dr. Logo...)

Examples:
```
?nodes
55702
?recycle
?nodes
59524
```

noformat

Action:
Removes comments from the workspace.

Syntax:
noformat

Explanation:
NOFORMAT removes comments from the workspace, to free more nodes. NOFORMAT works by removing any .FMT property pairs from a procedure's poperty list.

Examples:
```
?to bg.cycle :val ; Displays background colors
>if :val = 0
>   [print [Cycle complete.] setbg 0 stop]
>setbg :val
>(print [This is background color number] first screenfacts)
>wait 20000 ; 2 seconds on a 500-MHz PC
>bg.cycle :val - 1
>end
bg.cycle defined
?noformat
?po "bg.cycle :val
bg.cycle :val
if :val = 0 [print [Cycle complete.] setbg 0 stop]
setbg :val
(print [This is background color number] first screenfacts)
wait 20000
bg.cycle :val - 1
end
```

not

```
---

Action:
Outputs  TRUE if the input predicate outputs FALSE, FALSE if  input  predicate
outputs TRUE.

Syntax:
not pred_exp

Explanation:
NOT  reverses  the  value of a predicate expression. If  the  input  predicate
outputs  TRUE,  NOT outputs FALSE. If the input predicate outputs  FALSE,  NOT
outputs TRUE.

Examples:
?not 2 > 1
FALSE

?not 2 < 1
TRUE

?to inverse :n
>if not numberp :n
>   [(print :n [is not a number.]) stop]
>(print "1\ / :n "= 1 / :n
>end
inverse defined
?inverse 5
1 / 5 = 0.2
?inverse "five
five is not a number.


notrace
-------

Action:
Turns off trace monitoring of all or specified procedure(s).

Syntax:
notrace < procname | procname_list >

Explanation:
NOTRACE  turns off the monitoring of procedure execution initiated by a  TRACE
command.

Examples:
?to average :numbers
>make "total 0
>add.up :numbers
>print :total / count :numbers
>end
average defined
?to add.up :list
>if emptyp :list
>   [stop]
>make "total :total + first :list
>add.up butfirst :list
>end
add.up defined
?trace
?average [1 2 3]
[1] Evaluating average
[1] numbers is [1 2 3]
[2] Evaluating add.up
[2] list is [1 2 3]
[3] Evaluating add.up
[3] list is [2 3]
[4] Evaluating add.up
[4] list is [3]
[5] Evaluating add.up
[5] list is []
2
?notrace
?average [1 2 3]
2


nowatch
-------
```

Action:
Turns off watch monitoring of all or specified procedure(s).

Syntax:
nowatch < procname | procname_list >

Explanation:
NOWATCH turns off the monitoring of procedure execution initiated by a WATCH command.

Examples:
```
?to average :numbers
>make "total 0
>add.up :numbers
>print :total / count :numbers
>end
average defined
?to add.up :list
>if emptyp :list
>   [stop]
>make "total :total + first :list
>add.up butfirst :list
>end
add.up defined
?watch
?average [1 2 3]
[1] In average, make "total 0
[1] In average, add.up :numbers
[2] In add.up, if emptyp :list [stop]
[2] In add.up, make "total :total + first :list
[2] In add.up, add.up butfirst :list
[3] In add.up, if emptyp :list [stop]
[3] In add.up, make "total :total + first :list
[3] In add.up, add.up butfirst :list
[4] In add.up, if emptyp :list [stop]
[4] In add.up, make "total :total + first :list
[4] In add.up, add.up butfirst :list
[5] In add.up, if emptyp :list [stop]
[5] In average, print :total / count :numbers
2
?nowatch
?average [1 2 3]
2
```

numberp (= NUMBER Predicate)

Action:
Outputs TRUE if the input object is a number.

Syntax:
numberp object

Explanation:
NUMBERP outputs TRUE if the input object is a number; otherwise, NUMBERP outputs FALSE.

Examples:
```
?numberp 374.926
TRUE

?numberp "six
FALSE

?numberp first [2 4 6 8]
TRUE

?numberp butfirst [2 4 6 8]
FALSE

?to inverse :n
>if not numberp :n
>   [(print :n [is not a number.]) stop]
>(print "1\ / :n "= 1 / :n
>end
inverse defined
?inverse 2
1 / 2 = 0.5
?inverse "two
two is not a number.
```

open

Action:
Open a file or device.

Syntax:
open < fname | device >

Explanation:
Open the file or device to send or receive characters. OPEN must be used before accessing data in a file.

(ROCHE> The devices are: CON: NUL: PRN: and AUX: There is a system message, saying that "Only 4 files can be open".)

Examples:
?open "letters

or
--

Action:
Outputs FALSE if all input predicates outputs FALSE.

Syntax:
or pred_exp pred_exp (...)

Explanation:
OR outputs FALSE if all input predicates output FALSE. If any input predicate outputs FALSE, OR outputs TRUE.

Without punctuation, OR requires and accepts two input objects. OR can accept more or fewer inputs when you enclose the OR expression in parentheses ["(" and ")"]. If no other expressions follow the OR expression on the line, you do not need to type the closing right parenthesis [")"].

Examples:
?or "TRUE "TRUE
TRUE

?or "TRUE "FALSE
TRUE

?or "FALSE "FALSE
FALSE

?(or "FALSE "FALSE "FALSE "TRUE
TRUE

?to weather
>print [What is the temperature today?]
>make "answer readquote
>if or (:answer < 50) (:answer > 80)
> [print [Hmm, sounds uncomfortable!]]
> [print [Sounds nice!]]
>end
weather defined
?weather
What is the temperature today?
92
Hmm, sounds uncomfortable!

?weather
What is the temperature today?
78
Sounds nice!

.out

Action:
Sends number to I/O port.

Syntax:
.out port_n n

Explanation:
Assigns the input number (a byte value) to the specified I/O port. Port numbers range from 0 through 65535. THIS PRIMITIVE SHOULD BE USED WITH CAUTION!

Examples:
?.out 10 33

output op

Action:
Makes the input object the output of the procedure.

Syntax:
output object
op object

Explanation:
OUTPUT makes the input object the output of your procedure. Therefore, OUTPUT is valid only within a procedure. As soon as your procedure outputs an object, control returns to calling procedure or toplevel.

Examples:
```
?to Aurora
>output [Briar Rose]
>end
Aurora defined
?print sentence [Sleeping Beauty's name is] Aurora
Sleeping Beauty's name is Briar Rose

?to vowelp :object
>output memberp :object [a e i o u]
>end
vowelp defined
?vowelp "r
FALSE
?vowelp "e
TRUE

?to coin
>output first shuffle [heads tails]
>end
coin defined
?repeat 3 [show coin]
heads
heads
tails

?to add.up :list
>if emptyp :list
>   [output 0]
>output (add.up butfirst :list) + (first :list)
>end
add.up defined
?add.up [7 6 -2]
11
```

package

Action:
Puts the input-named objects into the input-named package.

Syntax:
package pkgname name | name list

Explanation:
PACKAGE puts the input-named procedures and variables into the package identified by the input package name. If the package does not exist, PACKAGE creates it. If the package already exists, PACKAGE adds the input-named objects to the package. You can enter either a single procedure or variable name, or a list of procedure and variable names as the second input to PACKAGE.

A procedure or variable can be in only one package at a time, but it is easy to move an object from one package to another. A procedure or variable resides in the package to which it was most recently assigned with a PACKAGE command.

PACKAGE works by adding to or changing properties in the property lists of the named objects. PACKAGE adds the .PKG property to the package name's property list, and the .PAK property to the property list of each input-named object.

Examples:
These examples assume you have the following in your workspace: six procedures named DRAW, CIRCLE, SQUARE, TRIANGLE, ADDUP, and GROW; and five variables named SIZE, BIG, SMALL, MEDIUM, and PENSTATE.

```
?pots
to draw
to circle :size
to square :size
to triangle :size
to add.up :list
to grow :size
?package "shapes [circle square triangle]
?pons
size is 100
big is 80
small is 20
medium is 40
penstate is [pd 1]
?package "sizes [big medium small]
?popkg
sizes
  "medium (VAL)
  "small (VAL)
  "big (VAL)
shapes
  to square :size
  to triangle :size
  to circle :size
?package "sizes "grow
?popkg
sizes
  to grow :size
  "medium (VAL)
  "small (VAL)
  "big (VAL)
shapes
  to square :size
  to triangle :size
  to circle :size
```

paddle (Not a primitive)

Action:
Outputs a number that represents a paddle (joystick) input coordinate.

Syntax:
paddle n

Explanation:
You can use PADDLE to read the position of a paddle or joystick. Dr. Logo supports up to two paddles or joysticks. If your IBM Personal Computer does not have a paddle or joystick when Dr. Logo encounters a PADDLE command, PADDLE returns 0. The range of values that paddle outputs depends on your particular paddle or joystick.

The number you input to PADDLE tells Dr. Logo which coordinate you want PADDLE to output. The input number must be in the range 0 to 3:

```
        PADDLE 0 outputs the X-coordinate of paddle1
        PADDLE 1 outputs the Y-coordinate of paddle1

        PADDLE 2 outputs the X-coordinate of paddle2
        PADDLE 3 outputs the Y-coordinate of paddle2
```

where PADDLE1 is the first paddle or joystick, and PADDLE2 is the second paddle or joystick.

Examples:
DRAW allows the user to guide the turtle with the joystick. DRAW contains calculations that convert X and Y coordinates returned by PADDLE into turtle coordinates. PADDLE returns a 190-number range for the X-coordinate of this particular joystick, and a 144-number range for the Y-coordinate of this particular joystick. AMOUNT smooths the turtle's response to the joystick's commands.

```
?to draw
>repeat 10000
>  [make "xin paddle 0
>   make "yin paddle 1
>   make "xin int ((:xin * (300 / 190)) - 150)
>   make "yin int ((:yin * (-200 / 144)) + 90)
>   setheading towards list :xin :yin
>   forward (amount * 0.1)
>   if buttonp 0 [stop]
>   if buttonp 1 [clean]
>end
draw defined
?to amount
>output int sqrt
>   ((abs :xin) * (abs :xin) +
>   ((abs :yin) * (abs :yin)))
>end
amount defined
?draw
```

pal (= PALette of colors)

Action:
Output a list of RGB values for a given pen index.

Syntax:
pal pen_index

Explanation:
Outputs a list of color components for the given pen_index. The range of values for pen_index are 0 through 255. Item 1 of the list represents the Red component of the color, item 2 represents the Green component, and item 3 is for Blue. The range of values for the RGB values are 0 through 63.

Examples:

```
?pal 0
[0 0 0]    ; Black
?pal 1
[0 0 42]   ; Blue
?pal 2
[0 42 0]   ; Green
?pal 3
[0 42 42]  ; Cyan
?pal 4
[42 0 0]   ; Red
?pal 5
[42 0 42]  ; Magenta
?pal 6
[42 21 0]  ; "Yellow"
?pal 7
[42 42 42] ; "White"
```

(ROCHE> To get the true colors, use 63. By default, Dr. Logo uses 42 for normal intensity, and 21 for half intensity. To get true Yellow, use [63 63 0]. And [63 63 63] for White. Normally, once you have Red, Green, and Blue, you are served. Personally, I prefer a Black background. You can use White or Yellow for the turtle.)

pause

Action:
Suspends the execution of the current procedure, to allow interaction with the interpreter or editor.

Syntax:
pause

Explanation:
PAUSE suspends the execution of the current procedure, and lets you interact with the interpreter or editor. Therefore, the PAUSE command is valid only within a procedure. When Dr. Logo encounters a PAUSE command, it suspends execution of the procedure, displays a "Pausing..." message, and allows you to interact with the interpreter. To end the PAUSE, enter CO.

PAUSE is one of the three ways you can cause a pause in the execution of your procedure.

A PAUSE expression causes a "breakpoint" in your procedure. For example, you can insert a PAUSE command just before an expression that has been regularly causing an error. When you execute the procedure, Dr. Logo will pause before the problem expression. During the pause, you can experiment by entering variations of the problem expression, and examine or change local variables whose values will be discarded when the procedure finishes executing. Enter CO to end the PAUSE and continue execution.

Examples:

```
?make "size 100
?to nautilus :size
>if (remainder :size 5) = 0
>   [pause]
>repeat 36
>  [forward :size right 10]
>right 15
>nautilus :size + 0.5
>end
nautilus defined
?clearscreen right 180
?nautilus 1
Pausing... in nautilus: [if [remainder :size 5) = 0 [pause]]
nautilus ?:size
5
nautilus ?co
Pausing... in nautilus: [if [remainder :size 5) = 0 [pause]]
nautilus ?:size
5.5
nautilus ?co
Pausing... in nautilus: [if [remainder :size 5) = 0 [pause]]
nautilus ?:size
10
nautilus ?stop
?:size
100
```

pendown pd

Action:
Puts the turtle's pen down; the turtle resumes drawing.

Syntax:

```
pendown
pd
```

Explanation:
PENDOWN makes the turtle put its pen down and begin drawing in the current pen color. The pen is down when you start Dr. Logo. Use PENDOWN to resume normal drawing after a PENUP, PENERASE, or PENREVERSE.

Examples:

```
?to pen
>output list item 4 turtlefacts item 5 turtlefacts
>end
pen defined
?pen
[pu 2]
?pendown pen
[pd 2]

?to halo :size
>repeat 36
>  [forward :size * 0.5 penup
>   forward :size * 0.1 pendown
>   forward :size * 0.2 penup
>   forward :size * 0.1 pendown
>   forward :size * 0.1 penup
>   back :size right 10 pendown]
>ht
>end
halo defined
?halo 100
```

penerase pe

Action:

Puts the turtle's eraser down; turtle erases drawn lines.

Syntax:
penerase
pe

Explanation:
PENERASE makes the turtle put its eraser down. With the eraser down, the turtle erases any drawn lines it passes over. To lift the eraser, use PENUP or PENDOWN.

Examples:
```
?to pen
>output list item 4 turtlefacts item 5 turtlefacts
>end
pen defined
?pen
[pd 2]
?penerase pen
[pe 2]

?to move.triangle
>clearscreen
>hideturtle
>repeat 36
>  [triangle40 penerase triangle40 pendown setpc 2 right 10]
>end
move.triangle defined
?to triangle40
>repeat 3
>  [forward 40 right 120]
>end
triangle40 defined
?move.triangle
```

penreverse px

Action:
Makes turtle erase where lines are drawn, and draw where there are no lines.

Syntax:
penreverse
px

Explanation:
PENREVERSE makes the turtle draw where there are no lines, and erase where there are lines. To stop using the reversing pen, enter PENUP or PENDOWN.

Examples:
```
?to pen
>output list item 4 turtlefacts item 5 turtlefacts
>end
pen defined
?pen
[pd 2]
?penreverse pen
[px 2]

?to moving.triangle
>clearscreen
>hideturtle
>penreverse
>repeat 26
>  [triangle40 triangle40 right 10]
>end
moving.triangle defined
?to triangle40
>repeat 3
>  [forward 40 right 120]
>end
triangle40 defined
?moving.triangle
```

penup pu

Action:
Picks the turtle's pen up; the turtle stops drawing.

Syntax:
penup
pu

Explanation:
PENUP makes the turtle pick its pen up and stop leaving a trace of its path.
Use PENDOWN to resume normal drawing after a PENUP command.

Examples:
```
?to pen
>output list item 4 turtlefacts item 5 turtlefacts
>end
pen defined
?pen
[pd 2]
?penup pen
[pu 2]

?to halo :size
>repeat 36
>  [forward :size * 0.5 penup
>   forward :size * 0.1 pendown
>   forward :size * 0.2 penup
>   forward :size * 0.1 pendown
>   forward :size * 0.1 penup
>   back :size right 10 pendown]
>ht
>end
halo defined
?pendown halo 100
```

pi
--

Action:
Outputs the value of pi.

Syntax:
pi

Explanation:
PI outputs the value of pi: 3.14159265358979.

Examples:
```
?(print char 227 "= pi     ; Code Page 850
π = 3.14159265358979

?to find.radius :n
>repeat 180
>  [forward :n left 2]
>make "radius (180 * :n) / (2 * pi)
>(print [Radius =] :radius
>left 90 forward radius
>end
find.radius defined
?penup sety -90 pendown right 90
?find.radius
radius = 85.9436692696235
```

piece

Action:
Outputs an object that contains the specified elements of the input object.

Syntax:
piece n n object

Explanation:
PIECE outputs an object made up of the elements of the input object. The two input numbers specify which elements are to be included in the output object. The range you specify is inclusive; for example, when you specify PIECE 3 8, the third and eight elements are included in the output object.

Examples:
```
?piece 4 7 "Kensington
sing
```

```
?piece 2 4 [Nana John Michael Wendy Tinkerbell]
[John Michael Wendy]

?to last.part :name
>if memberp ". :name
>   [make "readlist where]
>   [(print :name [in wrong format.]) stop]
>make "r2 count :name
>output piece (:r1 + 1) :r2 :name
>end
last.part defined
?last.part "Peter.Pan
Pan


pkgall  (= PacKaGe ALL)
------

Action:
Puts all procedures and variables not in packages into the specified package.

Syntax:
pkgall pkgname

Explanation:
PKGALL  assigns  all procedures and variables not already in packages  to  the
input-named package. If the package does not exist, PKGALL creates it. If  the
package does exist, PKGALL adds the unpackaged procedures and variables to the
existing package.

Examples:
These examples assume that you have the following in your workspace: a  buried
package named SHAPES that contains three procedures named CIRCLE, SQUARE,  and
TRIANGLE;  a  buried package named SIZES that contains three  variables  named
BIG,  MEDIUM,  and  SMALL;  and three unpackaged  objects:  a  variable  named
PEN.STATE and two procedures named DRAW and GROW.

?popkg
shapes is buried
  to square :size
  to circle :size
  to triangle :size
sizes is buried
  "medium (VAL)
  "small (VAL)
  "big (VAL)
?pots
to grow :size
to draw
?pons
pen.state is [pd 2]
?pkgall "other
?popkg
other
  to grow :size
  to draw
  "pen.state (VAL)
shapes is buried
  to square :size
  to circle :size
  to triangle :size
sizes is buried
  "medium (VAL)
  "small (VAL)
  "big (VAL)


plist   (= Property LIST)
-----

Action:
Outputs the property list of the input-named object.

Syntax:
plist name

Explanation:
PLIST  outputs  the  property  list of  the  input-named  object.  Section  5,
"Property Lists, Workspace, and Disks", describes property lists, and  defines
the  properties Dr. Logo gives to objects in the workspace. To understand  the
examples  below, remember that a property list is made up of  property  pairs.
```

The first element of the pair is the property; the second, its value. Dr. Logo gives defined properties the .DEF property, and defined variables the .APV property.

