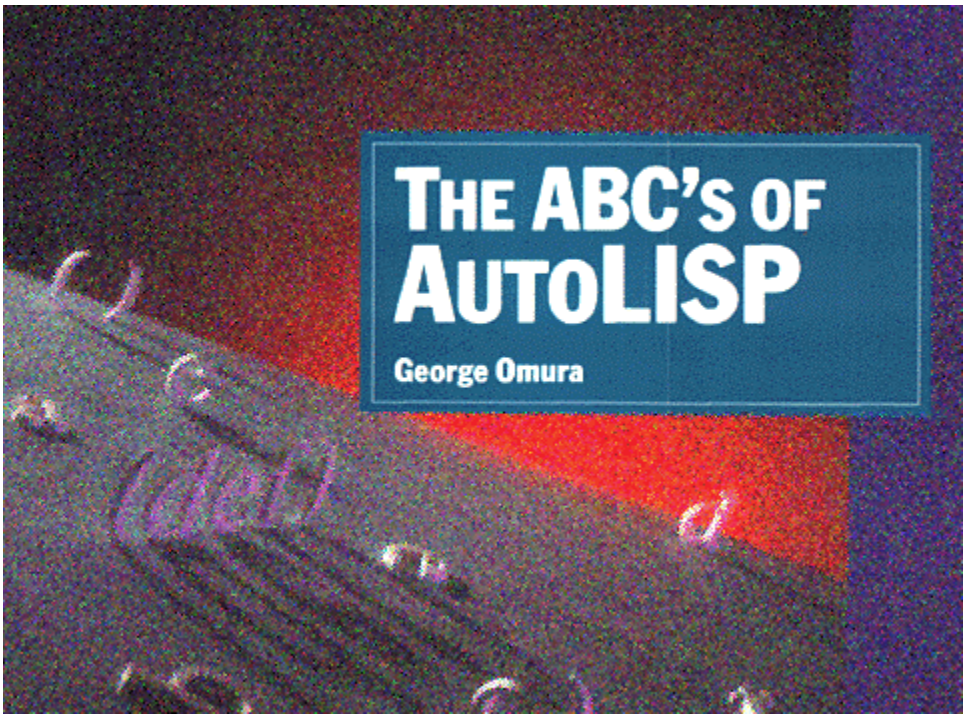


Introduction

One of AutoCAD's greatest assets is its adaptability. You can control just about every aspect of AutoCAD's operations from the appearance of its drawing editor to its variety of menus. A key element of this adaptability is its built-in programming language, AutoLISP. With AutoLISP, you can virtually write your own commands or redefine others.

You can think of AutoLISP as a very sophisticated macro-building facility. (Simple macros are like scripts that automate repetitive keystrokes.) You don't need to be a programmer to use AutoLISP. In fact, AutoLISP is designed so that everyday users of AutoCAD can start to use it after a minimum of training. This book makes AutoLISP accessible to AutoCAD users who are looking for a way to enhance and extend their use of AutoCAD.

Who should read this book

This book introduces nonprogrammers to the use of AutoLISP. If you are an intermediate level AutoCAD user, interested in learning about this powerful tool, then this is the book for you. If you are just beginning to learn AutoCAD, then you should probably become a bit more familiar with AutoCAD before attempting to learn AutoLISP. This book assumes that you have at least an intermediate level of expertise with AutoCAD and are acquainted with simple Windows operations.

How This Book Is Organized

The book is divided into 11 chapters. The first three chapters give you an introduction to programming in AutoLISP. The Chapter 1 introduces you to AutoLISP by showing you how to use it directly from the AutoCAD command prompt. The Chapter 2 shows you how to create and save programs in a file. Chapter 3 discusses ways of organizing your programming projects and how to manage your computers' memory.

The next four chapters show you how to use AutoLISP to do a variety of editing tasks. Chapter 4 discusses the functions that allow you to ask the user for input. Chapter 5 explains how to build decision-making capabilities into your programs. Chapter 6 shows you how to deal with geometric problems using AutoCAD. Chapter 7 discusses the manipulation of text.

The last four chapters show you how AutoCAD and AutoLISP interact. In Chapter 8, you will see how you can control many facets of AutoCAD through AutoLISP. Chapter 9 delves into lists, a fundamental component of all AutoLISP programs. Chapter 10 shows you ways of modifying AutoCAD objects by directly accessing the AutoCAD drawing database. And finally, Chapter 11 looks at ways to dig deeper into the drawing database to get information on complex drawing objects like polylines and block attributes.

In addition, five appendices are included as reference material to the book. In the original version of this book, these appendices contained the resources indicated by their title. In this electronic version, these appendices offer directions on how to find information in the AutoCAD help system. The first three show you how to find information on the AutoCAD menu structure, AutoLISP error messages, and AutoCAD group codes. The fourth appendix describes how to find informaton on the standard AutoCAD dimension variables and system variables. The fifth appendix describes how to find informaton on the Table group codes.

How to Use This Book

Each chapter offers exercises and sample programs that demonstrate some general concept regarding AutoLISP. Through these exercises, the book shows you how programs develop from ideas into finished, running programs. Also, the information you learn in one chapter will build on what you learned in the previous chapter. This way, your knowledge of AutoLISP will be integrated and cohesive, rather than fragmented. For this reason, the best way to use this book is to read each chapter in order and do all of the exercises. Since the topics are oriented toward accomplishing tasks rather than simply focusing on individual functions, you will have a good grasp of how to use AutoLISP in real world situations by the end of this book.

Where to Find the LISP Programs

As you read the chapters and do the exercise, you will be asked to enter program code into a file. If you are in a hurry, you can cut and paste the code directly from the chapter you are reading. This will save a good deal of time, but make sure you study the code that you cut and paste.

This book was originally published in 1990 by Sybex Inc. It has been reproduced here in an electronic format by the Author for the benefit of Mastering AutoCAD readers everywhere. Enjoy....

Chapter 1: Introducing AutoLISP

Featuring

If you have never programmed a computer before, you may think that learning AutoLISP will be difficult. Actually, when you use a program such as AutoCAD, you are, in a sense, programming your computer to create and manipulate a database. As you become more familiar with AutoCAD, you may begin to explore the creation of linetypes and hatch patterns, for example. Or you may customize your menu to include your own specialized functions and macros. (Macros are like scripts that the computer follows to perform a predetermined sequence of commands.) At this level, you are delving deeper into the workings of AutoCAD and at the same time programming your computer in a more traditional sense.

Using AutoLISP is really just extending your knowledge and use of AutoCAD. In fact, once you learn the basic syntax of AutoLISP, you need only to familiarize yourself with AutoLISP's built-in functions to start writing useful programs. (AutoLISP's *syntax* is the standard order of elements in its expressions.) You might look at AutoLISP functions as an extension to AutoCAD's library of commands. The more functions you are familiar with, the better equipped you are for using the program effectively.

AutoLISP closely resembles Common LISP, the most recent version of the oldest artificial intelligence programming language still in use today. AutoLISP is essentially a pared down version of Common LISP with some additional features unique to AutoCAD. Many consider LISP to be one of the easiest programming languages to learn, partly because of its simple syntax. Since AutoLISP is a subset of common LISP, it is that much easier to learn.

In this chapter, you will become familiar with some of the basic elements of AutoLISP by using AutoLISP directly from the AutoCAD command prompt to perform a few simple operations. While doing this, you will be introduced to some of the concepts you will need to know to develop your own AutoLISP applications.

Understanding the Interpreter and Evaluation

AutoLISP is accessed through the AutoLISP interpreter. When you enter data at the AutoCAD command prompt, the interpreter first reads it to determine if the data is an AutoLISP formula. If the data turns out to be intended for AutoLISP, then AutoLISP evaluates it, and returns an answer to the screen. This process of reading the command prompt, evaluating the data, then printing to the screen, occurs whenever anything is entered at the command prompt

and is an important part of how AutoLISP functions.

In some ways, the interpreter is like a hand-held calculator. Just as with a calculator, the information you wish to have AutoLISP evaluate must follow a certain order. For example, the formula 0.618 plus 1 must be entered as follows:

(+ 0.618 1)

Try entering the above formula at the command prompt. AutoLISP evaluates the formula (+ 0.618 1) and returns the answer, 1.618, displaying it on the prompt line.

This structure-+ 0.618 1-enclosed by parentheses, is called an *expression* and it is the basic structure for all AutoLISP programs. Everything intended for the AutoLISP interpreter, from the simplest expression to the most complex program, must be written with this structure. The result returned from evaluating an expression is called the *value* of the expression.

The Components of an Expression

An AutoLISP expression must include an operator of some sort followed by the items to be operated on. An *operator* is an instruction to take some specific action such as adding two numbers together or dividing one number by another. Examples of mathematical operators include the plus sign (+)for addition and forward slash (/) for division.

We will often refer to the operator as a *function* and the items to be operated on as the *arguments* to the function or simply, the arguments. So, in the expression (+ 0.618 1), the + is the function and the 0.618 and 1 are the arguments. All AutoLISP expressions, no matter what size, follow this structure and are enclosed by parentheses.

Parentheses are important elements of an expression. All parentheses must also be balanced, that is, for each left parenthesis, there must be a right parenthesis. If you enter an *un*balanced expression into the AutoLISP interpreter, you get the following prompt:

((_>

where the number of parentheses to the left is the number of parentheses required to complete the expression. If you see this prompt, you must enter the number of closing parentheses indicated in order to return to the command prompt. In this example, you would need to enter two right parentheses to complete the expression.

Double quotation marks enclosing text must also be carefully balanced. If an AutoLISP expression is unbalanced, it can be quite difficult to complete it and exit AutoLISP. **Figure 1.1** shows the components of the expression you just entered.

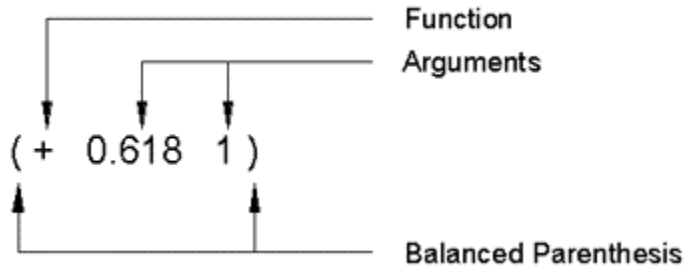


Figure1.1: The parts of and AutoLISP expression

Note that spaces are used to separate the functions and arguments of the expression. Spaces are not required between the parentheses and the elements of the expression though you can add spaces to help improve the readability of expressions when they become complex. However, it is very important to maintain spaces between the elements of the expression. Spaces help both you and AutoLISP keep track of where one element ends and another begins.

Using Arguments and Expressions

AutoLISP evaluates everything, not just expressions, but the arguments in expressions as well. This means that in the above example, AutoLISP evaluates the numbers 0.618 and 1 before it applies these numbers to the plus operator. In AutoLISP, numbers evaluate to themselves. This means that when AutoLISP evaluates the number 0.618, 0.618 is returned unchanged. Since AutoLISP evaluates all arguments, expressions can also be used as arguments to a function.

For example, enter the following at the command prompt:

(/ 1 (+ 0.618 1))

In this example, the divide function (/) is given two arguments-number 1 and an expression (+ 0.618 1). This type of expression is called a *complex* or *nested* expression because one expression is contained within another. So in our example, AutoLISP first evaluates the arguments of the expression, which are the expression (+ 0.618 1) and the number 1. It then applies the resulting value of the expression and the number 1 to the divide function and returns the answer of 0.618047 (see figure 1.2).

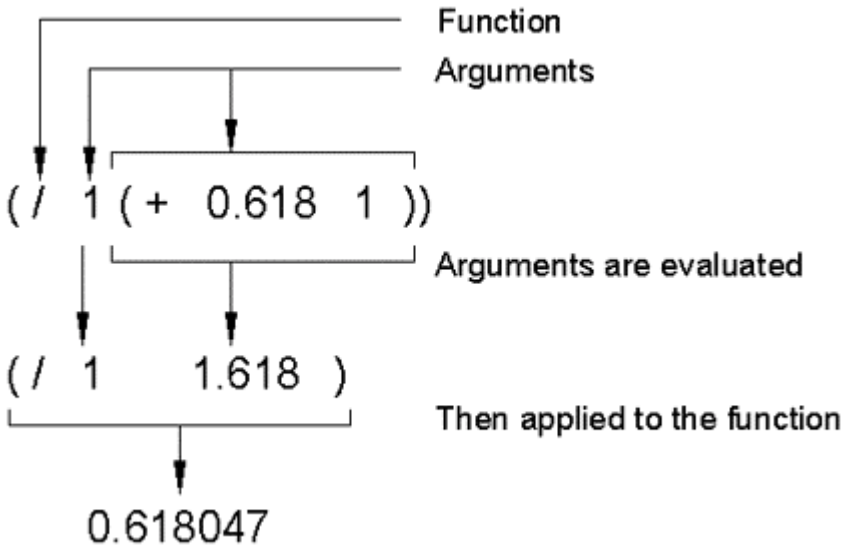


Figure 1.2: Evaluation of a nested expression

Using Variables

Another calculator-like capability of the interpreter is its ability to remember values. You probably have a calculator that has some memory. This capability allows you to store the value of an equation for future use. In a similar way, you can store values using variables.

A variable is like a container that holds a value. That value can change in the course of a program's operation. A simple analogy to this is the title of a government position. The position of president could be thought of as a variable. This variable can be assigned a value, such as Ronald Reagan or Bill Clinton.

Understanding Data Types

Variables can take on several types of values or data types. Here is what some of these data types look like in AutoLISP.

DATA TYPE	EXAMPLE
Integer	24
Real Number	0.618
String	``20 Feet 6 Inches"
List	(4.5021 6.3011 0.0)
File Descriptor	<File: a620>
Object Name	<Object name: 60000014c>
Selection Set	<Selection set: 1>
Symbols	Point1
Subrs	Setq

By separating data into types, the interpreter is better able to determine precisely how to evaluate the data and keep programs running quickly. Also, a computer stores different types of data differently, and so data types help AutoLISP to manage its memory more efficiently. Finally, data types help keep your programming efforts clear by forcing you to think of data as having certain characteristics. The following descriptions give you an idea of what each of these data types are.

Integers and Real Numbers

Integers are whole numbers from -32768 to + 32767. The value of an expression containing only integers is always an integer. For example, the value of the expression
(/ 25 2) is 12. The decimal value is dropped from the resulting value.

Real numbers are numbers that include a decimal value. If the same expression above is written using real numbers, (/ 25.0 2.0), its value will be expressed as the real number 12.5. Integers have a black and white quality about them. 24 will always equal 24. Real numbers (sometimes referred to as *reals*), on the other hand can be a bit less definite. For example, two real values, 24.001245781 and 24.001245782 are nearly identical but are not equal. If you were to drop the last decimal place in both these numbers, then they would be equal values. This definitive quality of integers makes them more suited to certain types of uses, like counting, while real numbers are better suited to situations that require exacting values such as coordinate values and angles. Also, computations performed on integers are faster than those performed on reals.

You may have noticed that in our previous examples, the real number 0.618 is preceded by a zero and not written as .618. In AutoLISP, real numbers with values between 1.0 and 0.0 must begin with zero. If you do not follow this rule, you will get an error message. Enter the following at the command prompt:

(+ .618 1)

Though the above expression looks perfectly normal, the following error message appears:

error: invalid dotted pair

Most beginners and even some experienced AutoLISP users might be completely baffled by the error message. We will look at what dotted pairs are later in this book but for now, just keep in mind that real values between 1.0 and 0.0 must be entered with a 0 preceding the decimal point.

Strings

The term string refers to text. Strings are often used as prompts in AutoLISP expressions but they can also be manipulated using AutoLISP. For example, using the **Strcat** AutoLISP function, you could combine two strings, "thirty seven feet" and "six inches", into one string "thirty seven feet six inches". Try entering this:

(strcat "thirty seven feet " "six inches")

The following is returned:

"thirty seven feet six inches"

Lists

Lists are data elements enclosed in parentheses. They are the basic data structure in AutoLISP. A list can be made up of any number of integers, real numbers, strings, and even other lists.

There are two types of lists. Those intended for evaluation and those intended as repositories for data. When a list contains a function as its first element, we can generally assume that it is an expression intended for evaluation. Such a list is often referred to as a *form*. An example of a list as a repository of data is a list that represents a coordinate location. For example, the list

(1.2 2.3 4.4)

contains three elements, an X, Y, and Z coordinate. The first element, 1.2, is the x coordinate, the second element, 2.3 is the y coordinate, and the third element, 4.4, is the z coordinate.

File Descriptors

AutoLISP allows you to read and write text files to disk. *File descriptors* are used in a program to access files that have been opened for processing. You might think of a file descriptor as a variable representing the file in question. We will discuss this data type in more detail in Chapter 7.

Object Names

Every object in an AutoCAD drawing has a name. The name is an alphanumeric code unique to that object. This name can be accessed by AutoLISP and used as a means of selecting individual objects for processing. Object names are provided by AutoCAD and are not user definable. Also Object names can change from one drawing session to another.

Selection Sets

Just as you can define a group of objects for processing using the AutoCAD Select command, you can also assign a group of objects, or a *selection set*, to a variable in AutoLISP for processing. Selection sets are given names by AutoCAD.

Symbols

AutoLISP treats everything as data to be evaluated. Therefore, *symbols*, or names given to variables, are also data types. Symbols are usually text, but they can also contain numbers like Point1 or dx2. A symbol must, however, start with a letter.

Subrs

Subrs are the built-in functions offered by AutoLISP. These functions perform tasks ranging from standard math operations such as addition and subtraction, to other more complex operations such as obtaining information from the drawing database about a specific object.

Atoms

There are really two classes of data, lists and atoms. You have already seen an example of a list. An *atom* is an element that cannot be taken apart into other elements. For example, a coordinate list can be "disassembled" into three numbers, the x value, the y value, and the z value, but the x, y and z values cannot be taken apart any further. In a coordinate list, the x, y, and z values are atoms. Symbols are also atoms because they are treated as single objects. So, in general, atoms are either numbers or symbols.

Assigning Values to Variables with Setq

Variables are assigned values through the use of the Setq function. As you have seen, a function can be a simple math operator such as plus or divide. A function can also consist of a set of complex instructions to perform more than one activity, like a small program.

The Setq function tells AutoLISP to assign a value to a variable. For example, Try the following exercise to assign the value 1.618 to the variable named Golden:

1. Enter the following at the command prompt:

   (setq golden 1.618)

   You can now obtain the value of a variable by preceding the variable name by an exclamation point. Now check the value of Golden.

2. Enter

   !golden

The value 1.618 is returned. You might think of the exclamation point as another way of saying "Display the contents of."

Setq will assign a value to a variable even if the variable already has a value assigned to it. See what happens when Golden is assigned a new value.

3. Enter the following:

(setq golden 0.618)

Golden is reassigned the value 0.618 and the old value, 1.618, is discarded. You can even reassign a value to a variable by using that variable as part of the new value as in the following expression

(setq golden (+ golden 1))

In this example, Golden is assigned a new value by adding 1 to its current value.

Preventing Evaluation of Arguments

But something doesn't seem quite right in the above example. Earlier, we said that AutoLISP evaluates the Arguments in an expression before it applies the arguments to the function. In the above example, we might expect AutoLISP to evaluate the variable Golden before it is applied to the **Setq** function. Since Golden is a variable whose value is 0.618, it would evaluate to 0.618. AutoLISP should then try to set 1.618 equal to 0.618, which is impossible. The value returned by the argument (+ golden 1) cannot be assigned to another number (see **Figure 1.3**).

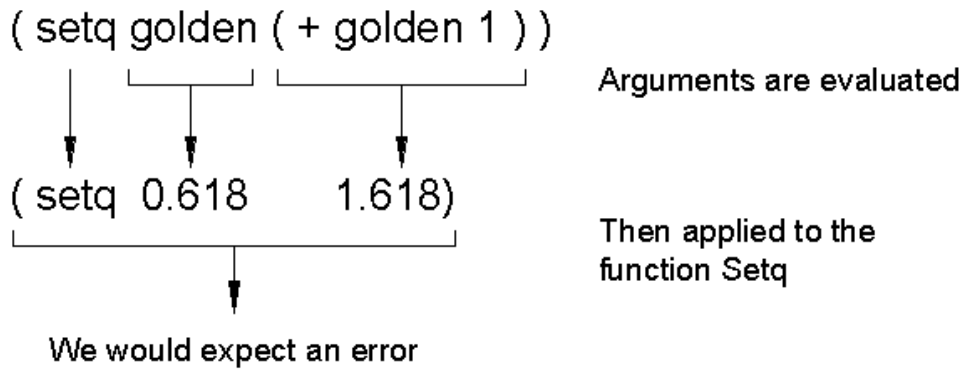


Figure 1.3: The expected outcome of setq

Here's why the above example works. Setq function is a special function that is a combination of two other functions, Set and Quote (hence the name Setq). As with Setq, the function Set assigns the value of the second argument to the value of the first argument. The Quote function provides a means of preventing the evaluation of an argument. So, both Setq and Set Quote prevent the evaluation of the first argument, which in the above example is the variable Golden.

You could write the above example as

(set quote golden (+ golden 1))

and get the same answer. Or you could abbreviate the Quote function to an apostrophe, as in the following:

(set 'golden (+ golden 1))

and get the same answer. **Figure 1.4** shows what happens when you use Set Quote. Any of these three forms work, but since Setq is the most concise, it is the preferred form.

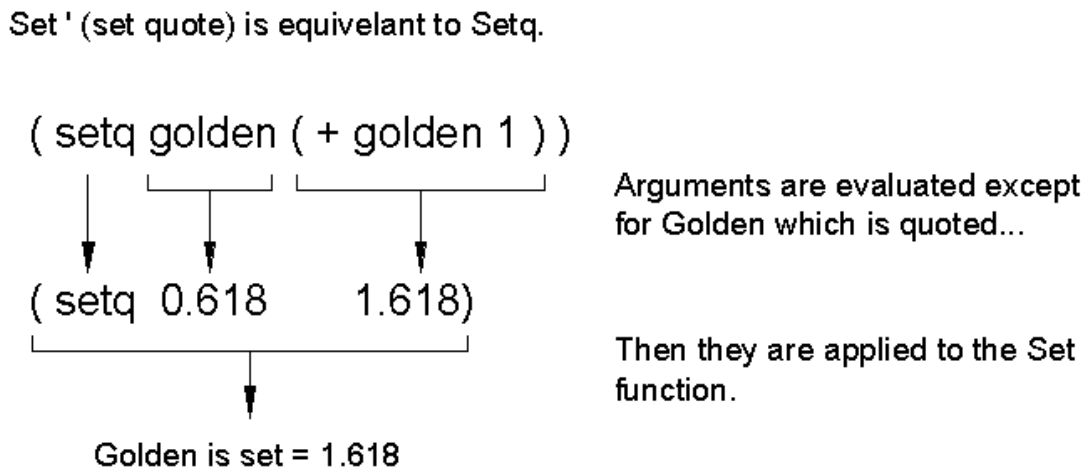


Figure 1.4: The Quote function prevents evaluation of an argument

To further illustrate the use of Quote, look at the following expression:

(setvar "snapunit" (12 12))

The function in this expression is Setvar. Setvar performs the same function as the AutoCAD setvar command-it changes the settings for system variables. Setvar accepts as its arguments a string value giving the name of the

setting to change ("snapunit") and a value representing the new settings (12 12). Here we are attempting to use Setvar to change the snap distance setting to 12 by 12.

Remember that AutoLISP evaluates each argument before it is passed to the function. As with numbers, Strings evaluate to themselves, so the string "snapunit" evaluates to "snapunit". But AutoLISP will also try to evaluate the list (12 12). AutoLISP always tries to evaluate lists as if they are expressions. As you saw earlier, the first element in an expression must be a function. Since the first element of the list (12 12) is not a function, AutoLISP will return an error message (see **figure 1.5**).

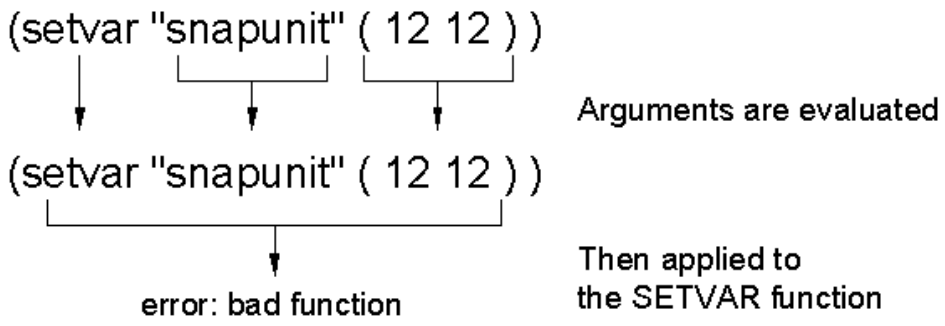


Figure 1.5: An error using Setvar

In this situation, we do not want this list (12 12) to be evaluated. We want it to be read "as is". To do this, we must add the Quote function as in the following:

(setvar "snapunit" '(12 12))

Now AutoLISP will not try to evaluate (12 12), and Setvar will apply the list to the snapunit system variable setting.

Quote provides a means to prevent evaluations when they are not desirable. Quote is most often used in situations where a list must be used as an argument to a function. Remember that there are two types of lists, those intended for evaluation and those used to store data. The list (12, 12) stores data, the width and height of the Snap distance. Because (12 12) does not have a function as its first element, it cannot be evaluated. Since AutoLISP blindly evaluates everything, Quote is needed to tell AutoLISP not to evaluate (12 12).

Applying Variables

The variable Golden can now be used within an AutoCAD command to enter a value at a prompt, or within another function to obtain other results. To see how this works, you'll assign the value 25.4 to a variable called Mill.

1. Enter

(setq mill 25.4)

at the command prompt.

Now find the result of dividing Mill by Golden.

2. Enter

(/ mill golden)

This returns the value **15.698393.**

Now assign this value to yet another variable.

3. Enter

(setq B (/ mill golden))

Now you have three variables, Golden, Mill, and B, which are all assigned values that you can later retrieve, either within an AutoCAD command by entering an exclamation point followed by the variable, or as an argument within an expression.

Our examples so far have shown numbers being manipulated, but text can also be manipulated in a similar way. Variables can be assigned text strings that can later be used to enter values in commands that require text input. Strings can also be joined together or *concatenated* to form new strings. Strings and numeric values cannot be evaluated together, however. This may seem like a simple statement but if you do not consider it carefully, it can lead to confusion. For example, it is possible to assign the number 1 to a variable as a text string by entering

(setq text1 "1")

Later, if you try to add this string variable to an integer or real number, AutoCAD will return an error message.

The examples used Setq and the addition and division functions. These are three functions out of many available to you. All the usual math functions are available, plus many other functions used to test and manipulate variables. Table 1.1 shows some of the math functions available.

Table 1.1: A partial list of AutoLISP functions

MATH FUNCTIONS THAT ACCEPT MULTIPLE ARGUMENTS

(+ *number number* ...)	add
(- *number number* ...)	subtract
(* *number number* ...)	multiply
(/ *number number* ...)	divide
(max *number number* ...)	find largest of numbers given
(min *number number* ...)	find smallest of numbers given
(rem *number number* ...)	find the remainder of numbers

MATH FUNCTIONS THAT ACCEPT SINGLE ARGUMENTS

(1+ *number*)	add 1
(1© *number*)	subtract 1
(abs *number*)	find the absolute value
(exp *nth*)	*e* raised to the *nth* power
(expt *number nth*)	*number* raised to the *nth* power
(fix *real*)	convert *real* to integer
(float *integer*)	convert *integer* to real
(gcd *integer integer*)	find greatest common denominator
(log *number*)	find natural log of *number*
(sqrt *number*)	find square root of *number*

FUNCTIONS FOR BINARY OPERATIONS

(~ *integer*)	find logical bitwise NOT of integer
(logand *int. int.* ...)	find logical bitwise AND of integers
(logior *int. int.* ...)	find logical bitwise OR of integers
(lsh *int. bits*)	find logical bitwise shift of int.by bits

Since AutoLISP will perform mathematical calculations, you can use it as a calculator while you are drawing. For example, if you need to convert a distance of 132 feet 6 inches to inches, you could enter

(setq inch1 (+ 6 (* 132 12)))

at the command prompt. The result of this expression is returned as 1590. The asterisk is the symbol for the multiplication function. The value 1590 is assigned to the variable Inch1, which can later be used as input to prompts that accept numeric values. This is a very simple but useful application of AutoLISP. In the next section, you will explore some of its more complex uses.

Accessing Single Elements of a List

When you draw, you are actually specifying points on the drawing area in coordinates. Because a coordinate is a group of values rather than a single value, it must be handled as a list in AutoLISP. You must use special functions to access single elements of a list. Two of these functions are **Car** and **Cadr**. The following example illustrates their use.

Suppose you want to store two point locations as variables called Pt1 and Pt2.

1. Enter the following two lines the command Prompt:

   (setq pt1 (list 5 6))
   (setq pt2 (list 10 12))

   The List function in these expressions combines the arguments to form a list. (You can see this in **Figure 1.6**).

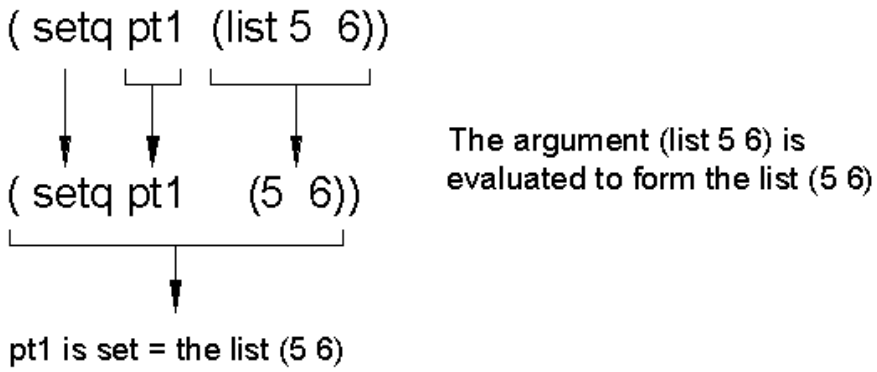


Figure 1.6: The list function.

These lists are assigned to the variable Pt1 and Pt2. As we have just seen, variables accept not only single objects as their value but also lists. In fact, variables can accept any data type as their value, even other symbols representing other variables and expressions.

2. To see the new value for pt1, enter the following:

!pt1

The list (5 6) appears.

Now suppose you want to get only the x coordinate value from this example.

3. Enter:

(car pt1)

The value 5 appears.

4. To get the y value, enter:

(cadr pt1)

which returns the value 6. These values can in turn be assigned to variables, as in the line

(setq x (car pt1))

Figure 1.7 may help you visualize what Car and Cadr are doing.

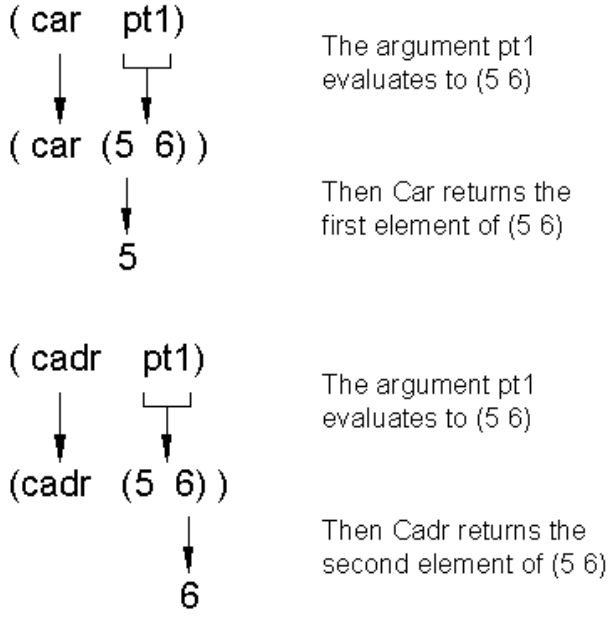


Figure 1.7: Car and Cadr of pt1

By using the List function, you can construct a point variable using x and y components of other point variables. For example, you may want to combine the y value of the variable Pt1 with the x value of a point variable Pt2.

5. Enter the following:

(list (car pt2) (cadr pt1))

You get the list (10 6) (see **figure 1.8**).

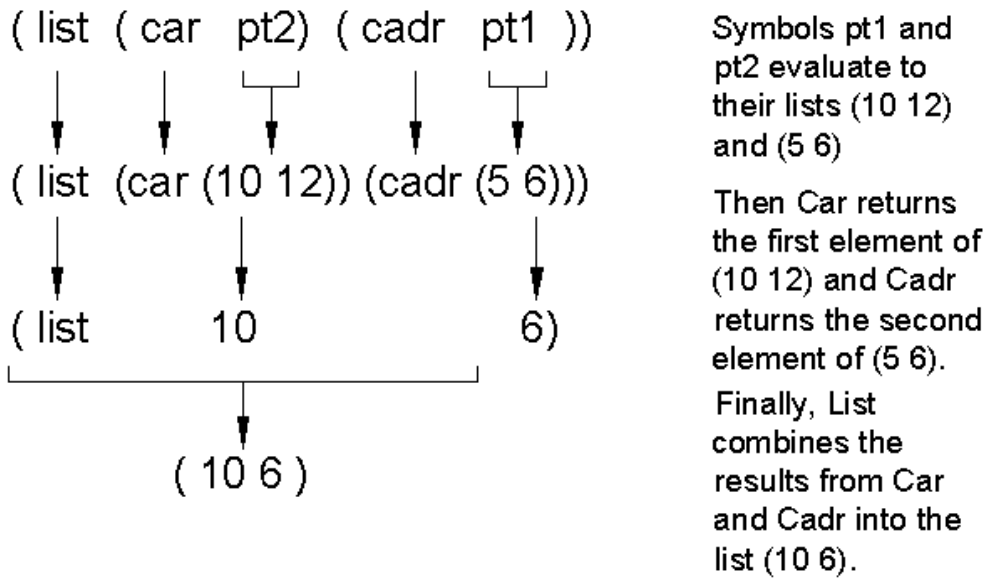


Figure 1.8: Deriving a new list from pt1 and pt2

These lists can be used to enter values during any AutoCAD command that prompts for points.

Actually, we have misled you slightly. The two primary functions for accessing elements of a list are CAR and CDR (pronounced *could*-er). You know that Car extracts the first element of a list. CDR, on the other hand, returns the value of a list with its first element removed.

6. Enter the following at the command prompt:

(cdr '(A B C))

The list (B C) is returned (see **figure 1.9**).

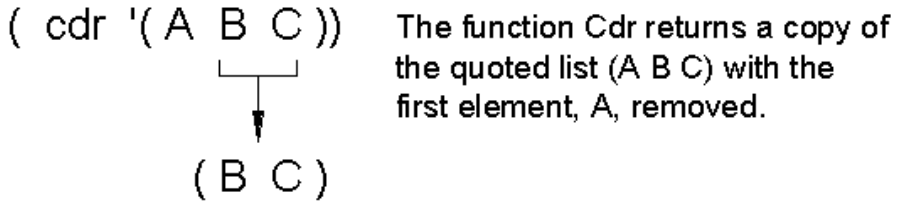


Figure 1.9: Using Cdr to remove the first element of a list.

When CDR is applied to the list (A B C) you get (B C) which is the equal to the list (A B C) with the first element, A, removed. Notice that in the above example, the list (A B C) was quoted. If the quote were left out, AutoLISP would try to evaluate (A B C). Remember that AutoLISP expect the first element of a list to be a function. Since A is variable and not a function, you would get the error message:

error: null function
(A B C)

Now try using CDR with the variable pt1.

7. Enter

(cdr pt1)

The list (6) is returned.

Remember that anything within a pair of parentheses is considered a list, even () is considered a list of zero elements. Since the value returned by CDR is a list, it cannot be used where a number is expected. Try replacing the CADR in the earlier example with a CDR:

8. Enter:

(list (car pt2) (cdr pt1))
(10 (6))

You get a list of 2 elements, 10 and (6) (see **figure 1.10**). Though this is a perfectly legal list, it cannot be used

as a coordinate list.

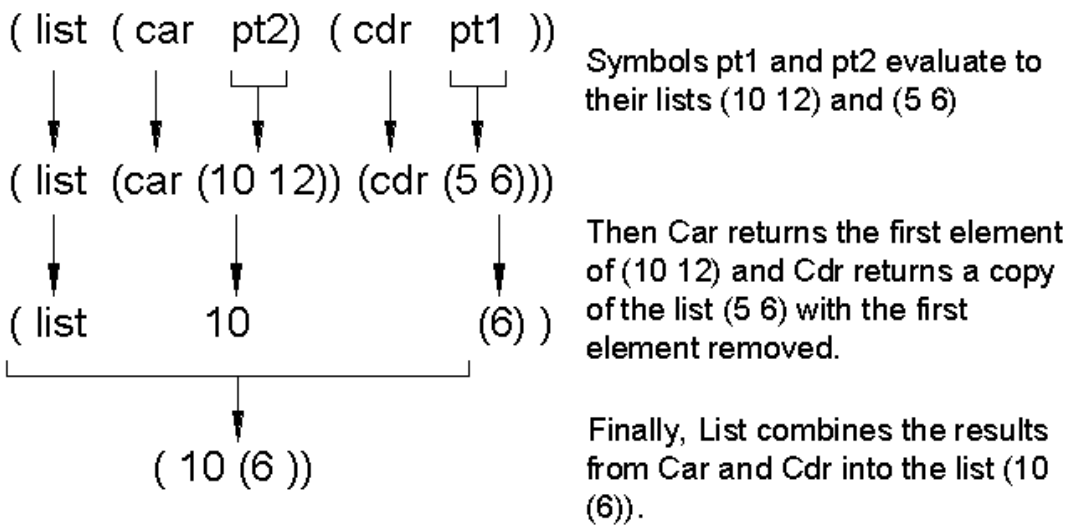


Figure 1.10: Using Car and Cdr together

So what exactly is CADR then. CADR is the contraction of CAR and CDR. You now know that CDR returns a list with its first element removed and that CAR returns the first element of a list. So to get 6 from the list held by the variable pt1, you apply CDR to pt1 to get (6) the apply car to (6) as in the following example:

```
(car (cdr pt1))
6
```

This CAR-CDR combination is abbreviated to CADR.

```
(cadr (pt1))
6
```

Figure 1.11 shows graphically how this works. You can combine CAR and CDR in a variety of ways to break down nested lists. **Figure 1.12** shows some examples of other CAR and CDR contractions.

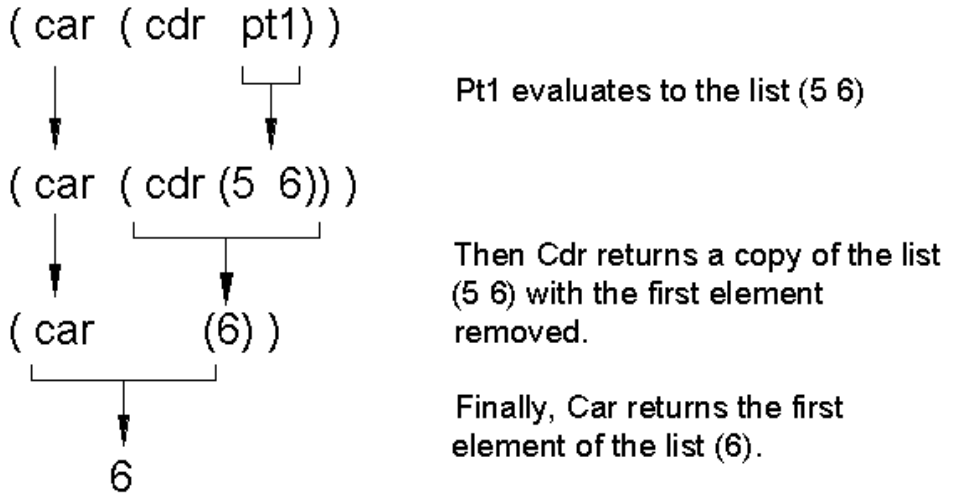
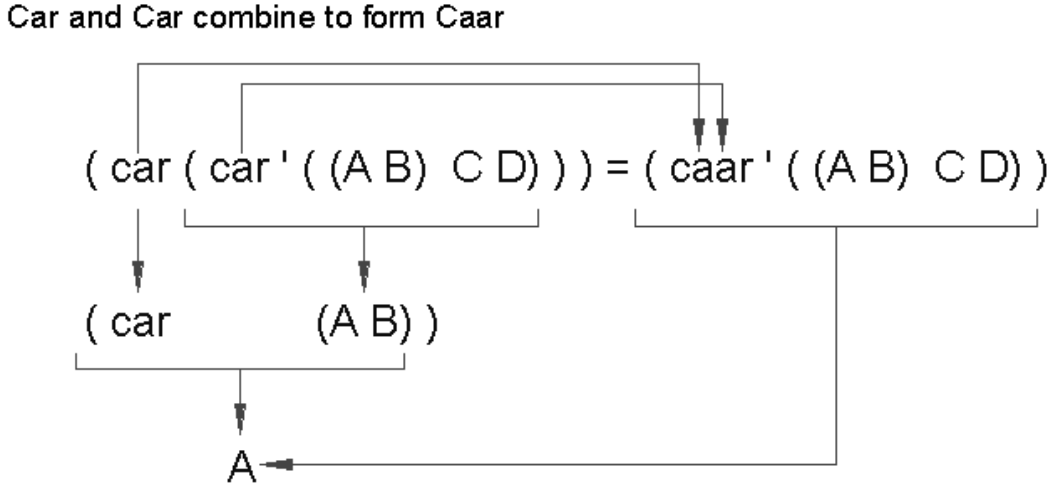


Figure 1.11: How CADR works



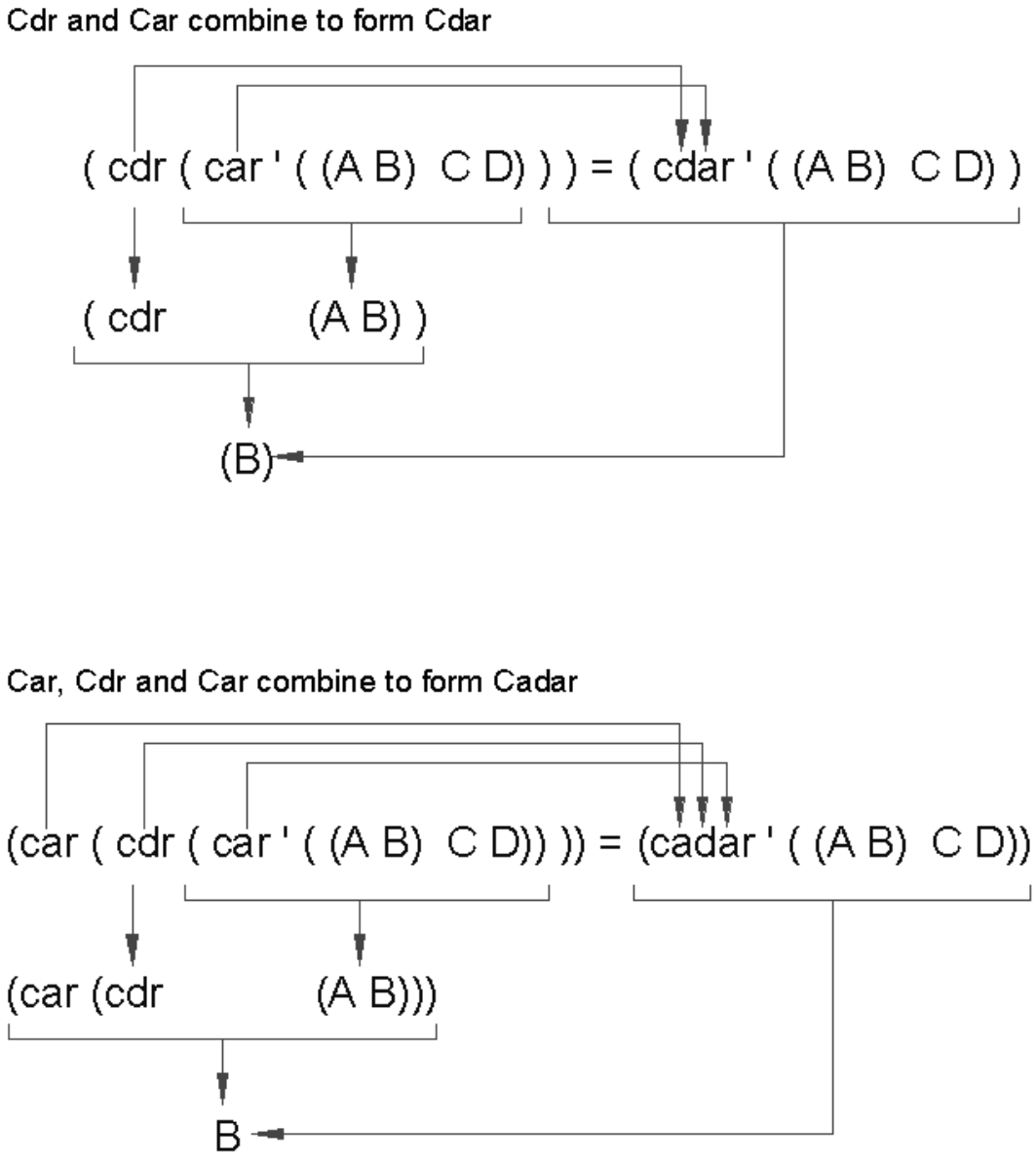


Figure 1.12: The CAR and CDR functions combined to extract elements of nested lists

Functions for Assigning Values to Variables

So far, you have been entering everything from the keyboard. However, you will most often want to bet information from the drawing area of the AutoCAD screen. AutoLISP offers a set of functions just for this purpose. These functions are characterized by their GET prefix. **Table 1.2** shows a list of these Get functions along with a brief description.

Table 1.2: Functions that pause to allow input

FUNCTION	DESCRIPTION
Getpoint	Allows key or mouse entry of point values. This always returns values as lists of coordinate values.
Getcorner	Allows selection of a point by using a window. this function requires a base point value defining the first corner of the window. The window appears, allowing you to select the opposite corner.
Getorient	Allows key or mouse entry of angles based on Units command setting for angles. Returns values in radians.
Getangle	Allows key or mouse entry of angles based on the standard AutoCAD compass orientation of angles. Returns values in radians.
Getdist	Allows key or mouse entry of distances. This always returns values as real numbers regardless of the unit format in use.

To see how one of these functions works, try the following exercise.

1. Turn your snap mode on by pressing the **F9** function key.

2. Turn on the dynamic coordinate readout by pressing the **F6** function key.

3. Enter the following at the command prompt:

(setq pt1 (getpoint))

This expression blanks the command line and waits until you enter a point. Just as with any standard AutoCAD command that expects point input, you can enter a relative or absolute point value through the keyboard or pick a point on the drawing area using the cursor. The coordinate of the point you pick will become the value assigned to the variable Pt1 in the form of a list.

4. Move your cursor until the coordinate readout lists the coordinate 4,5 then pick that point.

5. Check the value of pt1 by entering the following:

!pt1

The value (4.0 5.0 0.0) is returned and the command prompt appears once again

Note that a Z coordinate value of 0.0 was added and that all the elements of the coordinate list are reals.

Adding Prompts

All these Get functions allow you to create a prompt by following the function with the prompt enclosed by quotation marks. The following demonstrates the use of prompts in conjunction with these functions.

1. Enter the following expression:

(setq pt1 (getpoint ``Pick the first point:''))

The following prompt appears:

Pick the first point:

2. Move your cursor until the coordinate readout reads 3,4 then pick that point.

The Get functions allow you to specify a point from which the angle, distance, or point is to be measured.

3. Enter the following:

(setq pt2 (getcorner pt1 ``Pick the opposite corner:''))

the following prompt appears:

Pick the opposite corner:

Pt1 is the point variable that holds the coordinate for the last point you picked. A window appears from the coordinate defined by Pt1.

4. Move the cursor until the coordinate readout reads 6,7 then pick that point.

You can also enter a relative coordinate through the keyboard in the unit system currently used in your drawing.

Getangle and Getdist prompt you for two points if a point variable is not provided. Getcorner always requires a point variable (see **Figure 1.13**).

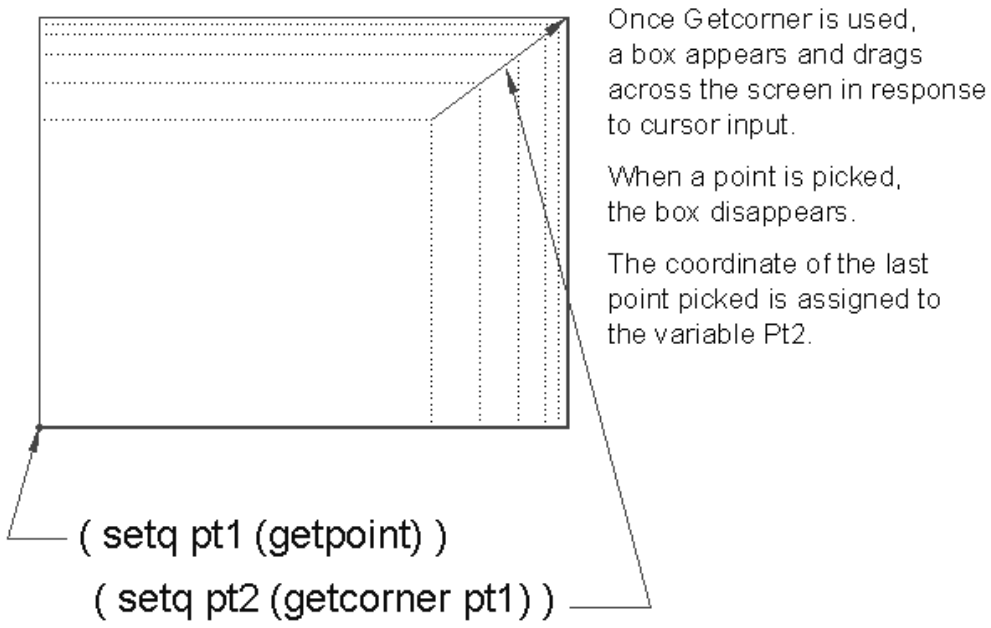


Figure 1.13: The Getcorner function as it appears on the drawing area

By using the Getpoint and getcorner functions, you can easily store point and angle values as variables. You can then refer to a stored point by entering its variable in response to a command prompt that accepts point input.

5. Issue the line command and at the From point prompt, enter:

!pt1

A rubber-banding line appears from the point previously defined as pt1 just as if you had selected that point manually.

6. Now enter the following:

!pt2

A line is drawn from the point stored by pt1 to the point stored by pt2.

Conclusion

So far, you have been introduced to the AutoLISP interpreter and to some of the terms used and a few of the function available in AutoLISP. You have also been introduced to six basic rules key to the use of AutoLISP. In summary, these rules are:

- The AutoLISP Expression is the fundamental structure of all AutoLISP programs.
- All AutoLISP expressions begin and end with parentheses with the first element of the expression being an operator or function followed by the arguments to the operator.
- All parentheses and double quotation marks enclosing strings must be balanced within an expression.
- AutoLISP evaluates everything. When it evaluates expressions, it does so by evaluating the arguments before applying the arguments to the function.
- Numbers and strings evaluate to themselves.
- Variables evaluate to the last value assigned to them.

You have seen how you can store values as variables and how you can use AutoLISP to perform math calculations. You may want to apply this knowledge to your everyday use of AutoCAD. Doing so will help you become more comfortable with AutoLISP and will give you further confidence to proceed with more complex programs.

You have also looked at how lists can be broken down and put together through the CAR, CDR, Quote and List functions. List manipulation can be a bit harry so take some time to thoroughly understand these functions. You may want to practice building lists just to get a better feel for this unusual data type.

In Chapter 2, you will learn how to create an AutoLISP program. You will also learn how to permanently store AutoLISP functions so that they can be later retrieved during subsequent editing sessions.

Chapter 2: Storing and Running Programs

Introduction

In the last chapter, you learned how to use the interpreter and in the process, you were introduced to some of the basic concepts of AutoLISP. You can now enter simple expressions into the interpreter to perform specific tasks. But once you exit the drawing editor, all of your work in AutoLISP is lost. AutoLISP would be difficult to use if there weren't some way of storing your functions for later retrieval. It would be especially difficult to create complex programs if you could only load them into the interpreter from the keyboard one line at a time. In this chapter, you will explore the development of programs through the use of AutoLISP files and in the process, review the AutoLISP concepts you learned in chapter 1.

Creating an AutoLISP Program

Instead of entering all of your functions directly into the interpreter, you have the option of writing them in a text file outside of AutoCAD. Later, when you want to use your function, you can quickly load and run them using the AutoLISP Load function. Functions you store and load in this way will act just as if you entered them into the interpreter manually. Since you can easily edit and review your functions in a word processor, you can begin to develop larger, more complex functions and programs.

What you Need

Before you can create an AutoLISP file, you need a word processor that will read and write ASCII files. ASCII stands for American Standard Code for Information Interchange. As the name implies, ASCII format was created to allow different computing systems to exchange data with each other. Most word processors allow you to generate files in this format. In this and the preceding chapters, whenever we say to open or create an AutoLISP file, we are asking you to open an ASCII file using your word processor. You can use the Windows Notepad to do most of your

AutoLISP work. Most other word processors will also save and read ASCII files, usually called TXT files in Windows.

Creating an AutoLISP File

The most common way to store AutoLISP programs is to save it as a text file with the extension .lsp. This file should contain the same information you would enter through the keyboard while using the interpreter interactively. We suggest that you create a directory called /LSP in which you can store all of your AutoLISP programs. By keeping your programs together in one directory, you are able to manage them as their numbers grow.

The program listed in figure 2.1 combines many of the concepts you have learned in chapter one into a single AutoLISP program. You will create a file containing this program then load and run the program into AutoCAD to see how it works. We use the term program to describe a function that performs a task when entered at the AutoCAD command prompt, even though programs such as the one in Figure 2.1 can still be considered functions

```
(defun c:BOX ( / pt1 pt2 pt3 pt4 )
(setq pt1 (getpoint "Pick first corner: "))
(setq pt3 (getcorner pt1 "Pick opposite corner: "))
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
(command "line" pt1 pt2 pt3 pt4 "c" )
)
```

Figure 2.1: A program to draw boxes

1. Use the Windows notepad to create a new text file called Box1.lsp

2. Carefully enter the first line from figure 2.1. Or you can simply cut and paste the data from this document into the Box1.lsp file.

(defun C:BOX (/ pt1 pt2 pt3 pt4)

Be sure you have entered everything exactly as shown before you go to the next line. Pay special attention to the spaces between elements of the line. Also note the capitalization of letters. Capitalization is not particularly important at this point however.

3. Press return to move to the next line.

4. Carefully, enter the second line again checking your typing and the spacing between elements before go to the next line. Also be sure to use the Shift-Apostrophe key for the double quotes, do not use two

apostrophes.

5. Continue entering each line as described in the previous steps. When you are done, double check your file for spelling errors and make sure the parentheses are balanced. Save and exit the **Box1.lsp** file. You now have a program that you can load and run during any AutoCAD editing session.

Loading an AutoLISP file

To load an AutoLISP file you use an AutoLISP

1.Start AutoCAD and create a new file called Box1.

2. When you are in the drawing editor, enter the following at the command prompt:

(load "box")

If the Box.lsp file is in a directory other than the current directory, the \lsp directory for example, enter

(load "/lsp/box")

The box program is now available for you to run.

As you can see, the Load function is used like any other AutoLISP function. It is the first element of an expression followed by an argument. The single argument to the load function is always a string value. Notice that within the string in the above example, the forward slash sign is used to designate a directory instead of the usual backslash. This is important to keep in mind as it is a source of confusion to both novice and experienced AutoLISP users. AutoLISP uses the backslash to denote special codes within strings. Whenever AutoLISP encounters a backslash within a string, it expects a code value to follow. These codes allow you to control the display of strings in different ways such as adding a carriage return or tab. If you use the backslash to designate directories, you must enter it twice as in the following example:

(load "\\lsp\\box")

Once the file is loaded, you will get the message:

C:BOX

You may have noticed that Load uses a string data type for its argument. Just as with numbers, strings evaluate to themselves, so when AutoLISP tries to evaluate "/lsp/box" the result is "/lsp/box".

Running a Loaded Program

Once you have loaded the box program, you can run it at any time during the current editing session. However, once you exit AutoCAD the program is not saved with the file. You must re-load the program file in subsequent editing sessions before it can be used again. Now try running the program.

1. First, set the snap mode and the dynamic coordinate readout on.

2. Enter the word Box at the command prompt. You should get the following prompt:

Pick first corner:

3. If your screen is in text mode, use the F2 key to shift to the graphic screen. Move the cursor so that the coordinate readout reads 2.0000,3.0000 and pick that point. The next prompt appears:

Pick opposite corner:

4. Now move your cursor. A window follows the motion of your cursor (see figure 2.2). Move the corner of the window to the so that the coordinate 8.0000,6.000 is displayed on the coordinate readout then pick that point. The box is drawn and the Command prompt returns. Figure 2.3 gives you a general description of how this box program works.