Examples:
These examples assume you have, in your workspace, a procedure named TRIANGLE and a variable named STAR.

```
?plist "triangle
[.DEF [[size] [repeat 3 [forward :size right 120]]]]

?plist "star
[.APV evening]

?to remove :name
>if emptyp plist :name
>   [stop]
>make "prop first plist :name
>run (sentence "remprop "quote :name "quote :prop)
>remove :name
>end
remove defined
?make "bird "blue
?:bird
blue
?remove "bird
?:bird
bird has no value
```

po (= Print Out)

Action:
Displays the definition(s) of the specified procedure(s).

Syntax:
po procname | procname_list

Explanation:
PO displays the definition of a procedure, or the definitions of a list of procedures. When you do not have your procedures grouped in an appropriate package and cannot use POPS, you can input a list of procedure names to PO to display the definitions of a group of procedures.

Examples:

```
?to triangle :size
>repeat 3
>  [forward :size right 120]
>end
triangle defined
?to square :size
>repeat 4
>  [forward :size right 90]
>end
square defined
?po "square
to square :size
repeat 4
  [forward :size right 90]
end
?po [triangle square]
to triangle :size
repeat 3
  [forward :size right 120]
end
to square :size
repeat 4
  [forward :size right 90]
end
```

poall (= Print Out ALL)

Action:
Displays the definitions of all procedures and variables in the workspace or input-named package(s).

Syntax:
poall < pkgname | pkgname list >

Explanation:
Without an input, POALL displays the definitions of all procedures and variables in the workspace. POALL with a package name as input displays the definitions of all procedures and variables in the input-named package. You can input a list of package names to POALL to display the contents of several packages.

Examples:
These examples assume you have two packages named SHAPES and SIZES in your workspace.

```
?poall
```

POALL displays the definitions of all procedures and variables in the workspace.

```
?poall "shapes
```

POALL displays the definitions of all procedures and variables in the package named SHAPES.

```
?poall [shapes sizes]
```

POALL displays the definitions of all procedures and variables in the packages named SHAPES and SIZES.

pocall (= Print Out CALLed procedure)

Action:
Displays the names of the procedures called by the input-named procedure.

Syntax:
pocall procname

Explanation:
POCALL displays the names of the procedures called by the input-named procedure. The name of each called procedure is indented on the line beneath the name of the procedure that calls it. The procedure names appear in the order in which they are called.

Examples:

```
?to wheel
>repeat 12
>  [flag left 30]
>end
wheel defined
?to tri
>repeat 3
>  [right 120 forward 25]
>end
tri defined
?to flag
>forward 50 tri back 50
>end
flag defined
?pocall "wheel
wheel
  flag
    tri
?to add.up :list
>if emptyp :list
>   [output 0]
>output (add.up butfirst :list) + (first :list)
>end
add.up defined
?pocall "add.up
add.up :list
  add.up :list
```

pons (= Print Out NameS)

Action:
Displays the names and values of all variables in the workspace or specified package(s).

Syntax:

pons < pkgname | pkgname_list >

Explanation:
Without an input, PONS displays the names and values of all the variables defined in the workspace. With a package name as input, PONS displays the names and values of all the variables contained in the specified package. You can input a list of package names to PONS to display variable names and values from several packages.

Examples:
These examples assume you have the following in your workspace: three variables named BIG, MEDIUM, and SMALL in a package named sizes; two variables named BIRD and SNAKE in a package named ANIMALS; and a variable named PENSTATE.

```
?to pen
>output list item 4 turtlefacts item 5 turtlefacts
>end
pen defined
make "pen.state pen
?pons
medium is 40
small is 20
big is 80
bird is blue
pen.state is [pd 2]
snake is green
?pons "sizes
medium is 40
small is 20
big is 80
?pons [sizes animals]
medium is 40
small is 20
big is 80
bird is blue
snake is green
```

popkg (= Print Out PacKaGe)

Action:
Displays the name and contents of each package in the workspace.

Syntax:
popkg < pkgname | pkgname_list >

Explanation:
POPKG displays the name and contents of each package in the workspace. The names of variables and procedures in the package are indented under the package name. Variable names are quoted, and are followed by (VAL). Procedure names are preceded by the word TO and are followed by their inputs, if any. Object names that have properties are quoted, and are followed by (PROP). POPKG includes the names and contents of buried packages, and indicates that the package is buried.

POPKG accepts a package name or a list of package names as input. When you specify a package or list of packages, POPKG displays the contents of the specified package only.

Examples:
These examples assume you have four packages named WHEEL, SIZES, AVERAGE, and SHAPES in your workspace.

```
?popkg
wheel
  to wheel
  to flag
  to tri
sizes
  "big (VAL)
  "small (VAL)
  "medium (VAL)
average
  to add.up :list
  to average :list
shapes
  to circle :size
  to triangle :size
  to square :size
```

```
?bury "average
?popkg
wheel
  to wheel
  to flag
  to tri
sizes
  "big (VAL)
  "small (VAL)
  "medium (VAL)
average is buried
  to add.up :list
  to average :list
shapes
  to circle :size
  to triangle :size
  to square :size
```

pops (= Print Out ProcedureS)

Action:
Displays the names and definitions of all procedures in the workspace or specified package(s).

Syntax:
pops < pkgname | pkgname_list >

Explanation:
Without an input, POPS displays the names and definitions of all the procedures defined in the workspace. With a package name as input, POPS displays the names and definitions of all the procedures contained in the specified package. You can input a list of package names to POPS to display procedure names and definitions from several packages.

Examples:
These examples assume you have the following in your workspace: three procedures named WHEEL, FLAG, and TRI in a package named WHEEL; two procedures named AVERAGE and ADDUP in a package named average; and a procedure named SQUARE.

```
?pops
to wheel
repeat 12
  [flag left 30]
end
to square :size
repeat 4
  [forward :size right 90]
end
to tri
repeat 3
  [right 120 forward 25]
end
to flag
forward 50 tri back 50
end
to add.up :list
if emptyp :list
   [output 0]
output (add.up butfirst :list) + (first :list)
end
to average :list
output (add.up :list) / (count :list)
end
?pops "wheel
to wheel
repeat 12
  [flag left 30]
end
to tri
repeat 3
  [right 120 forward 25]
end
to flag
forward 50 tri back 50
end
?pops [wheel average]
to wheel
repeat 12
```

```
    [flag left 30]
  end
  to tri
  repeat 3
    [right 120 forward 25]
  end
  to flag
  forward 50 tri back 50
  end
  to add.up :list
  if emptyp :list
     [output 0]
  output (add.up butfirst :list) + (first :list)
  end
  to average :list
  output (add.up :list) / (count :list)
  end


  poref   (= Print Out REFerenced procedures)
  -----

  Action:
  Displays the names of the procedures that call the input-named procedure(s).

  Syntax:
  poref procname | procname_list

  Explanation:
  POREF  displays  the  names  of  any  procedures  that  call  the  input-named
  procedure.  You can use POREF to check that other procedures will be  affected
  when you change the input-named procedure.

  Examples:
  ?to wheel
  >repeat 12
  >  [flag left 30]
  >end
  wheel defined
  ?to tri
  >repeat 3
  >  [right 120 forward 25]
  >end
  tri defined
  ?to flag
  >forward 50 tri back 50
  >end
  flag defined
  ?poref "tri
  to flag
  ?poref "flag
  to wheel
  ?to add.up :list
  >if emptyp :list
  >   [output 0]
  >output (add.up butfirst :list) + (first :list)
  >end
  add.up defined
  ?to average :list
  >output (add.up :list) / (count :list)
  >end
  average defined
  ?poref "add.up
  to average :list
  to add.up :list


  pos
  ---

  Action:
  Outputs a list that contains the coordinates of the turtle's current position.

  Syntax:
  pos

  Explanation:
  POS outputs a two-element list that identifies the turtle's current  position.
  The  format  of the list is suitable for input to SETPOS. When the  turtle  is
  plotting  on screen, the first element of the list is the X-coordinate in  the
  range  -150 to +149, and the second element is the Y-coordinate in  the  range
```

-99 to +100. When the turtle is off the screen, either the X or Y coordinate or both will be greater than the visible range.

Examples:

```
?clearscreen pos
[0 0]

?to save.place
>make "place pos
>end
save.place defined
?to return
>setpos :place
>end
return defined
?setpos list (random 150) (random 100)
?pos
[90 22]
?save.place
?setpos list (random 150) (random 100)
?pos
[55 73]
?return

?to find.center :n
>repeat 180
>  [forward :n left 2]
>make "radius (180 * :n) (2 * pi)
>left 90 forward :radius
>show pos
>end
find.center defined
?penup back 80 right 90 back 90 pd
?find.center 2
[-89.9891 -22.6934]
```

potl (= Print Out Top Level procedures)

Action:
Displays the names of the procedures that are not called by any other procedures in the workspace.

Syntax:

```
potl
```

Explanation:
POTL displays the names of toplevel procedures, procedures that are not called by any other procedures in the workspace, and are executed only by direct command at the interpreter's ? prompt. Use POTL to discover which procedures do not affect other procedures, or which procedure names are good candidates for subsequent POCALL command.

Examples:

```
?to wheel
>repeat 12
>  [flag left 30]
>end
wheel defined
?to tri
>repeat 3
>  [right 120 forward 25]
>end
tri defined
?to flag
>forward 50 tri back 50
>end
flag defined
?to add.up :list
>if emptyp :list
>   [output 0]
>output (add.up butfirst :list) + (first :list)
>end
add.up defined
?to average :list
>output (add.up :list) / (count :list)
>end
average defined
?potl
to wheel
```

```
to average :list
```

pots (= Print Out TitleS)

Action:
Displays the names and inputs of all procedures in the workspace or specified package(s).

Syntax:
pots < pkgname | pkgname_list >

Explanation:
Without an input, POTS displays the names and inputs of all the unburied procedures in the workspace. With a package name as input, POTS prints the names and inputs of all the procedures in the package, even if the package is buried. You can input a list of package names to POTS to display the names and inputs of procedure in several packages.

Examples:
This example assumes you have the following in your workspace: three procedures named WHEEL, FLAG, and TRI in a package named WHEEL; two procedures named average and ADDUP in a package named AVERAGE; and three procedures named SQUARE, CIRCLE, and TRIANGLE.

```
?pots
to wheel
to tri
to square :size
to flag
to add.up :list
to average :list
to circle :size
to triangle :size
?pots "average
to add.up :list
to average :list
?pots [average wheel]
to wheel
to flag
to tri
to add.up :list
to average :list
```

pprop (= Put PROperty Pair)

Action:
Puts the input property pair into the input-named object's property list.

Syntax:
pprop name prop object

Explanation:
PPROP puts a property pair into the input-named object's property list. PPROP requires three inputs: the name of the object to which the property is to be added, the name of the property, and the object that is to be the value of the property. PPROP makes the input property pair the first pair in the input-named object's property list.

You can use PPROP to add standard Dr. Logo system properties to a property list. For example, if you use PPROP to add the .APV property to an object's property list, you create a variable the same as if you had used NAME or MAKE.

You must use PPROP to add your own special properties to a property list. However, the erasing commands (ERASE, ERN, ERALL, ERPS, and ERNS) remove only standard Dr. Logo system properties from property lists. This means that, if you have assigned a non-standard property to an object's property list, that object will still be in the workspace after an ERALL commmand. Use REMPROP to remove your special properties from a property list.

Examples:

```
?pprop "dungeonmaster ".APV "Scott
?:dungeonmaster
Scott

?to make.character :name
>make "abilities [strength intelligence wisdom dexterity constitution charisma]
```

```
>make   "other   [class  hitpoints  armorclass  alignment   level   experience
goldpieces]
>assign :abilities
>assign :other
>(print [You have given] :name [the following characteristics:]
>print plist :name
>end
make.character defined
?to assign :list
>if emptyp :list
>   [stop]
>print word first :list "?
>make "value readlist
>pprop :name first :list :value
>assign butfirst :list
>end
assign defined
?make.character "Borg
Strength?
17
Intelligence?
8
(...)
You have given Borg the following characteristics:  goldpieces [10] experience
[0] level [1] alignment [lawful] armorclass [3] hitpoints [6] class  [fighter]
charisma  [6]  constitution [15] dexterity [7] wisdom  [10]  intelligence  [8]
strength [17]
```

pps (= Property PairS)

Action:
Displays the non-system property pairs of all objects in the workspace or specified packages.

Syntax:
pps < pkgname | pkgname_list >

Explanation:
Without an input, PPS prints out the special property pairs you have assigned with a PPROP command to any object in the workspace. If you have not assigned any special properties, PPS prints nothing. With a package name as input, pps prints the special property pairs from the specified package, even if the package is buried. You can input a list of package names to PPS to display the names and inputs of procedures in several packages.

Examples:

```
?pprop "Kathy "extension 82
?pps
Kathy's extension is 82
```

prec

Action:
Outputs the number of significant digits.

Syntax:
prec

Explanation:
Outputs the number of significant digits displayed in an output number.

Examples:

```
?prec
6
```

primitivep (= PRIMITIVE Predicate)

Action:
Outputs TRUE if the input object is a primitive name.

Syntax:
primitivep object

Explanation:
PRIMITIVEP outputs TRUE if the input object is a primitive name; otherwise,

PRIMITIVEP outputs FALSE.

Examples:
?primitivep "phone
FALSE

?primitivep "home
TRUE

print pr

Action:
Displays the input object(s) on the text screen.

Syntax:
print object (...)
pr object (...)

Explanation:
PRINT displays the input object on the screen, followed by a Carriage Return. PRINT removes the outer square brackets ("[" and "]") from an input list. Compare print with TYPE and SHOW. You can input any number of objects by preceding them with a left parenthesis ["("]. When preceded by a left parenthesis, PRINT displays all its inputs on the same line, and follows only the last input with a Carriage Return.

Examples:
?print [This is a message.]
This is a message.

?make "variable "silly
?(print [This is a] :variable "message.)
This is a silly message.

proclist (= PROCedure LIST)

Action:
Outputs a list that contains the names of all defined procedures.

Syntax:
proclist

Explanation:
PROCLIST outputs a list that contains the names of all procedures currently defined in the workspace.

Examples:
This example assumes you have the following procedures in your workspace: WHEEL, FLAG, TRI, ADDUP, and AVERAGE.

```
?pots
to wheel
to flag
to tri
to add.up
to average
?proclist
[wheel flag tri add.up average]
?package "current proclist
?popkg
current
  to wheel
  to flag
  to tri
  to add.up :list
  to average :list
```

product *

Action:
Outputs the product of the input numbers.

Syntax:
product n n (...)
* n n (...)

Explanation:
PRODUCT outputs the product of the input numbers. PRODUCT is equivalent to the * arithmetic operator.

Without punctuation, PRODUCT requires and accepts two input objects. PRODUCT can accept more or fewer inputs when you enclose the PRODUCT expression in parentheses ["(" and ")"]. If no other expressions follow the PRODUCT expression on the line, you do not need to type the closing right parenthesis [")"].

Examples:
```
?product 7 6
42

?* 7 6
42

?(product 2 pi 5)
31.4159265358979

?(* 2 pi 5
31.4159265358979

?(product 7)
7

?(* 7)
7

?to cube :n
>output (product :n :n :n)
>end
cube defined
?cube 3
27
```

quotient

Action:
Outputs the quotient of the two input numbers.

Syntax:
quotient n n

Explanation:
QUOTIENT outputs the number that results when the first input number is divided by the second. QUOTIENT truncates any input decimal number to an integer. Unlike the / operator, if the result ends in a decimal fraction, QUOTIENT truncates the result to an integer.

Examples:
```
?quotient 10 4
2

?10/4
2.5

?quotient -10 5
-2

?quotient 5 0
Can't divide by zero

?quotient 5 .9
Can't divide by zero
```

radians

Action:
Outputs the number of radians in the input number of degrees.