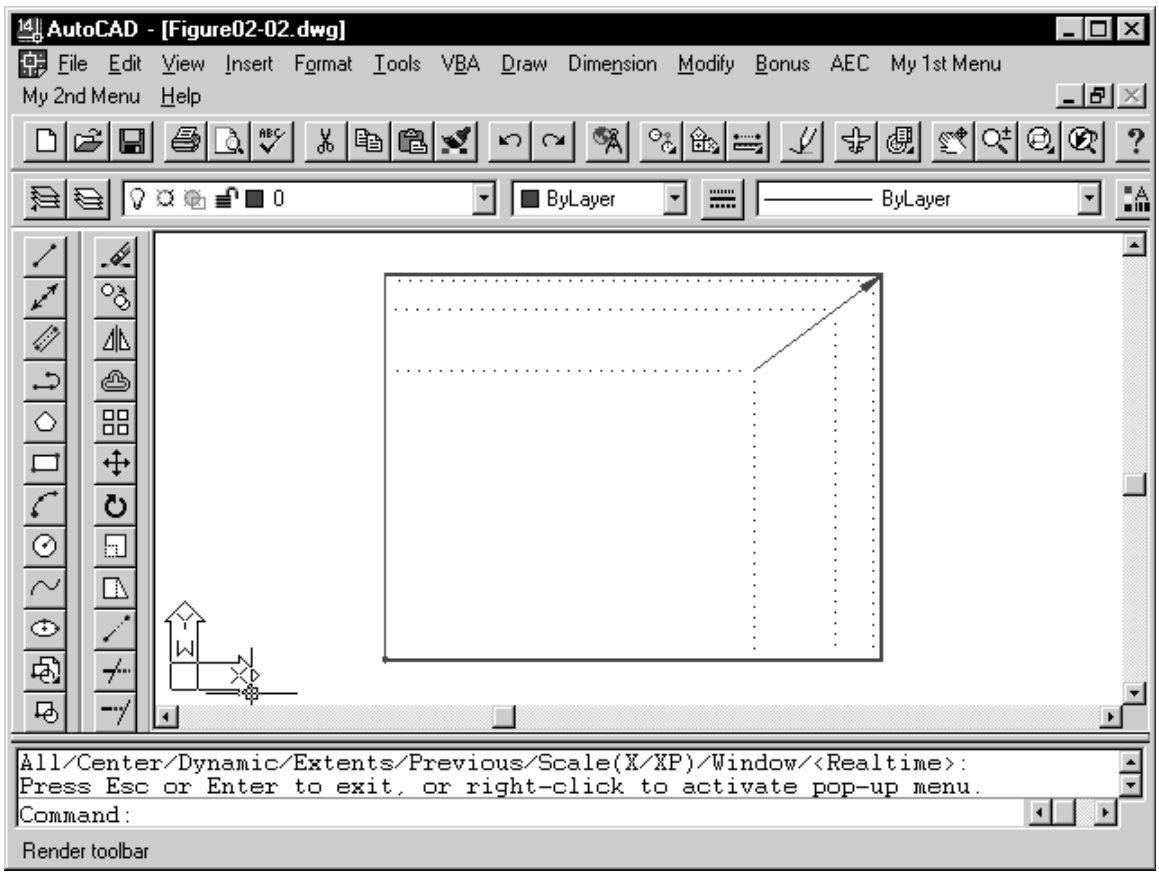


Figure 2.2: The Getcorner window

Understanding How a Program Works

Up until now, you have been dealing with very simple AutoLISP expressions that perform simple tasks such as adding or multiplying numbers or setting system variables. Now that you know how to save AutoLISP code in a file, you can begin to create larger programs. The box program is really nothing more than a collection of expressions that are designed to work together to obtain specific results. In this section, we will examine the Box program to see how it works.

The Box program draws the box by first obtaining a corner point using Getpoint:

(setq pt1 (getpoint pt1 "Pick first corner: "))

The user will see only the prompt portion of this expression:

Pick first corner:

Next, the opposite corner point is obtained using Getcorner (see Figure 2.3).

(setq pt3 (getcorner pt1 "Pick opposite corner: "))

Again, the user only sees the prompt string:

Pick opposite corner:

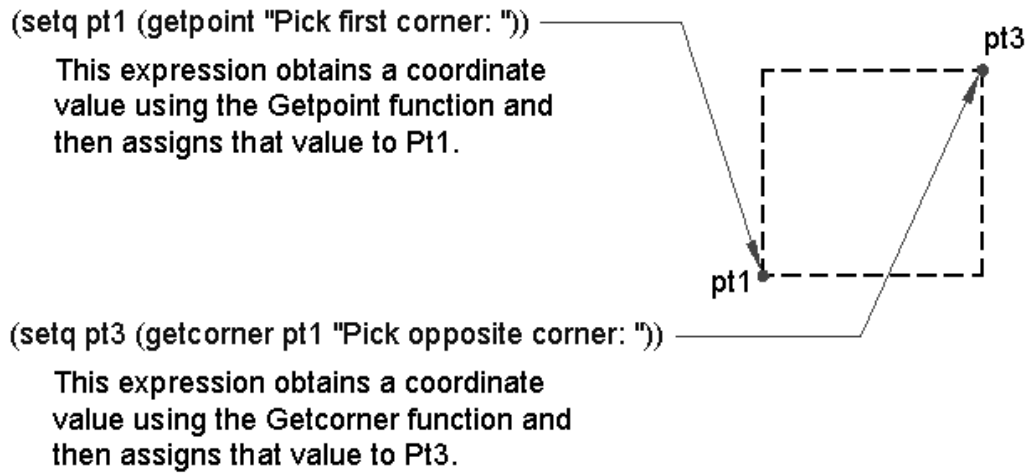


Figure 2.3: The workings of the box program

You may recall that Getcorner will display a window as the user move the cursor. In this box program, this window allows the user to visually see the shape of the box before the opposite corner is selected (see figure 2.2). Once the

second point is selected, the Box program uses the point coordinates of the first and opposite corners to derive the other two corners of the box. This is done by manipulating the known coordinates using Car, Cadr, and List (see Figure 2.4).

pt2 (list (car pt3) (cadr pt1)))

pt4 (list (car pt1) (cadr pt3)))

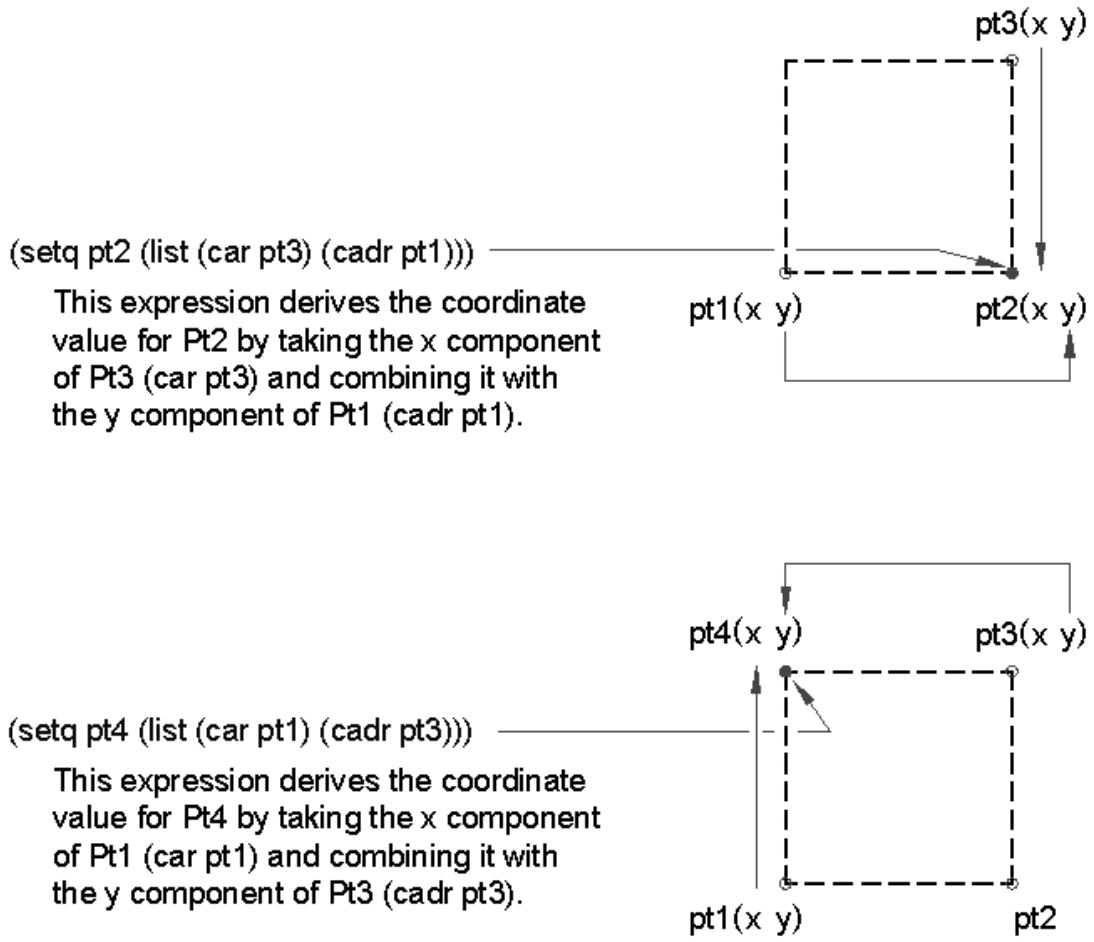


Figure 2.4: Using Car, Cadr, and List to derive the remaining box corners

Pt2 is derived by combining the X component of Pt3 with the Y component of Pt1. Pt 4 is derived from combining the X component of Pt1 with the Y component of Pt3 (see figure 2.5).

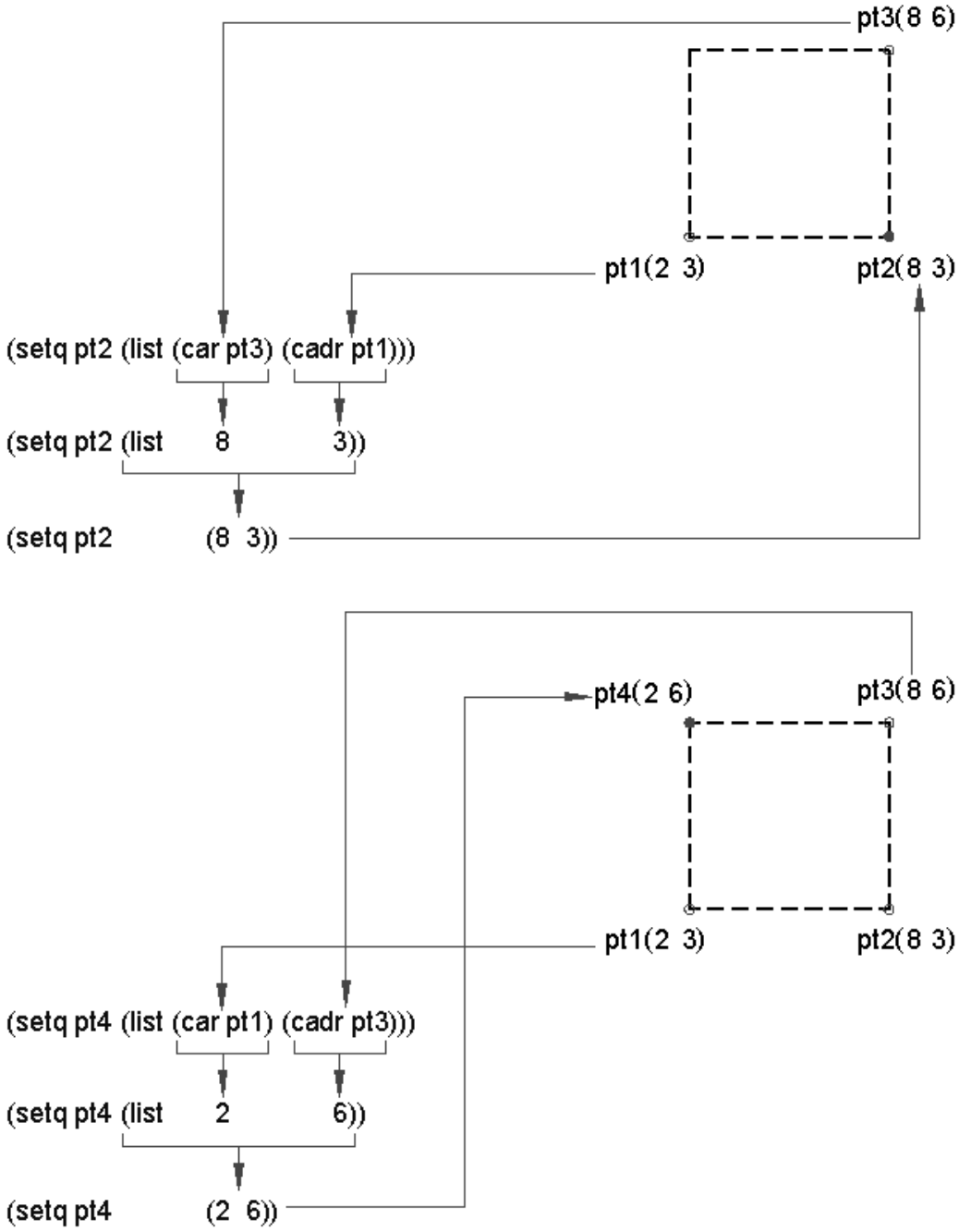


Figure 2.5: Using Car and Cadr to derive Pt2 and Pt4

Using AutoCAD Commands in AutoLISP

The last line in the box program:

(command "line" pt1 pt2 pt3 pt4 "c")

shows you how AutoCAD commands are used in an AutoLISP expression (see figure 2.6). Command is an AutoLISP function that calls standard AutoCAD commands. The command to be called following the Command function is enclosed in quotation marks. Anything in quotation marks after the Command function is treated as keyboard input. Variables follow, but unlike accessing variables from the command prompt, they do not have to be preceded by an exclamation point. The C enclosed in quotation marks at the end of the expression indicates a Close option for the Line command (see Figure 2.7.

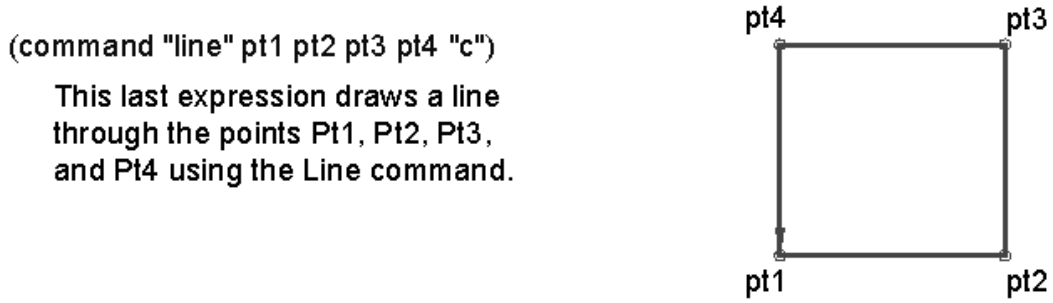


Figure 2.6: Using AutoCAD commands in a function.

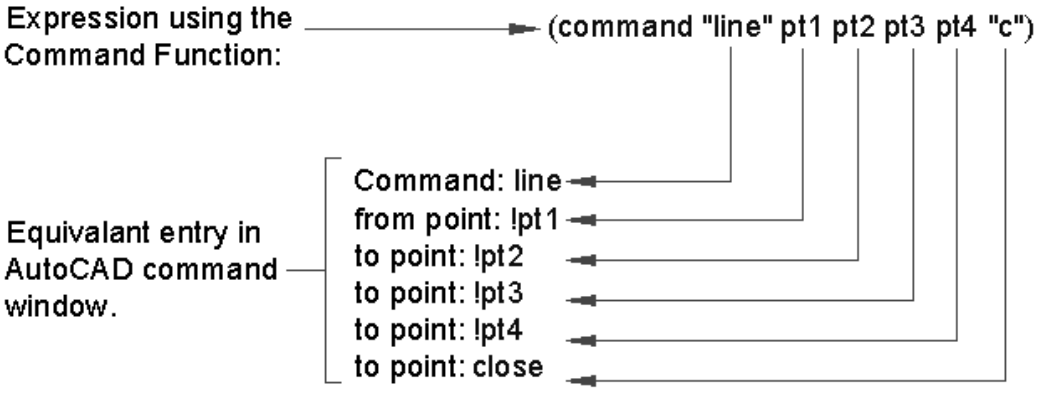


Figure 2.7: Variables help move information from one expression to another.

How to Create a Program

The box program is like a collection of expressions working together to perform a single task. Each individual expression performs some operation who's resulting value is passed to the next expression through the use of variables (see figure 2.8).

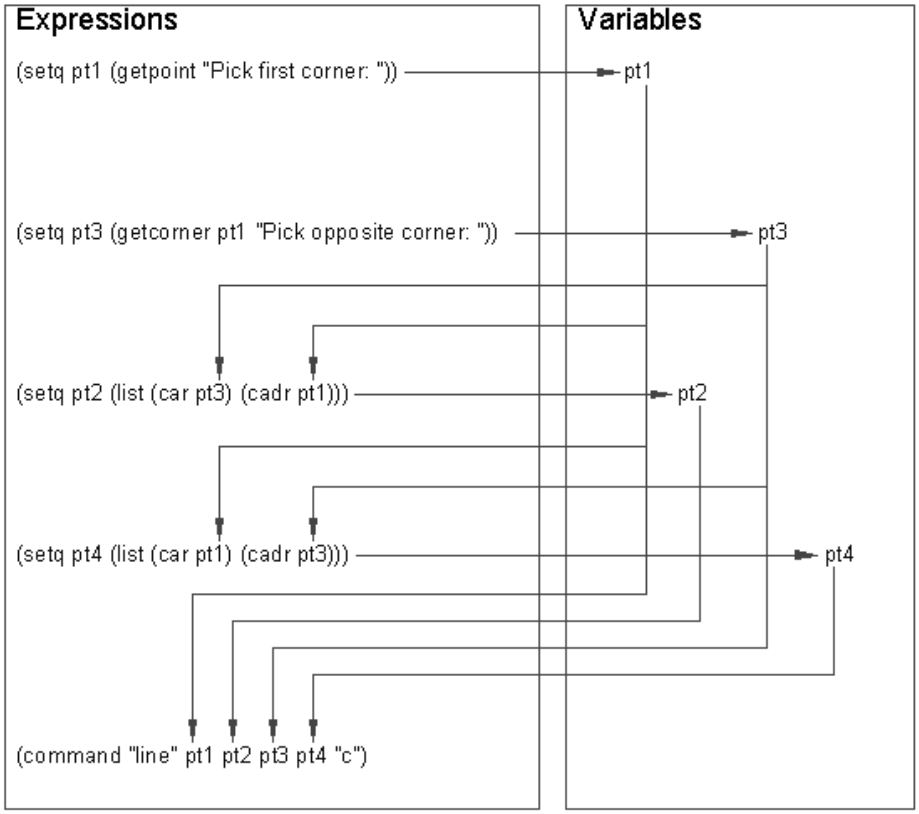


Figure 2.8: Arguments are assigned to variables in the function

The first line in the box program:

(defun c:BOX (/ pt1 pt2 pt3 pt4)

ties the collection of expressions that follow into a command called Box. Defun is a special function that allows you to define other functions. The arguments to defun are first the function name, in this case, C:BOX, followed by the argument list. The Quote function is automatically applied to the first argument. The c: in the name tells AutoLISP that this function is to act like an AutoCAD command. This means that if the function name is entered at the

AutoCAD command prompt, the function will be executed as an AutoCAD command. The name following the C: should entered in upper case letters. Care should be take not to give your functions names reserved for AutoLISP's built in functions and atoms. If, for example, you were to give the box function the name setq, then the setq function would be replaced by the box function and would not work properly.

Table 1.1 shows a list of AutoLISP function names that can easily be mistaken for user defined variable names.

abs	if	or
and	length	pi
angle	list	read
apply	load	repeat
atom	member	reverse
distance	nil	set
eq	not	t
equal	nth	type
fix	null	while
float	open	

Table 1.1 AutoLISP function names

The list that follows the name Box is an argument list. An argument list is used for two purposes. First, it is used where the function is called from another function to evaluate a set of values. For example, you could define a function that adds the square of two variables. Try entering the following function directly into the AutoLISP interpreter.

```
(defun ADSQUARE (x y)
(+ (* x x) (* y y))
)
```

1. Carefully enter the first line. Pay special attention to the parentheses and spaces.

   (defun ADSQUARE (x y)

2. Once you are sure everything is correct, press Return. You will see the following prompt:

(>

This tells you that your expressing is missing one closing parenthesis.

3. Enter the rest of the function at the 1> prompt again checking your typing before pressing Return.

In this example, the c: is left out of the Defun line. By doing this, you create a function that can be used in other functions like a subprogram or during an AutoCAD command. We'll discuss this item in more detail later. Note that the variables X and Y are included in the parentheses after the name of the function, ADSQUARE. This is the argument list. To use this function, enter the following at the command prompt:

(adsquare 2 4)

AutoLISP returns 20. The variables X and Y in the ADDSQUARE function take on the arguments 2 and 4 in the order they are listed. X takes on the value of 2, and Y takes on the value of 4 (see figure 2.9).

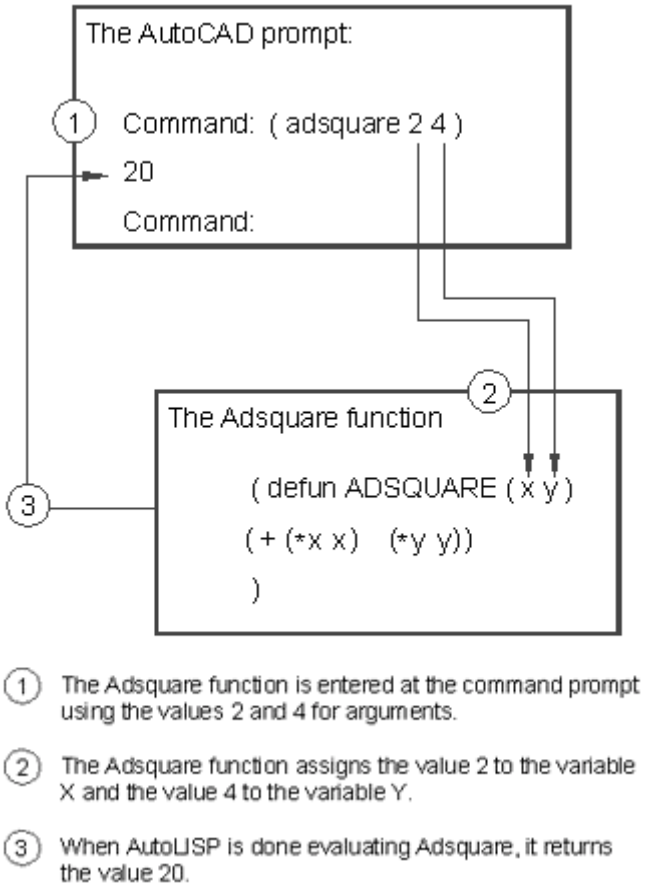


Figure 2.9: Arguments are evaluated before they are assigned to variables in the function

Variables can also be used to pass values to the function. For example, if you have a variable called A whose value is 2 and a variable B whose value is 4, you could use A and B in place of the 2 and 4. Enter the following:

(setq a 2)

(setq b 4)

(adsquare a b)

AutoLISP returns 20. Remember that AutoLISP evaluates the arguments before applying them to the function. This rule applies even to functions that you create yourself. In this case, the A is evaluated to 2 before it is passed to the X variable B is evaluated to 4 before it is passed to the Y variable.

Local and Global Variables

The second use for the argument list is to determine global and local variables. Global variables maintain their value even after a function has finished executing. In chapter 1, when you assign the value 1.618 to the variable Golden, Golden holds that value no matter where it is used. Any function can evaluate Golden to get its value, 1.618. Enter the following:

(setq golden 1.618)

(adsquare 2 golden)

The value of 6.61792 is returned. A local variable, on the other hand, holds its value only within the function it is found in. For example, the variable X in the Adsquare function above holds the value 2 only while the Square function is evaluated. Once the function is finished running, X's value of 2 is discarded. Enter the following:

!x

Nil is returned. The variables A, B, and Golden, however, are global and will return a value. Enter the following:

!golden

The value 1.618 is returned. This temporary assigning of a value to a variable with a function is called binding. This term should not be confused with the term bound which often refers to the general assignment of a value to a variable. In the example of X above, we say that a binding is created for the value X within the function Adsquare. In order for binding to take place, a variable must be used within a function that includes that variable in its argument list. Global variables cannot have bindings since they are not by their very definition confined to individual functions (see figure 2.10).

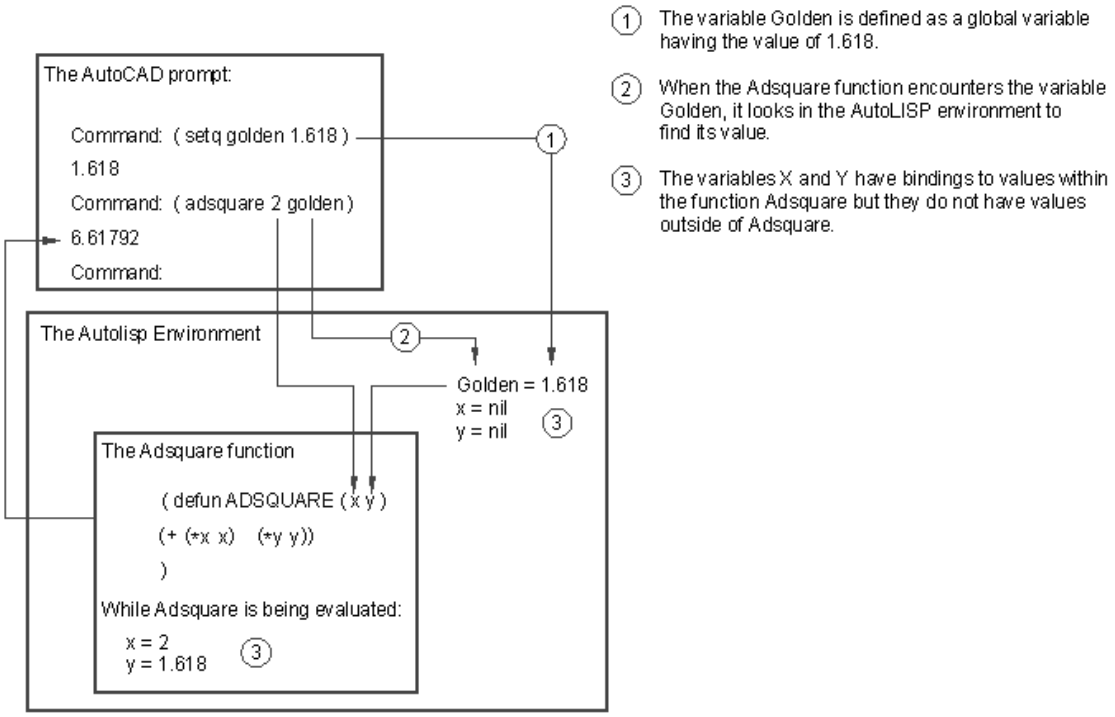


Figure 2.10: Local and Global variables

Since an argument list is used for two purposes, the forward slash symbol is used to separate variables used locally from function arguments to which values are passed when the function is called. The variables used to hold the value of arguments are listed first. Then a slash is entered, then the list of local variables as in the following example:

```
(defun square2 (x y / dx dy)
(setq dx (* x x))
(setq dy (* y y))
(+ dx dy)
)
```

X and Y are variables that will be assigned values when the function is called initially, as in the Square function given earlier. Dx and Dy, on the other hand, are variables assigned values within the function, so they follow the slash sign. In either case, the variables are local. Arguments left out of the list become global. If, for example, Dy is left out of the argument list, it's value will remain in the AutoLISP system memory for as long as the current editing

session lasts, and can be retrieved through evaluation by any other function. However, all of the functions and variables you have created in this session will be lost as soon as you exit the drawing editor.

We should mention that you must include a space before and after the slash sign in the argument list. If these spaces are not present, you will get an error message.

In both the adsquare and Square2 functions above we left out the c:. As we mentioned, this allows the function to be used by other functions like a subprogram. You can also use functions defined in this way in the middle of other commands by entering the name of the function enclosed by parentheses at the command prompt. For example, you could define a function that converts centimeters to inches. Carefully enter the following function:

```
(defun CMTOI (cm)

(* cm 0.3937)

)
```

Now suppose you started the Insert command to insert a symbol file into a drawing. When the prompt

X scale factor (1) / Corner / XYZ:

appears, you could enter

(cmtoi 90)

to signify that your symbol is to be given the scale of 90 centimeters.

The AutoLISP function Cmtoi will convert the 90 to 35.433 and enter this value for the X scale factor prompt. This can be quite useful where a value conversion or any other type of data conversion is wanted.

Automatic Loading of Programs

Eventually, you will find that some of your AutoLISP programs are indispensable to your daily work. You can have your favorite set of AutoLISP programs automatically load at the beginning of every editing session by collecting all of your programs into a single file called Acad.lsp. Be sure that Acad.lsp is in your AutoCAD directory. By doing this, you don't have to load your programs every time you open a new file. AutoCAD will look for Acad.LSP when it enters the drawing editor, and if it exists, AutoCAD will load it automatically.

1. Exit AutoCAD and check to see if you already have a file called Acad.lsp in your Acad directory. If so, rename it to Acadtemp.lsp.

2. Next, rename the Box.lsp file to Acad.lsp. Place Acad.lsp in your AutoCAD directory if it isn't there already.

3. Start AutoCAD and open any file. When the drawing editor loads, notice the following message in the prompt area:

Loading acad.lsp...

Now, the box program is available to you without having to manually load it.

Though the Acad.lsp file only contained the box program, you could have included several programs and functions in that single file. Then, you would have access to several AutoLISP programs by loading just one file.

Managing Large Acad.lsp files

As you begin to accumulate more AutoLISP functions in your ACAD.lsp file, you will notice that AutoCAD takes longer to load them. This delay in loading time can become annoying especially when you just want to quickly open a small file to make a few simple revisions. Fortunately, there is an alternative method for automatically loading programs that can reduce AutoCAD's start-up time.

Instead of placing the programs code in Acad.lsp, you can use a program that loads and runs the program in question. For example, you could have the following line in place of the box program in the Acad.lsp file:

(defun c:BOX () (load "/lsp/box") (c:box))

We will call this a box loader function. Once the above function is loaded, entering Box at the command prompt will start it. This box loader function then loads the real Box program which in turn replaces this box loader function. The (c:box) in the box loader function is evaluated once the actual box program has been loaded thus causing the box program to run. C:BOX is the symbol representing the program BOX so when it evaluated, like any function, it will run.

As you can see, this program takes up considerably less space that the actual box program and will therefore load faster at start-up time. You can have several of these loading programs, one for each AutoLISP function or program you wish to use on a regular basis. Imagine that you have several programs equivalent in size to the box program. the Acad.lsp file might be several pages long. A file this size can take 30 seconds to load. If you reduce each of those programs to one similar to the box loader function above, you substantially reduce loading time. Several pages of programs could be reduced to the following:

(defun C:PROGM1 () (load "/lsp/progm1") (C:PROGM1))

(defun C:PROGM2 () (load "/lsp/progm2") (C:PROGM2))

(defun C:PROGM3 () (load "/lsp/progm3") (C:PROGM3))

(defun C:PROGM4 () (load "/lsp/progm4") (C:PROGM4))

(defun C:PROGM5 () (load "/lsp/progm5") (C:PROGM5))

(defun C:PROGM6 () (load "/lsp/progm6") (C:PROGM6))

(defun C:PROGM7 () (load "/lsp/progm7") (C:PROGM7))

If you imagine that each of the functions being called from the above example is several lines long then you can see

that as the list of programs in Acad.lsp grows, the more space you will save. By setting up your Acad.lsp file in this way, you also save memory since functions are loaded only as they are called.

Using AutoLISP in a Menu

There are two reasons why you might write AutoLISP code directly into the menu file. The first is to selectively load external AutoLISP programs as they are needed. You may have several useful but infrequently used programs that take up valuable memory. You might prefer not load these programs at startup time. By placing them in the menu file, they will only load when they are selected from the menu. In fact, this is what the AutoShade and 3dobjects menu options do. When you pick Ashade from either the screen or pull down menu, and AutoShade is present on your computer, an AutoLISP program called Ashade.lsp is loaded.

The code of the program can be present in the menu file or you can use a method similar to the one described earlier to load external AutoLISP files. However, if you use the menu system to load external AutoLISP files, you must a slightly different method.

In the example we gave for loading programs from external AutoLISP file, the loader program is replaced by the fully operational program of the same name. But if you were to place the following expression in a menu, the program would load every time the menu option was selected.

[box]^C^C(load "box");box

There is nothing wrong with loading the program each time it is run but if the AutoLISP file is lengthy, you may get tired of waiting for the loading to complete every time you select the item from the menu. A better way to load a program from the menu is to use the If function as in the following:

[box]^C^C(if (not C:box)(load "box")(princ "Box is already loaded. ");box

In this example, we show three new functions, If, Not and Princ. The If functions checks to see if certain conditions can be met then evaluates an expression depending on the result. The If functions expects the first argument to test the condition that is to be met while the second argument is the expression to be evaluated if the condition is true. A third expression can optionally be added for cases where you want an expression to be evaluated when the test condition returns nil. The Not function returns a T for true if its argument evaluates to nil, otherwise it returns nil (see figure 2.11).

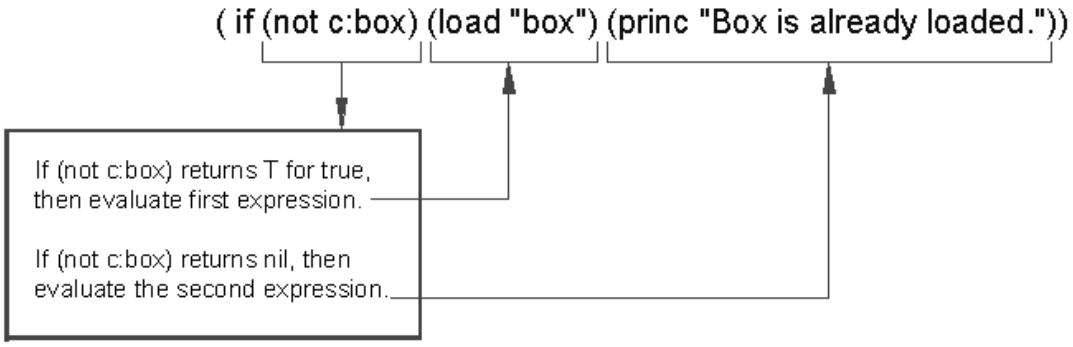


Figure 2.11: Using the If function

So, in the menu sample above, if C:BOX does not exists, Not will return T for true and the If function evaluates the (load "box") expression thereby loading the Box program. If C:BOX has already been loaded, then Not function returns nil and box.lsp will not be loaded again. Instead, the second argument will be evaluated. This expression:

(princ "Box is already loaded. ")

simply displays the string:

Box is already loaded.

on the command prompt.

You may have noticed that the If function does not conform to the standard rules of evaluation. Where If is used, the second or third argument is evaluated depending on the value of the first argument.

The second reason for placing code in the menu is speed. Instead of creating a function using Defun, you can set up a program to be read and executed line by line. This saves time since the interpreter reads and executes each expression of your program as they occur in the menu listing instead of reading the entire set of expressions then executing the program. Memory is also saved since Defun is not used to define a new function. Figure 2.11 shows a listing of how the box program from chapter 1 would look as a menu option.

```
[BOX] ^C^C(setvar "menuecho" 1);+
(setq pt1 (getpoint "Pick first corner: "));\+
(setq pt3 (getcorner pt1 "Pick opposite corner: "));\+
(setq pt2 (list (car pt3) (cadr pt1)));+
(setq pt4 (list (car pt1) (cadr pt3)));+
line; pt1; pt2; pt3; pt4;C;)
```

Figure 2.12: The box program as a menu option

NOTE that the Defun function is absent from the listing. When the box option is picked from the menu, AutoCAD reads the associated text just as it would any menu option. Since the text in this case is AutoLISP code, the AutoLISP interpreter evaluates each expression as if it were entered through the keyboard.

NOTE that the semicolon is used to indicate the enter key at the end of each expression. A backslash is used to pause for input, just as you would have a backslash in other commands that require mouse or keyboard input. Also note the use of the plus sign indicating the continuation of the menu item. Finally, note that the last line of the menu item uses the exclamation point to enter the values of the variables as responses to the Line command. The last C is the Close option of the Line command. You may have noticed a new expression:

(setvar "menuecho" 1)

The Setvar function in the above expression does the same thing as the Setvar command in AutoCAD. In this case, it sets the menuecho system variable to 1. This setting prevents the actual AutoLISP code from being displayed on the command prompt.

1. Using your word processor, copy the above listing into a file called Box.mnu, again being careful to input the listing exactly as shown above.

2. Get back into the AutoCAD drawing editor then use the Menu command to load the Box menu you just created. The AutoCAD menu will disappear and will be replaced by the single word Box.

3. Pick the Box option from the menu, and you will see the prompts you entered when you created the Box menu above. This program will work in the same way as the Box.lsp program.

4. To get the AutoCAD menu back, enter the command Menu and enter acad at the menu name prompt.

Since Defun is not used in this example, no argument list is used. Any variables used in the listing becomes global. For this reason, when using menus for AutoLISP programs, it especially important to keep track of variable names so they do not conflict with other variables from other programs.

Using Script Files

AutoCAD has a useful feature that allows you to write a set of pre-defines sequence of command and responses stored as an external text file. This scripting ability is similar to writing a macro in the menu file. The main difference between menu macros and scripts is that scripts do not allow you to pause for input. Once a script is issued, it runs until it is completed. Also, unlike menu macros, scripts can exist as independent files that are called as they are needed from the command prompt.

Although it is not commonly done, you can put your AutoLISP programs in a script file. In fact, you can directly convert your AutoLISP program files directly to a script file simply by changing the file extension from .LSP to .SCR. To load your programs, you would use the AutoCAD Script command instead of the AutoLISP load function. There may be times when you want to embed AutoLISP code into a script you have written. You can either write the code directly into the script file or use the load function to load an AutoLISP file from the script.

Conclusion

You have been introduced to some new ways of getting more usefulness from AutoLISP.

- The function Defun allows you to create programs that can be invoked from the AutoCAD command line.
- Variables can be used to pass information between expressions and functions.
- Functions can be created that can be used to respond to AutoCAD commands.
- Functions and programs can be stored as files on disk to be easily retrieved at a later date.
- Frequently used functions can be stored in a file called Acad.lsp to be loaded automatically.

We encourage you to try creating some simple functions and save them to disk. You can start out with the functions presented in this chapter or you can try your hand at writing some functions on your own. You may also want to try using the menu file to store programs or you may want to keep each of your programs as separate files and load them as they are needed.

In the next chapter, you will be introduced to some of the ways you can plan, organize and execute a programming project.

Chapter 3: Organizing a Program

Introduction

When you write a program, you are actually setting out to solve a problem. It is often helpful to develop a plan of how you want to solve your problem. The plan may need revisions along the way, but at least it gives you a framework out of which you can get both an overall and detailed understanding of the problem you are trying to solve. Often the most monumental problem can be broken into parts that can be easily dealt with.

By taking some time to analyze your goals and writing down some preliminary plans, you can shorten your program development time considerably. In this chapter we will discuss some methods for planning your programming efforts and along the way, you will get a more detailed look at how AutoLISP works.

Looking at a Programs Design

As simple as the box.lsp program is, it follows some basic steps in its design. These steps are as follows:

1. Establish the name of the program or function
2. Obtain information by prompting the user
3. process the information
4. Produce the output

Lets look at a breakdown of the program to see these steps more clearly. The first line defines the program by giving it a name and listing the variables it is to use.

(defun c:box (/dx dy pt1 pt2 pt3 pt4)

The second and third line of the program obtain the minimum information needed to define the box.

(setq pt1 (getpoint "Pick first corner: "))

(setq pt3 (getcorner "Pick opposite corner: "))

The fourth, fifth, and sixth lines process the information to find the other two points needed to draw the box.

(setq pt2 (list (car pt3) (cadr pt1)))

(setq pt4 (list (car pt1) (cadr pt3)))

The last line draws the box in the drawing editor.

(command "line" pt1 pt2 pt3 pt4 "c")

This four step process can be applied to nearly every program you produce.

You should also consider how standard AutoCAD commands work. A program that has an unusual way of prompting for information will seem jarring and unfamiliar to the user. By designing your prompts and prompt sequence to match closely those of AutoCAD's, your programs will seem more familiar and therefore easier to use. For example, when a command requires editing of objects, you are first prompted to select the objects to be edited. So when you design a program that performs some editing function, you may want to follow AutoCAD's lead and have your program select objects first (we will cover functions that allow you to select objects later in this book).

Outlining Your Programming Project

But before you get to actual writing of code, you will want to chart a path describing what your program is to do. The box program might be planned as the following sequence of statements:

1. Get the location of one corner of the box. Save that location as a variable called pt1.

2. Get the location of the other corner of the box. Save that location as a variable called pt3.

3. calculate the other two corners by using information about the known corner locations pt1 and pt3.

4. Draw the box

The above list outlines the procedures needed to accomplish your task. This type of list is often called Pseudocode. It is a plain language description of the code your program is to follow. It acts like an outline of your programs code. You could even incorporate your pseudocode as comments to the actual code of the program. Just as with an outline, you may have to go through several iterations of lists like this before actually hitting on one that seem workable.

Along with the pseudocode of a program, you may also want to sketch out what your program is supposed to do, especially if your program is performing some graphic manipulation. Figure 3.1 shows a sketch done as an aid to developing the box program.

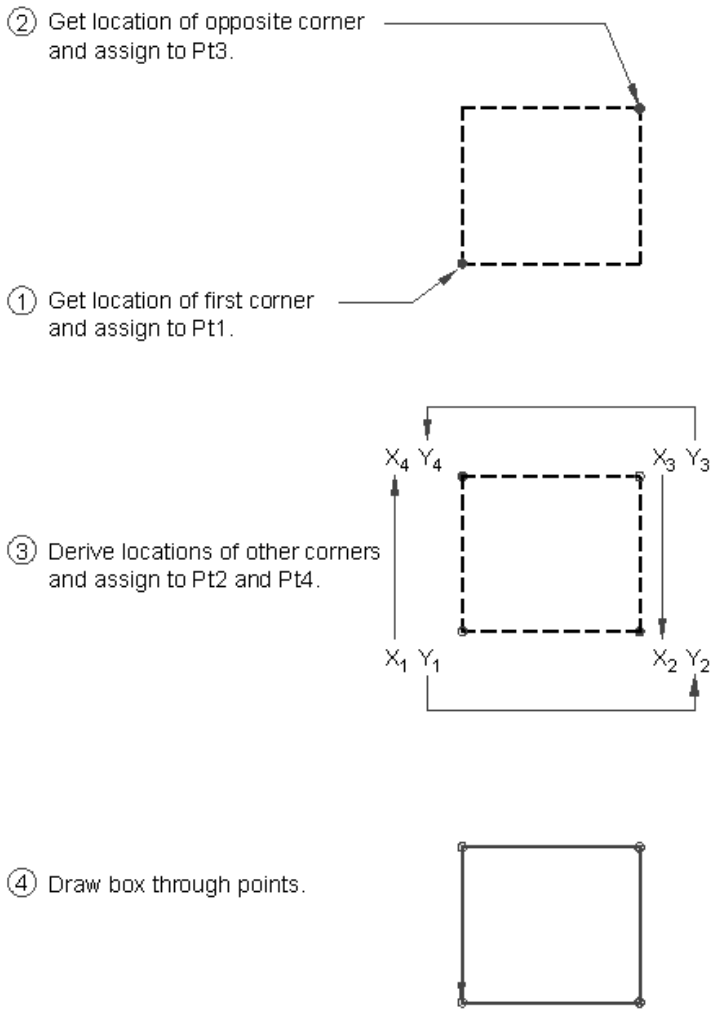


Figure 3.1: Sample sketch used to develop box.lsp

Using Functions

We have broken the box program down conceptually into three processes. We can also break it down physically by turning each process into a function as in the following:

```
 (defun getinfo ()
(setq pt1 (getpoint "Pick first corner: "))
(setq pt3 (getcorner pt1 "Pick opposite corner: "))
)
(defun procinfo ()
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
)
(defun output ()
(command "line'' pt1 pt2 pt3 pt4 "c'' )
)
```

You now have three functions that can be called independently. We now need a way to tie these functions together. We can write a main program that does just that:

```
 (defun C:BOX1 (/ pt1 pt2 pt3 pt4)
(getinfo)
(procinfo)
(output)
)
```

Let's see how this new version of the Box drawing program works.

1. Open a new AutoLISP file called box1.lsp.

2. Enter the above three functions, Getinfo, Procinfo, and output, along with the new box program. Your file should look like Figure 3.2.

3. Open a new AutoCAD file called chapt3= and load the Box1.lsp file.

4. Run the C:BOX1 program. You will see that it acts no differently from the first box program in chapter 1.

```
(defun getinfo ()
(setq pt1 (getpoint "Pick first corner: "))
(setq pt3 (getcorner pt1 "Pick opposite corner: "))
)

(defun procinfo ()
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
)

(defun output ()
(command "pline" pt1 pt2 pt3 pt4 "c" )
)

(defun C:BOX1 (/ pt1 pt2 pt3 pt4)
(getinfo)
(procinfo)
(output)
)
```

Figure 3.2: The contents of box1.lsp

The C:Box1 program listed above acts as a sort of master organizer that evaluates each of the three functions, getinfo, procinfo, and output, as they are needed. This modular approach to programming has several advantages.

First, since each function exists independently, they can be used by other programs thereby reducing the amount of overall code needed to run your system. Though in the above example, we actually increased the size of the Box program, as you increase the number of program you use, you will find that you can use functions from this program in other programs. Functions you write in this way can serve as tools in building other programs.

Second, while writing programs, you can more easily locate bugs since they can be localized within one function or another. Whenever AutoCAD encounters an error, it displays an error message along with the offending expression.

Third, smaller groups of code are more manageable making your problem solving task seem less intimidating. The actual Box program represents the problem in general terms while the functions take care of the details. You can get a clearer idea of how your programs work because clarity is built into the program by virtue of this modular structure.

Finally, features can be added and modified more easily by "plugging" other functions either into the main program or into the individual functions that make up the program.

Adding a Function

As great a program AutoCAD is, it does have some glaring shortcomings. One is the fact that it does not dynamically display relative X and Y coordinates. It does display relative polar coordinates dynamically, but often, it is helpful to see distances in relative X and Y coordinates. One example where this would be useful is in the creation of floor plans where a dynamic reading of relative X and Y coordinates could help you size rooms in a building. It would be also helpful if this dynamic readout would display the rooms area as the cursor moved. Figure 3.3 Shows a function that displays relative X and Y coordinates in the status line.

```
(defun RXY (/ pt lt x last pick lpt1)
(if (not pt1)(setq lpt1 (getvar "lastpoint"))(setq lpt1 pt1))
   (while (/= pick t)
    (setq pt (cadr (setq lt (grread t))))
     (if (= (car lt) 5)(progn
      (setq x (strcat
         (rtos (- (car pt) (car lpt1))) " x "
         (rtos (- (cadr pt) (cadr lpt1))) " SI= "
         (rtos (*(- (car pt) (car lpt1))
                 (- (cadr pt) (cadr lpt1))
                )
          2 2)
             )
      )
        (grtext -2 x)
      )
     )
     (setq pick (= 3 (car lt)))
   )
(cadr lt)
)
```

Figure 3.3: The RXY function

1. Exit AutoCAD temporarily either by using the AutoCAD Shell command or by using the End command to exit AutoCAD entirely.

2. Open an AutoLISP file called Rxy.lsp and copy the program listed in figure 3.4. Check your file carefully against the listing to be sure it is correct.

3. Return to the Chapt3 AutoCAD file.

4. Turn on the snap and dynamic coordinate readout.

5. Load Rxy.LSP then start the line command.

6. At the First point prompt, pick the coordinate 2,2 then, at the next point prompt, enter the following:

(rxy)

Notice how the coordinate readout now dynamically displays the relative XY coordinates from the first point you picked.

7. Move the cursor until the coordinate readout lists 6.0000,7.0000 then pick that point. A line is draw with a displacement from the first point of 6,7.

We won't go into a detailed explanation of this function quite yet. Imagine, however, that you have just finished writing and debugging this function independent of the box program and you want to add it permanently to the box program.

1. Exit AutoCAD temporarily either by using the AutoCAD Shell command or by using the End command to exit AutoCAD entirely.

2. Open the box1.lsp file and change the Getinfo function to the following:

(defun getinfo ()

(setq pt1 (getpoint "Pick first corner: "))

(princ "Pick opposite corner: ")

(setq pt3 (rxy))

)

Your new Box1.lsp file should look like figure 3.4. We have replaced the expression that obtains PT3 with two lines, one that displays a prompt:

(princ "Pick opposite corner: ")

and another that uses the RXY function to obtain the location of PT3:

(setq pt3 (rxy))

Now when you use the box program, the width and height of the box along with its area are displayed on the status prompt.

3. Return to the **Chapt3** file.

4. Load the Rxy.lsp and Box1.lsp files.

5. Run the C:BOX1 program by entering box1 at the command prompt.

6. Make sure the Snap mode and Dynamic coordinate readout are both on.

7. At the Pick first corner prompt, pick the coordinate 2,3.

8. At the Opposite corner prompt, move the cursor and note how the coordinate readout responds. If you move the cursor to the first point you picked, the coordinate readout will list 0,0 telling you that you are a relative distance of 0,0 from the first corner.

9. Now move the cursor until the coordinate readout lists 5.0000,5.0000 then pick that point. You have drawn a box exactly 5 units in the x axis by 5 units in the y axis.

```
(defun getinfo ()
(setq pt1 (getpoint "Pick first corner: "))
(princ "Pick opposite corner: ")
(setq pt3 (rxy))
)

(defun procinfo ()
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
)

(defun output ()
(command "pline" pt1 pt2 pt3 pt4 "c" )
)

(defun C:BOX1 (/ pt1 pt2 pt3 pt4)
(getinfo)
(procinfo)
(output)
)
```

Figure 3.4: The revised Box1.lsp file

In the above exercise, after you pick the first corner of the box, the window no longer appears. Instead, the status line changes to display the height and width of you box dynamically as you move the cursor. It also displays the square inch area of that height and width. By altering a function of the main C:BOX1 program, you have added a new features to it. Of course, you had to do the work of creating Rxy.lsp but Rxy.lsp can also be used to dynamically display relative X Y coordinates in any command that reads point values.

Reusing Functions

You have seen that by breaking the box program into independent functions, you can add other functions to your program to alter the way the program works. In the following section, you will create two more programs, one to draw a 3D rectangle, and one to draw a wedge, using those same functions from the box program.

Creating an 3D Box program

To create a 3D box from a rectangle, all you really need to do is extrude your rectangle in the Z axis then add a 3dface to the top and bottom of the box. Extruding the box can be accomplished using the Change command. Figure 3.5 shows how this would be done with standard AutoCAD commands.

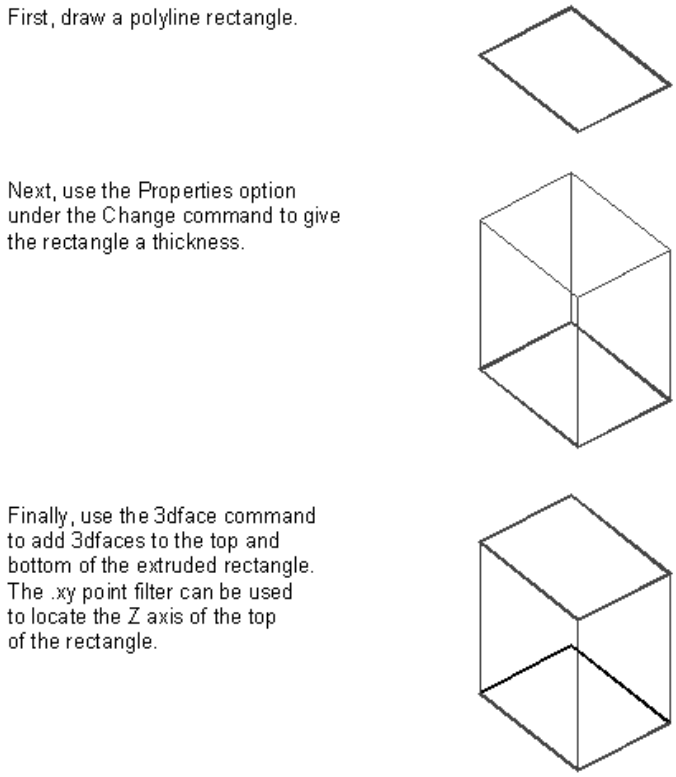


Figure 3.5: Drawing a three-dimensional box manually

Your box program also needs a way to prompt the user for a height. We can accomplish these things by creating a modified version of the C:BOX1 main program. Figure 3.6 shows such a program.

```
(defun C:3DBOX (/ pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of box: "))
(procinfo)
(output)
(command "change" "L" "" "P" "th" h ""
         "3dface" pt1 pt2 pt3 pt4 ""
         "3dface" ".xy" pt1 h ".xy" pt2 h
         ".xy" pt3 h ".xy" pt4 h ""
)
)
```

Figure 3.6: A program to draw a 3D box.

In the following exercise, you will add the 3D box program to the Box1.lsp file then run C:3DBOX.

1. Exit AutoCAD temporarily either by using the AutoCAD Shell command or by using the End command to exit AutoCAD entirely.

2. Open the box1.lsp file again and add the program listed in figure 3.6 to the file. Your Box1.lsp file should look like figure 3.7.

3. Return to the Chapt3 AutoCAD file.

4. Load Box1.lsp again. If you had to Exit AutoCAD to edit Box1.lsp, Load Rxy.lsp also.

5. Start the C:3DBOX program by entering 3dbox at the command prompt.

6. At the Pick first corner prompt, pick a point near the coordinate 2,3.

7. At the Pick opposite corner prompt, move the cursor until the coordinate readout lists 7.0000,5.0000 then pick that point.

8. At the Enter height prompt, enter 6. A box appears.

9. Issue the Vpoint command and at the prompt:

Rotate/<View point> <0.0000,0.0000,1.0000>:

10. Enter **-1,-1,1.** Now you can see that the box is actually a 3d box.

11. Issue the Hide command. The box appears as a solid object.

```
(defun getinfo ()
(setq pt1 (getpoint "Pick first corner: "))
(princ "Pick opposite corner: ")
(setq pt3 (rxy))
)

(defun procinfo ()
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
)

(defun output ()
(command "pline" pt1 pt2 pt3 pt4 "c" )
)

(defun C:BOX1 (/ pt1 pt2 pt3 pt4)
(getinfo)
(procinfo)
(output)
)

(defun C:3DBOX (/ pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of box: "))
(procinfo)
(output)
(command "change" "L" "" "P" "th" h ""
         "3dface" pt1 pt2 pt3 pt4 ""
         "3dface" ".xy" pt1 h ".xy" pt2 h
         ".xy" pt3 h ".xy" pt4 h ""
)
)
```

Figure 3.7: The Box1.lsp file with the C:3DBOX program added.

In the C:3DBOX program, a line is added to obtain height information.

(setq h (getreal "Enter height of box: "))

In most cases, it is impractical to try to enter a 3D point value using a cursor, though it can be done. We use the Getreal function here to allow the input of a real value for the height since this is the simplest route to obtain the height information.

To actually draw the box, other AutoCAD commands have been added to give the box a third dimension:

```
(command "change" "L" "" "P" "th" h ""
"3dface" pt1 pt2 pt3 pt4 ""
"3dface" ".xy" pt1 h ".xy" pt2 h
".xy" pt3 h ".xy" pt4 h ""
)
```

This expression issues several AutoCAD commands to turn the simple rectangle drawn by the Output function into a 3 dimensional box. the first line changes the thickness of the box previously drawn by the Output function. The second line draws a 3dface at the base of the box to close the bottom of the box. The third and fourth lines draw a 3dface at the top of the box by using the .xy point filter to designate the x and y coordinate of each corner, then entering the Z coordinate using the H variable.

```
(defun C:3DBOX (/  pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of box: "))
```

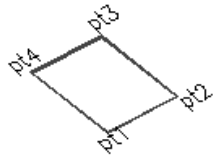
The user is prompted for information by the Getinfo function and the third line of the program.

The AutoCAD prompt:

Command: Pick first corner:
Pick opposite corner:
Enter height of box:

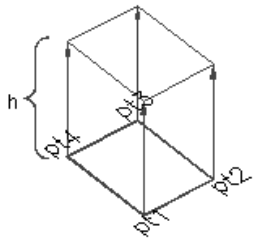
```
(procinfo)
(output)
```

The two-dimensional box is drawn using the Procinfo and Output functions.




```
(command "change" "Last" " " "Properties" "thickness" h " "
```

The two-dimensional box is extruded using the Change command.




```
"3dface" pt1 pt2 pt3 pt4 " "
"3dface" ".xy" pt1 h ".xy" pt2 h
         ".xy" pt3 h ".xy" pt4 h " "
)
)
```

A 3dface is added to the bottom of the extruded rectangle. A 3dface is also added to the top of the extruded rectangle using the .xy point filter. The Hide command is issued to see the box with hidden lines removed.

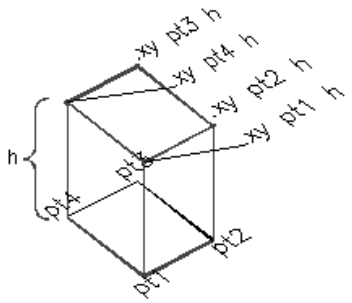


Figure 3.8: A 3D box using the C:3DBOX program

You may have noticed that although you added the program C:3DBOX to the Box1.lsp file, you still called up the program by entering its name, 3DBOX. We should emphasize here that AutoLISP files are only use as the vessel to hold your program code. AutoLISP makes no connection between the program name and the name of the file that holds the program. You could have called Box1.lsp by another name like Myprogm.lsp or Mybox.lsp, and the programs would still run the same. Of course, you would have to give the appropriate name when loading the file.

Creating a 3D Wedge Program

Let's create a new program that draws a wedge shape based on similar information given for the 3d box. Figure 3.9 shows how you might draw a wedge based on the rectangle drawn from the box program.