Syntax:
radians degrees_n

Explanation:
RADIANS outputs the number of radians in the input number of degrees, where

```
  degrees = radians * (180 / pi).

  Examples:
  ?radians 90
  1.5707963267949

  ?radians 180
  3.14159265358979

  ?pi
  3.14159265358979

  ?radians 450
  7.85398163397449


  random
  ------

  Action:
  Outputs a random non-negative integer less than the input number.

  Syntax:
  random n

  Explanation:
  RANDOM outputs a random non-negative integer less than the input number.  This
  means  random  might output any integer from zero to one less than  the  input
  number. For example, random 2 will output either 0 or 1.

  Examples:
  ?random 10
  4

  ?repeat 20 [(type random 10 char 9)]
  3   0    2    3    1    4    6    4    4   7    9    4    5    1    8    2

  ?to draw.spinner :n
  >if (:n > 330)
  >   [stop]
  >penup right 15 forward 25
  >turtletext (word (int (:n / 36)))
  >back 25 right 21 forward 25 pendown forward 5 penup back 30
  >draw.spinner :n + 36
  >end
  draw.spinner defined
  ?to spin
  >repeat 72 * random 6
  >  [right 5]
  >repeat random 45
  >  [right 5]
  >print int (heading / 36)
  >end
  spin defined
  ?draw.spinner 0
  ?spin
  1


  readchar rc
  -----------

  Action:
  Outputs the first character typed at the keyboard.

  Syntax:
  readchar
  rc

  Explanation:
  READCHAR  outputs  the  first character typed at the  keyboard.  READCHAR  can
  output any character you can type, including control characters, except  Ctrl-
  G. Ctrl-G stops execution and returns to toplevel.

  During  a procedure's execution, READCHAR does not move the cursor or  display
  the  input  character on the screen. If no character is waiting  to  be  read,
  REACHAR  waits until you type something. If something is waiting to  be  read,
  READCHAR  immediately outputs the first character in the keyboard buffer.  The
  description of KEYP tells how to check if a character is waiting to be read.

  Examples:
```

```
?readchar
z

?to quiz
>print [Do you know any Martians?]
>if lowercase readchar = "y
>   [print [Takes one to know one!]]
>   [print [Me either!]]
>end
quiz defined
?quiz
Do you know any Martians? y
Takes one to know one!

?to poly.cycle :n
>(print [Press any key to draw a] :n [sided figure.])
>sink readchar cs
>repeat :n
>  [forward 40 right (360 / :n)]
>if :n > 8
>   [stop]
>poly.cycle :n + 1
>end
poly.cycle defined
?to sink :object
>end
sink defined
?poly.cycle 3
Press any key to draw a 3 sided figure.
Press any key to draw a 4 sided figure.
Press any key to draw a 5 sided figure.
Press any key to draw a 6 sided figure.
Press any key to draw a 7 sided figure.
Press any key to draw a 8 sided figure.
Press any key to draw a 9 sided figure.


readeofp        (= READ End-Of-File Predicate)
--------

Action:
Outputs TRUE if end of data file reached.

Syntax:
readeofp

Explanation:
Outputs TRUE if the current data file is at the end; otherwise, outputs FALSE.
You must use OPEN and SETREAD before you use READEOFP.

(ROCHE> There is a system message, saying that "Only 4 files can be open".)

Examples:
?open "telnos
?setread "telnos
?readeopf
FALSE


reader
------

Action:
Outputs current open file name.

Syntax:
reader

Explanation:
Outputs the current file name that is open for reading.

(ROCHE> There is a system message, saying that "Only 4 files can be open".)

Examples:
?reader
[A:BOOKLIST.DAT]


readlist rl
-----------
```

Action:
Outputs a list that contains a line typed at the keyboard.

Syntax:
readlist
rl

Explanation:
READLIST outputs a list that contains a line typed at the keyboard. READLIST always displays the input line on the screen before outputting the list.

READLIST can read a line only after you press the Carriage Return (Enter) key. If no line is waiting to be read, READLIST waits for something to be typed. If a line is waiting to be read, READLIST outputs the line immediately. The description of KEYP tells how to check if something is waiting to be read.

Examples:
```
?readlist
yippee ti yi yo
[yippee ti yi yo]

?repeat 5 readlist
forward 40 right 72

?to interpret
>print [What next, boss?]
>run readlist
>interpret
>end
interpret defined
?to my.message
>catch "error [interpret]
>(print "Oops, first butfirst error [!!!])
>print [What do you want to do about that?]
>run readlist
>my.message
>end
my.message defined
?my.message
What next, boss?
fence
What next, boss?
forward 200
Oops, Turtle out of bounds !!!
What do you want to do about that?
back 100
What next, boss?
to quit
Oops, I don't know to to !!!
What do you want to do about that?
stop
```

readquote rq

Action:
Outputs a word that contains a line typed at the keyboard.

Syntax:
readquote
rq

Explanation:
READQUOTE outputs a word that contains a line typed at the keyboard. READQUOTE always displays the input line on the screen. READQUOTE can read a line only if the line is ended with a Carriage Return (Enter) keystroke.

If the line contains words separated by spaces, READQUOTE inserts backslant characters ("\") in front of the spaces, so that the spaces are treated as literal characters. This makes the line one word. You can see the backslant characters if you load an object created with READQUOTE into the editor.

When no line is waiting to be read, READQUOTE waits until something is typed. If a line is waiting to be read, READQUOTE outputs the line immediately. You can use keyp to see if a line is waiting to be read.

Examples:
```
?readquote
I don't think we are in Kansas anymore...
I don't think we are in Kansas anymore...
```

recycle

Action:
Reorganizes the workspace, to free as many nodes as possible.

Syntax:
recycle

Explanation:
RECYCLE reorganizes and cleans up the workspace, freeing as many nodes as possible. After RECYCLE, NODES can tell you how much of the workspace is filled with procedures, variables, and other defined objects.

RECYCLE works by calling the garbage collector. Dr. Logo automatically calls the garbage collector when it is needed. However, if the timing of one of your procedures is critical, you can run RECYCLE before executing the procedure, to ensure that your procedure will not be interrupted by an automatic garbage collection. Section 5, "Property Lists, Workspace, and Disks", describes the garbage collector, and how it reorganizes and cleans up the workspace.

Examples:
```
?nodes
55702
?recycle
?nodes
59524

?to spi :side :angle :inc
>forward :side
>right :angle
>spi :side + :inc :angle :inc
>end
spi defined
?spi 10 100 2
I'm out of space in spi: spi :side + :inc :angle :inc
!recycle
?nodes
55380
```

redefp (= REDEFine Predicate) (Not a primitive)

Not a primitive, but a system variable.

When REDEFP is TRUE, primitives can be redefined. But the new definition will take all the characteristics of the primitive; it cannot be printed out or edited.

remainder

Action:
Outputs the integer remainder obtained when the first input number is divided by the second.

Syntax:
remainder n n

Explanation:
REMAINDER outputs the integer that is the remainder obtained when the first input number is divided by the second input number.

Examples:
```
?remainder 7 2
1

?to evenp :n
>if 0 = remainder :n 2
>   [output "TRUE]
>   [output "FALSE]
>end
evenp defined
?evenp 11
FALSE
?evenp 6
TRUE
```

remprop (= REMove PROPerty)

Action:
Removes the specified property from the input-named object's property list.

Syntax:
remprop name prop

Explanation:
REMPROP removes the specified property and its value from the input-named object's property list. You may use REMPROP to remove any non-system properties you put in an object's property list with PPROP. You must remove all properties from an object's property list to completely remove the object from the workspace.

Examples:
```
?pons
horse1 is Tinderbox
horse2 is Muffy
?plist "horse1
[color bay .APV Tinderbox]
?ern "horse1
?plist "horse1
[color bay]
?remprop "horse1 "color
?plist "horse1
[]
?plist "horse2
[color [liver chestnut] class [hunter jumper] .APV Muffy]

?to remove :name
>if emptyp plist :name
>   [stop]
>make "prop first plist :name
>run (sentence "remprop "quote :name "quote :prop)
>remove :name
>end
remove defined
?remove "horse2
?plist "horse2
[]
```

repeat

Action:
Executes the input instruction list the input number of times.

Syntax:
repeat n instr_list

Explanation:
REPEAT executes the input instruction list the input number of times. The input number must be positive. If the input number is not an integer, REPEAT truncates it to an integer.

If you want a procedure to execute continuously, such as the DRAW procedure shown in the description of PADDLE, you can use REPEAT 1/0 instead of a recursive call. This minimizes interruptions from the garbage collector. We recommend that you use a recursive call as you write your procedure, so that you can take advantage of Dr. Logo's line-by-line debugging facilities. Then, when your procedure is working correctly, remove the recursive call and enclose the entire procedure definition within a single line: REPEAT 1/0 with a lengthy input instruction list.

Examples:
```
?repeat 100 [print [I will not chew gum in class.]
I will not chew gum in class.
I will not chew gum in class.
I will not chew gum in class.
I will not chew gum in class.
I will not chew gum in class.
I will not chew gum in class.
(...)

?make :sides 3
?repeat :sides [forward 40 right (360 / :sides)]
```

```
?to spi :side :angle :inc
>forward :side
>right :angle
>spi :side + :inc :angle :inc
>end
spi defined
?spi 10 100 2
I'm out of space in spi: spi :side + :inc :angle :inc
!recycle
?to rep.spi :side :angle :inc
>repeat 200
>  [forward :side
>   right :angle
>   make "side :side + :inc]
>end
rep.spi defined
?rep.spi 10 100 2
```

.replace

Action:
Replaces the specified item of the list with the object.

Syntax:
.replace item_n varlist object

Explanation:
Replaces the specified item of the list with the object. THIS PRIMITIVE SHOULD BE USED WITH CAUTION!

Examples:

```
?make "varlist [A B C D E F]
?.replace 4 :varlist [1 2 3]
?:varlist
[A B C [1 2 3] E F]
```

.reptail

Action:
Replaces all items following the specified item in the list with the object.

Syntax:
.reptail item_n varlist object

Explanation:
Replaces all items following the specified item in the list with the object. THIS PRIMITIVE SHOULD BE USED WITH CAUTION!

Examples:

```
?make "varlist [A B C D E F]
?.reptail 4 :varlist [1 2 3]
?:varlist
[A B C D [1 2 3]]
```

rerandom

Action:
Makes a subsequent RANDOM expression reproduce the same random sequence.

Syntax:
rerandom

Explanation:
RERANDOM makes a subsequent RANDOM expression reproduce the same random sequence.

Examples:

```
?repeat 10 [(type random 10 char 9)] print []
2   3   7   5   3   2   0   4   2   6
?repeat 10 [(type random 10 char 9)] print []
8   9   9   1   0   6   1   3   5   1
?rerandom
?repeat 10 [(type random 10 char 9)] print []
6   2   9   0   3   1   6   2   3   7
```

```
?rerandom
?repeat 10 [(type random 10 char 9)] print []
6   2   9   0   3   1   6   2   3   7
```

right rt

Action:
Rotates the turtle the input number of degrees to the right.

Syntax:
right degrees_n
rt degrees_n

Explanation:
RIGHT rotates the turtle the input number of degrees to the right. Usually, you input a number of degrees between 0 and 359. If the input number is greater than 359, the turtle appears to move the input number minus 360 degrees to the right. If you input a negative number to right, the turtle turns to the left.

Examples:

```
?repeat 36 [right 10]

?repeat 36 [right -10]

?to around.R :s
>repeat 360
>  [forward :s right 10 make "s :s + .01]
>end
around.R defined
?around.R 2
```

round

Action:
Outputs the input number rounded off to the nearest integer.

Syntax:
round n

Explanation:
ROUND outputs the input number rounded off to the nearest integer. If the fractional portion of the input number is 0.5 or greater, ROUND rounds up to the next integer. To truncate a decimal number to an integer, use INT.

Examples:

```
?round 3.333333
3

?int 28753/12
2149

?round -75.482
-75

?round 0.5
1
```

run

Action:
Executes the input instruction list.

Syntax:
run instr_list

Explanation:
RUN executes the input instruction list. If the input object outputs an object, RUN outputs that object. You can use LIST or SENTENCE to assemble an instruction list for run to execute.

Examples:

```
?run [print [tricolor]]
tricolor
```

```
?run [equalp 0 9]
FALSE

?to use :what
>if memberp lowercase :what [green red yellow]
>   [make "colornumber where
>    make "state "pd]
>if "eraser = lowercase :what
>   [make "state "pe]
>run (sentence "setpen "list "quote :state "quote :colornumber)
>end
use defined
?use "green forward 60 right 120
?use "red forward 60 right 120
?use "yellow forward 60 right 120

?to while :condition :instr_list
>if "TRUE = run :condition
>   [run :instr_list]
>   [stop]
>while :condition :instr_list
>end
while defined
?make "side 5
?while [:side < 50] [forward :side right 60 make "side :side + 2]


save
----

Action:
Writes  the  contents of the workspace or specified package(s) to  the  input-
named disk file.

Syntax:
save fname < pkgname | pkgname_list >

Explanation:
SAVE  writes  the  contents of the workspace or specified  package(s)  to  the
input-named  disk file. If there is already a file with the specified name  on
the disk, SAVE displays the message "File already exists." Choose another name
for  the  new  file, delete the old file with ERASEFILE,  or  rename  it  with
CHANGEF. If there is no file with the specified name on the disk, SAVE creates
one  with  the  file type LOG. File names in the disk's  directory  cannot  be
longer  than eight characters. So, if you specify a file name with  more  than
eight characters, SAVE truncates the file name to eight characters.

If  you do not specify a package name as input, SAVE saves all procedures  and
variables, except those in buried packages. If you specify a package or a list
of  packages, only the procedures and variables in the input-named  package(s)
are  saved.  SAVE can save buried procedures and variables if  you  specify  a
buried package name.

Examples:
These examples assume you have three packages named FLY, SHAPES, and SIZES  in
your workspace.

?popkg
fly is buried
  to buzz
  to fly
  to zoom
shapes
  to circle :size
  to triangle :size
  to square :size
sizes
  "big (VAL)
  "medium (VAL)
  "small (VAL)
?save "Sheila
?save "fly "fly
?save "shapes [shapes sizes]
?dir
[SHAPES.LOG FLY.LOG SHEILA.LOG]

(ROCHE> I found the following procedure useful... For multiple saves.)

to sav :fname
erasefile :fname
save :fname
```

```
end


savepic
-------

Action:
Writes the picture (on the screen) to the input-named disk file.

Syntax:
savepic fname fname

Explanation:
SAVEPIC  writes  the contents of the screen to the input-named disk  file.  If
there is already a file with the specified name on the disk, SAVEPIC  displays
the  message  "File  already exists." Choose another name for  the  new  file,
delete the old file with ERASEFILE, or rename it with CHANGEF. If there is  no
file  with the specified name on the disk, SAVEPIC creates one with  the  file
type  PC0.  (ROCHE> for SETRES 0 images. But images saved under SETRES  1  are
given  the  PC1 file type... and DIRPIC is then unable to find them!  This  is
clearly a bug. The solution is to use an ambiguous filespec, with .PC? for the
file  type.)  File names in the disk's directory cannot be longer  than  eight
characters.  So, if you specify a file name with more than  eight  characters,
SAVEPIC truncates the file name to eight characters.

Examples:
?savepic "shapes
?dirpic
[SHAPES.PC0 PLAID.PC0 PIGLATIN.PC0]
?dirpic "p???????.pc?
[PLAID.PC0 PIGLATIN.PC0 PLAID.PC1 PIGLATIN.PC1]


screenfacts sf
--------------

Action:
Outputs a list that contains information about the screen.

Syntax:
screenfacts
sf

Explanation:
Outputs a list that contains: Background color number of the viewport;  Screen
state; Split size; Window state; Scrunch ratio; Zoom factor; X-coordinate  and
Y-coordinate of viewport center; Current resolution setting.

Examples:
?screenfacts
[0 ts 5 window 1 1 0 0 0]

?to screen.facts
>print []
>type [Background Color Number:\ ] print item 1 screenfacts
>type [Screen state:\ ] print item 2 screenfacts
>type [Split Size:\ ] print item 3 screenfacts
>type [Window State:\ ] print item 4 screenfacts
>type [Scrunch Ratio:\ ] print item 5 screenfacts
>type [Zoom Factor:\ ] print item 6 screenfacts
>(type [(X,Y) of Viewport Center:] (word "\( item 7 sf ", item 8 sf "\))
>print []
>type [Current Resolution Setting:\ ] print item 9 screenfacts
>print []
>end
screen.facts defined
?setres 0
?screen.facts

Background Color Number: 0
Screen state: ts
Split Size: 5
Window State: window
Scrunch Ratio: 1
Zoom Factor: 1
(X,Y) of Viewport Center: (0,0)
Current Resolution Setting: 0

?setres 1
?screen.facts
```

```
Background Color Number: 0
Screen state: ts
Split Size: 10
Window State: window
Scrunch Ratio: 0.5
Zoom Factor: 2
(X,Y) of Viewport Center: (0,0)
Current Resolution Setting: 1

?


sentence se
-----------

Action:
Outputs a list made up of the input objects.

Syntax:
sentence object object (...)
se object object (...)

Explanation:
SENTENCE  outputs a list made up of the input objects. SENTENCE is  like  LIST
but removes the outermost brackets from the input objects.

Without punctuation, SENTENCE requires and accepts two input objects. SENTENCE
can  accept more or fewer inputs when you enclose the SENTENCE  expression  in
parentheses  ["("  and  ")"].  If no other  expressions  follow  the  SENTENCE
expression on the line, you do not need to type the closing right  parenthesis
[")"].

There are two ways of creating lists in Dr. Logo:

        1) using square brackets ("[" and "]")
        2) using LIST or SENTENCE

Each  way  results  in a different kind of list. When you  create  a  list  by
enclosing  elements in square brackets, you create a "literal" list. Dr.  Logo
treats the elements of the list literally; it does not evaluate expressions or
look up the values of variables named in a literal list.

When  you  use LIST or SENTENCE to create a list, you can  use  variables  and
expressions to specify the elements Dr. Logo will put in the list. You can use
LIST and SENTENCE to create a list for input to most procedures. However,  IF,
IFFALSE, and IFTRUE require literal lists as input.

Examples:
?sentence "tortoise "hare
[tortoise hare]

?sentence "turtle []
[turtle]

?make "turtle (sentence readlist readlist readlist)
pecans
caramel
chocolate
?:turtle
[pecans caramel chocolate]

?sentence "hare [rabbit bunny]
[hare rabbit bunny]

?sentence [Slow and steady] [wins the race.]
[Slow and steady wins the race.]

?to use :what
>if memberp lowercase :what [green red yellow]
>   [make "colornumber where
>   make "state "pd]
>if "eraser = lowercase :what
>   [make "state "pe]
>run (sentence "setpen "list "quote :state "quote :colornumber)
>end
use defined
?use "green forward 60 right 120
?use "red forward 60 right 120
?use "yellow forward 60 right 120
```

setbg (= SET BackGround)

Action:
Sets the graphic screen background to the color represented by the input number.