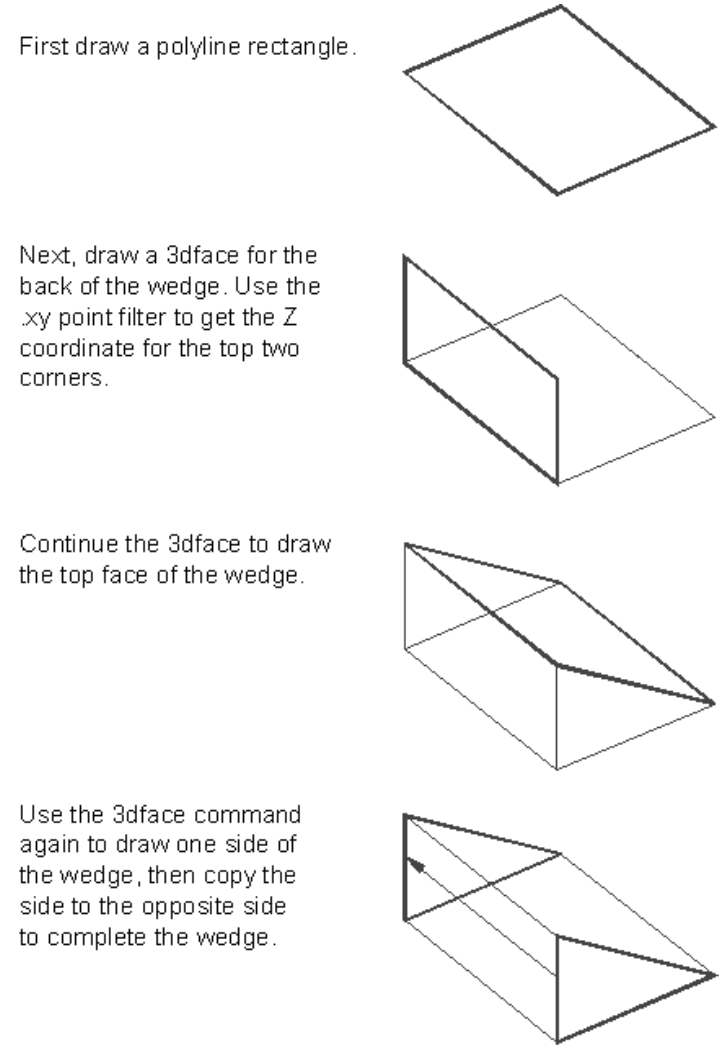


Figure 3.9: A program to draw a 3D wedge

The procedure outlined in Figure 3.9 was converted into the following AutoLISP program.

```
(defun C:3DWEDGE (/  pt1 pt2 pt3 pt4 h)
  (getinfo)
  (setq h (getreal "Enter height of wedge: "))
  (procinfo)
  (output)
    (command "3dface" pt1 pt4 ".xy" pt4 h ".xy" pt1 h pt2 pt3 ""
             "3dface" pt1 pt2 ".xy" pt1 h pt1 ""
             "copy" "L" "" pt1 pt4
    )
)
```

Try the following exercise to see how it works.

1. Exit AutoCAD temporarily either by using the AutoCAD Shell command or by using the End command to exit AutoCAD entirely.

2. Open the box1.lsp file again and add the program listed above to the file.

3. Return to the Chapt3 AutoCAD file.

4. Load Box1.lsp again. If you had to Exit AutoCAD to edit Box1.lsp, Load Rxy.lsp also.

5. Erase the box currently on the screen.

6. Start the C:3DWEDGE program by entering 3dwedge at the command prompt.

7. At the Pick first corner prompt, pick a point at the coordinate 2,3.

8. At the Pick opposite corner prompt, pick a point so the wedge's base is 7 units wide by 5 units wide.

9. At the Enter height prompt, enter 6. A wedge appears.

10. Issue the hide command. The wedge appears as a solid object.

This Wedge program is nearly identical to the C:3DBOX program with some changes to the last expression.

(command "3dface" pt1 pt4 ".xy" pt4 h ".xy" pt1 h pt2 pt3 ""

"3dface" pt1 pt2 ".xy" pt1 h pt1 ""

"copy" "L" "" pt1 pt3

)

The first line of this expression draws a 3d face on the top and vertical face of the wedge. It does this by using AutoCAD's .xy point filter to locate points in the Z coordinate. The second line draws a 3dface on the triangular side of the wedge again using the .xy point filter. The last line copies the triangular face to the opposite side of the wedge. Figure 3.10 shows the entire process.

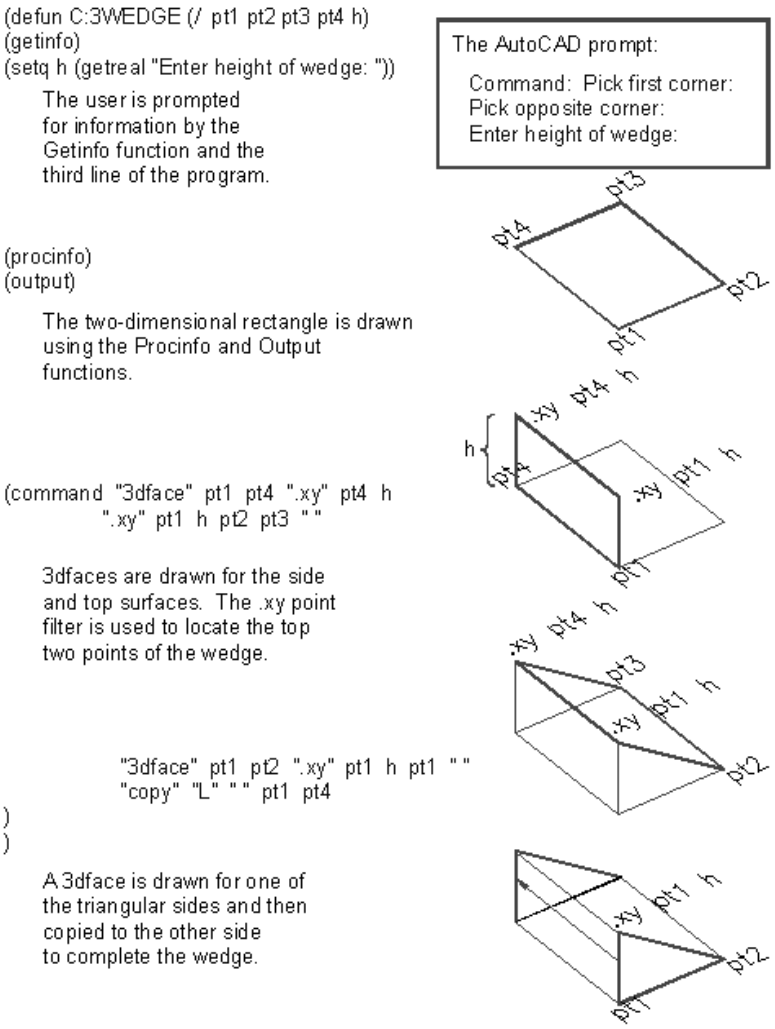


Figure 3.10: Box1.lsp with the C:3DWEDGE program added

Now you have three programs which use as their basic building blocks the three functions derived from your original Box program. You also have a function, rxy, which can be used independently to dynamically display relative x y coordinates.

Making Your Code More Readable

The box program is quite short and simple. As your programs grow in size and complexity, however, you will find that it becomes more and more difficult to read. Breaking the program down into modules help to clarify your code. There are other steps you can take to help keep your code readable.

Using Prettyprint

The term Prettyprint is used to describe a way to format your code to make it readable. Indents are used to offset portions of code to help the code's readability. Figure 3.12 shows three examples of the C:3DBOX program. The first example is arranged randomly. The second lists each expression as a line while the third makes use of prettyprint to organize the code visually.

```
(defun C:3DBOX (/ pt1 pt2 pt3 pt4 h) (getinfo)
(setq h (getreal "Enter height of box: "))(procinfo)(output)
(command "change" "Last" "" "Properties" "thickness" h
"" "3dface" pt1 pt2 pt3 pt4 "" "3dface" ".xy" pt1 h ".xy" pt2
h ".xy" pt3 h ".xy" pt4 h ""))


(defun C:3DBOX (/ pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of box: "))
(procinfo)
(output)
(command "change" "Last" "" "Properties" "thickness" h ""
"3dface" pt1 pt2 pt3 pt4 ""
"3dface" ".xy" pt1 h ".xy" pt2 h ".xy" pt3 h ".xy" pt4 h ""))
```

Figure 3.12 (continued on next page): Three ways to format the 3dbox program code

```
 (defun C:3DBOX (/ pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of box: "))
(procinfo)
(output)
     (command "change" "Last" "" "Properties" "thickness" h ""
              "3dface" pt1 pt2 pt3 pt4 ""
              "3dface" ".xy" pt1 h ".xy" pt2 h
              ".xy" pt3 h ".xy" pt4 h ""
      )
)
```

Figure 3.12 (continuet)

In the third example, notice how the listing is written under the command function. Each new command to be issued using the command functions is aligned with the next in a column. This lets you see at a glance the sequence of commands being used. Look at the RXY.lsp function in figure 3.3. As you progress through the book, make note of how the listings are written.

Using Comments

It often helps to insert comments into a program as a means of giving a verbal description of the code. Figure 3.13 shows the box program from chapter 1 including comments. The comments start with a semicolon and continue to the end of the line. When AutoLISP encounters a semicolon, it will ignore everything that follows it up to the end of the line. Using the semicolon, you can include portions of your Pseudocode as comments to the program code.

```
;Function to draw a simple 2 dimensional box
;---------------------------------------------------------------------


(defun c:BOX (/ pt1 pt2 pt3 pt4)                  ;define box function
(setq pt1 (getpoint "Pick first corner: "))       ;pick start corner
(setq pt3 (getpoint pt1 "Pick opposite corner: "));pick other corner
(setq pt2 (list (car pt3) (cadr pt1)))            ;derive secondcorner
(setq pt4 (list (car pt1) (cadr pt3)))            ;derive fourth corner
(command "line" pt1 pt2 pt3 pt4 "c")                ;draw box
)                                                  ;close defun

;Function to display relative XY coordinates in status line
;---------------------------------------------------------------------



(defun RXY (/ pt lt x last pick lpt1)
(if (not pt1)(setq lpt1 (getvar "lastpoint"))(setq lpt1 pt1)) ;get last point
    (while (/= pick t)
        (setq pt (cadr (setq lt (grread t))))                 ;read cursor
        (if (= (car lt) 5)
            (progn                                            ;location
               (setq x (strcat
                    (rtos (- (car pt) (car lpt1))) " x "      ;get X coord
                    (rtos (- (cadr pt) (cadr lpt1))) " SI= "  ;get Y coord
                    (rtos (*(- (car pt) (car lpt1))           ;get area
                            (- (cadr pt) (cadr lpt1))
                          )      ;close mult
                       2 2)      ;close rtos
                    )            ;close strcat
                )                ;close setq x
                (grtext -2 x)                                 ;display status
             )                   ;close progn
         )                       ;close if
      (setq pick (= 3 (car lt)))                              ;test for pick
     )                                                        ;close while
(cadr lt)                                                     ;return last
)                                                             ;coordinate
```

Figure 3.13: Sample of code using comments.

Using Capitals and Lower Case Letters

In programming, case sensitivity is a term that means a programming language treats upper an lower case letters differently. Except where string variables occur, AutoLISP does not have any strict requirements when it comes to using upper and lower case letters in the code. However, you can help keep you code more readable by using upper case letters sparingly. For example, you may want to use all uppercase letters for defuned function names only. That way you can easily identify user created functions within the code. You can also mix upper and lower case letters for variable names to help convey their meaning. This can help give the variable name more significance while still keeping the name short to conserve space. For example, You might give the name NewWrd for a variable that represents a new string value. NewWrd is more readable than say newwrd.

The only time AutoLISP is case sensitive is where strings variables are concerned. The string "Yes" is different from "yes" so you must take care when using upper and lower case letters in string variable types. We will cover this topic in more detail in the next chapter.

Dynamic Scoping

You may have noticed that in the functions getinfo, procinfo, and output, the argument list is empty (see figure 3-2). There are no variables local to those functions. The variables used in the programs C:BOX1, C:3DBOX, and C:3DWEDGE appear in their argument lists rather than in the functions being called by these programs. A binding is created between variables and their values within these programs so when the program ends, the variables loose the value assigned to them.

On the other hand, there are no variable bindings created within the individual functions called by the main programs so when the functions assign a value to a variables, the variables are free to all the functions used by the main program. This ability of a function to access variables freely from the calling function is known as Dynamic Scoping.

Whenever a function looks for a variables value, it will look at its own local variables first. If the variables value is not found, it then looks to the calling function or program for the value. Finally, if no value is found there, the function will look at the global environment for a value.

We saw this occur in chapter 2 with the Adsquare function. Adsquare looked in the AutoLISP global environment for the value of Golden since golden was not a formal argument to Adsquare (see figure 2.10).

In the case of the C:BOX1 program, all the variables used by the program and the functions it calls are included in its argument list. For this reason, variables have a binding within C:BOX1 but are free tp any of the functions called by C:BOX1.

An interesting affect of Dynamic Scoping is the ability to maintain two variables of the same name, each holding different values. To see how this works, do the following.

1. Erase the wedge currently on the screen.

2. Enter the following expression:

   (setq pt1 (getpoint))

3. pick a point near the coordinate 6,1. This assigns the coordinate you pick to the variable pt1.

4. Start the C:BOX1 program.

5. At the Pick first corner prompt, pick a point near the coordinate 2,3.

6. At the Pick opposite corner prompt, pick a point near so the box is about 7 units wide by 5 units wide. The 2D box appears.

7. Issue the line command and at the first point prompt, enter:

!pt1

9. The line will start at the point you selected in step 3 of this exercise.

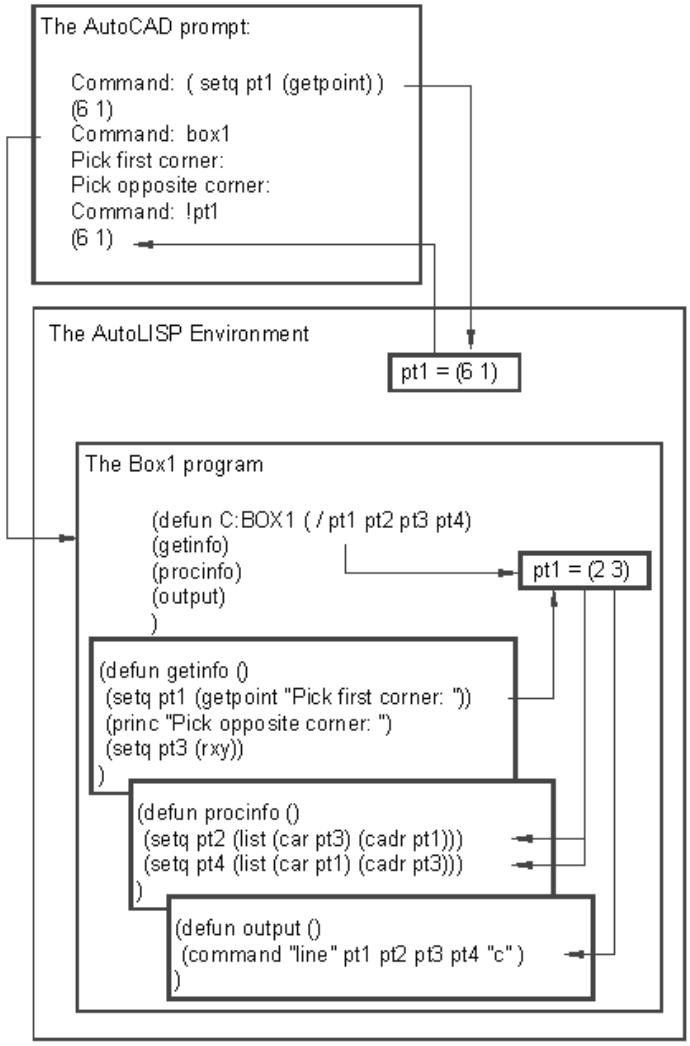


Figure 3.13: Dynamic Scoping

In the pervious example, you assigned a point location to the variable pt1. Then you ran the C:BOX1 program which also assigns a value to a variable called pt1. Pt1 in the C:BOX1 program is assigned the coordinate of the first corner point you pick. After the box is drawn, you start the line command and enter the variable pt1 as the starting point. You might expect the line to start from the corner of the box since it is the point last assigned to pt1. Instead, the line begins at the location you had assigned to pt1 before you ran the wedge program. When you ran the C:BOX1 program, two versions of pt1 existed, The global version of pt1 you created before you ran C:BOX1, and the version of pt1 created by the C:BOX1 program. The C:BOX1 version of pt1 lives and dies within that program and has no affect on the global version of pt1 (see figure 3.13).

Dynamic Scoping of variables can simplify your program coding efforts since you don't have to create an argument list for every functions you write. It can also simplify your management of variables. Dynamic Scoping can also create some interesting side affects as seen in the last exercise. For this reason, you should try and keep track of your variables and use global variables sparingly.

Conclusion

In this chapter you examined methods for designing your programs in an organized fashion. The box example, though simple enough to leave as a single program, allowed you to explore the concept of modular programming. It also showed how a program might be structured to give a clear presentation to both the creator of the program and those who might have to modify it later.

Chapter 4: Interacting with the Drawing Editor

Introduction

In the first three chapters of the book, you learned the basics of AutoLISP. You now have a framework within which you can begin to build your programs. We can now discuss each built-in AutoLISP functions without loosing you in AutoLISP nomenclature. In this and subsequent chapters, You will be shown how each of the built-in functions work. Since the AutoLISP interpreter can be used interactively, you can enter the sample expressions shown in this chapter at the AutoCAD command prompt to see first hand how they work.

A key element in a user defined function is how it obtains information from the user. In this chapter, you will look at some built in functions that expedite your programs information gathering. You have already seen the use of two of these functions, Getpoint and Getcorner, in chapter 2 and 3.

Prompting the user for Distances

You will often find the need to prompt the user to obtain a distance value. At times, it is easier for the user to select distances directly from the graphic screen by using the cursor. AutoLISP provides the Getdist function for this purpose.

How to use Getdist

The syntax for Getdist is:

(getdist [optional point value] [optional prompt string])

You can optionally supply one or two arguments to Getdist. These arguments can be either string values which will be used as prompts to user when the program runs, or point values indicating a position from which to measure the distance. Getdist will accept both keyboard input as well as cursor input and it always returns values in real numbers regardless of what unit style is current. This means that if you have your drawing set up using an architectural unit style, Getdist will return a value in decimal units. The following exercises will demonstrate these uses.

There are three ways to use Getdist. First, you can use it to input a distance by picking two points using the cursor. For example:

1. Open a file called Chapt4=. Set the snap mode and dynamic coordinate readout on.

2. Enter the following:

(setq dist1 (getdist "Pick point or enter distance: "))

The prompt line will display:

Pick first point or enter distance:

This prompt is the same as the string value used as the argument to Getdist. At this prompt, you can enter a numeric value in the current unit style or in decimal units or you can pick a beginning point using the cursor.

3. Pick a point at coordinate 3,3 with your cursor. You will get the prompt:

Second point:

and a rubber banding line will appear from the first point selected.

4. Pick a second point at coordinate 8,3 with you cursor, the distance between the two points selected is assigned to the variable dist1.

5. Enter the following:

!dist1

The distance value stored by dist1 is displayed.

5.0

Using the same expression as above, you can also enter a distance value at the keyboard.

1. Enter the following:

(setq dist1 (getdist "Pick point or enter distance: "))

The prompt line will display:

Pick first point or enter distance:

2. In the previous exercise, you picked a point at this prompt and Getdist responded by asking for a second point. Instead of picking a point, enter the value of 6.5. Once you have done this, 6.5 is assigned to the variable dist1.

3. Enter the following:

!dist1

The distance value stored by dist1 is displayed.

6.5

A third way to use Getpoint is to supply a point variable as an argument.

1. Enter the following at the command prompt:

(setq pt1 (getpoint "Enter a point: "))

When the Enter a point prompt appears, pick a point at the coordinate 3,3. The coordinate 3,3 is assigned to the variable pt1.

2. Enter the following expression:

(setq dist1 (getdist pt1 "Enter a second point: "))

Notice that the variable pt1 is used as an argument to getdist. A rubber-banding line appears from pt1 and the prompt displays the string prompt argument you entered with the expression.

3. Pick a point at the coordinate 9,6. The distance from pt1 to the point 9,6 is assigned to the variable dist1.

4. Enter the following to get the distance stored by dist1:

!dist1

6.7082

As you can see, getdist is quite flexible in the way it will accept input. This flexibility makes it ideal when you need

to get sizes of objects from the user. The user has the flexibility to either enter a size value at the keyboard or visually enter a size by picking points from the drawing area. The rubber-banding line provided by getdist allows the user to visually select a size. Several AutoCAD commands act in a similar manner. For example, the Text command allows you to select a text size either by entering a height or by visually selecting a height using your cursor. Figure 4.1 summarizes the three ways you can use Getdist.

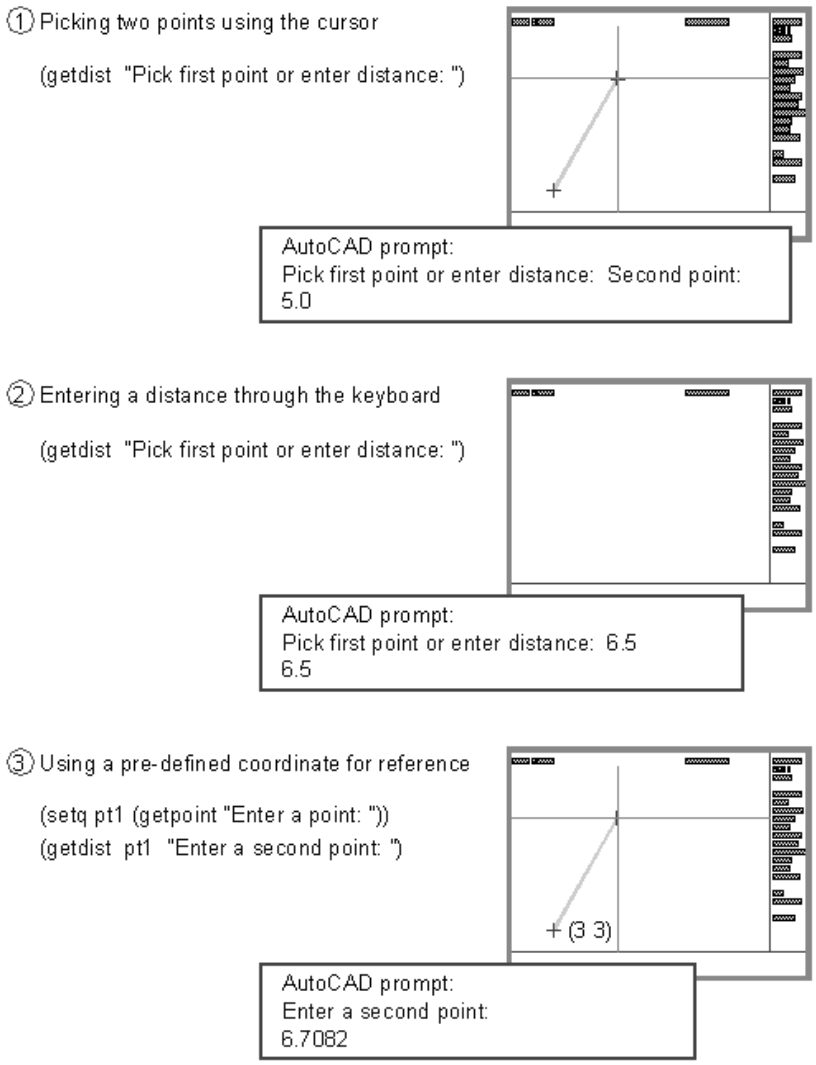


Figure 4.1: Three methods that Getdist accepts for input

A Sample Program Using Getdist

At times, you may need to find the distance between two points in a format other than the current unit style. Figure 4.2 lists a function that displays distances in decimal feet, a common engineering system of measure that AutoCAD does not directly support in release 10.

```
;Function to get distance in decimal feet -- Decft.lsp-----------------------
(defun decft (/ dst1 dst1s)
(setq dst1 (getdist "Pick first point or enter distance: ")) ;get distance
(setq dst1s (rtos (/ dst1 12.0) 2 4)) ;convert real value to string
(terpri) ;advance prompt one line
(strcat dst1s " ft.") ;return string plus "ft."
)
```

Figure 4.2: A program to display decimal feet

1. Exit AutoCAD by using the End command and open an AutoLISP file called Decft.lsp

2. Copy the file listed in Figure 4.1 into the Decft.lsp file. When you are done, and you are certain you have corrected all transcription errors, close the file.

3. Open an AutoCAD file called Chapt4. Be sure add the = sign at the end of the file name.

4. Use the Setup option on the main menu to set up you drawing using the Architectural unit style at a scale of 1/8 inch equals 1 foot. Select a sheet size of 11 by 17.

5. Set the snap distance to 12 by 12 and turn on the Dynamic coordinate readout.

6. Load the Decft.lsp file.

7. Enter Decft at the Command prompt. The following prompt appears:

   Pick first point or enter distance:

8. Pick a point at the coordinate 17'-0",9'-0". The next prompt appears:

   Second point:

9. Pick a point at the coordinate 100'-0",70'-0". The prompt displays the distance **103.0049 ft**.

The first line defines the function. The second line obtains the distance using the Getdist function.

```
(setq dst1 (getdist "Pick first point: "))
```

The third line uses the Rtos function to convert the distance value from a real to a string (see Chapter).

```
(setq dst1s (rtos (/ dst1 12.0) 2 4))
```

This conversion is necessary because we want to append the string "ft" to the numeric distance value obtained from getdist. The fourth line enters a return to the prompt line. The last line combines the string distance value with the string "ft.".

```
(terpri)

(strcat dst1s " ft")
```

AutoLISP will return the value of the last expression evaluated so the result of combining the distance with "ft" is displayed on the command prompt. The result value could be assigned to a variable for further processing as in the following line:

```
(setq dist2s (decft))
```

In this simple expression, the final value returned by Decft is assigned to the variable Dist2s.

How to Get Angle Values

When you are manipulating drawings with AutoLISP, you will eventually need to obtain angular information. AutoLISP provides the Getangle and Getorient functions for this purpose. These functions determine angles based on point input. This means that two point values are required before these functions will complete their execution. Getangle and Getorient will accept keyboard input of relative or absolute coordinates or cursor input to allow angle selection from the graphic screen. Getangle and Getorient always return angles in radians, regardless of the current AutoCAD Units settings. So even if you are using Architectural units, Getangle and Getorient will return a distance in radians (see chapter___ for a discussion of radian to degree conversion).

Using Getangle and Getorient

The difference between Getangle and Getorient is that Getangle will return an angle value relative to the current Unit setting for the 0 angle while Getorient will return an angle based on the "standard" 0 angle setting. For example, the default or "standard" orientation for 0 degrees is a direction from left to right but you can use the Units command or the Angbase or Angdir system variables to make 0 degrees a vertical direction. If a drawing is set up with 0 degrees being a direction from bottom to top, Getangle will return a value relative to this orientation while Getorient will return a value based on the "standard" orientation regardless of the Units, Angbase, or Angdir settings. Figure 4.3 illustrates these differences.

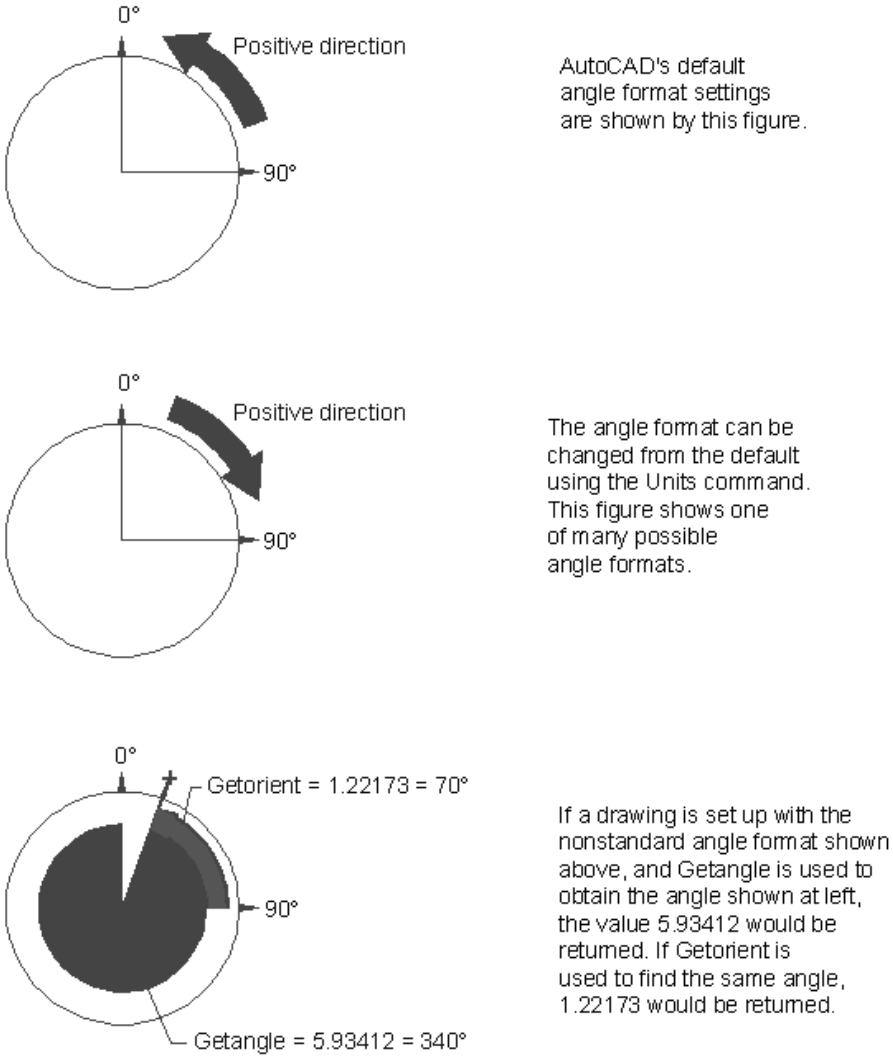


Figure 4.3: Comparing Getorient and Getangle

NOTE THAT Getorient ignores the angle direction setting. Even though the hypothetical setting uses a clockwise direction for the positive angle direction, getorient still returns angles using the counter-clockwise direction for positive angle.

The syntax for Getangle and Getorient are:

(getangle [optional point value] [optional prompt string])

(getorient [optional point value] [optional prompt string])

You can optionally supply one or two arguments to these functions. These arguments can be either string values which will be used as prompts to user when the program is run, or point values indicating a position from which to measure the angle.

Getangle and getorient accept three methods of input. These methods are similar to those offered by getdist. In the first method, you can enter two points.

1. Enter the following:

(setq ang1 (getangle "Pick a point or enter angle: "))

2. Pick a point at coordinate 3,3. A rubberbanding line appears from the picked point and you get the prompt:

Second point:

3. Pick another point at coordinate 6,6. The angle obtained from getangle is then assigned to the variable ang1.

4. Find the value of ang1 by entering the following:

!ang1

If the value returned by ang1 looks unfamiliar, it is because it is in radians. Radians are a way of describing an angle based on a circle whose radius is one unit. Such a circle will have a circumference of 2 pi. An angle of 90 degrees would describe a distance along the circle equivalent to 1/4 of the circles circumference or pi/2 or 1.5708 (see figure 4.4). This distance is the radian equivalent of the angle 90 degrees. We will discuss radians and their conversion to degrees in more detail in the next chapter.

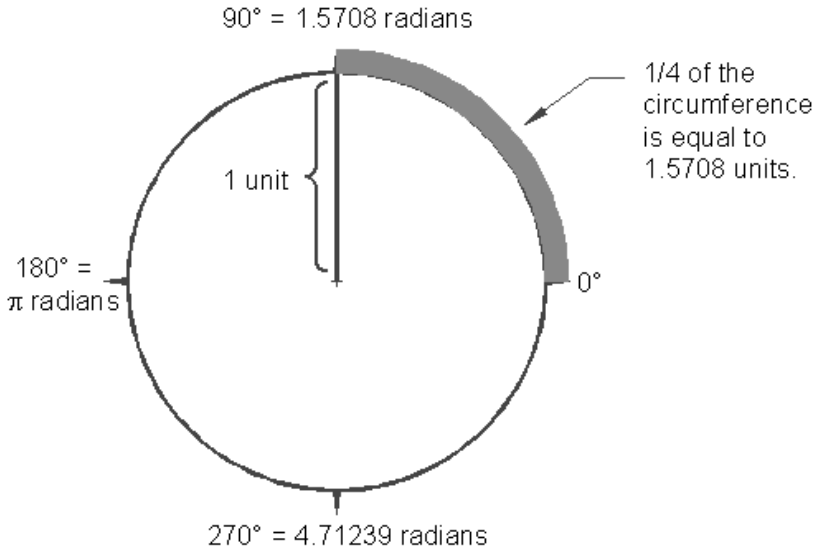


Figure 4.4: 90 degrees described in radians

Just as with getdist, if a point is selected using a cursor, you are prompted for a second point and a rubberbanding line appears. The rubberbanding line allows you to visually see the angle you are displaying in the drawing area. The second method is to enter an angle from the keyboard.

1. Enter the same expression you did previously:

   (setq ang1 (getangle "Pick a point or enter angle: "))

2. Enter the following:

   <45

   The angle of 45 degrees is applied to the variable ang1 in radians.

3. Enter the following to find the value of Ang1:

   !ang1

The value **0.785398** is returned. This is the radian equivalent of 45 degrees.

Just as with getdist, you can supply a point variable as an argument to getangle.

1. Enter the following expression:

   (setq pt1 '(3 3))

   This assigns the coordinate 3,3 to the variable pt1.

2. Enter the expression:

   (setq ang1 (getangle pt1 "Indicate angle: "))

   A rubber-banding line appears from the coordinate 3,3.

3. Pick a point at coordinate 7,6. The angle from pt1 to 7,6 is assigned to ang1.

4. To display the value of ang1, enter:

   !ang1

   The value **0.6463501** is returned. This is the radian equivalent to 37 degrees.

By accepting both keyboard and cursor input, these two functions offer flexibility to the user. Angles can be

specified as exact numeric values or visually using the cursor. The Rotate command works in a similar way by accepting angular input as well as allowing the user to visually select an angle of rotation. Figure 4.5 summarizes the three ways you can use Getanglel

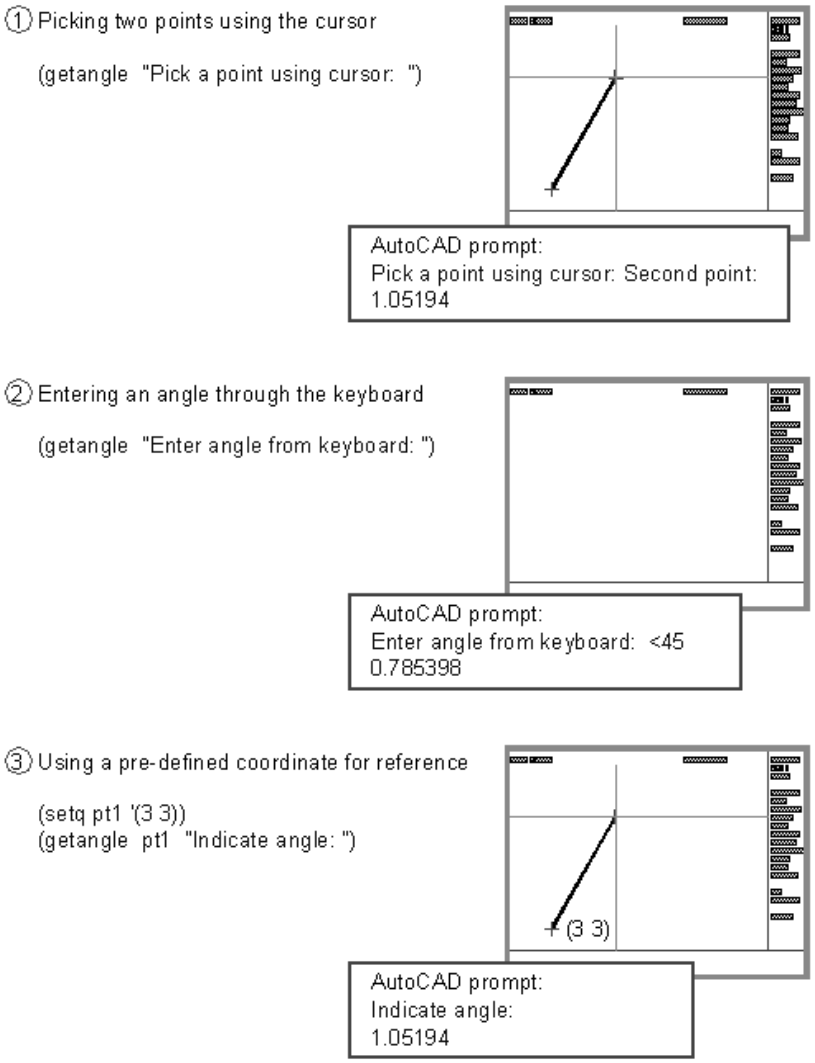


Figure 4.5: Three methods Getangle accepts for input

How to Get Text Input

You can prompt the user for text input using the Getstring and Getkword functions. These two functions always return String variables even if a number is entered as a response. Though both Getstring and Getkword are used for obtaining String values, They operate in slightly ways. Getstring will accept almost anything as input so it can be used to acquire words or sentences of up to 132 characters long with no particular restriction to content. Getkword, however, is designed to work in conjunction with the Initget function, (described in detail later in this chapter), to accept only specific strings, or keywords, as input. If the user tries to enter a string to Getkword that is not a predetermined keyword, the user is asked to try again.

Using Getstring

The syntax for Getstring is:

(getstring [optional variable or number] [optional prompt string])

You can optionally supply one or two arguments to Getstring. You can supply a string value for a prompt to the user or you can also supply a non-nil variable or number, to signify that Getstring is to accept spaces in the user input. For example, in many AutoCAD commands, pressing the space bar on your keyboard is the same as pressing the return key. Normally, Getstring will also read a space bar as a return thus disallowing more than a single word to be input to the Getstring function. Enter the following expression:

(setq str1 (getstring "Enter a word: "))

the following prompt appears:

Enter a word:

Try entering your first and last name. As soon as you press the space bar, the getstring function and the command prompt returns read your first name. Display the value of str1 by entering:

!str1

Since the space bar is read as a return, once it is pressed, the entered string is read and AutoLISP goes on to other business. If you provide non-nil variable or number argument, however, getstring will read a space bar input as spaces in a text string thereby allowing the user to enter whole sentences or phrases rather than just single words. Enter the following:

(setq str2 (getstring T "Enter a sentence: "))

The following prompt appears:

Enter a sentence:

Try entering your first and last name again. This time you are able to complete your name. Display the value of str2 by entering:

!str2

An integer or real value or anything that will not evaluate to nil could replace the T argument in the above example.

We would like to restate that you can enter numbers in response to Getstring but they will be treated as strings therefore, you won't be able to apply math functions to them. You can, however, convert a number that is a string variable type into a real or integer type using the Atof, Atoi and Ascii functions (see Chapter 7 for details).

Using Getkword

The syntax for Getkword is:

(getkword [options prompt string])

Getkword offers only one argument option, the string prompt. Unlike most other functions, Getkword must be used in conjunction with the Initget function. Initget is used to restrict the user by allowing only certain values for input. If a disallowed value is entered, the user is prompted to try again. In the case of Getkword, Initget allows you to specify special words that you expect as input. The most common example of this situation would be the Yes/No prompt. For example, you may want to have your program perform one function or another based on the user entering a Yes or No to a prompt. Enter the following:

(initget "Yes No")

(setq str1 (getkword "Are you sure? <Yes/No>: "))

The following prompt appears:

Are you sure? <Yes/No>:

If the user does not enter a Yes, Y, No or N, Getkword will continue to prompt the user until one of the keywords is entered. Try entering MAYBE. You get the message:

Invalid option keyword.

Are you sure? <Yes/No>:

Now try entering Y. Getkword accepts the Y as a valid keyword and the whole keyword Yes is assigned to the variable str1. If you had entered n or N for No, then the entire keyword No would have been assigned to str1.

The words Yes and No are first established as keywords using the initget function. Getkword then issues the prompt and waits for the proper keyword to be entered. If capital letters are used in the Initget function's string argument, they too become keywords. This is why Y is also allowed as keyword in the above example (see Initget for more details). It doesn't matter if you enter a capital or lower case Y. It only matters that you enter the letter that is capitalized in the Initget string argument.

How to Get Numeric Values

At times, you may want to prompt the user for a numeric value such as a size in decimal units or the number of times a functions is to be performed. Getreal and Getint are the functions you would use to obtain numeric input.

Using Getreal and Getint

Getreal always returns reals and Getint always returns integer values. The syntax for Getint is:

(getint [optional prompt string])

The syntax for Getreal is:

(getreal [optional prompt string])

Getreal and getint can be supplied an optional argument in the form of a prompt string. Just as with all the previous get functions, this prompt string is displayed as a prompt when an expression using getreal or getint is evaluated.

How to Control User Input

You can add some additional control to the Get functions described in this chapter by using the Initget function. In version 2.6 of AutoCAD, Initget was only used to provide keywords for the Getkword function (see Getkword earlier in this chapter). With Version 9, other functions are added to give more control over input to the other Get functions. For example, with Initget, you can force a Get function not to accept zero or negative values. Unlike most functions, Initget always returns nil.

Using Initget

The syntax for Initget is:

(initget [optional bit code] [optional string input list])

You must supply at least one argument to initget. The bit code option controls the kinds of input that are restricted, or, in some cases, how rubber-banding lines or windows are displayed. Table 4.1 lists the bit codes and their meaning.

CODE	MEANING
1	null input not allowed
2	zero values not allowed
4	negative values not allowed
8	do not check limits for point values
16	return 3D point rather than 2D point
32	Use dashed lines for rubber-banding lines and windows

Table 4.1: The Initget bit codes and what they mean

Bit codes can be used individually or added together to affect several options at once. For example, if you do not want a Get function to accept null, zero and negative values for input, you can use the following:

```
(initget 7)
(setq int1 (getint "Enter an integer: "))
```

In this example, though there is no formal 7 bit code value, for initget, 7 is 1 plus 2 plus 4 so the restrictions associated with bit codes 1, 2, and 4 are applied to the Getint that follows. You can also write the Initget expression above as:

```
(initget (+ 1 2 4))
```

Not all initget bit codes are applicable to all Get functions. Table 4.2 shows which code works with which Get function.

FUNCTION	***can be used with initget bit code:***
getint	1,2,4
getreal	1,2,4
getdist	1,2,4,16,32
getangle	1,2,32
getorient	1,2,32
getpoint	1,8,16,32
getcorner	1,8,16,32
getkword	1
getstring	no initget codes honored

Table 4.2: The Bit codes and related Get functions

Prompting for Dissimilar Variable Types

The Initget functions allows Get functions to accept string input even though a Get function may expect data in another format. For example, you may want your function to accept either a point or string value from a prompt. To do this you might have the following expressions:

```
(initget 1 "Next")

(setq pt1 (getpoint "Next/<pick a point>: "))
```

In this example, the user will see:

Next/<pick a point>:

Initget sets up the word Next as a keyword. Once this is done, The Getpoint function in the following expression is allowed to accept either a point value or the string "next" or "n". Note that the user need not enter his or her response specifically in upper or lower case letters.

Using Multiple Keywords

You can have more than one keyword for situations where multiple choices are offered. For example, you might have a situation where the user is asked to choose from several fonts as in the following expressions:

```
(initget "Roman Gothicg Scripts")

(setq str1 (getkword "Font style = Roman/Gothicg/Scripts: "))
```

The user will see the following prompt when the expressions are read:

```
Font style = Roman/Gothicg/Scripts:
```

In this case, the user can enter one of the three keywords listed in the prompt or their capital letters.

Capitalization is important in specifying the keyword as initget will allow both the whole word and the capitalize letters of a word to be used as keywords.

```
(initget "Style STeak STROKE")

(setq kword (getkword "Style/Steak/STROKE: "))
```

In the above example, the user can either enter S for style, ST for string or STROKE for stroke. Initget expects the input keyword to be as long as the capitalized portion of the keyword definition. Therefor, to enter the Style option, the user only needs to enter s since that is capitalized portion of the keyword. For the Steak option, the user must enter at least the ST portion of the word, once again, because the ST is capitalized in the argument. Finally, the user must enter the entire word Stroke to select that keyword since it is all capitalized in the argument.

Another way to specify keywords is to capitalize all the characters of the word then follow it with a comma and the abbreviation as in the following:

```
(initget "STYLE,S STEAK,ST STROKE")
```

This sample has the same affect as the previous example.

How to Select Groups of Objects

AutoCAD provides the Select command to allow you to pre-select a group of objects to be edited in some way. Select acts the same way as other commands when the Select object prompt is presented. For example, when you issue the Move command, you are prompted to Select objects. You can then use any number of options to select groups of objects to move. These options range from windows to single object selection or de-selection. The group of objects you select is called a selection set.

AutoLISP offers a similar facility in the Ssget function. You can think of ssget as an abbreviation for selection set

get. When this function is used as part of an expression, the Select objects prompt appears and you can go about selecting a single object or groups of objects to be processed by your program in much the same way as the standard select objects prompt.

Using Ssget

The syntax for Ssget is:

(ssget [optional selection mode][optional point][optional point])

Three optional arguments affect the way ssget selects objects. the first option, selection mode, allows you to predetermine the method ssget uses for selection. For example, if you want the user to use a window to select objects, you would include the string "W" as the first argument to ssget as in the following:

```
(setq obj1 (ssget "W"))
```

Other options are:

"P" select a previous selection set

"L" select the last object added to database

"C" select objects using a crossing window

"X" select objects using a filter list.

Most of these options should be familiar to you as standard selection options. The "X" mode however is probably new to you. This mode allows you to select objects based on their properties such as layer, linetype color and so on. We won't discuss this option until later in this book as it is rather involved.

You can also specify points either to select objects at a known location or as input to the window or crossing mode options. For example, if you want to select an object you know is at point 1,1, you can place the following in your program:

```
(setq obj1 (ssget '(1 1)))
```

You can also indicate a window by indicating two point as in the following:

```
(setq obj1 (ssget "W" '(1 1) '(9 12)))
```

Point specification need not be in the form of a quoted list. You can supply a variable as well. Suppose two points have been previously defined in your program:

```
(setq pt1 (getpoint "Pick a point: "))

(setq Pt2 (getcorner pt1 "Pick another point: "))
```

.

.

.

The periods in the sample above indicate other expressions in your program Later in your program, You can then use those two point to select objects with a window:

.

.

.

(setq obj1 (ssget "W" pt1 pt2))

If you provide points as arguments, however, ssget does not pause for user input. It assumes that the points provided as arguments indicate the location of the objects to be included in the selection set.

Finally, if you do not provide any arguments, ssget will allow the user to select the mode of selection. If the following appears in your program:

(setq obj1 (ssget))

Ssget displays the prompt:

Select objects:

The user can either pick objects with a single pick, or enter W or C to select a standard or crossing window. The use can also use the Remove mode to de-select objects during the selection process. In fact, all the standard object selection options are available. When the user is done, he or she can press return to confirm the selection.

We must caution you that AutoLISP only allows you to have six selection set variables at any given time. This shouldn't be a problem so long as you do not make your selection set variables global. Remember that a variable is made global by not including its symbol in the functions argument list.

A Sample Program Using Ssget

Figure 4.8 shows two AutoLISP programs. The first one called Group uses the ssget function to store a set of objects as a group. This program is similar to the Select AutoCAD command only Group allows the user to give that grouping a name so that it can be recalled any time during the current editing session. Also, with the Group program in earlier versions of AutoCAD, you can only store up to 6 sets of groups. After 6, any group you try to store will return nil. The groups saved by the Group program are not saved when you exit the file however.

```
(defun collct ()
(if (not gp1)(setq gp1 (getstring "enter name of group: ")))
(if (<= *gpcnt 4)(setq *gplst (cons gp1 *gplst)))
(set (read gp1) (ssget))
)


(defun C:Group (/ gp1)
(if (not *gpcnt)(setq *gpcnt 0))
(setq *gpcnt (1+ *gpcnt))
(if (<= *gpcnt 4)(collct)(freegp))
(princ)
)

(defun FREEGP ()
(prompt "\nYou have exceeded the number of groups allowed: ")
(prompt "\nEnter group name to re-use ")
(princ *gplst)
(setq gp1 (getstring ": "))
(set (read gp1) nil)
(gc)
(collct)
)
```

Figure 4.8: Examples of Ssget

The second program called Freegp will free up memory taken by a group so that if exceed 6 selection sets, you can remove one or more selection sets from memory to allow the use of the Group function again. It is up to you, however, to remember the names of your groups.

1. Exit the Chapt4 file and open an AutoLISP file called Group.lsp. Copy the program shown in figure 4.4 into the file.

2. Go back to your Chapt4 AutoCAD drawing file and load Group.lsp.

3. Draw eight vertical lines in the drawing area.

4. Enter Group at the command prompt. The following prompt appears:

Enter name of group:

5. Enter the following for a group name:

group1

6. The **Select objects** prompt appears. pick the leftmost three lines. You can use a window or pick them individually. Once you are done selecting objects, press return. You will see a message similar to the following:

<Selection set: n>

Command:

The value of n will depend on the number of selection sets previously used in the current editing session.

7. Start the move command. At the **Select objects** prompt, enter the name of the group you selected previously:

!group1

The objects you picked using the Group function are highlighted just as if you had picked them manually while in the move command.

The Group program works by first prompting you for a name to give your group:

(setq gp1 (getstring "enter name of group: "))

This name is saved as a variable gp1. Next, it uses ssget to assign a selection set to a variable with the name you entered during the previous expression.

(set (read gp1) (ssget))

This is accomplished using the set and read functions. Read is a function that reduces a string to a symbol. When the variable gp1 is applied to read, the string, "group1" which you entered in the previous expression, is returned as the variable name or symbol group1. The set function then applies the selection set from the ssget function to the symbol group1. You may recall that set works just like setq only set will evaluate both its arguments. Since set evaluates its first argument, you can use an expression such as (read gp1) to derive a variable name .

Since the variable created using set and read is created while the program is executing, it cannot be included in the argument list of the program. For this reason it becomes global variable.

As mentioned earlier, AutoLISP allows up to 6 selection sets to be available at once. If you try to create more than 6, ssget will return nil instead of a selection set. Any program that uses the ssget function will not work properly once the maximum number of concurrent selection sets is exceeded. To recover the use of ssget, you must set at least one of the selection set variables to nil then perform what is called a "garbage collection" using the GC function. This is precisely what the Freegp program does.

1 Enter **Freegp** at the command prompt. At the prompt:

Enter name of group to delete:

2. Enter **group1**.

Freegp will set group1 to nil and recover any node space group1 may have take.

Freegp uses an AutoLISP function called gc. To help understand what gc does, think of nodes, portions of memory used to store AutoLISP symbols and values, are of two types, Bound and Free. Bound nodes are those that are being used to store symbols and value. Free nodes are unassigned. When a new symbol is created, it is assigned a free node. That node is then bound to that symbol. Even if a symbol is eventually assigned a nil value, it will still be bound to a node. Gc, short for garbage collection, releases node space that is bound to a symbol with a nil value. Frequent use of Gc is not recommended, as it can be time consuming. In the case of the ssget function however, it is the only function that will allow you to recover memory and regain ssget's use.

Conclusion

In this chapter, you were introduced to the many ways AutoLISP allows you to interact with the user to gather information. In summary, the following points were discussed:

- Several functions allow you to pause your program to allow the user to input data.
- Many of these functions accept data either from the keyboard or from the cursor
- You can place controls on the type of data being input through the initget function.

In addition, you saw how objects can be selected using the Ssget function. Though there are only a handful of functions that give AutoLISP its interactive capabilities, the flexibility of these functions give a complete range of possibilities for gathering data from the user.

In the next chapter, you will explore how you can make your program do more of your work for you by making decisions and performing repetitive tasks.

Chapter 5: Making Decisions with AutoLISP

Introduction

As we mentioned in chapter 3, AutoLISP is designed to help us solve problems. One of the key elements to problem solving is the ability to perform one task or another based on some existing condition. We might think of this ability as a way for a program to make decisions. Another element of problem solving is repetitive computation. Something that might be repetitive, tedious, and time consuming for the user to do may be done quickly using AutoLISP. In this chapter, we will look at these two facilities in AutoLISP.

Making Decisions

You can use AutoLISP to create macros which are like a predetermined sequence of commands and responses. A macro building facility alone would be quite useful but still limited. Unlike macros, AutoLISP offers the ability to perform optional activities depending on some condition. AutoLISP offers two conditional functions that allow you to build-in some decision making capabilities into your programs. These are the if and cond functions. If and cond work in very similar ways with some important differences.

How to Test for Conditions

The if functions works by first testing to see if a condition is met then it performs one option or another depending on the outcome of the test. This sequence of operations is often referred to as an **if-then-else** conditional statement. **if** a condition is met, **then** perform computation A, **else** perform computation B. As with all else in AutoLISP, the if function is used as the first element of an expression. It is followed by an expression that provides the test. A second

and optional third argument follows the test expression. The second argument is an expressions that is to be evaluated if the test condition is true. If the test returns false or nil, then **if** evaluates the third argument if it exists, otherwise if returns nil. The following shows the general syntax of the if function.

(if (test expression)

(expression) (optional expression)

)

The test expression can be use any function but often you will use two classes of functions called predicates and logical operators. Predicates and logical operators are functions that return either true or false. Since these functions don't return a value the way most functions do, the atom T is used by predicates and logical operators to represents a non-nil or true value. There are several functions that return either a T or nil when evaluated.

FUNCTION	**RETURNS T (TRUE) IF...**
Predicates	
<	a numeric value is less than another
>	a numeric value is greater than another
<=	a numeric value is less than or equal to another
>=	a numeric value is greater than or equal to another
=	two numeric or string values are equal
/=	two numeric or string values are not equal
eq	two values are one in the same
equal	two expressions evaluate to the same value
atom	an object is an atom (as opposed to a list)
boundp	a symbol has a value bound to it
listp	an object is a list
minusp	a numeric value is negative
numberp	an object is a number, real or integer
zerop	an object evaluates to zero

Table 5.1 A list of AutoLISP predicates and logical operators.

Logical Operators

and	all of several expressions or atoms return non-nil
not	a symbol is nil
null	a list is nil
or	one of several expressions or atoms return non-nil

Table 5.1 (continuet) A list of AutoLISP predicates and logical operators.

You may notice that several of the predicates end with a p. The p denotes the term predicate. Also note that we use the term object in the table. When we say object, we mean any lists or atoms which include symbols and numbers. Numeric values can be numbers or symbols that are bound to numbers.

All predicates and logical operators follow the standard format for AutoLISP expressions. They are the first element in an expression followed by the arguments as in the following example:

(> 2 4)

The greater than predicate compares two numbers to see if the one on the left is greater than the one on the right. The value of this expression is nil since two is not greater than four.

The predicates >,<,>=, <= all allows more than two arguments as in the following:

(> 2 1 5 8)

When more than two arguments are used, > will return T only if each value is greater than the one to is right. The above expression returns nil since 1 is not greater than 5.

The functions and, not, null and or are similar to predicates in that they too return T or nil. But these functions, called logical operators, are most often used to test predicates (see table 5.1). For example, you could test to see if a value is greater than another:

(setq val1 1)

(zerop val1)

nil

We set the variable val1 to 1 then test to see if val1 is equal to zero. The zerop predicate returns nil. But suppose you want to get a true response whenever val1 does not equal zero. You could use the not logical operator to "reverse" the result of zerop:

(setq val1 1)

(not (zerop val1))

T

Since not returns true when its argument returns nil, the end result of the test is T. Not and null can also be used as predicates to test atoms and lists.

Using the If function

In chapter 2, you saw briefly how the if function worked with the not logical operator to determine whether to load a program or not. Looking at that expression again (see Figure 5.1), you see a typical use of the if function.

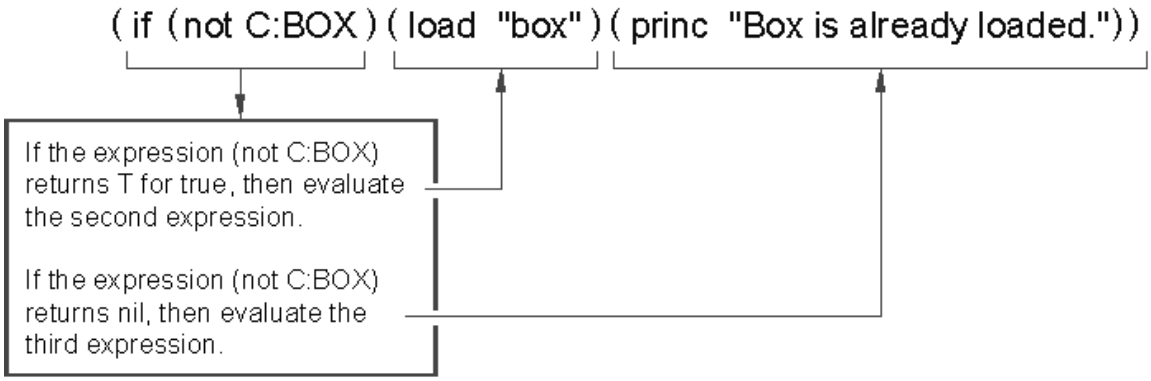


Figure 5.1: An expression showing the syntax for the If function

This function tests to see if the function C:BOX exists. If C:BOX doesn't exist, it is loaded. This simple program decides to load a program based on whether the program has already been loaded. The test function in this case is **not**. Lets look at how you might apply **if** to another situation. In the following exercise, you will add an expression to decide between drawing a 2 dimensional or 3 dimensional box.

1. Open the Box1.lsp file.

2. Change the first line of the Box program so it reads as follows:

(defun BOX1 (/ pt1 pt2 pt3 pt4)

By removing the C: from the function name, Box1 now becomes a function that can be called from another function.

3. Change the first line of the 3dbox program so it reads as follows:

(defun 3DBOX (/ pt1 pt2 pt3 pt4)

4. Add the program listed in boldface print in figure 5.2 to the end of the Box1.lsp file. Your Box1.lsp file should look like figure 5.2. You may want to print out Box1.lsp and check it against the figure.