Syntax:
setbg n

Explanation:
SETBG sets the graphic screen background to the color represented by the input number. The input number must be in the range from 0 to 63. The IBM Personal Computer supports eight background colors in two levels of intensity, although your color monitor might or might not display different intensities. The numbers SETBG accepts represent colors as follows:

Low intensity

0	16	32	48	Black
1	17	33	49	Blue
2	18	34	50	Green
3	19	35	51	Cyan
4	20	36	52	Red
5	21	37	53	Magenta
6	22	38	54	Yellow
7	23	40	55	White

High intensity

8	24	40	56	Black
9	25	41	57	Blue
10	26	42	58	Green
11	27	42	59	Cyan
12	28	43	60	Red
13	29	45	61	Magenta
14	30	46	62	Yellow
15	31	47	63	White

Each of the four numbers for a background color specifies a different pen for the turtle to use (see SETPC).

Examples:
When you first start Dr. Logo, FIRST SCREENFACTS returns 0, representing Black, the default background color, with the turtle, pen, and text all written in White. (Use SETBG 1 for a Blue background, with a Yellow turtle leaving a White trail with its pen.) The BG.CYCLE procedure displays background colors.

```
?to bg.cycle :val ; Displays background colors
>if :val = 0
>   [print [Cycle complete.] setbg 0 stop]
>setbg :val
>(print [This is background color number] first screenfacts)
>wait 20000 ; 2 seconds on a 500-MHz PC
>bg.cycle :val - 1
>end
bg.cycle defined
?bg.cycle 7
This is background color 7
This is background color 6
This is background color 5
This is background color 4
This is background color 3
This is background color 2
This is background color 1
Cycle complete.
```

setcursor

Action:
Positions the cursor at the location specified by the input text screen coordinate list.

Syntax:
setcursor coord_list

Explanation:
SETCURSOR positions the cursor at the location specified in the input text

screen coordinate list. A text screen coordinate list has two elements: the first element is the column number; the second, the line number. The line number must be in the range 0 to 24. The column number must be in the range 0 to 79.

If either of the two input numbers exceed the allowed range, SETCURSOR uses the highest allowable value: 24 for line number, or 79 for column number. If you are using a splitscreen or other 40 column text screen, and specify a column number greater than 40, SETCURSOR wraps the cursor to the next line, in effect adding 1 to the input line number, and subtracting 40 from the input column number.

Examples:
```
?to diagonal.type
>type readchar
>setcursor list (3 + first cursor) (1 + last cursor)
>diagonal.type
>end
diagonal.type defined
?cleartext
?setcursor [0 0]
?diagonal.type
H
 e
  l
   l
    o
     ,

       w
        o
         r
          l
           d
            !
```

setd (= SET Drive name)

Action:
Makes the specified drive the default drive.

Syntax:
setd d:

Explanation:
SETD makes the specified drive the default drive. Dr. Logo looks in the directory of the disk in the default drive when you do not specify a drive name in a disk command such as SAVE, LOAD, ERASEFILE, CHANGEF, or DIR. (ROCHE> You need to type the colon (":") after the drive name. Normally, colon is used to separate the drive from the file name. You say: "drive A", not "drive A:".)

Examples:
```
?setd "b:
?defaultd
B:
?setd "a:
?defaultd
A:
```

setheading seth

Action:
Turns the turtle to the absolute heading specified by the input number of degrees.

Syntax:
setheading degrees_n
seth degrees_n

Explanation:
SETHEADING turns the turtle to the absolute heading specified by the input number of degrees. If the input number is positive, SETHEADING turns the turtle clockwise (right) from North to the input number of degrees, regardless of the turtle's current heading. If the input number is negative, SETHEADING turns the turtle counter-clockwise.

Examples:

```
?setheading 90

?setheading 180

?setheading -90

?to draw.compass :n
>if :n > 330
>   [setheading 0 stop]
>penup setheading :n forward 50
>turtletext (word :n)
>back 50 right 23 forward 45 pendown forward 5 penup back 50
>drawcompass :n + 45
>end
draw.compass defined
?draw.compass 0
```

setpal (= SET PALette of RGB colors)

Action:
Assigns RGB colors to a pen_index.

Syntax:
setpal [pal_n RGB_list]

Explanation:
The colors are stored in a table called a palette. The palette has room for 256 colors. SETPAL assigns the color represented by RGB_LIST to the pen index designated by PAL_N. The range of values for PAL_N are 0 through 256. RGB_LIST must be made of 3 elements. Each element control one of the basic components of the color.

The first element controls the Red component, the second element controls the Green component, and the last element controls the Blue component. Each component of the RGB_LIST can be from 0 to 63. A 0 turns off the component color, and a 63 turns it at its maximum. The standard setting is 42. Half intensity is 21.

Examples:

```
setpal [0  0  0  0]  ; Black
setpal [0  0  0 42]  ; Blue
setpal [0  0 42  0]  ; Green
setpal [0  0 42 42]  ; Cyan
setpal [0 42  0  0]  ; Red
setpal [0 42  0 42]  ; Magenta
setpal [0 42 21  0]  ; "Yellow"
setpal [0 42 42 42]  ; "White"
```

(ROCHE> To get real Yellow, use [0 63 63 0]. For White, use [0 63 63 63].)

setpan

Action:
Establishes the center point of the viewport.

Syntax:
setpan coord_list

Explanation:
Establishes the center point of the viewport.

Examples:

```
?setpan [50 50]
```

setpc (= SET Pen Color)

Action:
Sets the turtle's pen to the color specified by the input number.

Syntax:
setpc n

Explanation:
SETPC sets the turtle's pen to the color specified by the input number. The turtle has four pens. Each pen has four unique colors of ink, one of which is

the background color that the turtle uses for erasing. Which pen and set of inks the turtle uses depends on the number you input to SETBG to specify the background color.

Low intensity

0	16	32	48	Black
1	17	33	49	Blue
2	18	34	50	Green
3	19	35	51	Cyan
4	20	36	52	Red
5	21	37	53	Magenta
6	22	38	54	Yellow
7	23	40	55	White

High intensity

8	24	40	56	Black
9	25	41	57	Blue
10	26	42	58	Green
11	27	42	59	Cyan
12	28	43	60	Red
13	29	45	61	Magenta
14	30	46	62	Yellow
15	31	47	63	White

When the background color is in the range 0 to 15:

- pencolor 1 represents dark green ink
- pencolor 2 represents dark red ink
- pencolor 3 represents dark yellow ink

When the background color is in the range 16 to 31:

- pencolor 1 represents bright green ink
- pencolor 2 represents bright red ink
- pencolor 3 represents bright yellow ink

When the background color is in the range 32 to 47:

- pencolor 1 represents dark cyan ink
- pencolor 2 represents dark magenta ink
- pencolor 3 represents dark grey ink

When the background color is in the range 48 to 63:

- pencolor 1 represents bright cyan ink
- pencolor 2 represents bright magenta ink
- pencolor 3 represents bright white ink

For all background color numbers, pencolor 0 represents background color (erasing) ink. The turtle itself is drawn with pencolor 3.

Examples:

```
?to pc.cycle :penc :bkgr
>if :penc > 3
>   [change.bg]
>if :bkgr > 50
>   [stop]
>setpc :penc
>(print [This is pencolor] item 5 tf)
>repeat 36
>  [forward 4 right 10] back 20
>wait 20000 ; 2 seconds on a 500-MHz PC
>pencolor.cycle (:penc + 1) :bkgr
>end
pc.cycle defined
?to change.bg
>make "bkgr 16 + :bkgr
>if :bkgr > 50
>   [stop]
>setbg :bkgr
>(print [Background color] output first sf)
>make "penc 1
>end
change.bg defined
?clearscreen home left 90 forward 120 clean splitscreen setbg 1
?pc.cycle 1 1
This is pencolor number 1
This is pencolor number 2
This is pencolor number 3
```

```
Background color 17
This is pencolor number 1
This is pencolor number 2
This is pencolor number 3
Background color 33
This is pencolor number 1
This is pencolor number 2
This is pencolor number 3
Background color 49
This is pencolor number 1
This is pencolor number 2
This is pencolor number 3
```

setpen

Action:
Sets the turtle's pen to the state and color specified in the input list.

Syntax:
setpen list

Explanation:
SETPEN sets the turtle's pen to the state and color specified in the input list. You can use SETPEN to change the pen's state and color with a single command; for example, SETPEN [PD 2] is equivalent to PENDOWN SETPC 2.

The first element of the input list must be one of PENDOWN, PENERASE, PENUP, or PENREVERSE. The second element must be a number that represents a pencolor (see SETPC).

Examples:

```
?to pen
>output list item 4 turtlefacts item 5 turtlefacts
>end
pen defined
?pen
[pd 1]
?setpen [pe 0]
?pen
[pe 0]
?setpen [pd 1]

?to dandelion
>setheading first shuffle list (random 45) (-1 * (random 45))
>make "penstate pen
>forward 20 + random 80
>setpc 1 + random 3
>repeat 36
>  [forward 10 back 10 right 10]
>penup setpos :root
>setpen :penstate
>dandelion
>end
dandelion defined
?make "root [0 -50]
?setpos :root clean
?dandelion
```

setpos (= SET turtle POSition)

Action:
Moves the turtle to the position specified in the input coordinate list.

Syntax:
setpos coord_list

Explanation:
SETPOS moves the turtle to the position specified in the input coordinate list. The input list has the same two-element form as the list output by pos. The first element is the X-coordinate, in the range -150 to +149. The second element is the Y-coordinate, in the range -99 to +100.

If the graphic screen is FENCEd, values outside these ranges generate a "Turtle out of bounds" message. When WINDOW is set, you can input coordinates outside the ranges, to make the turtle plot off screen. When WRAP is set, SETPOS converts any input coordinates that are outside the visible range, so that the turtle WRAPs and remain visible.

```
Examples:
?clearscreen
?setpos [80 50]

?setpos [0 30]

?setpos [80 -50]

?setpos [0 0]

?to dandelion
>setheading first shuffle list (random 45) (-1 * (random 45))
>make "penstate pen
>forward 20 + random 80
>setpc 1 + random 3
>repeat 36
>  [forward 10 back 10 right 10]
>penup setpos :root
>setpen :penstate
>dandelion
>end
dandelion defined
?make "root [0 -50]
?setpos :root clean
?dandelion


setprec         (= SET PRECision of numbers)
-------

Action:
Sets the number of significant digits displayed in an output number.

Syntax:
setprec n

Explanation:
Sets the number of significant digits displayed in an output number.

Examples:
?setprec 10


setread         (= SET file to READ)
-------

Action:
Sets the data file from which to receive input.

Syntax:
setread fname

Explanation:
Sets  the data file from which to receive input. After this command,  you  use
READLIST, READCHAR, and READQUOTE to read the data from the file or device.

(ROCHE> There is a system message, saying that "Only 4 files can be open".)

Examples:
?open "telnos
?setread "telnos
?readlist
[TERRY CLOTH]


setres          (= SET the RESolution of the screen)
------

Action:
Sets the resolution of the viewport.

Syntax:
setres n

Explanation:
Sets  the resolution of the viewport. Available resolutions are  0  (320x200),
and  1  (640x480).  Changes the default and maximum values  for  many  of  the
parameters that affect the appearance of the text and graphic displays.

                 +100                          +239
```


```
Examples:
?screenfacts
[0 ts 5 window 1 1 0 0 0]

?to screen.facts
>print []
>type [Background Color Number:\ ] print item 1 screenfacts
>type [Screen state:\ ] print item 2 screenfacts
>type [Split Size:\ ] print item 3 screenfacts
>type [Window State:\ ] print item 4 screenfacts
>type [Scrunch Ratio:\ ] print item 5 screenfacts
>type [Zoom Factor:\ ] print item 6 screenfacts
>(type [(X,Y) of Viewport Center:] (word "\( item 7 sf ", item 8 sf "\))
>print []
>type [Current Resolution Setting:\ ] print item 9 screenfacts
>print []
>end
screen.facts defined
?setres 0
?screen.facts

Background Color Number: 0
Screen state: ts
Split Size: 5
Window State: window
Scrunch Ratio: 1
Zoom Factor: 1
(X,Y) of Viewport Center: (0,0)
Current Resolution Setting: 0

?setres 1
?screen.facts

Background Color Number: 0
Screen state: ts
Split Size: 10
Window State: window
Scrunch Ratio: 0.5
Zoom Factor: 2
(X,Y) of Viewport Center: (0,0)
Current Resolution Setting: 1

?
```

(ROCHE> You will note the following differences between Resolution 0 and Resolution 1:

	Res0	Res1
1) Split Size:	5	10
2) Scrunch Ratio:	1	0.5
3) Zoom Factor:	1	2

I don't know if it is a bug, but SETRES 1 "scrunch" the picture vertically by half. If you use SETSCRUNCH 1, you recognize your Resolution 0 image. If you then use SETZOOM 1, the image fills the screen, just like the Resolution 0 image, but with finer pixels. However, it would be silly to use less than half the resolution of VGA. So, the only logical thing to do is:

1) SETRES 1
2) SETSCRUNCH 1
3) SETZOOM 1
4) SETSPLIT 5 (optional)

to get a real VGA graphics screen (640x480), then rewrite all your old 300x200 pixels (EGA) programs to fill this VGA screen. And don't forget that the workspace is four times bigger than the 8-bits version of Dr. Logo... If you want to see the increase in pixels between SETRES 0 and SETRES 1, run the following procedure, after FRAME1 (see FENCE).)

```
to frame
pu setpos [-160 99] pd
setx 159
sety -100
```

```
setx -160
sety 100
pu home st
end
```

.setseg

Action:
Sets segment value.

Syntax:
.setseg segment_n

Explanation:
Sets segment value to be used by subsequent .DEPOSIT and .EXAMINE expressions.
Does not change the segment register.

Examples:
?.setseg 23

setscrunch (= SET SCRUNCH ratio)

Action:
Sets the viewport vertical aspect ratio.

Syntax:
setscrunch n

Explanation:
Sets the viewport vertical aspect ratio to the input number that can be from 1 through 5.

Examples:
?setscrunch 5

setsplit

Action:
Sets the number of lines in the splitscreen's text window.

Syntax:
setsplit n

Explanation:
SETSPLIT sets the number of lines in the splitscreen screen window. The input number must be in the range 0 to 25.

Examples:
?repeat 12 [repeat 5 [forward 50 right 72] right 30]
?setpc 1
?splitscreen
?setsplit 15

?setsplit 1

?setsplit 5

setwrite (= SET file to WRITE)

Action:
Sets the destination of outputs.

Syntax:
setwrite fname | device

Explanation:
Sets the destination of outputs from PRINT, TYPE, and SHOW to the data file or system device. The file or device must already be OPEN. SETWRITE sets the file position at the top of the file.

(ROCHE> The devices are: CON: NUL: PRN: and AUX: There is a system message, saying that "Only 4 files can be open".)

Examples:
?open "phones
?setwrite "phones

setx (= SET X-coordinate)

Action:
Moves the turtle horizontally to the X-coordinate specified by the input number.

Syntax:
setx n

Explanation:
SETX moves the turtle horizontally to the X-coordinate specified by the input number. The Y-coordinate is not changed.

Examples:
?to frame
>penup setpos [-150 100] pd
>setx 149
>sety -99
>setx -150
>sety 100
>end
frame defined
?frame

sety (= SET Y-coordinate)

Action:
Moves the turtle vertically to the Y-coordinate specified by the input number.

Syntax:
sety n

Explanation:
SETY moves the turtle vertically to the Y-coordinate specified by the input number. The X-coordinate is not changed.

Examples:
?to frame
>penup setpos [-150 100] pd
>setx 149
>sety -99
>setx -150
>sety 100
>end
frame defined
?frame

setzoom (= SET ZOOM ratio)

Action:
Sets the screen to show more or less of the picture.

Syntax:
setzoom n

Explanation:
Causes the viewport to show a greater or lesser portion of the graphic plane, thereby expanding or contracting subsequent turtle motion. Does not clear the viewport, nor alter anything previously drawn.

Examples:
?setzoom 2

show

Action:
Displays the input object on the text screen.

Syntax:

show object

Explanation:
SHOW displays the input object on the text screen, followed by a Carriage Return. SHOW does not remove the outer brackets from an input list. Compare show with PRINT and TYPE.

You can input any number of objects by preceding SHOW with a left parenthesis ["("]. When preceded by a parenthesis, SHOW displays all its inputs on the same line, and follows only the last input with a Carriage Return.

Dr. Logo's interpreter has an implicit SHOW command. This means that, when you enter an expression that outputs an object to the ? prompt, Dr. Logo shows the output object, instead of complaining that it does not know what to do with it.

Examples:

```
?show "K
K

?show [crab scallops clams]
[crab scallops clams]

?3 * 10
30

?to demo :list
>make "success "FALSE
>catch "error [do.it]
>if :success = "TRUE
>   [stop]
>make "error1 error
>if 52 = first :error1
>   [do.it]
>   [print first butfirst :error1]
>end
demo defined
?to do.it
>type "? print :list
>run :list
>make "success "TRUE
>end
do.it defined
?to do.it1
>show run :list
>make "success "TRUE
>end
?do.it1 defined
?to logo.demo
>print []
>print [Here's how Dr. Logo responds to some commands:]
>print []
>demo [make "side 80]
>demo [repeat 4 [forward :side right 90]]
>demo [fence]
>demo [:side]
>demo [repeat 2 [back :side]]
>print []
>print [That's all, folks!]
>print []
>end
logo.demo defined
?logo.demo

Here's how Dr. Logo responds to some commands:

?make "side 80
?repeat 4 [forward :side right 90]
?fence
?:side
80
?repeat 2 [back :side]
Turtle out of bounds

That's all, folks!

?
```

showturtle st

Action:
Makes the turtle visible if hidden.

Syntax:
showturtle
st

Explanation:
SHOWTURTLE makes the turtle visible. When invisible, the turtle draws faster, and does not distract visually from the drawing. To make the turtle invisible, enter HIDETURTLE (HT).

Examples:
```
?hideturtle
?showturtle
?to face
>hideturtle forward 90
>make "chin pos
>setpc 3 repeat 36 [forward 6 left 10]
>setpc 1 left 90 penup forward 30 left 90 forward 20 setheading 0 eye
>setheading 90 forward 30 setheading 0 eye
>setpc 2 setheading 150 forward 10 setheading 180 mouth
>setpos :chin setheading 0
>forward 23 left 5
>end
face defined
?to eye
>pendown repeat 18 [forward 1 right 10]
>right 100
>repeat 8
>  [forward 1.5 left 6]
>pu
>end
eye defined
?to mouth
>pd
>repeat 18
>  [forward 2 right 10]
>right 100
>repeat 8
>  [forward 3 left 6]
>pu
>end
mouth defined
?face
?showturtle
```

shuffle

Action:
Outputs a list that contains the elements of the input list in random order.

Syntax:
shuffle list

Explanation:
SHUFFLE outputs a list that contains the elements of the input list in random order.

Examples:
```
?shuffle [a b c d]
[c b d a]

?to pick.a.card
>output word
>   first shuffle [A K Q J 10 9 8 7 6 5 4 3 2]
>   char first shuffle [3 4 5 6]
>end
pick.a.card defined
?repeat 4 [(type pick.a.card char 9)]
7Hearts  7Clubs  KDiamond  JSpade
```

(ROCHE> On the IBM PC, the values 3, 4, 5, and 6 (of Code Page 850) generate the symbols of the playing cards (Hearts, Diamond, Clubs, and Spade, respectively). Since WordStar uses control codes for its inner working, they are replaced, in the above example, by their names. When run on an IBM PC, the above program would dislay 7* 7* K* J*, where * would be the symbol.)

sin

Action:
Outputs the sine of the input number of degrees.

Syntax:
sin degrees_n

Explanation:
SIN outputs the trigonometric sine of the input number of degrees. SIN outputs a decimal number between 0 and 1.

Examples:
```
?sin 90
1

?to plot.sine
>make "val 0
>make "x -150
>make "inc (300 / 60)
>setx 150 setx :x
>plot.s :val
>end
plot.sine defined
?to plot.s :val
>if :x > 150
>   [stop]
>make "y (90 * (sin :val))  ; 90 makes plot visible
>setheading towards list :x :y
>setpos list :x :y
>make "x :x + :inc
>make "val :val + 6
>plot.s :val
>end
plot.s defined
?plot.sine
```

sort

Action:
Outputs a list of input words sorted into ascending order.

Syntax:
sort list

Explanation:
SORT takes a list as input, then outputs, in ascending order, the elements of the input list.

Examples:
```
?sort .contents

?sort glist ".PRM
```

splitscreen ss

Action:
Displays a window of text on the graphic screen.

Syntax:
splitscreen
ss

Explanation:
SPLITSCREEN displays a window of text on the graphic screen. A SPLITSCREEN command is equivalent to a Ctrl-S keystroke. Use SETSPLIT to specify the number of lines of text in the text window.

Examples:
```
?clearscreen repeat 12 [repeat 4 [forward 60 right 90] right 30
?splitscreen

?repeat 12 [repeat 4 [forward 60 right 90] right 30]
?splitscreen
```

sqrt (= SQuare RooT)

Action:
Outputs the square root of the input number.

Syntax:
sqrt n

Explanation:
SQRT outputs the square root of the input number.

Examples:
```
?sqrt 2
1.4142135623731

?to measure :xypair1 :xypair2
>make "x1 first :xypair1
>make "x2 first :xypair2
>make "xdif :x2 - :x1
>make "y1 last :xypair1
>make "y2 last :xypair2
>make "ydif :y2 - :y1
>make "xdif2 :xdif * :xdif
>make "ydif2 :ydif * :ydif
>output sqrt (:xdif2 + :ydif2)
>end
measure defined
?make "p1 [-75 -49]
?setpos :p1 clean
?make "p2 [75 30]
?setpos :p2
?measure :p1 :p2
169.53170794869
?make "p3 [50 -90]
?setpos :p3
?measure :p2 :p3
122.576506721313
?setpos :p1
?measure :p3 :p1
131.552270980018
```

STARTUP (Not a primitive)

A STARTUP file is a file that Dr. Logo automatically loads into the workspace when you start your Dr. Logo system. Dr. Logo interprets each line loaded from the file as if you typed it at the keyboard to the ? prompt. This makes any procedures stored in the STARTUP.LOG file available immediately when you start your Dr. Logo system. If you use EDF to create the STARTUP.LOG file, you can include stand-alone expressions that automatically execute the procedures in the file.

Your STARTUP.LOG file must always be on your system disk, so that Dr. Logo can find it when you are starting your system. To see the contents of the STARTUP.LOG file, enter:

```
?edf "startup
```

You can change the STARTUP.LOG file to contain your own procedures and execute your own commands. Sometimes, it is useful to PACKAGE and BURY procedures in the STARTUP.LOG file, so that they will not be displayed by workspace management commands, or included in other files saved during the session. Note that Dr. Logo does not automatically make a backup copy of your original file on disk, so use EDF carefully.

stop

Action:
Stops the execution of the current procedure, and returns to toplevel or the calling procedure.

Syntax:
stop

Explanation:
STOP stops the execution of the current procedure, and returns to the caller.

The caller is either the calling procedure or, if the procedure name was typed at the ? prompt, toplevel. STOP has an effect only within a procedure. At toplevel, STOP has no effect; it simply returns to toplevel. Compare STOP with CATCH, THROW, OUTPUT, and END.

Examples:
```
?to vanish :object
>if emptyp :object
>   [stop]
>print :object
>vanish butfirst :object
>end
vanish defined
?vanish "emperor
emperor
mperor
peror
eror
ror
or
r
```

sum +

Action:
Outputs the sum of the input numbers.

Syntax:
sum n n (...)
+ n n (...)

Explanation:
SUM outputs the sum of the input numbers. SUM is equivalent to the + arithmetic operator. Without punctuation, SUM requires and accepts two input objects. SUM can accept more or fewer inputs when you enclose the SUM expression in parentheses ["(" and ")"]. If no other expressions follow the SUM expression on the line, you do not need to type the closing right parenthesis [")"].

Examples:
```
?sum 5 9
14

?+ 5 9
14

?sum 2.565 7.9
10.465

?+ 2.565 7.9
10.465

?(sum 6 4 -7 9 3 -2 8
21

?(+ 6 4 -7 9 3 -2 9
21
```

tan

Action:
Outputs the tangent of the input angle.

Syntax:
tan degrees_n

Explanation:
TAN outputs the tangent of the angle represented by the input number of degrees.

Examples:
```
?tan 0
0

?arctan 45
1
```

```
?to plot.tan
>make "val -pi
>make "inc pi / 37.5
>make "x -150
>setx 150
>setx :x
>plot.t :val
>end
plot.tan defined
?to plot.t :val
>if :x > 150
>   [stop]
>forward 1000 * tan :val
>sety 0
>setx :x + 4
>make "x :x + 4
>make "val :val + :inc
>plot.t :val
>end
plot.t defined
?plot.tan
```

test

Action:
Remembers whether the input predicate is TRUE or FALSE for subsequent IFFALSE and IFTRUE expressions.

Syntax:
test pred_exp

Explanation:
TEST remembers whether the input predicate is TRUE or FALSE for subsequent IFFALSE and IFTRUE expressions. You can use TEST, IFFALSE, and IFTRUE instead of IF to control the flow of execution within your procedure when you need Dr. Logo to evaluate expressions after it evaluates a predicate expression, but before it executes the chosen instruction list.

Examples:
The COIN5 procedure is similar to the COIN procedures shown as examples under IF, but shows how to use TEST, IFFALSE, and IFTRUE. COIN5 evaluates an expression after it evaluates a predicate expression, but before it executes the chosen instruction list.

```
?to coin5
>test 1 = random 2
>if 1 = random 1000000
>   [print [Landed on edge!] stop]
>iftrue [type "heads]
>iffalse [type "tails]
>print [\ side up]
>end
coin5 defined
```

text

Action:
Outputs the definition list of the specified procedure.

Syntax:
text procname

Explanation:
TEXT outputs the definition list of the specified procedure. The format of the definition list is suitable for input to DEFINE. TEXT works by outputting the value of the system property .DEF from the procedure's property list.

Examples:

```
?to star2  ; Five pointed star
>repeat 5
>  [forward 30 left 217 forward 30 left 70]
>end
star2 defined
?star
?package "figures "star
?popkg
figures
```

```
  to star2  ; Five pointed star
?plist "star2
[.PAK figures PKG TRUE.DEF [[] [repeat 5 [forward 30 left 217 forward 30  left
70]]] .FMT [[0]]
?text "star2
[[] [repeat 5 [forward 30 left 217 forward 30 left 70]]]
```

textscreen ts

Action:
Selects a full text screen.

Syntax:
textscreen
ts

Explanation:
TEXTSCREEN devotes the entire monitor to text. It can return to a full text screen from a full graphic screen or splitscreen. If you have a single monitor system, TEXTSCREEN is equivalent to a Ctrl-T keystroke. If you have a two-monitor system (both a monochrome and color monitor), TEXTSCREEN is the only way to display the text screen on the color monitor; Ctrl-T displays the text screen on the monochrome monitor.

Examples:
```
?repeat 12 [repeat 4 [forward 60 right 90] right 30]
?splitscreen
?textscreen
```

thing

Action:
Outputs the value of the input-named variable.

Syntax:
thing varname

Explanation:
THING displays the contents or value of the input-named variable. THING is equivalent to a colon (":") before a variable name; for example, THING "KAREN is equivalent to :KAREN. THING works by outputting the value of the system property .APV from the input-named variable's property list.

Examples:
```
?name "chocolate "flavor
?thing "flavor
chocolate

?:flavor
chocolate

?make "chocolate "semi\-sweet
?thing "chocolate
semi-sweet

?thing "flavor
chocolate

?thing :flavor
semi-sweet
```

throw

Action:
Executes the line identified by the input name in a previous CATCH expression.

Syntax:
throw name

Explanation:
THROW works with the CATCH primitive to let your procedure handle special conditions. A THROW expression is valid only within the scope of a CATCH command. The description of CATCH explains how to use CATCH and THROW.

CATCH and THROW each require a name as input. To pair a CATCH expression with

a THROW expression, you must give the CATCH and THROW expressions the same input name. When a THROW command is executed, Dr. Logo returns to the procedure that contains the CATCH command identified by the throw name. Dr. Logo then executes the line that follows the CATCH command. THROW can accept the special word "toplevel to return to the ? prompt (compare with STOP).

Examples:
The COIL procedure asks the user to enter increasingly larger numbers as the turtle draws a coil on the screen. If the user types a number that is not bigger than the last one entered, COIL reminds the user what to type, and continues working.

```
?to coil
>print [Enter a small number.]
>make "previous 0
>forward grow.number
>right 30
>trap
>end
coil defined
?to grow.number
>make "growth first readlist
>if :growth < :previous
>   [throw "not.bigger]
>make "previous :growth
>output :growth
>end
grow.number defined
?to trap
>catch "not.bigger [draw.coil]
>(print [Enter a number bigger than] :previous)
>trap
>end
trap defined
?to draw.coil
>print [Enter a bigger number.]
>forward grow.number
>right 30
>draw.coil
>end
draw.coil defined
```

The THROW "NOT.BIGGER instruction in the GROW.NUMBER procedure always returns Dr. Logo to the TRAP procedure. If a STOP instruction had been used instead of THROW, Dr. Logo would return to the procedure that called GROW.NUMBER, which might be either COIL or DRAW.COIL.

The following procedures allow the user to type commands just as he normally would to the Dr. Logo interpreter. However, if the user enters a command incorrectly, the MY.MESSAGE procedure traps the normal Dr. Logo error message and prints a custom message.

```
?to my.message
>catch "error [interpret]
>print "Oops! first butfirst error [!!!]
>print [What do you want to do about that?]
>run readlist
>my.message
>end
my.message defined
?to interpret
>print [What next, boss?]
>run readlist
>interpret
>end
interpret defined
?my.message
```

to (Not a primitive)
--

Action:
Indicates the beginning of a procedure definition.

Syntax:
to procname < inputs >

Explanation:
TO is a special word that indicates the beginning of a procedure definition. At toplevel, TO signals Dr. Logo that you are starting to define a procedure,

and puts you in the procedure editor (the > prompt).

TO is not part of a procedure's definition list, and is not a primitive. You can use "to" as a procedure or variable name, if you are confident that the name will not cause undue confusion.

Examples:
```
?to pent
>repeat 5
>  [forward 25 left 72]
>end
pent defined
```

tones

Action:
Outputs a tone of the frequency and duration specified in the input list.

Syntax:
tones note_list

Explanation:
TONES outputs a note of the frequency and duration specified in the input note_list. The input list must contain two numbers. TONES interprets the first number as the frequency of the desired note. For example, 440 is the frequency of concert A. TONES interprets the second number as the number of milliseconds the tone is to last.

Examples:
```
?tones [440 250]

?to scale :freq
>repeat 14
>  [type list " int 0.5 + :freq
>  tones list 0.5 + :note 300
>  make "note :note * :c]
>end
scale defined
?make "c (2 ^ (1 / 12))
?scale 440
 440 466 494 523 554 587 622 659 698 740 784 831 880

?to play :song :speed
>make "note first :song
>if memberp :note :notes
>   [make interval where]
>(type " :note)
>if :note = "R
>   [tones list 0 :speed]
>   [tones list 440 * (:c ^ :interval) :speed]
>play butfirst :song :speed
>end
play defined
?make  "notes [A A# B C C# D D# E F F# G G# A' A#' B' C' C#' D' D#' E' F'  F#'
G' G#']
?make "hb [G A' G C' B' R G A' G D' C' R' G G#' E' C' B' A' F' E' C' D' C']
?play :hb 60

?to play1 :song :speed
>make "note first :song
>if memberp :note :notes
>   [make interval where]
>(type " :note)
>if :note = "R
>   [tones list 0 :speed]
>   [tones list 440 * (:c ^ :interval) :speed]
>if emptyp first butfirst :song
>   [stop]
>make "note2 first butfirst :song
>if memberp :note2 :notes
>   [make interval where]
>(type " :note2)
>if :note2 = "R
>   [tones list 0 :speed * 4]
>   [tones list 440 * (:c ^ :interval) :speed * 4]
>play butfirst butfirst :song :speed
>end
play1 defined
?make  "turtle.song [D G A' A#' A' A#' R R A' A#' R R R D' R C' A#' A' G A'  G
```

```
A' R R G F G A' R R C' A#' A' G F G F G R R F D# F G R R G G A' R G F# R D'
?play :turtle.song 14
 D G A' A#' A' A#' R R A' A#' R R R D' R C' A#' A' G A' G A' R R G F G A' R  R
C' A#' A' G F G F G R R F D# F G R R G G A' R G F# R D'
```

towards

Action:
Outputs a heading that would make the turtle face the position specified in the input coordinate list.

Syntax:
towards coord_list

Explanation:
TOWARDS outputs a heading that would make the turtle face the position specified in the input coordinate list. To make the turtle turn towards the position, use the output of TOWARDS as the input to SETHEADING.

Examples:

```
?cs
?towards [75 50]
56.3099324740202

?towards [-75 -50]
236.30993247402

?to plot.sine
>make "val 0
>make "x -150
>make "inc (300 / 60)
>setx 150 setx :x
>plot.s :val
>end
plot.sine defined
?to plot.s :val
>if :x > 150
>   [stop]
>make "y (90 * (sin :val)) ; 90 makes plot visible
>setheading towards list :x :y
>setpos list :x :y
>make "x :x + :inc
>make "val :val + 6
>plot.s :val
>end
plot.s defined
?plot.sine
```

trace

Action:
Turns on trace monitoring of all or specified procedure(s).

Syntax:
trace < procname | procname_list >

Explanation:
TRACE turns on trace monitoring of procedure execution. Tracing displays the name of each procedure as it is called, and the name and value of each variable as it is defined. It also displays the level number, the number of procedures that have been called since a procedure was initiated at toplevel.

Tracing lets you observe details of your procedure's execution without interrupting with PAUSEs. Use WATCH or Ctrl-Z if you want to PAUSE during procedure execution.

Examples:

```
?to average :numbers
>make "total 0
>add.up :numbers
>print :total / count :numbers
>end
average defined
?to add.up :list
>if emptyp :list
>   [stop]
>make "total :total + first :list
```

```
>add.up butfirst :list
>end
add.up defined
?trace
?average [1 2 3]
[1] Evaluating average
[1] numbers is [1 2 3]
[2] Evaluating add.up
[2] list is [1 2 3]
[3] Evaluating add.up
[3] list is [2 3]
[4] Evaluating add.up
[4] list is [3]
[5] Evaluating add.up
[5] list is []
2
?notrace
?average [1 2 3]
2
```

turtlefacts tf

Action:
Outputs a list of information about the turtle.

Syntax:
turtlefacts
tf

Explanation:
Outputs a list that contains: Turtle's X-coordinate; Turtle's Y-coordinate; Turtle's heading; Pen state; Pen's color number; TRUE if the turtle is visible, FALSE if not.

Examples:

```
?turtlefacts
[15 30 60 PE 3 FALSE]

?to turtle.facts
>print []
>type [Turtle's X\-coordinate:\ ] print item 1 tf
>type [Turtle's Y\-coordinate:\ ] print item 2 tf
>type [Turtle's Heading:\ ] print item 3 tf
>type [Pen State:\ ] print item 4 tf
>type [Pen Color:\ ] print item 5 tf
>type [Turtle Visible:\ ] print item 6 tf
>print []
>end
turtle.facts defined
?turtle.facts

Turtle's X-coordinate: 0
Turtle's Y-coordinate: 0
Turtle's Heading: 0
Pen State: pd
Pen Color: 15
Turtle Visible: TRUE
```

(ROCHE> SETRES does not modify the values of TURTLEFACTS. Note that, by default, pen color is 15 (White) and background color is 0 (Black). Using SETBG 1 will display a Blue background, with a Yellow turtle leaving a White trail with its pen.)

turtletext tt

Action:
Displays the input object at the turtle's current location on the graphic screen.