5. Save and Exit Box1.lsp.

6. Open an AutoCAD file called chapt5. Be sure to use the = suffix with the file name.

7. Load the box1.lsp file.

8. Run the Mainbox program by entering **mainbox** at the command prompt. You will see the following prompt:

Do you want a 3D box <Y=yes/Return=no>?

9. Enter y. The 3dbox function executes.

```
(defun getinfo ()
(setq pt1 (getpoint "Pick first corner: "))
(princ "Pick opposite corner: ")
(setq pt3 (rxy))
)
 (defun procinfo ()
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
)

(defun output ()
(command "line" pt1 pt2 pt3 pt4 "c" )
)

(defun BOX1 (/ pt1 pt2 pt3 pt4)
(getinfo)
(procinfo)
(output)
)
 (defun 3DBOX (/ pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of box: "))
(procinfo)
(output)
(command "change" "L" "" "P" "th" h ""
         "3dface" pt1 pt2 pt3 pt4 ""
         "3dface" ".xy" pt1 h ".xy" pt2 h
         ".xy" pt3 h ".xy" pt4 h ""
)
)

(defun C:3DWEDGE (/ pt1 pt2 pt3 pt4 h)
(getinfo)
(setq h (getreal "Enter height of wedge: "))
(procinfo)
(output)
(command "3dface" pt1 pt4 ".xy" pt4 h ".xy" pt1 h pt2 pt3 ""
         "3dface" pt1 pt2 ".xy" pt1 h pt1 ""
         "copy" "L" "" pt1 pt4
)
)
(defun C:MAINBOX ()
(setq choose (getstring "\nDo you want a 3D box <Y=yes/Return=no>? "))
    (if (or (equal choose "y")(equal choose "Y"))(3dbox)(box1))
)
```

Figure 5.2: The BOX1.LSP file with C:MAINBOX added.

In this example, you first turned the programs C:BOX1 and C:3DBOX into functions by removing the C: from their names. Next, you created a control program called C:MAINBOX that prompts the user to choose between a 2 dimensional or 3 dimensional box. The first line in the C:MAINBOX program, as usual, gives the program its name and determines the local variables. The next line uses the Getstring function to obtain a string value in response to a prompt:

(setq choose (getstring "\nDo you want a 3D box <Y=yes/Return=no>? "))

The prompt asks the user if he or she wants a 3 dimensional box and offers the user two options, Y for yes or Return for no. The third line uses the if function to determine whether to run the BOX1 or 3DBOX function. Notice that the or and the equal predicates are used together.

(if (or (equal choose "y")(equal choose "Y"))(3dbox)(box1))

The **or** function returns T if any of its arguments returns anything other than nil. Two arguments are provided to **or**. One test to see of the variable **choose** is equal to the lower case y while the other checks to see if **choose** is equal to an upper case y. If the value of either expression is T, then or returns T. So, if the user responds by entering either an upper or lower case y to the prompt in the second line, then the or predicate returns T. Any other value for choose will result in a nil value from **or** (see figure 5.3).

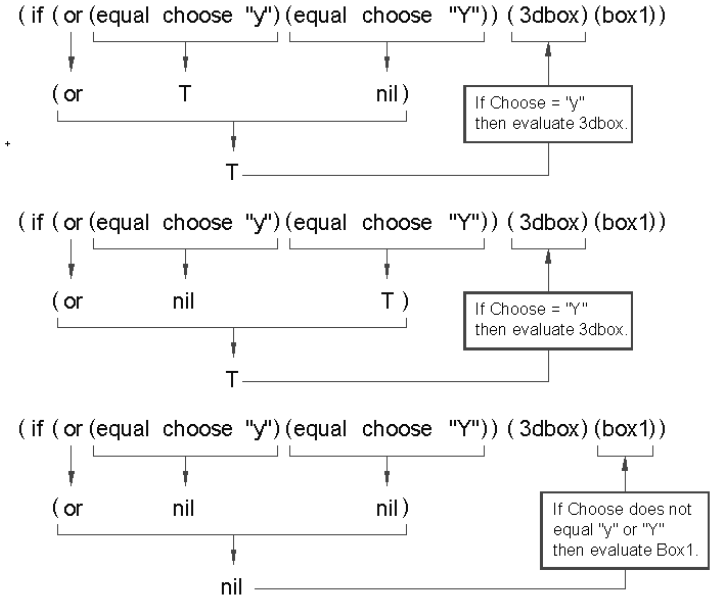


Figure 5.3: Using the logical operator Or

Once the or function returns a T, the second argument is evaluated which means that a 3dbox is drawn. If or returns nil, then the third argument is evaluated drawing a 2 dimensional box. The **and** function works in a similar way to **or** but and requires that all its arguments return non-nil before it returns T. Both **or** and will accept more that two arguments.

How to Make Several Expressions Act like One

There will be times when you will want several expressions to be evaluated depending on the results of a predicate. As an example, lets assume that you have decided to make the 2 dimensional and 3 dimensional box programs part of the C:MAINBOX program to save memory. You could place the code for these functions directly in the C:MAINBOX program and have a file that looks like figure 5.5. When this program is run, it acts exactly the same way as in the previous exercise. But in figure 5.4, the functions BOX1 and 3DBOX are incorporated into the **if** expression as arguments.

```
(defun C:MAINBOX (/ pt1 pt2 pt3 pt4 h choose)
(setq choose (getstring "\nDo you want a 3D box <Y=yes/Return=no>? "))
   (if (or (equal choose "y")(equal choose "Y"))
    (progn                                ;if choose = Y or y then draw a 3D
box
     (getinfo)
     (setq h (getreal "Enter height of box: "))
     (procinfo)
     (output)
       (command "change" "Last" "" "Properties" "thickness" h ""
                "3dface" pt1 pt2 pt3 pt4 ""
                "3dface" ".xy" pt1 h ".xy" pt2 h
                ".xy" pt3 h ".xy" pt4 h ""
          );end command
     );end progn
   (progn ;if choose /= Y or y then draw a 2D box
     (getinfo)
     (procinfo)
     (output)
    );end progn
   );end if
);end MAINBOX
```

Figure 5.4: The C;Mainbox program incorporating the code from Box1 and 3dbox

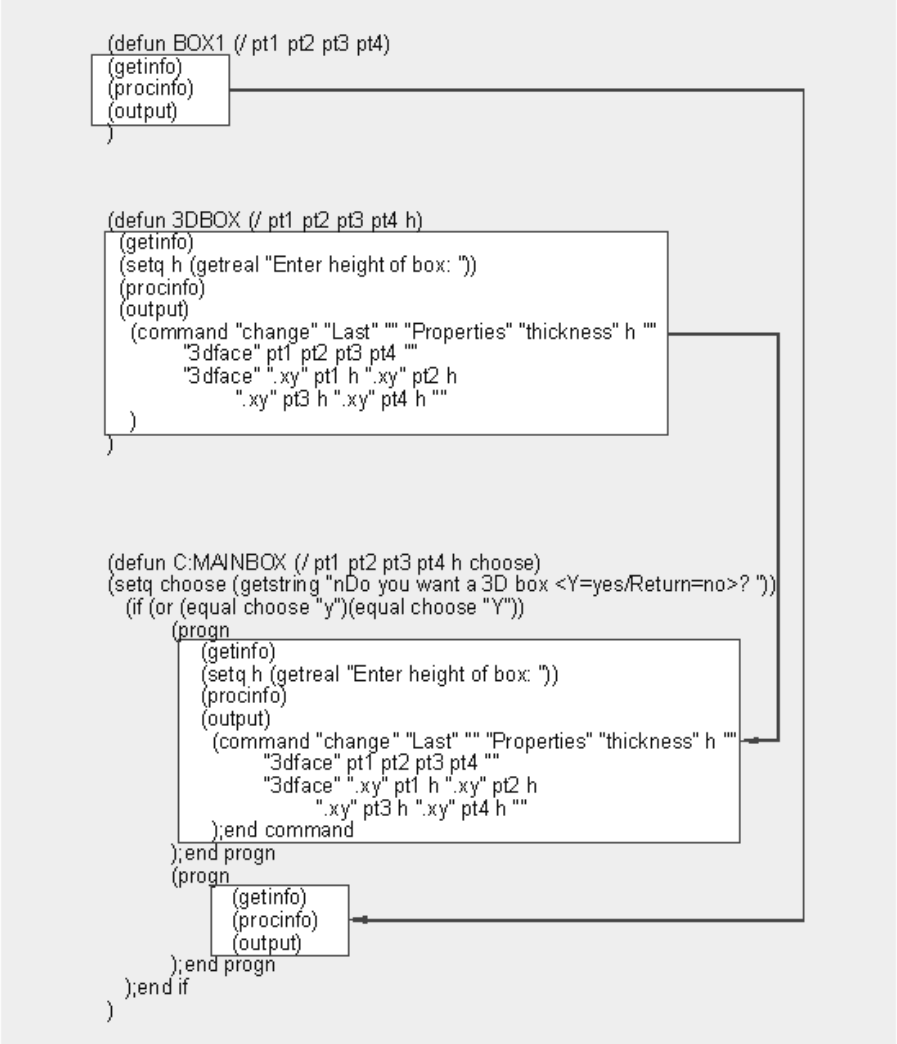


Figure 5.5: Using the progn function

We are able to do this using the progn function. Progn allows you to have several expression where only one is expected. Its syntax is as follows:

(progn

(expression1) (expression2) (expression3) . . .

)

Figure 5.5 shows how we arrived at the program in figure 5.4. The calls to functions BOX1 and 3DBOX were replaced by the actual expressions used in those functions.

How to Test Multiple Conditions

There is also another function that acts very much like the if function called Cond. Arguments to cond consists of one or more expressions that have as their first element a test expression followed by an object that is to be evaluated if the test returns T. Cond evaluates each test until it comes to one that returns T. It then evaluates the expression or atom associated with that test. If other test expressions follow, they are ignored.

Using the Cond function

The syntax for cond is:

(cond

((test expression)[expression/atom][expression/atom]...)

((test expression)[expression/atom][expression/atom]...)

((test expression)[expression/atom][expression/atom]...)

((test expression)[expression/atom][expression/atom]...)

)

Figure 5.6 shows the cond function used in place of the if function. Cond's syntax also allows for more than one expression for each test expression. This means that you don't have to use the Progn function with cond if you want several expressions evaluated as a result of a test.

```
(defun C:MAINBOX (/ choose)
(setq choose (getstring "\nDo you want a 3D box <Y=yes/Return=no>? "))
   (cond
      ( (or (equal choose "y") (equal choose "Y")) (3dbox))
      ( (or (/= choose "y") (/= choose "Y")) (box1))
   )
)
```

Figure 5.6: Using cond in place of if

Figure 5.7 shows another program called Chaos that uses the Cond function. Chaos is an AutoLISP version of a mathematical game used to demonstrate the creation of fractals through an iterated function. The game works by following the steps shown in Figure 5.8.

```
;function to find the midpoint between two points
(defun mid (a b)
(list (/ (+ (car a)(car b)) 2)
(/ (+ (cadr a)(cadr b)) 2)
)
)

;function to generate random number
(defun rand (pt / rns rleng lastrn)
(setq rns (rtos (* (car pt)(cadr pt)(getvar "tdusrtimer"))))
(setq rnleng (strlen rns))
(setq lastrn (substr rns rnleng 1))
(setq rn (* 0.6 (atof lastrn)))
(fix rn)
)

;The Chaos game
(defun C:CHAOS (/ pta ptb ptc rn count lastpt randn key)
(setq pta '( 2.0000 1 )) ;define point a
(setq ptb '( 7.1962 10)) ;define point b
(setq ptc '(12.3923 1 )) ;define point c
(setq lastpt (getpoint "Pick a start point:")) ;pick a point to start
    (while (/= key 3) ;while pick button not pushed
        (setq randn (rand lastpt)) ;get random number
            (cond ;find midpoint to a b or c
               ( (= randn 0)(setq lastpt (mid lastpt pta)) ) ;use corner a if 0
               ( (= randn 1)(setq lastpt (mid lastpt pta)) ) ;use corner a if 1
               ( (= randn 2)(setq lastpt (mid lastpt ptb)) ) ;use corner b if 2
               ( (= randn 3)(setq lastpt (mid lastpt ptb)) ) ;use corner b if 3
               ( (= randn 4)(setq lastpt (mid lastpt ptc)) ) ;use corner c if 4
               ( (= randn 5)(setq lastpt (mid lastpt ptc)) ) ;use corner c if 5
            );end cond
        (grdraw lastpt lastpt 5) ;draw midpoint
        (setq key (car (grread T))) ;test for pick
     );end while
);end Chaos
```

Figure 5.7: The Chaos game program

Cond can be used anywhere you would use if. For example:

(if (not C:BOX) (load "box") (princ "Box is already loaded. "))

can be written:

(cond

((not C:BOX) (load "box"))

((not (not C:BOX)) (princ "Box is already loaded. "))

)

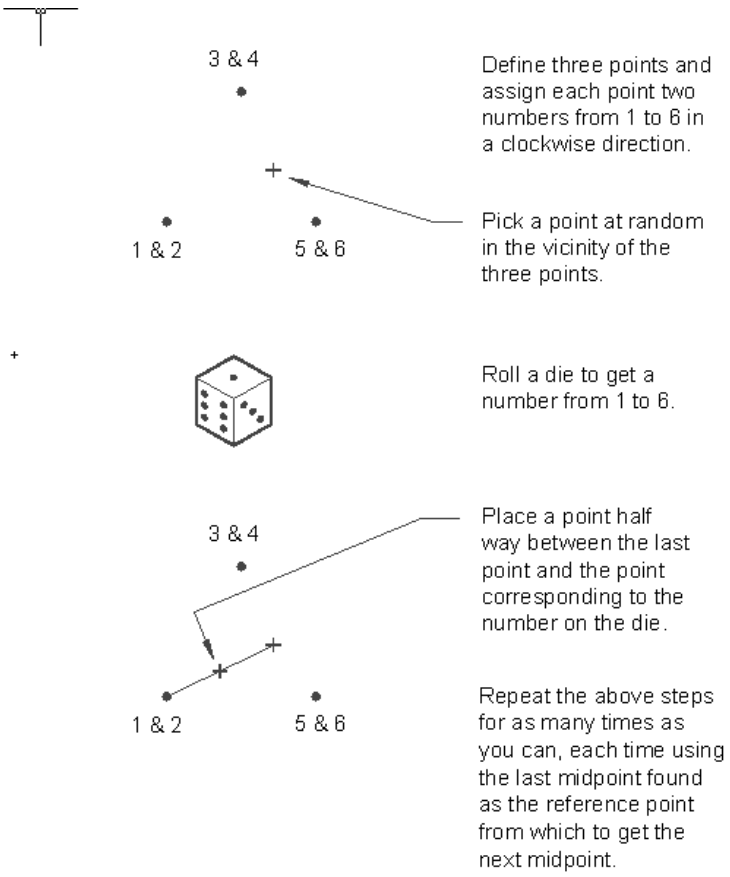


Figure 5.8: How to play the Chaos game

Cond is actually the primary conditional function of AutoLISP. The if function is offered as an alternative to cond since it is similar to conditional functions used in other programming languages. Since the If function syntax is simpler and often more readable, you may want to use if where the special features of cond are not required.

How to Repeat parts of a Program

Another useful aspect to programs such as AutoLISP is their ability to perform repetitive tasks. For example, suppose you want to be able to record a series of keyboard entries as a macro. One way to do this would be to use a series of Getstring functions as in the following:

(Setq str1 (getstring "\nEnter macro: "))

(Setq str2 (getstring "\nEnter macro: "))

(Setq str3 (getstring "\nEnter macro: "))

(Setq str4 (getstring "\nEnter macro: "))

.

Each of the str variables would then be combined to form a variable storing the keystrokes. Unfortunately, this method is inflexible. It requires that the user input a fixed number of entries, no more and no less. Also, this method would use memory storing each keyboard entry as a single variable.

It would be better if you had some facility to continually read keyboard input regardless of how few or how many different keyboard entries are supplied. The While function can be used to repeat a prompt and obtain data from the user until some test condition is met. The program in figure 5.9 is a keyboard macro program that uses the while function.

```
;Program to create keyboard macros -- Macro.lsp

(defun C:MACRO (/ str1 macro macname)
(setq macro '(command))                                    ;start list with command
(setq macname (getstring "\nEnter name of macro: "))       ;get name of macro
    (while (/= str1 "/")                                   ;do while str1 not eq.
         (setq str1 (getstring "\nEnter macro or / to exit: " )) ;get keystrokes
            (if (= str1 "/")
                   (princ "\nEnd of macro ")              ;if / then print message
                   (Setq macro (append macro (list str1)) ;else append keystrokes
               )                                          ;end if macro list
        );end while
   (eval (list 'defun (read macname) '() macro))          ;create function
   );end macro
```

Figure 5.9: A program to create keyboard macros

Using the While Function

The syntax for while is:

(while (test expression)

(expression 1)(expression 2)(expression 3) ....

)

The first argument to while is a test expression. Any number of expressions can follow. These following expressions are evaluated if the test returns a non-nil value.

Open an AutoLISP file called Macro.lsp and copy the contents of figure 5.8 into this file. Since this is a larger program file than you worked with previously, you may want to make a print out of it and check your print out against figure 5.8 to make sure you haven't made any transcription errors. Go back to your AutoCAD Chapt5 file. Now you will use the macro program to create a keyboard macro that changes the last object drawn to a layer called hidden. Do the following:

1. Draw a diagonal line from the lower left corner of the drawing area to the upper right corner.
2. Load the Macro.lsp file
3. Enter Macro at the command prompt.
4. At the following prompt:

   Enter name of macro:

5. Enter the word **chlt**. At the prompt:

   Enter macro or / to exit:

6. Enter the word **CHANGE**.

The Enter macro prompt appears again. Enter the following series of words at each Enter macro prompt:

Enter macro or / to exit: L

Enter macro or / to exit: [press return]

Enter macro or / to exit: P

Enter macro or / to exit: LT

Enter macro or / to exit: HIDDEN

Enter macro or / to exit: [press return]

Enter macro or / to exit: /

This is where the **while** function takes action. As you enter each response to the Enter macro prompt, **while** test to see if you entered a forward slash. If not, it evaluates the expressions included as its arguments. Once you enter the backslash, **while** stops its repetition. You get the prompt:

End of macro CLAYER

The input you gave to the Enter macro prompt is exactly what you would enter if you had used the change command normally. Now run your macro by entering:

chlt

The line you drew changes to the hidden line type.

When Macro starts, it first defines two variables def and macro.

```
(setq def "defun ")
```

```
(setq macro '(command))
```

Def is a string variable that is used later to define the macro. Macro is the beginning of a list variable which is used to store the keystrokes of the macro. The next line prompts the user to enter a name for the macro.

```
(setq macname (getstring "\nEnter name of macro: "))
```

The entered name is then stored with the variable macname. Finally, we come to the while function.

```
(while (/= str1 "/")
```

The **while** expression is broken into several lines. The first line contains the actual while function along with the test expression. In this case, the test compares the variable str1 with the string "/" to see if they are not equal. So long as str1 is not equal to "/", while will execute the arguments that follow the test. The next four lines are the expressions to be evaluated. The first of these lines prompts the user to enter the text that compose the macro.

```
(setq str1 (getstring "\nEnter macro or / to exit: " ))
```

When the user enters a value at this prompt, it is assigned to the variable str1. The next line uses an if function to test if str1 is equal to "/".

```
(if (= str1 "/")
```

If the test results in T, the next line prints the string End of macro.

(princ "\nEnd of macro ")

This expression prints the prompt **End of macro** to the prompt line. If the test results in nil, the following line appends the value of str1 to the existing list macro.

(Setq macro (append macro (list str1)))

The append function takes one list and appends its contents to the end of another list. In this case, whatever is entered at the Enter macro prompt is appended to the variable macro. Each time a new value is entered at the Enter macro prompt, it is appended to the variable macro creating a list of keyboard entries to be saved (see figure 5.10).

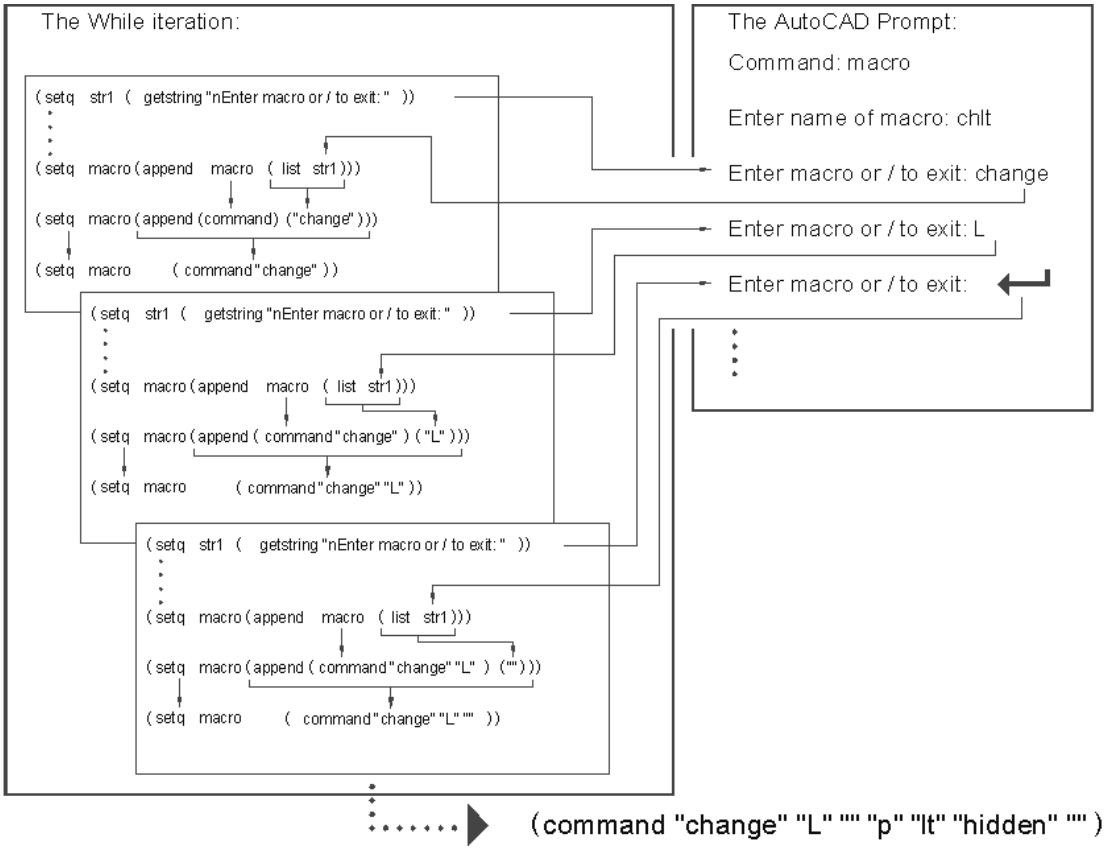


Figure 5.10: Appending lists to other lists

The next two lines close the if and while expressions. Note that comments are used to help make the program easier to understand.

);end if

);end while

The last line combines all the elements of the program into an expression that, when evaluated, creates a new macro program.

(eval (list (read def) (read macname) '() macro))

)

Figure 5.11 shows gives breakdown of how this last line works.

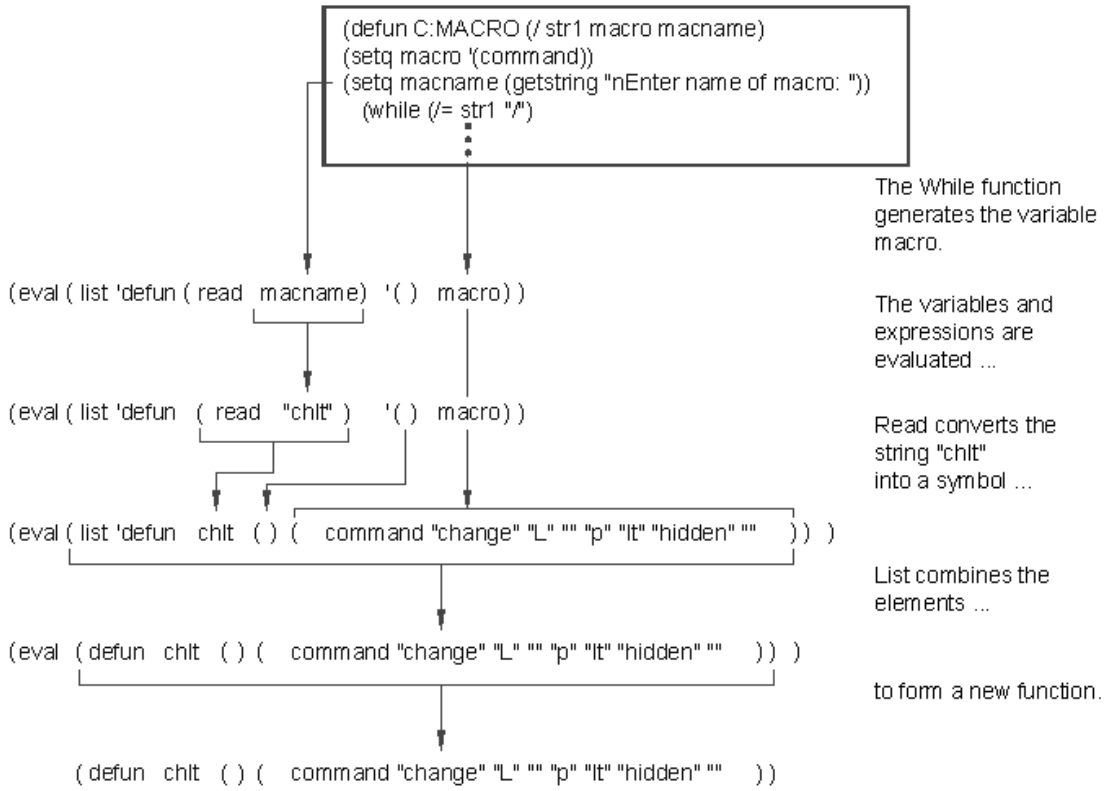


Figure 5.11: How the macro is defined

The read function used in this expression is a special function that converts a string value into a symbol. If a string argument to read contains spaces, read will convert the first part of the string and ignore everything after the space.

The while expression does not always include prompts for user input. Figure 5.12 shows a simple program that inserts a sequence of numbers in increasing value. Each number is increased by one and they are spaces 0.5 units apart. The user is prompted for a starting point and a first and last number. Once the user inputs this information, the program calculates the number to be inserted, inserts the number using the text command, calculates the next number and so on until the last number is reached.

```
;Program to draw sequential numbers -- Seq.lsp
(defun C:SEQ (/ pt1 currnt last)
(setq pt1 (getpoint "\nPick start point: "))
(setq currnt (getint "\nEnter first number: "))
(setq last (getint "\nEnter last number: "))
(command "text" pt1 "" "" currnt)                    ;write first number
     (while (< currnt last)                          ;while not last number
        (setq currnt (1+ currnt))                    ;get next number
          (command "text" "@.5<0" "" "" currnt)      ;write value of currnt
      );end while
);end seq
```

Figure 5.12: Sequential number program

This program expects the current text style to have a height of 0.

Using the Repeat Function

Another function that performs recursions is the repeat function. Repeat works in a similar way to While but instead of using a predicate to determine whether to evaluate its arguments, repeat uses an integer value to determine the number of times to perform an evaluation. The syntax for repeat is:

(repeat [n]

(expression 1)(expression 2) (expression 3) ...

)

The n above is an integer or a symbols representing an integer.

The program in Figure 5.13 shows the sequential number program using repeat instead of while. When run, this program appears to the user to act in the same way as the program that uses while.

```
;Program to write sequential numbers using Repeat
(defun C:SEQ (/ pt1 currnt last)
(setq pt1 (getpoint "\nPick start point: "))
(setq currnt (getint "\nEnter first number: "))
(setq last (getint "\nEnter last number: "))
(command "text" pt1 "" "" currnt)                ;write first number
   (repeat (- last currnt)                       ;repeat last - currnt times
      (setq currnt (1+ currnt))                  ;add 1 to currnt
      (command "text" "@.5<0" "" "" currnt)      ;write value of currnt
   );end repeat
);end seq
```

Figure 5.13: Sequential number program using repeat

Using Test Expressions

So far, we have shown you functions that perform evaluations based on the result of some test. In all the examples, we use predicates and logical operators for testing values. While predicates and logical operators are most commonly used for tests, you are not strictly limited to these functions. Any expression that can evaluate to nil can also be used as a test expression. Since virtually all expressions are capable of returning nil, you can use almost any expression as a test expression. The following function demonstrates this point:

(defun MDIST (/ dstlst dst)

(setq dstlst '(+ 0))

(while (setq dst (getdist "\nPick distance or Return to exit: "))

(Setq dstlst (append dstlst (list dst)))

(princ (Eval dstlst))

);end while

);end MDIST

This function gives the user a running tally of distances. The user is prompted to pick a distance or press Return to exit. if a point is picked, the user is prompted for a second point. The distance between these two points is displayed on the prompt. The Pick distance prompt appears again and if the user picks another pair of points, the second distance is added to the first and the total distance is displayed. This continues until the user presses return. The following discussion examines how this function works.

As usual, the first line defines the function. The second line creates a variable called dstlst and gives it the list value (+ 0).