Syntax:
turtletext object
tt object

Explanation:
TURTLETEXT displays the input object on the graphic screen. The first character of the input object appears to the right of the turtle's center line, plus zero to four turtle steps (pixels) to align with the closest

character cell.

Like PRINT, TURTLETEXT removes the outer brackets from any input list, and follows the last input item with a Carriage Return. Without punctuation, TURTLETEXT requires and accepts one input object. TURTLETEXT can accept more inputs when you enclose the TURTLETEXT expression is parentheses ["(" and ")"]. If no other expressions follow the TURTLETEXT expression on the line, you do not need to type the closing right parenthesis [")"].

Examples:
```
?cs
?turtletext "home forward 10
?turtletext readquote forward 10
I want to go
```

type

Action:
Displays the input objects on the screen.

Syntax:
type object (...)

Explanation:
TYPE displays the input object on the screen, but does not follow the last input object with a Carriage Return. TYPE removes the outer square brackets ("[" and "]") from an input list. You can input any number of objects by preceding TYPE with a left parenthesis ["("]. When preceded by a parenthesis, TYPE displays all its inputs on the same line. Compare TYPE with PRINT and SHOW.

Examples:
```
?type [This is the turtle's position:] pos
This is the turtle's position: [-19 -21]

?to new.prompt :prompt
>(type :prompt " )
>run readlist
>new.prompt :prompt
>end
new.prompt defined
?new.prompt [Dr.\ Logo\>]
Dr. Logo> repeat 5 [forward 40 left 72]
Dr. Logo>
```

unbury

Action:
Restores the specified package(s) to workspace management commands.

Syntax:
unbury pkgname | pkgname_list

Explanation:
UNBURY restores the specified package or packages to workspace management commands. UNBURY works by removing the bury property (.BUR) from the package's property list. The description of BURY command tells how workspace management commands treat buried packages.

Examples:
These examples assume you have the following in your workspace: a package named FIGURES that contains two variables named BIG and SMALL and two procedures named SQUARE and TRIANGLE, and a package named TITLES that contains two procedures named PRAUTHOR and PRDATE.

```
?popkg
figures
  "big (VAL)
  "small (VAL)
  to square
  to triangle
titles
  to prauthor
  to prdate
?bury "titles
?popkg
figures
```

```
    "big (VAL)
    "small (VAL)
    to square
    to triangle
  titles is buried
    to prauthor
    to prdate
  ?pots
  to square
  to triangle
  ?unbury "titles
  ?popkg
  figures
    "big (VAL)
    "small (VAL)
    to square
    to triangle
  titles
    to prauthor
    to prdate
  ?pots
  to square
  to triangle
  to prauthor
  to prdate
```

uppercase uc

Action:
Outputs the input word with all alphabetic characters in uppercase.

Syntax:
uppercase word
uc word

Explanation:
UPPERCASE outputs the input word with all alphabetic characters converted to uppercase.

Examples:

```
?uppercase "jones
JONES

?uppercase "BeckyAnn
BECKYANN

?to quiz
>print [Can you play the ocarina?]
>if "N = uppercase first rq
>   [print [Me neither!]]
>   [print [Wow, I have never met anyone who did!]
>end
quiz defined
?quiz
Can you play the ocarina?
Not really
Me neither!
```

wait

Action:
Stops execution for the amount of time specified by the input number.

Syntax:
wait n

Explanation:
WAIT stops execution for the amount of time specified by the input number. WAIT interprets the input number as the number of 1/60ths of a second it is to wait. For example, to stop for one second, use WAIT 60.

(ROCHE> This was the case for the IBM PC in the USA running at 4-MHz in 1981, where the electrical current's frequency is 60-Hz. (In Europe, it is 50-Hz.) However, on my 500-MHz European PC, WAIT 10000 waits 1 seconds.)

Examples:

```
?wait 60
```

```
?repeat 10 [type "tick wait 30 print "tock wait 30]
ticktock
ticktock
ticktock
ticktock
ticktock
ticktock
ticktock
ticktock
ticktock
ticktock

?to bg.cycle :val ; Displays background colors
>if :val = 0
>   [print [Cycle complete.] setbg 0 stop]
>setbg :val
>(print [This is background color number] first screenfacts)
>wait 20000 ; 2 seconds on a 500-MHz PC
>bg.cycle :val - 1
>end
bg.cycle defined
?bg.cycle 7
This is background color 7
This is background color 6
This is background color 5
This is background color 4
This is background color 3
This is background color 2
This is background color 1
Cycle complete.


watch
-----

Action:
Turns on watch monitoring of all or specified procedure(s).

Syntax:
watch < procname | procname_list >

Explanation:
WATCH  turns  on expression-by-expression monitoring of  procedure  execution.
WATCH  displays each expression before execution, and PAUSEs until  you  press
the  (Enter)  key.  During  the PAUSE, you can examine  the  values  of  local
variables,  and  experiment  with  variations of  the  expression  before  the
expression  is  actually executed. If you want the values of variables  to  be
displayed automatically, enable TRACE as well as WATCH.

WATCH  also  displays  a number in square brackets ("[" and  "]")  before  the
expression. This level number tells you how many procedures your procedure has
called since it began execution.

If you give WATCH a procedure name or a list of procedure names as input, only
the specified procedures are monitored. Otherwise, any procedure you  initiate
at toplevel or call from within another procedure is monitored.

Normally,  anything  an expression displays on the text  appears  interspersed
with the information that WATCH displays on the screen.

To stop the WATCH step-by-step monitoring, enter NOWATCH.

Examples:
?to average :numbers
>make "total 0
>add.up :numbers
>print :total / count :numbers
>end
average defined
?to add.up :list
>if emptyp :list
>   [stop]
>make "total :total + first :list
>add.up butfirst :list
>end
add.up defined
?watch
?average [1 2 3]
[1] In average, make "total 0
[1] In average, add.up :numbers
```

```
[2] In add.up, if emptyp :list [stop]
[2] In add.up, make "total :total + first :list
[2] In add.up, add.up butfirst :list
[3] In add.up, if emptyp :list [stop]
[3] In add.up, make "total :total + first :list
[3] In add.up, add.up butfirst :list
[4] In add.up, if emptyp :list [stop]
[4] In add.up, make "total :total + first :list
[4] In add.up, add.up butfirst :list
[5] In add.up, if emptyp :list [stop]
[1] In average, print :total / count :numbers
2
?nowatch
?average [1 2 3]
2
```

where

Action:
Outputs the item number of the most recent successful MEMBERP expression.

Syntax:
where

Explanation:
WHERE outputs a number that identifies the location of an element within a word or list if a MEMBERP expression containing that element and word or list outputs TRUE. WHERE outputs the item number of the most recent successful MEMBERP expression.

Examples:

```
?memberp "v" river
TRUE
?show where
3

?to use :what
>if memberp lowercase :what [green red yellow]
>   [make "colornumber where
>    make "state "pd]
>if "eraser = lowercase :what
>   [make "state "pe]
>run (sentence "setpen "list "quote :state "quote :colornumber)
>end
use defined
?use "green forward 80 right 120
?use "red forward 80 right 120
?use "yellow forward 80 right 120
```

window

Action:
Allows the turtle to plot outside the visible graphic screen.

Syntax:
window

Explanation:
WINDOW allows the turtle to plot outside the visible graphic screen. When you first start Dr. Logo, the turtle can go beyond the visible screen and return. You can enter WRAP or FENCE to limit the turtle to on-screen plotting. To resume offscreen plotting after a WRAP or FENCE command, enter WINDOW.

Examples:

```
?fence
?to squiral :side
>repeat 4
>  [forward :side right 90]
>right 20
>squiral :side + 5
>end
squiral defined
?squiral 20
Turtle out of bounds in squiral: repeat 4 [forward :side right 90]
?window
?squiral 20
Ctrl-G
```

word

Action:
Outputs a word made up of the input words.

Syntax:
word word word (...)

Explanation:
WORD outputs a word made up of the input words. Without punctuation, WORD requires and accepts two input objects. WORD can accept more or fewer inputs when you enclose the WORD expression in parentheses ["(" and ")"]. If no other expressions follow the WORD expression on the line, you do not need to type the closing right parenthesis [")"].

Examples:
```
?word "Hocus "Pocus
HocusPocus

?word 23 "skiddoo
23skiddoo

?(word "ab "ra "ca "da "bra
abracadabra

?to make.string :list
>if emptyp :list
>   [output "]
>output (word first :list char 32 make.string butfirst :list)
>end
make.string defined
?repeat 2 make.string [forward 40 right 160]
?forward 40 right 160
?forward 40 right 160
```

wordp (= WORD Predicate)

Action:
Outputs TRUE if the input object is a word or a number.

Syntax:
wordp object

Explanation:
WORDP outputs TRUE if the input object is a word or a number. Otherwise, WORDP outputs FALSE.

Examples:
```
?wordp "Naima
TRUE

?wordp 50
TRUE

?wordp [word]
FALSE

?wordp butfirst [green red yellow]
FALSE

?to list.memberp :word :list
>if emptyp :list
>   [output "FALSE]
>if wordp first :list
>   [if :word = first :list
>       [output "TRUE]
>       [output list.memberp :word butfirst :list]]
>if list.memberp :word first :list
>   [output "TRUE]
>output list.memberp :word butfirst :list
>end
list.memberp defined
?make "address.book [[name [Mr. President]] [street [1600 Pennsylvania
Avenue]] [city/state [Washington D.C.]]]
?list.memberp "Washington :address.book
TRUE
```

```
?list.memberp "Oregon :address.book
FALSE
```

wrap

Action:
Makes the turtle re-appear on the opposite side of the screen when it exceeds the boundary.

Syntax:
wrap

Explanation:
WRAP makes the turtle re-appear on the opposite side of the graphic screen when it moves beyond an edge. While WRAP is set, the turtle never leaves the visual field. To allow the turtle to plot offscreen, enter WINDOW.

Examples:

```
?wrap
?forward 180
?cs

?to plaid
>wrap
>setpc 3 right 40 forward 10965
>setpc 2 right 90 forward 5000
>setpc 1 penup right 90 forward 6 pendown ht
>repeat 625
>  [forward 3 left 90 forward 1 right 90 back 3 left 90 forward 1 right 90]
>end
plaid defined
?plaid
```

writer

Action:
Outputs the current data file.

Syntax:
writer

Explanation:
Outputs the current data file that is open for writing.

(ROCHE> There is a system message, saying that "Only 4 files can be open".)

Examples:

```
?writer
[A:ADDRESS.DAT]
```

xcor

Action:
Outputs the X-coordinate of the turtle's current position.

Syntax:
xcor

Explanation:
XCOR outputs the X-coordinate of the turtle's current position. XCOR is equivalent to a FIRST POS expression.

Examples:

```
?clearscreen xcor
0
?to jump
>setpos list
>   random 150 * first shuffle [1 -1]
>   random 100 * first shuffle [1 -1]
>end
jump defined
?jump xcor
145
?jump xcor
-64
```

```
ycor
----

Action:
Outputs the Y-coordinate of the turtle's current position.

Syntax:
ycor

Explanation:
YCOR  outputs  the  Y-coordinate of the turtle's  current  position.  YCOR  is
equivalent to a LAST POS expression.

Examples:
?clearscreen ycor
0
?to jump
>setpos list
>   random 150 * first shuffle [1 -1]
>   random 100 * first shuffle [1 -1]
>end
jump defined
?jump ycor
36
?jump ycor
49


EOF
```

DRLRMA.WS4 (= "Dr. Logo Reference Manual", Appendix A)

(Retyped by Emmanuel ROCHE.)

Appendix A: Dr. Logo error messages

Table A-1. Dr. Logo error messages

Number	Message
2	Number too big
6	(symbol) is a primitive
7	Can't find lable (symbol)
8	Can't (symbol) from the editor
9	(symbol) is undefined
11	I'm having trouble with the disk
12	Disk full
13	Can't divide by zero
15	File already exists
17	File not found
21	Can't find catch for (symbol)
23	Out of space
25	(symbol) is not true or false
29	Not enough inputs to (procedure)
30	Too many inputs to (procedure)
32	Too few items in (list)
34	Turtle out of bonds
35	I don't know how to (symbol)
36	(symbol) has no value
37	) without (
38	I don't know what to do with (symbol)
40	Disk is write protected
41	(procedure) doesn't like (symbol) as input
42	(procedure) didn't output
43	I don't know how to do that yet
44	!! Dr. Logo system bug !! (Should not occur. Please write to Digital Research if it does.)
45	The word is too long
46	I don't have enough buffer space
47	If wants [ ]'s around instruction list
48	(varies according to disk error)
49	(symbol) isn't a parameter
50	I can't (symbol) while loading
51	The file is write protected
52	I can't find the disk drive

EOF

DRLRMB.WS4 (= "Dr. Logo Reference Manual", Appendix B)

(Retyped by Emmanuel ROCHE.)

Appendix B: Dr. Logo control and escape character commands

Table B-1. Dr. Logo control and escape character commands

Format: Character
Effect

(* indicates the character is valid within screen editor only.)

Ctrl-A
Moves the cursor to the beginning of the line.

Ctrl-B
Moves the cursor [B]ack one character; that is to say, it moves the cursor one position to the left.

Ctrl-C *
Exits the screen editor; updates Dr. Logo's workspace with definitions of all procedures and variables from the screen editor's buffer.

Ctrl-D
[D]eletes the character indicated by the cursor.

Ctrl-E
Moves the cursor to the [E]nd of the line.

Ctrl-F
Moves the cursor [F]orward one character; that is to say, it moves the cursor one position to the right.

Ctrl-G
Outside the screen editor, it immediately terminates the currently executing procedure. Inside the screen editor, it exits the screen editor without updating Dr. Logo's workspace, discarding any changes made during the screen editing session.

Ctrl-H
Deletes the character to the left of the cursor.

Ctrl-I
[I]nserts a tab (three spaces).

Ctrl-J
(No effect.)

Ctrl-K

[K]ills the remaining line; that is to say, it deletes all characters right of the cursor to the end of the line. Deleted characters are stored in buffer.

Ctrl-L
Outside screen editor, displays a full graphic screen, devoting the monitor to graphics. Inside screen editor, it readjusts the display so that the line currently indicated by the cursor is positioned at the center of the screen. If the cursor is less than 12 lines from the beginning of the buffer, the screen editor simply beeps when Ctrl-L is pressed.

Ctrl-M
Generates a Carriage Return.

Ctrl-N
Moves the cursor to the [N]ext line; in the screen editor, the cursor moves down one line towards the end of the buffer.

Ctrl-O
[O]pens a new line. In the screen editor, it is equivalent to pressing Enter followed by Ctrl-B.

Ctrl-P
Moves the cursor to the [P]revious line; the cursor moves up one line towards the beginning of the buffer.

Ctrl-Q
Generates the [Q]uoting character ("\") that makes Dr. Logo treat a delimiter character as a literal character. Delimiter characters are [ ] ( ) " : ; = < > + / ^

Ctrl-R
(No effect.)

Ctrl-S
Displays a [S]plitscreen; opens a text window on the graphic screen.

Ctrl-T
Displays a full [T]ext screen, devoting the monitor to text.

Ctrl-U
(No effect.)

Ctrl-V *
Displays the next screen full of text in the screen editor's buffer, the next 24 lines towards the bottom of the buffer.

Ctrl-W
Interrupts the scrolling of a text display; [W]aits until the next keystroke to continue scrolling the display.

Ctrl-X
(No effect.)

Ctrl-Y

[Y]anks text from the buffer; that is to say, it redisplays the line most recently stored in the buffer by an Enter or a Ctrl-K keystroke.

Ctrl-Z
Interrupts the currently executing procedure; displays a pause prompt to allow interactive debugging. Enter "co" to continue the execution of the interrupted procedure.

ESC-V *
Displays the previous screen full of text in the screen editor's buffer, the previous 24 lines towards the beginning of the buffer.

ESC-< *
Positions the cursor at the beginning of the screen editor's buffer.

ESC-> *
Positions the cursor at the end of the screen editor's buffer.

EOF

DRLRMC.WS4 (= "Dr. Logo Reference Manual", Appendix C)

(Retyped by Emmanuel ROCHE.)

Appendix C: Functional command list

Word and list processing

ascii word
butfirst, bf object
butlast, bl object
char n
count object
emptyp object
equalp object object
first object
fput object object
item n object
last object
list object object (...)
listp object
lowercase, lc word
lput object object
memberp object object
numberp object
piece n object
proclist
quote object
sentence, se object object (...)
shuffle list
uppercase, uc word
where
word word word (...)
wordp object

Arithmetic operations

abs n
arctan n
cos degrees_n
degrees radians_n
exp n
int n
log n
log10 n
pi
product n n (...)

quotient n n
radians degrees_n
random n
remainder n n
rerandom
round n
sin degrees_n
sqrt n
sum n n (...)
tan degrees_n
+ a b
- a b
* a b
/ a b
^ a b

Logical operations
---

and pred_exp pred_exp (...)
not pred_exp
or pred_exp pred_exp (...)
= a b
< a b
> a b

Variables
---

local varname (...)
make varname object
name object varname
namep word
thing varname

Defining procedures
---

copydef new_procname old_procname
defined procname defin_list
definedp object
end
primitivep object
text procname
to procname <inputs>

Editing
---

edall <pkgname | pkgname_list>

edit, ed <name | name_list>
edns <pkgname | pkgname_list>
edps <pkgname | pkgname_list>

Text screen
---

cleartext, ct
cursor
print, pr object (...)
setcursor coord_list
show object
textbg n
textfg n
textscreen
twoscreen
type object (...)

Graphic screen
---

background, bg
clean
clearscreen, cs
dot coord_list
fence
fullscreen
pen
pencolor, pc
pendown, pd
penerase, pe
penreverse, px
penup, pu
setbg n
setpc n
setpen list
setsplit n
splitscreen
turtletext, tt object (...)
window
wrap

Graphic movement
---

back, bk distance_n
forward, fd distance_n
heading
hideturtle, ht
home
left, lt degrees_n

pos
right, rt degrees_n
setheading, seth degrees_n
setpos coord_list
setx n
sety n
shownp
showturtle, st
towards coord_list
xcor
ycor

Workspace management
---

bury pkgname | pkgname_list
erall <pkgname | pkgname_list>
erase procname | procname_list
ern varname | varname_list
erns <pkgname | pkgname_list>
erps <pkgname | pkgname_list>
follow procname procname
nodes
noformat
noprim
package pkgname name | name_list
pkgall pkgname
po procname | procname_list
poall <pkgname | pkgname_list>
pons <pkgname | pkgname_list>
popkg
poprim
pops <pkgname | pkgname_list>
potl
pots <pkgname | pkgname_list>
recycle
unbury pkgname

Property lists
---

glist prop <pkgname | pkgname_list>
gprop name prop
plist name
pprop name prop object
pps <pkgname | pkgname_list>
remprop name prop

Disks
---

copyd dest_d: source_d:
defaultd
initd d: n
resetd <d:>
setd d:
spaced <d:>

Files
---

changef new_fname old_fname
copyf dest_fname source_fname
erf fname
getfs <d:>
load fname
save fname <pkgname | pkgname_list>

Keyboard
---

fkey n instr_word
keyp
readchar, rc
readlist, rl
readquote, rq

Printer
---

copyoff
copyon
printscreen

Sound, lightpen and joystick
---

tones note_list
lpen
lpenp
buttonp paddle_n
paddle n

Conditionals and flow of control
---

bye
co <object>
go word
if pred_exp instr_list <instr_list>

iffalse, iff instr_list
iftrue, ift instr_list
label word
output, op object
repeat n instr_list
run instr_list
stop
test pred_exp
wait n

Error handling and debugging
---

catch name instr_list
debug
error
nodebug
notrace <procname | procname_list>
nowatch <procname | procname_list>
pause
throw name
trace <procname | procname_list>
watch <procname | procname_list>

EOF

DRLRMD.WS4 (= "Dr. Logo Reference Manual", Appendix D)

(Retyped by Emmanuel ROCHE.)

Appendix D: Glossary

address
Location in memory.

ambiguous file name
File name containing a wildcard character. In Dr. Logo, the wildcard character is a question mark ("?"). An ambiguous file name is used to access one or more files. For example, getfs "z? displays all file names that begin with the letter z. You can put the ? after any number of characters or as the only character, but the ? must be the last character in a file name. A file name cannot contain more than eight characters. See wildcard character.

artificial intelligence
Imitation of human-like information processing performed by a computer.

ASCII
Acronym for American Standard Code for Information Interchange. ASCII is a standard code for the computer representation of the numbers, letters, and symbols that appear on most keyboards.

assembly language
Human readable form of machine language. See machine language.

backup or back-up
Duplicate copy of data to be used in case the original is lost, destroyed, or accidentally altered. The process of duplicating a disk or file.

BASIC
Programming language that is widely used in microcomputers because of its English-like structure and ease of use. BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code.

bit
Fundamental unit of information that a computer understands. Abbreviated from Binary digIT. A bit can have one of two values, 0 or 1, meaning off or on, respectively. See byte.

boot
Act of starting up a computer system.

buffer
Area of memory that temporarily stores information during the transfer of information.

bug

Error in a procedure that prevents the procedure from executing as expected.

byte
Unit of memory or disk storage that usually contains eight bits. In ASCII code, each character is represented by one byte.

cathode ray tube, CRT
Picture tube that shows information being entered or output from the computer; functionally similar to those used in standard televisions. Commonly abbreviated CRT; also referred to as the monitor.

cell
See character cell.

central processing unit, CPU
Brain of the computer, commonly abbreviated CPU. Your IBM Personal Computer contains an Intel 8088 CPU, an integrated circuit that contains an arithmetic/logic unit that controls and manipulates the flow of information and its storage in memory.

character
Single letter, number, symbol, space, or punctuation mark. A character is usually stored in one byte of the computer's memory, or on disk.

character cell
Unit of space on the monitor that can hold one character. A cell is located at the intersection of each horizontal row and each vertical column. There are 1,000 character cells on a graphic display, and 2,000 character cells on a text display. Dr. Logo lets you control the background color and foreground color of each character cell on the graphic and text display.

code
Sequence of expressions written in a programming language that instructs the computer to perform a task.

color monitor
Output device that allows you to see the visual field of both the graphic screen and the text screen.

command
Instruction that makes Dr. Logo initiate an action.

comments
Remarks or explanatory notes set off from the rest of a procedure's definition by a semicolon (";").

compiler
Language translator that interprets the text of a high-level language, like BASIC, into machine language code (1s and 0s) understandable to the computer.

concatenation
Joining of two or more objects together, end to end. For example, the primitive word outputs a single word made up of multiple inputs, as in word "sun "shine outputs "sunshine".

constant
Object, word, number, or list with a value that does not change when a procedure runs.

control character
Non-printing character combination that sends a simple command to Dr. Logo. To enter a control character, hold down the control (Ctrl) key, and press the specified character key.

CP/M
Operating system controlling the operation of Dr. Logo. CP/M stands for Control Program for Microcomputers.

CPU
See Central Processing Unit.

crash
Severe hardware or software malfunction. A head crash, for example, generally causes an irretrievable loss of the data on a disk.

CRT
See Cathode Ray Tube.

cursor
Blinking underline symbol on the text display. The cursor designates the position where the next keystroke at the keybaord has an effect. As you type, the cursor automatically advances. You can also move the position of the cursor with the cursor control keys located on the numeric key pad, the tab key ("-->|"), and the backspace key ("<--"). On the graphic display, the turtle designates the point of action.

data
Information that is created, changed, or stored by the computer.

debug
Process of locating and correcting errors in a procedure.

default
Value supplied in the absence of user input. For example, when you start your computer system, drive A: is the default drive. Dr. Logo looks for files on the disk in the default drive, unless you specify another drive.

delimiter
Blank space or any of these special characters used to set another character, word, or number off from the next: [ ] ( ) ; < > + - * / ^.

directory
List of the contents of the disk. Use the getfs primitive to display a list of all the Dr. Logo file names on a data disk.

disk
Magnetic media used to store information. Sometimes called diskette. Programs and information are stored and retrieved like music on a record. The term

diskette usually refers to floppy disks 5 1/4 or 8 inches in diameter. The term disk can refer to a diskette, a removable cartridge disk, or a fixed hard disk.

disk drive
Peripheral device that stores and retrieves information on disks.

display
Visual field of a screen, or to show the contents of the screen on the monitor.

double sided
Disk format that stores information on both sides; a disk drive that can read and write on both sides of a double-sided disk.

Dr. Logo
Digital Research's version of Logo, a programming language with extensions that make it suitable for commercial and academic applications.

edit
Process of adding, modifying, or deleting text in the definition of a user-defined procedure.

editor
Utility program that allows you to add, modify, or delete information from a text screen. See also line editor, procedure editor, and screen editor.

element
Single piece of an object. For example, a numeral is an element of a number, a character is an element of a word, and a word is an element of a list.

end
Special word that indicates the conclusion of a procedure's definition. The word "end" must stand alone on the last line of a procedure.

enter
Act of transmitting information to the computer. The computer does not recognize or process any information typed at the keyboard since the last prompt, until you press the Enter key ("<--+"). The Enter key on an IBM Personal Computer is the equivalent to a Carriage Return ("RETURN") on a standard typewriter keyboard.

evaluate
Process Dr. Logo goes through to execute an expression.

executing a procedure
See running a procedure.

expression
In Dr. Logo, an expresion is a procedure name followed by its inputs.

file
Collection of information on a disk.

file name
Name you assign to a file. The file name can consist of up to eight alphabetic or numeric characters. The first character in a file name must be alphabetic.

floating point
Method Dr. Logo uses internally to express numbers that have a decimal point that can shift to the left or right.

floppy disk
Flexible magnetic disk used to store information. A floppy disk is a thin piece of mylar, coated with magnetic particles. The disk drive causes the floppy disk to rotate inside its paper jacket. Floppy disks come in two sizes: 5 1/4 and 8 inches in diameter.

flowchart
Graphic diagram that uses special symbols to indicate the input, processing, output, and flow of a procedure.

function
Operation or specific task that is called into action by referencing the procedure by name. The result of a function is always a single value. See operation.

function keys
Ten multiple-purpose keys located along the left side of the keyboard, labeled F1 through F10, that you can program to recall command lines of up to 16 characters. Press the Enter key to execute the recalled command line.

graphic displays
Visual representations that illustrate, map, or plot mathematical curves, diagrams, graphs, or charts. Dr. Logo allows you to draw graphic designs with the turtle.

graphic screen
Construct in memory where graphic data is stored that can extend beyond the range of your monitor. You can display the visual field of the graphic screen on a color monitor. Dr. Logo allows you to draw graphic designs on the graphic screen with the turtle.

graphic window
Visual display of the graphic screen on a color monitor; that portion of the monitor dedicated to the graphic screen. Dr. Logo's splitscreen primitive allows you to have both text and graphic windows on the same color monitor.

hardware
Physical components of a computer, such as the central processing unit, random access memory, disk storage devices, monitor, keyboard, printer, lightpen, paddle, or joystick.

high-level language
Computer language written in procedural form that is easily comprehensible by humans. Many machine language instructions are generated for each high-level expression. Dr. Logo is a high-level language.

infix operation
Operations where the primitive name or identifier is positioned between its inputs, as in a + b.

initialize
Act of preparing or formatting a disk, in order to read and write information on it.

input
Object that a procedure requires to complete its task; information entered into the computer by an operator typing at the terminal or by a procedure reading from the disk, or to enter such information.

input device
Mechanism that accepts information from the outside world and converts that information into a form that the computer can use. A keyboard is an input device.

input/output devices
Combination of input and output devices. Commonly abbreviated I/O devices.

integer
Positive or negative whole number with no decimal point.

interface
Object that allows two independent systems to communicate with each other, as an interface between hardware and software in a microcomputer.

interpreter
Computer program that evaluates each line as it is typed at the keyboard, and each line of a procedure every time it is run. Dr. Logo is an interpreted computer programming language.

interpreter prompt
Question mark ("?") signaing that Dr. Logo is expecting you to type something at your keyboard; the interpreter is waiting for input.

I/O
Abbreviation for input/output. This term commonly refers to combinations of input and output devices.

jacket
Stiff paper container that holds a floppy disk.

joystick
See paddle.

kilobyte
1024 bytes, commonly denoted as 1K.

language
System of words and symbols used to write programs that communicate with a computer.

level number
Number enclosed in square brackets ("[" and "]") that precedes each line displayed by the trace and watch primitives. The number indicates the position of the procedure on the stack. See stack.

lightpen
Input device that allows you to interact with the computer without the keyboard. The lightpen sends a screen location to the computer when you touch the screen's surface with the tip of the lightpen.

line editor
Dr. Logo interpreter's utility that allows you to modify the current line of text without erasing and retyping the entire line. Dr. Logo allows you to use certain control characters or function keys to move the cursor left and right over the current line of text to make modifications before you press the Enter key.

LISP
Programming language derived from the words LISt Processing. LISP dominates artificial intelligence programming because of its powerful symbolic and list processing capabilities. Logo evolved from LISP.

list
Type of object used as input to a Dr. Logo procedure. A series of objects (words, numbers, or lists) separated by spaces and enclosed in square brackets ("[" and "]").

load
To move procedures or information from permanent storage on the disk into the computer's memory buffer or workspace.

logical operator
Mathematical symbols representing equals ("="), less-than ("<"), or greater-then (">") as well as the primitives "and", "not", and "or" that are used in predicate expressions; expressions that evaluate to either TRUE of FALSE.

Logo
Name of a programming language derived drom the Greek word logos, which means word. Logo is a computer language that evolved from LISP and is designed to teach the fundamentals of computer programming.

machine language
Sequence of instructions to the machine, written in terms of 1s and 0s, and generally not understandable by humans. See bit.

megabyte
One million bytes, or 1024K bytes. See kilobyte.

memory
That part of a computer system that temporarily stores information. Also called random access memory (RAM) or working storage. Dr. Logo automatically loads into memory when you start up your system. Once Dr. Logo is loaded, it allocates part of memory for your workspace.

microprocessor
Miniaturized integrated circuit usually on a silicon chip that is the brain of the microcomputer.

monitor
Output device that displays the visual field of your graphic or text screen. A color monitor displays both graphic and text screens; a monochrome monitor displays only text.

monochrome monitor
Output device that displays the text screen.

name
Type of object used as input to a Dr. Logo procedure. A special word that identifies a procedure, a variable, a package, a file, or a disk.

number
Type of object used as input to a Dr. Logo procedure. In Dr. Logo, a number is a kind of word and can be used as a variable name if it is preceded by a quotation mark, as in "8. A number cannot be used as the first or only character in a procedure name. In arithmetic operations, you can input negative or positive decimals numbers with up to 15 significant digits.

numeric constant
Real or integer quantity that does not change as the procedure is run.

numeric pad
Separate set of keys, arranged like a 10-key adding machine, located on the right of the IBM Personal COmputer keyboard. Provided because they are useful for entering large amounts of numeric information. Press the NUM LOCK key, located above the numeric pad, to make the keys function as cursor control keys.

object
Type of input to a Dr. Logo procedure. A Logo object can be a word, a number, a list, or anything that is not a procedure name. Variables can represent objects.

operation
Procedure that outputs an object, or a mathematical process such as addition, multiplication, subtraction, or division. In Dr. Logo, an operation is equivalent to a function.

operating system
Master program that supervises the execution of other programs, and manages the computer system's resources. An operating system provides an orderly input/output environment between the computer and its peripheral devices. Dr. Logo runs under the CP/M operating system, which is compatible with many different computer systems.

optional input
Inputs that are not required. In the primitive expression syntax descriptions, optional inputs are enclosed in angle brackets ("<" and ">").

output
Object returned to the caller by a procedure; data that the processor sends to the console, printer, or disk after processing is complete. To send information to the console, printer, or disk. The computer outputs the requested object or data.

output device
Mechanism by which a computer transfers its information to the outside world. The printer is an output device.

package
Group of related procedures or variables. Packaging helps you manage your workspace. When you organize your procedures and variables into packages, you can display, edit, save, or erase a whole group without affecting other procedures or variables in the workspace.

paddle
Input device that allows you to interact with the computer without the keyboard. Common on electronic computer games and also called a joystick. The coordinate scale that the paddle or joystick sends to the computer depends on the kind of device you purchase. Dr. Logo supports up to two paddles or joysticks, each with two buttons.

peripheral device
Mechanisms that are external to the CPU. Terminals printers, disk drives, paddle (joystick), and lightpen are peripheral devices.

permanent storage
Location outside of the computer's memory where data can be stored, usually on disk. When you save a file, you copy the information from the workspace onto the disk.

pixel
Smallest element of a monitor's display; a point within a character cell. Also referred to as a dot. A turtle step, as in the command forward 1, is equivalent to one pixel.

precedence
Order in which Dr. Logo processes arithmetic operations.

predicate expression
Expression that contains a logical operator and outputs either TRUE or FALSE. A predicate expression can be a type of input to a Dr. Logo procedure.

prefix operation
Operations where the primitive name or identifier precedes the inputs, as in - a b, or print [Hi there!].

primitives
Procedures, operations, or commands that make up Dr. Logo; the built-in procedures.

printer
Peripheral device used to put computer information on paper.

printout
Printed material or listing produced by the printer.

procedure
A series of expressions that tell Dr. Logo how to perform a task.

procedure editor
Dr. Logo interpreter's utility that allows you to teach Dr. Logo a new task, that is to say, to define a new procedure. The procedure editor is identified by the greater-than sign (">"), a prompt that appears when you enter the word "to" at the interpreter's question mark prompt ("?"). All the line editor's control character commands function within the procedure editor.

procedure editor prompt
Greater-than sign (">") that tells you Dr. Logo is expecting you to enter a procedure definition line. Dr. Logo does not immediately evaluate the expressions typed at the > prompt.

program
Complete set of instructions designed to tell the computer to perform a specific task. To define a Dr. Logo procedure.

prompt
Cue displayed on the monitor telling the user that Dr. Logo is expecting input. A prompt can be a symbol, such as ?, >, or !. Or a prompt can be a message, such as "Is this what you want (y/n)?". All prompts expect a response from the user. For example, when you see the ! prompt, you are extremely low on free nodes in your workspace. Your response should be to enter the primitive "recycle" to call the garbage collector. See interpreter prompt and procedure editor prompt.

property
Quality or attribute of an object, procedure, or package.

property list
List of property pairs that represent the qualities or attributes connected with an object, procedure, or package. Use the pprop primitive to assign both the Dr. Logo system properties and your own non-system properties. Use the plist primitive to display a property list.

property name
First member of a property pair; the label representing the quality or attribute of an object, procedure, or package. For example, the .BUR property name in a package name's property list signifies that the package is buried.

property pair
Two parts of a property. The first element of a property pair is the property name; the second element is its value. For example, in a procedure's property list is a property pair consisting of the property name .DEF and the value of .DEF, the actual definition of the procedure.

property value
Second member of a property pair; the qualities of or attributes connected

with an object, procedure, or package. For example, the information following a .PAK property name in a property list is the property value, the name of the package to which this object belongs.

RAM
See random access memory.

random access
Process of reading or writing informaion in memory or on a disk in any order.

random access memory
Temporary memory location inside a computer. Also called working storage. Commonly abbreviated as RAM. Size of RAM is measured in kilobytes.

random number
Number selected by chance from a set of numbers. The random primitive returns a random number in Dr. Logo.

read
Process of transferring prestored information from a storage device, such as a floppy disk, into the computer's memory.

Read-Only
Attribute of a disk or a disk drive that allows you to read but not write to the disk or disk drive.

Read-Only disk
Attribute assigned to a disk that allows you to read from that disk, but not change it. To assign the Read-Only attribute to a 5 1/4 inch floppy disk, disable the Read-Write notch on the upper left side of the disk by placing a sticker over it.

Read-Only disk drive
Attribute assigned to a disk drive that allows you to read any file on the disk but prevents you from adding a new file, erasing a file, changing a file, or renaming a file. A drive has the Read-Only attribute when a new disk is inserted. Use the primitive resetd to give the drive a Read-Write capability.

Read-Write
Attribute of a disk or a disk drive that allows you to read from and write to the disk or the disk drive. A 5 1/4 inch floppy disk can be set to Read-Write by cutting out the Read-Write notch on the upper left corner of the disk. A drive can be set to Read-Write by entering the resetd command after inserting a new disk.

real number
Numeric value specified with a decimal point, internally represented in floating point notation.

reserved word
Keyword that has specific meaning to a given language or operating system. In Logo, the keywords are called "primitives". A reserved word cannot be used as a procedure or variable name, unless you have set the system variable REDEFP to TRUE.

running a procedure
Process of Dr. Logo evaluating and performing the expressions in a procedure.

save
To transfer information from random access memory into permanent storage on the disk.

screen
Construct in memory where text or graphic data is stored. Dr. Logo supports two kinds of screens: a text screen and a graphic screen. A color monitor can display both kinds of screens, and a monochrome monitor can display only the text screen.

screen editor
Dr. Logo utility that allows you to define a procedure or modify a procedure without retyping the complete definition. In the screen editor, you can use special screen editing control characters that enable you to move from line to line in a procedure, as well as all of the line editor's control character commands. You can start the screen editor by typing the commands ed, edall, edps, or edns at Dr. Logo's ? prompt.

single-sided
Disk format that stores information on only one side; a disk drive that can read or write on only one side of a single-sided disk.

sixteen-bit computer
Computer system that is capable of processing information sixteen bits at a time, potentially twice as fast as an eight-bit system. Sixteen-bit systems can usually accomodate more RAM than eight-bit systems, making some applications more efficient.

software
Set of instructions that tells the hardware how to complete a particular task. Unlike hardware, software is not an electronic product, and does not perform any physical work. Software is a communication tool that allows you to interact with the hardware.

splitscreen
Display where a text window is opened on the graphic screen of a color monitor.

stack
Area in memory used by the interpreter to keep track of which procedure is currently running.

store
To save information. See save.

superprocedure
Procedure that is never called by any other procedure. You call a superprocedure into action by entering the procedure name at Dr. Logo's ? prompt. Use the potl primitive to display the names of superprocedures.

syntax
Format for entering an expression.

syntax error
Error that results from entering an expression that does not conform to the syntax rules.

temporary storage
See random access memory.

terminal
Input/output device consisting of a monitor and a keyboard.

text screen
Area in memory where text data is stored. You can display the visual field of the text screen on either a monochrome or color monitor.

text window
Visual display of the text screen on either a color monitor or a monochrome monitor; that portion of the display dedicated to the text screen. Dr. Logo's splitscreen primitive allows you to have both text and graphic windows on the same color monitor.

toplevel
Interpreter's prompt. When Dr. Logo displays a question mark ("?"), there are no procedures on the stack, and the level number is 0.

trace
Option used for debugging a procedure while it is running. The trace option displays the name of each procedure as it is called, and the name and value of each variable as it is defined. Trace allows you to observe the details of the procedure's execution without interruption.

turtle
Graphic symbol that functions as a cursor appearing on the Dr. Logo graphic screen.

turtle graphics
Dr. Logo's graphics environment. The Logo language allows you to draw on the graphic screen by directing the turtle's movement. As the turtle moves, it leaves a trace of its path on the screen.

turtle step
One pixel or dot on the IBM Personal Computer.

user
Person who operates a computer.

user-friendly
Quality of a piece of software that makes it simple for the inexperienced person to use.

utility program
Software tool that enables the user to perform certain operations. For

example, the editor is a utility program that enables you to add, change, and delete text in a Dr. Logo procedure.

value
Quantity expressed by an integer or real number; the object assigned to a variable or property name.

variable
Name to which Dr. Logo can assign an object as a value. A variable can be thought of as a container that can hold a value.

visual field
Portion of the screen displayed on the monitor.

watch
Option used for debugging a procedure while it is running. The watch option displays the procedure's name and the expression that the interpreter will evaluate next, one line at a time. It waits for you to press the Enter key before it executes that line. This feature allows you to interact with the interpreter or edito during the execution of a procedure.

wildcard character
Question mark ("?") character that gives Dr. Logo a pattern to match when searching in a disk directory for a file name. A ? placed in a file name creates an ambiguous file name. See ambiguous file name.

window
Visual portion of the screen. See graphic window and text window.

word
Type of object used as input to a Dr. Logo procedure. A group of one or more consecutive characters separated from other characters on the line by a delimiter.

working storage
See random access memory.

workspace
Dr. Logo's temporary storage area for information, such as your procedures, variables, and property lists.

EOF

DRLRME.WS4 (= "Dr. Logo Reference Manual", Appendix E)

(Retyped by Emmanuel ROCHE.)

Appendix E: Getting Started

This section quickly leads you through a complete session with Dr. Logo. It shows how to start up and shut off Dr. Logo, how to initialize a disk, define a Dr. Logo procedure, save and load files, specify the default drive, and back up your data disk and files.

The beginning of this book explain why you must enter quotation marks and other punctuation; to keep this introduction brief, the explanations are not repeated here. To complete this session, simply type each of the examples exactly as it is shown, and then press the Enter key ("<--+"). If you make a typing error, use the backspace key ("<--") to correct it, or retype the line. Do not worry about mistakes; you can't hurt Dr. Logo or your computer with typing errors.

If, while reading this section, you find a computer-related term you do not understand, you can find its definition in Appendix D, the glossary.

E.1 Starting up Dr. Logo

Your computer automatically loads Dr. Logo into memory when you turn on your computer. You can start up Dr. Logo in one of two ways, depending on whether your computer is powered OFF or ON.

If the power is OFF

1. Open the door of drive A: by lifting the load latch outwards.

2. Insert the Dr. Logo system disk with the label face up and the disk jacket seams underneath. Hold the label in your hand as you slide the disk into the slot. See Figure E-1.

   Figure E-1. Inserting a disk (not shown)

3. Close the disk drive door by pulling the load latch down.

4. Turn ON the printer, if you have one, the color monitor and monochrome monitor if you have one, and finally the computer system unit. (If you have just powered off, but want to use your computer again, you must count slowly to five before you power ON.)

If the power is ON
---

1. Insert the Dr. Logo system disk into drive A: and close the door.

2. Hold down the Ctrl and Alt keys, and press the Del key. Then, release all three keys. This sequence is called system reset. See Figure E-2.

Figure E-2. System keyboard (not shown)

In either start sequence, wait a moment while the system does some self-testing. Watch for the red light on drive A: to flash, and listen for the drive to read Dr. Logo from the system disk into your computer's memory. While Dr. Logo is being loaded, it displays the following message on your monitor:

```
+--------------------------------------+
|  Welcome to Dr. Logo, Version V.V    |
| Copyright (C) 1983, Digital Research |
|      Pacific Grove, California       |
|                                      |
|             Please Wait              |
|                                      |
|                                      |
|                                      |
|                                      |
+--------------------------------------+
```

Figure E-3. Dr. Logo banner

The version number, represented above by V.V, tells you the major and minor revision level of the Dr. Logo version that you own. After this greeting disappears, a question mark ("?") prompt and the flashing underline ("_") display on your text screen. The question mark tells you that Dr. Logo is waiting for you to type something at your keyboard. The underline tells you where the next character you type will appear.

Once in memory, Dr. Logo allocates a part of memory for your workspace. When you start Dr. Logo, the workspace is empty, ready for you to type in new procedures or load previously saved procedures from a disk.

E.2 Defining Dr. Logo procedures

A procedure is a list of instructions that tells Dr. Logo how to do a task. Dr. Logo performs the task when you enter the procedure's name. The list of instructions is the definition of the procedure. You use primitives and previously-defined procedures to define a new procedure. For example, here is how to enter a procedure that makes Dr. Logo draw a square on the graphic screen with the turtle. Type the following:

```
?to square
>repeat 4 [forward 60 right 90]
>end
```

```
square defined
?
```

When you type the statement "to square", Dr. Logo responded with a > prompt on the next line. The special word "to" signals to Dr. Logo that you are starting to define a new procedure. The greater-than sign (">") indicates that Dr. Logo is remembering your instructions as a procedure definition.

After you define the procedure, you can make Dr. Logo perform the new task by typing the procedure name at Dr. Logo's ? prompt and pressing the Enter key. For example, type

```
?square
?
```

cs stands for clearscreen, and erases the square from the screen. You can teach Dr. Logo to draw a triangle as follows:

```
?cs
?to triangle
>right 45 forward 35
>right 90 forward 35
>left 225 forward 50
>end
triangle defined
?triangle
?
```

Now, you can use the triangle and the square procedures in the definition of another procedure. For example,

```
?cs
?to draw.house
>square
>forward 50
>triangle
>end
draw.house defined
?draw.house
?
```

You can look at the names of the procedures you have defined by asking Dr. Logo to print out titles, pots in short:

```
?pots
to square
to triangle
to draw.house
?
```

When you turn off your computer, everything in your workspace disappears. If you were to turn off your computer now, these procedures would be lost. To be able to use them later, you must save them on a disk.

But, before you try to save your procedures, remember that your Dr. Logo system disk is write-protected. That means you can only read Dr. Logo from it, and cannot save procedures on it. You can take your Dr. Logo system disk out of the A: drive as soon as Dr. Logo is loaded. Store your system disk in a safe place, away from heat and magnetic devices such as telephones and vacuum cleaners.

Before you can save procedures, you must create a data disk, a disk that can store procedures in files, and insert it in a drive. A disk fresh from the box is not ready to store files; you must initialize (format) a new disk before you can read or write any data on it.

E.3 Initializing a disk

You must always initialize a disk that is fresh from the box. If you have an old disk that you want to reuse, you can initialize the disk to remove all the old data on the disk and reformat it, as if it were new.

Before you begin

1. Make sure that the read/write notch is cut out of the upper left edge of the disk jacket.

2. Check to see if you have single- or double-sided disk drives. If you have single-sided disk drives, you must initialize your disks to single-sided format. A double-sided drive can handle either single- or double-sided disks.

You will need to initialize two new disks, one to be your data disk and the other to be your back-up disk.

If you have a one-drive system

1. If you have not already done so, remove your Dr. Logo system disk, and store it in a safe place, away from heat and magnetic devices, such as telephones and vacuum cleaners. Insert the disk to be formatted in drive A: and close the door.

2. If you have a single-sided drive, you must initialize the disk to single-sided. Type

   ```
   ?initd "a: 1
   ```

   Dr. Logo displays the following prompt:

   ```
   I will erase the diskette in drive A
   and will make it single-sided.
   Is this what you want (y/n)?
   ```

If you have a double-sided drive, you can initialize the disk to double-sided. Type

```
?initd "a: 2
```

Dr. Logo displays the following prompt:

```
I will erase the diskette in drive A
and will make it double-sided.
Is this what you want (y/n)?
```

3. Type y (yes) to format the disk. Type n (no) if you have not removed the Dr. Logo system disk or, for some other reason, wish to return to Dr. Logo's ? prompt. After you type y, watch for the drive's red light to flash, and listen as Dr. Logo formats the disk. When the formatting is complete, Dr. Logo's ? reappears.

4. Format a second disk, repeating steps 1 through 3, for a back-up disk.

If you have a two-drive system
---

1. You can leave your Dr. Logo system disk in drive A:. Insert the disk to be formatted in drive B: and close the door.

2. If you have single-sided drives, you must initialize the disk to single-sided. Type

```
?initd "b: 1
```

Dr. Logo displays the following prompt:

```
I will erase the diskette in drive B
and will make it single-sided.
Is this what you want (y/n)?
```

If you have double-sided drives, you can initialize the disk to double-sided. Type

```
?initd "b: 2
```

Dr. Logo displays the following prompt:

```
I will erase the diskette in drive B
and will make it double-sided.
Is this what you want (y/n)?
```

3. Type y (yes) to format the disk. Type n (no) if you wish to return to Dr. Logo's ? prompt. After you type y, watch for the B: drive's red light to flash, and listen as Dr. Logo formats the disk. When the formatting is complete, Dr. Logo's ? reappears.

4. Format a second disk, repeating steps 1 through 3, for a back-up disk.

E.4 Saving Dr. Logo files

Saving a file means copying the contents of your workspace to permanent storage on the disk.

1. If you have a single drive system with a data disk in drive A:, type the following command to save the contents of the workspace in a disk file.

    ```
    ?save "home
    ```

    If you have a two-drive system with a data disk in drive B:, type

    ```
    ?save "b:home
    ```

    You have now made a copy of your three procedures square, triangle, and draw.house in a file named "home" on the disk in either drive A: or drive B:. Your procedures are still in Dr. Logo's workspace, as you can see with a pots command.

    ```
    ?pots
    to square
    to triangle
    to draw.house
    ?
    ```

2. To verify that Dr. Logo has saved your procedures, ask it to show you the names of Dr. Logo files on the disk.

    ```
    ?getfs
    [HOME]
    ```

    Or, if you saved the file on drive B:, type

    ```
    ?getfs "b:
    [HOME]
    ```

3. Now that you have saved your procedures, you can erase them from your workspace with confidence they can be restored. Type

    ```
    ?erall
    ```

    Now, to verify that the workspace has been erased, ask Dr. Logo to print out the titles of procedures in the workspace by typing

    ```
    ?pots
    ?
    ```

    Dr. Logo displays only the ? prompt, because nothing is in the workspace.

E.5 Loading Dr. Logo files

To load a file means to move the procedures or other information from permanent storage on the disk into the Dr. Logo workspace. To load a file, you must have saved it on the disk, and be able to see its name with a getfs command.

1. To load the file you just saved, type

```
?load "home
```

   Or, if you saved the file on drive B:, type

```
?load "b:home
```

   Listen ad Dr. Logo moves the procedures in the file from the disk into the workspace, and displays the following list on your screen:

```
square defined
triangle defined
draw.house defined
```

2. Verify that the procedures are in your workspace by executing them, or by using pots to print out their titles.

```
?draw.house
?pots
to square
to triangle
to draw.house
?
```

E.6 Specifying the default drive

When you do not specify a drive in a disk command like save, load, or getfs, Dr. Logo automatically looks for files, or writes files, on the disk in the default drive. When you start Dr. Logo, A: is the default drive. If you have a one-drive system, you only have drive A:, and do not need to specify a drive name in any of your commands. If you have two drives and leave your system disk in drive A:, enter the following commands to change yor default drive to B:.

1. Determine which drive is the default drive by typing

```
?defaultd
A:
```

2. Change the default drive to drive B: by typing

```
?setd "b:
?defaultd
B:
```

3. To verify that Dr. Logo now looks on drive B: for files, type

```
?getfs
[HOME]
```

E.7 Backing up a disk

To back up a disk means to make a duplicate copy of its contents. Professional programmers avoid losing programs by making copies of valuable disks and files. If your working copy is accidentally damaged or erased, you can restore it from the back-up copy.

The frequency of making copies varies with each programmer, but as a general rule make a copy when it would take 10 to 20 times longer to re-enter the information than to make the copy.

To make a back-up disk, you need an initialized or formatted disk on which to write. Make sure your data disk and your back-up disk are the same format: double-sided or single-sided.

If you have a one-drive system

1. Start with the data disk that you plan to copy in the drive. This is your source disk.

2. To start the copying process, type

```
?copyd "a: "a:
```

Figure E-4. System unit with one drive (not shown)

If you have a two-drive system

1. Insert the back-up disk into drive B: and the data disk into drive A:.

2. To make an exact, track-to-track copy of the disk in drive A: onto the disk in drive B:, type

```
?copyd "b: "a:
```

Dr. Logo displays the following message before it begins to copy:

```
I will copy from the diskette in drive A
to the diskette in drive B, erasing any
```

existing data on the diskette in drive B.
Is this what you want (y/n)?

3. Type y (yes) to copy the data disk onto the back-up disk. Type n (no) to return to Dr. Logo's ? prompt without making the copy. If you type y, watch for the red lights to flash back and forth from drive A: to drive B:, and listen as the data is transferred from one disk to the other.

Figure E-5. System with two drives (not shown)

E.8 Copying a file

Another way to back up a file is to make a copy of just that file, without making a copy of the entire disk.

If you have a one-drive system

1. Start with the data disk containing the original file in the drive. This is called your source disk.

2. To copy the file, type

```
?copyf "home "home
```

3. After Dr. Logo displays a message, remove the source data disk, and insert your back-up data disk into the drive.

If you have a two-drive system

1. To copy a file from the data disk in drive A: to the back-up disk in drive B:, type

```
?copyf "b:home "a:home
```

2. Verify that the file has been copied with a getfs command.

```
?getfs "b:
[HOME]
```

E.9 Shutting off Dr. Logo

Before you turn off the power to your computer, be sure to check that you have performed the following steps:

1. Save the contents of your workspace on disk, so that you can load it

later.

2. Make back-up copies of your files on a back-up disk.

3. Take your disks out of the drives. NEVER turn off your system with the disk engaged in the drive; this can permanently damage your disk.

4. Turn the power switches on your monitor, printer, and system unit to OFF.

EOF