(defun MDIST (/ dstlst dst)

(setq dstlst '(+ 0))

The next line begins the **while** portion of the program. Instead of a predicate test, however, this expression uses a setq function.

(while (setq dst (getdist "\nPick point or Return to exit: "))

As long as points are being picked, getdist returns a non-nil value to setq and **while** repeats the evaluation of its arguments. When the user presses return, getdist returns nil and **while** quits evaluating its arguments. We see that **while** is really only concerned with nil and non-nil since the test expression in this example returns a value other than T.

The next few lines append the current distance to the list dstlst then evaluates the list to obtain a total:

(Setq dstlst (append dstlst (list dst)))

(princ (eval dstlst))

The function princ prints the value obtained from (eval dstlst) to the prompt (see Figure 5.14).

```
;Program to measure non-sequential distances -- Mdist.lsp
(Defun C:MDIST (/ dstlst dst)
(setq dstlst '(+ ))                              ;create list with plus
;while a return is not entered ...
    (while (setq dst (getdist "\nPick point or Return to exit: "))
       (Setq dstlst (append dstlst (list dst)))     ;append distance value
       (princ (Eval dstlst))                        ;print value of list
    );end while
);end mdist
```

Figure 5.14: The Mdist function

Conclusion

You have been introduced to several of the most useful functions available in AutoLISP. You can now begin to create functions and programs that will perform time consuming, repetitive tasks quickly and easily. You can also build-in some intelligence to your programs by using decision making functions. You may want to try your hand at modifying the programs in this chapter. For example, you could try to modify the Mdist function to save the total distance as a global variable you can later recall.

In the next chapter, you will get a brief refresher course in geometry.

Chapter 6: Working With Geometry

Introduction

It is inevitable that your work with AutoLISP will involve some geometric manipulations. With the box program in chapter 2, you have already created a program that derives new point locations based on user input. There, you learned how to take coordinate lists apart, then re-assemble them to produce a new coordinate. In this chapter, you will be introduced to other AutoLISP functions that will help you determine locations in your drawings coordinate system and in the process, you will review some basic trigonometry.

How to find Angles and Distances

In chapter 4, you learned how to prompt the user for angles and distances. At time, however, you will want to find angles and distances based on the location of existing point variables rather than relying on user input every time you need to find an angle.

Suppose you want to find a way to break two parallel lines between two points in a manner similar to the standard AutoCAD break command. In addition, you would like this function to join the ends of the two broken portions of each line to form an opening. Figure 6.1 shows a drawing of the process along with a description of what must occur. This drawing can be developed into pseudocode for your program. A function similar to this is commonly used in architectural drawings to place an opening in a wall.

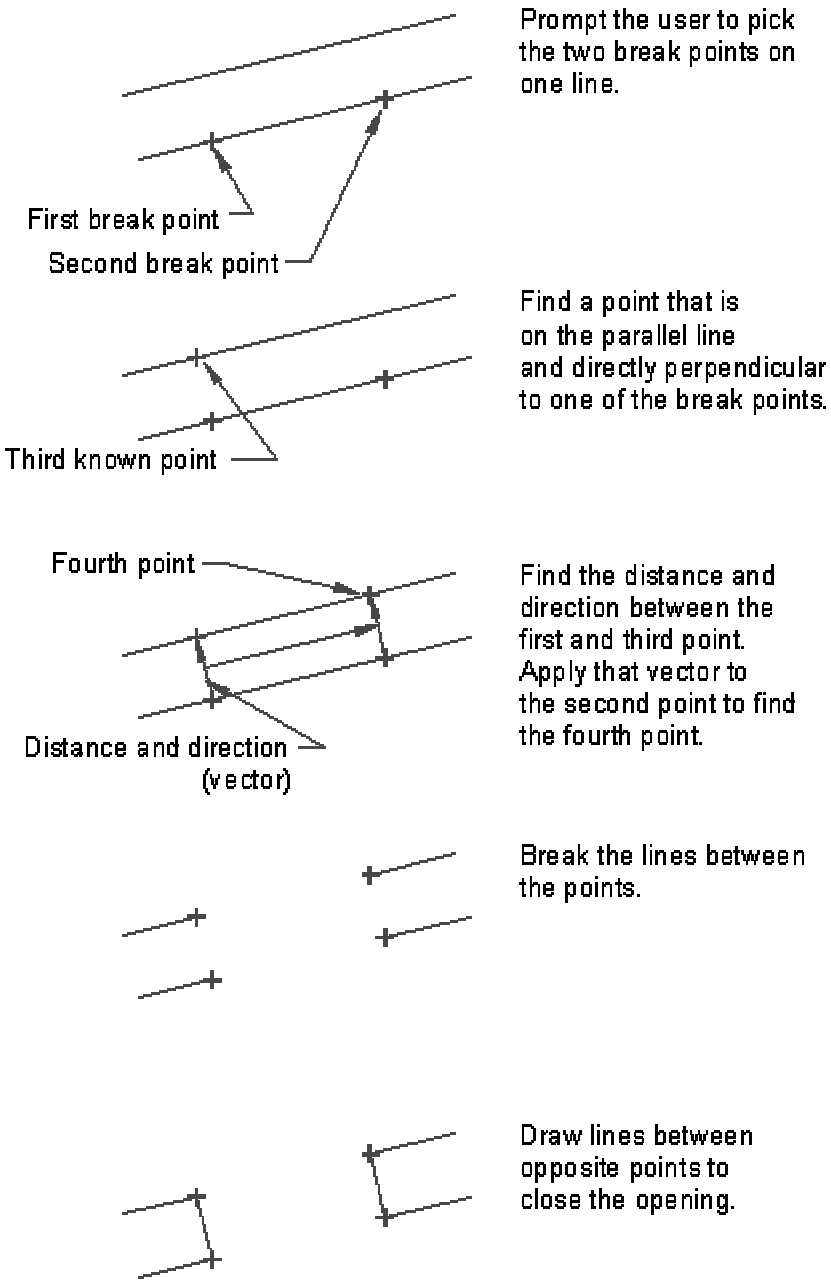


Figure 6.1: Sketch of the parallel line break program

In Chapter 3, we discussed the importance of designing your program to be simple to use. This program is designed

to obtain the information needed to perform its task using the minimum of user input. Since it is similar to the break command, it also tries to mimic the break program to some degree so the user feels comfortable with it. As you read through this section, pay special attention to the way information is gathered and used to accomplish the final result.

Open an AutoLISP file called Break2.lsp and copy the program in figure 6.2. Open a new AutoCAD file called Chapt6. Draw a line from point 2,4 to 12,4 then offset that line a distance of 0.25 units. Your screen should look like figure 6.3.

```
;Program to break 2 parallel lines -- Break2.lsp
(defun c:break2 (/ pt1 pt2 pt3 pt4 pt0 ang1 dst1)
   (setvar "osmode" 512)                              ;near osnap mode
   (setq pt1 (getpoint "\nSelect object: "))          ;get first break point
   (setq pt2 (getpoint pt1 "\nEnter second point: "))  ;get second break point
   (setvar "osmode" 128)                              ;perpend osnap mode
   (Setq pt3 (getpoint pt1 "\nSelect parallel line: "));get 2nd line
   (Setvar "osmode" 0)                                ;no osnap mode
   (setq ang1 (angle pt1 pt3))                        ;find angle btwn lines
   (setq dst1 (distance pt1 pt3))                     ;find dist. btwn lines
   (setq pt4 (polar pt2 ang1 dst1))                   ;derive pt4 on 2nd line
      (command
            "break" pt1 pt2                           ;break 1st line
            "break" pt3 pt4                           ;break 2nd line
            "line" pt1 pt3 ""                         ;close ends of lines
            "line" pt2 pt4 ""
      )
)
```

Figure 6.2: The parallel line break program

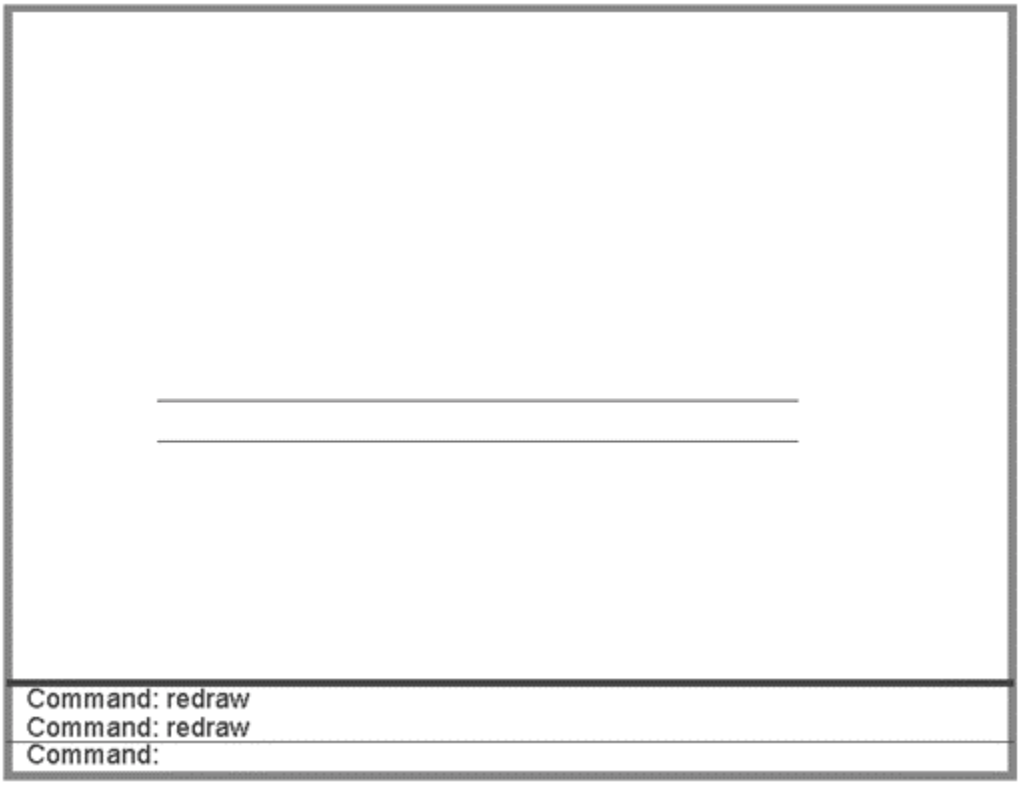


Figure 6.3: Two parallel lines drawn

1. Load the Break2.lsp file and enter **break2** at the Command prompt.

2. At the prompt:

Select object:

The osnap cursor appears. Pick the lowermost line near coordinate 5,4.

3. At the next prompt:

Enter second point:

Pick the lowermost line again near coordinate 10,4.

4. Finally, at the prompt:

Select parallel line:

pick the upper line near its midpoint. The two line break and are joined at their break points to form an opening (see figure 6.4)

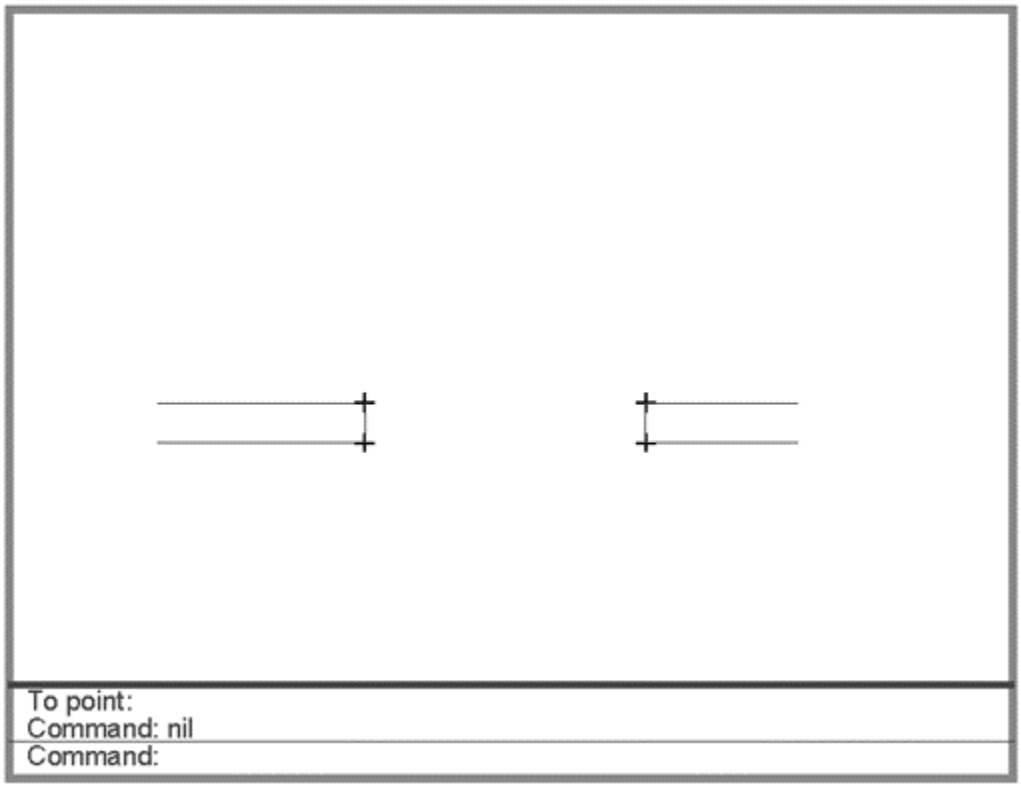


Figure 6.4: The lines after using Break2

Now draw several parallel lines at different orientations and try the Break2 program on each pair of lines. Break2 places an opening in a pair of parallel lines regardless of their orientation. Let's look at how break2 accomplishes this.

Understanding the Angle, Distance, and Polar Functions

The first line after defun function uses the setvar function to set the osnap system variable to the nearest mode.

(defun C:BREAK2 (/ pt1 pt2 pt3 pt4 pt0 ang1 dst1)

(setvar "osmode" 512)

This ensures that the point the user picks at the next prompt is exactly on the line. It also give the user a visual cue to pick something since the osnap cursor appears.

When setvar is used with the osmode system variable, a numeric code must also be supplied. This code determines the osnap mode to be used. Table 6.1 shows a list of the codes and their meaning.

Code	**Equivalent Osnap mode**
1	Endpoint
2	Midpoint
4	Center
8	Node
16	Quadrant
32	Intersection
64	Insertion
128	Perpendicular
256	Tangent
512	Nearest
1024	Quick

Table 6.1: Osmode codes and their meaning

The next line prompts the user to select an object using the getpoint function:

(setq pt1 (getpoint "\nSelect object: "))

Here, the variable pt1 is used to store the first point location for the break. Since the nearest osnap mode is used, the osnap cursor appears and the point picked falls exactly on the line (see Figure 6.5).

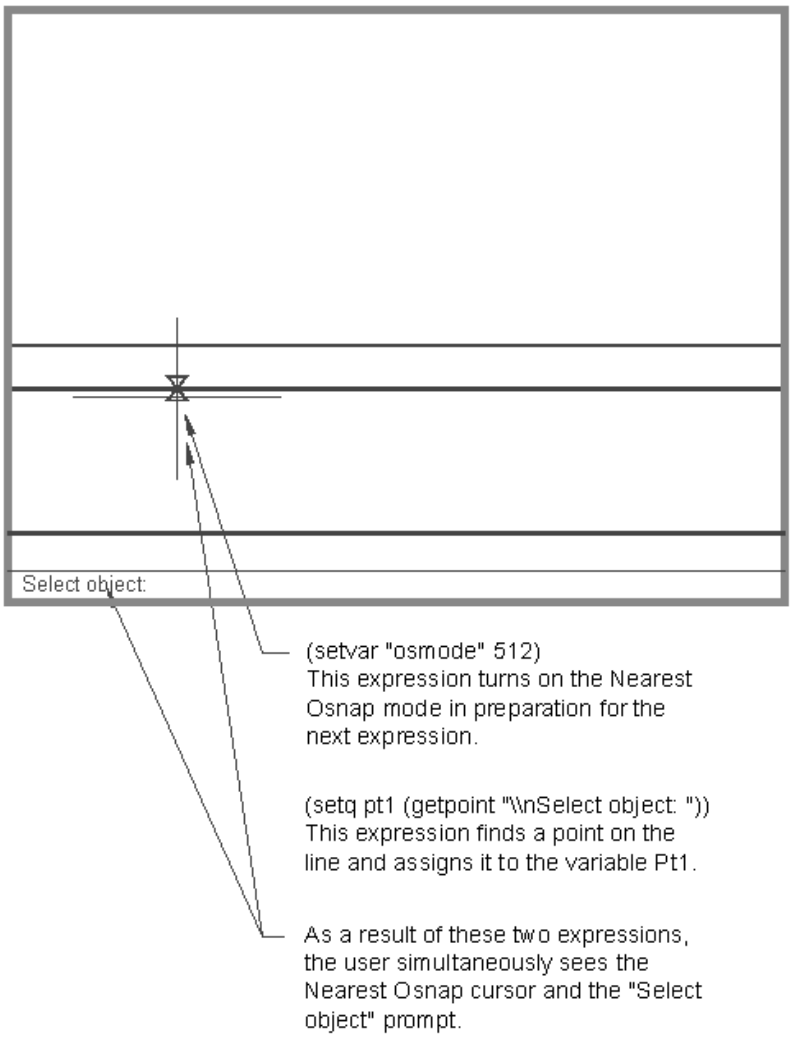


Figure 6.5: Getting a point using the "nearest" osnap mode.

Next, another prompt asks the user to pick another point:

(setq pt2 (getpoint pt1 "\nEnter second point: "))

The variable pt2 is assigned a point location for the other end of the break. The next line:

(setvar "osmode" 128)

sets the osnap mode to perpendicular in preparation for the next prompt:

(Setq pt3 (getpoint pt1 "\nSelect parallel line: "))

Here, the user is asked to select the line parallel to the first line. The user can pick any point along the parallel line and the perpendicular osnap mode ensures that the point picked on the parallel line is "perpendicular" to the first point stored by the variable pt1. The perpendicular osnap mode will only work, however, if a point argument is supplied to the Getpoint function. In this case, the point pt1 is supplied as a reference from which the perpendicular location is to be found (see figure 6.6). This new point variable pt3 will be important in calculating the location of the two break points on the parallel line.

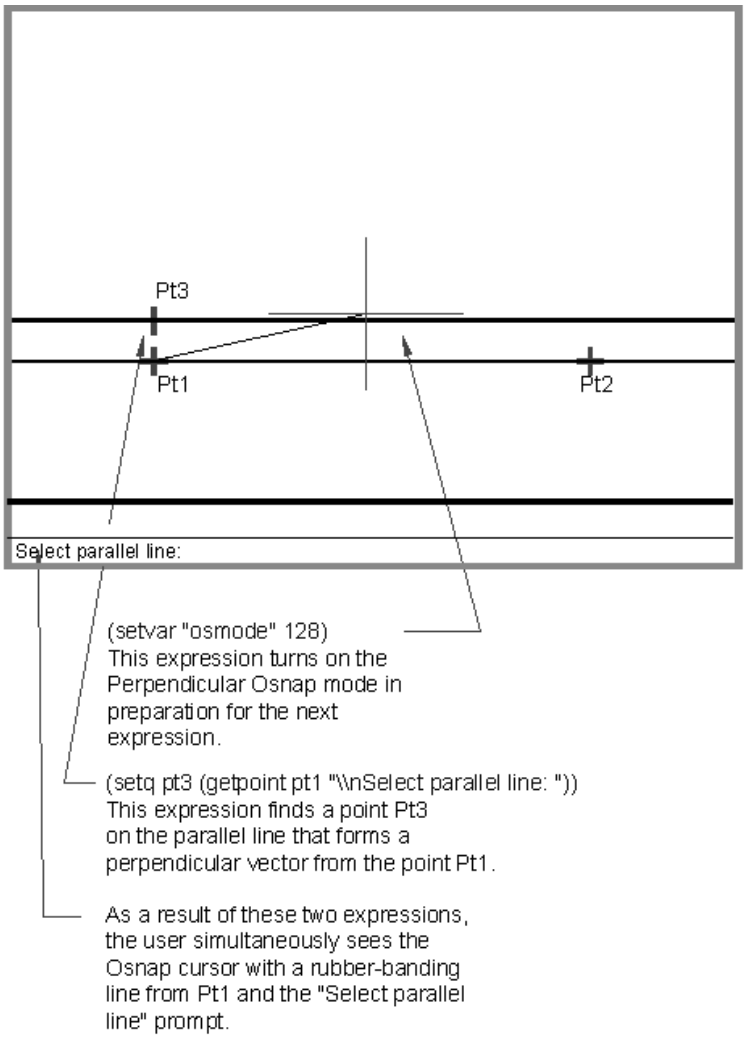


Figure 6.6: Picking the point perpendicular to the first point.

The next line sets the osnap mode back to "none":

(Setvar "osmode" 0)

The next two lines find the angle and distance described by the two point variables pt1 and pt3.

(setq ang1 (angle pt1 pt3))

(setq dst1 (distance pt1 pt3))

You can obtain the angle described by two points using the Angle function. Angles syntax is:

(angle [coordinate list][coordinate list])

The arguments to angle are always coordinate lists. The lists can either be variables or quoted lists.

Angle returns a value in radians. You looked at radians briefly in chapter 4. A radian is system to measure angles based on a circle of 1 unit radius. In such a circle, an angle can be described as a distance along the circle's circumference. You may recall from high school geometry that a circle's circumference is equal to 2 times pi times its radius. since the hypothetical circle has a radius of 1, we drop the one from the equation.

circumference = 2pi

90 degrees is equal to one quarter the circumference of the circle or pi/2 radians or 1.5708 (see figure 6.7). 180 degrees is equal to half the circumference of a circle or 1 pi radians or 3.14159. 360 degrees is equal the full circumference of the circle or to 2 pi radians or 6.28319.

A simple formula to convert degrees to radians is:

radians * 57.2958

To convert degrees to radians the formula is:

degrees * 0.0174533

Angle uses the current UCS orientation as its basis for determining angles. Though you can supply 3 dimensional point values to angle, the angle returned is based on a 2 dimensional projection of those points on the current UCS (see figure 6.7). Finally, radians are always measured with a counterclockwise directions being positive.

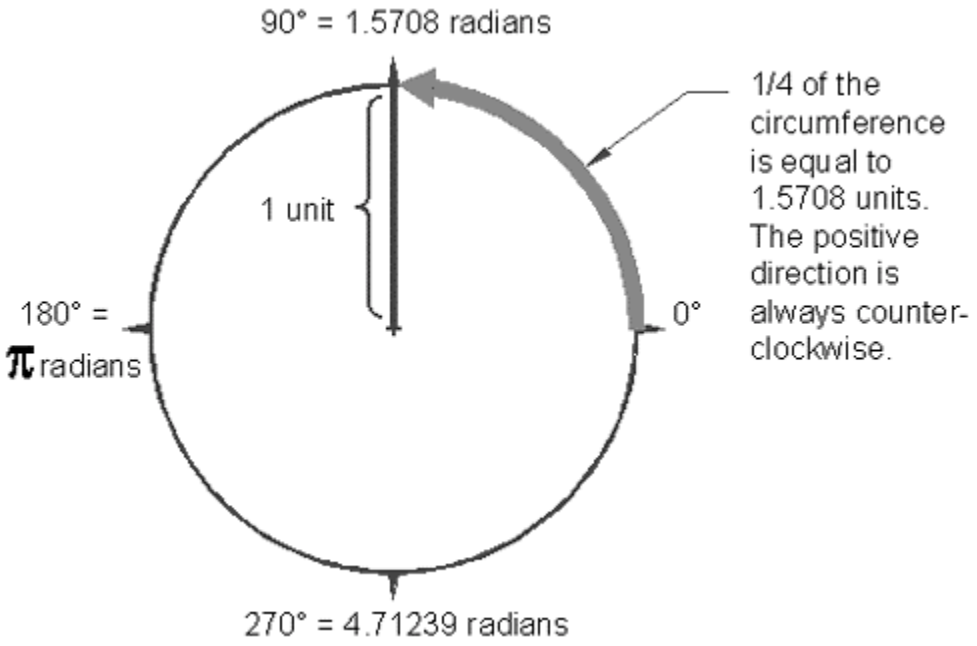


Figure 6.7: Degrees and radians

The distance function is similar to angle in that it requires two point lists for arguments:

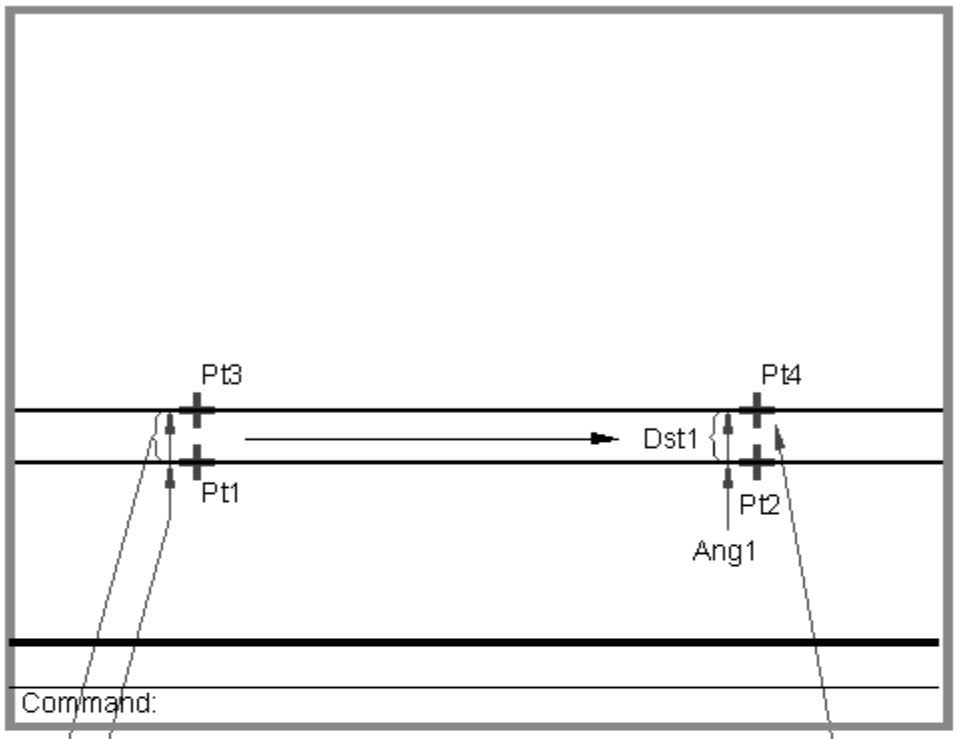
(distance [coordinate list][coordinate list])

The value returned by distance is in drawing units regardless of the current unit style setting. This means that if you have the units set to Architectural, distance will return distances in one inch units.

By storing the angle and distance values between pt1 and pt3, the program can now determine the location of the second break point on the parallel line. This is done by applying this angle and distance information to the second break point of the first line using the following expression:

(setq pt4 (polar pt2 ang1 dst1))

Here the angle variable ang1 and the distance variable dst1 are used as arguments to the polar function to find a point pt4. Polar returns a point value (see figure 6.8).



(setq ang1 (angle pt1 pt3))
This expression finds the angle
from Pt1 to Pt3 and assigns the
value to the variable Ang1.

(setq dst1 (distance pt1 pt3))
This expression finds the distance
between Pt1 and Pt3 and assigns
the value to the variable Dst1.

(setq pt4 (polar pt2 ang1 dst1))
This expression finds a point Pt4
by using the Polar function to
apply the angle Ang1 and
distance Dst1 to the point Pt2.

Figure 6.8: Using the polar function to find a new point.

The syntax for polar is:

(polar [point value][angle in radians][distance])

Polar is used to find relative point locations. It requires a point value as its first argument followed by an angle and distance value. The new point value is calculated by applying the angle and distance to the point value supplied. This is similar to describing a relative location in AutoCAD using the at sign. For example, to describe a relative location of .25 units at 90 degrees from the last point entered you would enter the following in AutoCAD:

@.25<90

The same relative location would look like the following using the polar function:

```
(setq pt1 (getvar "lastpoint"))

(polar pt1 1.5708 .25)
```

The first expression in this example uses the getvar function to obtain the last point selected. Setq then assigns that point to the variable pt1. Polar is used in the next line to find a point that is 1.5708 radians (45 degrees) and 2 units away from the lastpoint.

The last several lines of the Break2 program use all the point variables to first break the two lines then draw the joining lines between them.

```
(command

"break" pt1 pt2

"break" pt3 pt4

"line" pt1 pt3 ""

"line" pt2 pt4 ""

)

)
```

In the Break2 program, you could have used the getdist and getangle functions but to do so would mean including an additional prompt. By using the combination of the perpend osnap mode along with the getpoint function, you establishes a point value from which both the angle and distance value is derived. In general, if you know that you will need to gather both distance and angle information, it is better to establish coordinate variables first then derive angles and distances from those coordinates using the distance and angle functions.

Using Trigonometry to Solve a Problem

The Break2 function is relatively simple as far as its manipulation of data is concerned. But at times, you will need to enlist the aid of some basic trigonometric functions to solve problems. Suppose you want a function that will cut a

circle along an axis defined by the user. Figure 6.9 shows a sketch along with a written description of a program that does this.

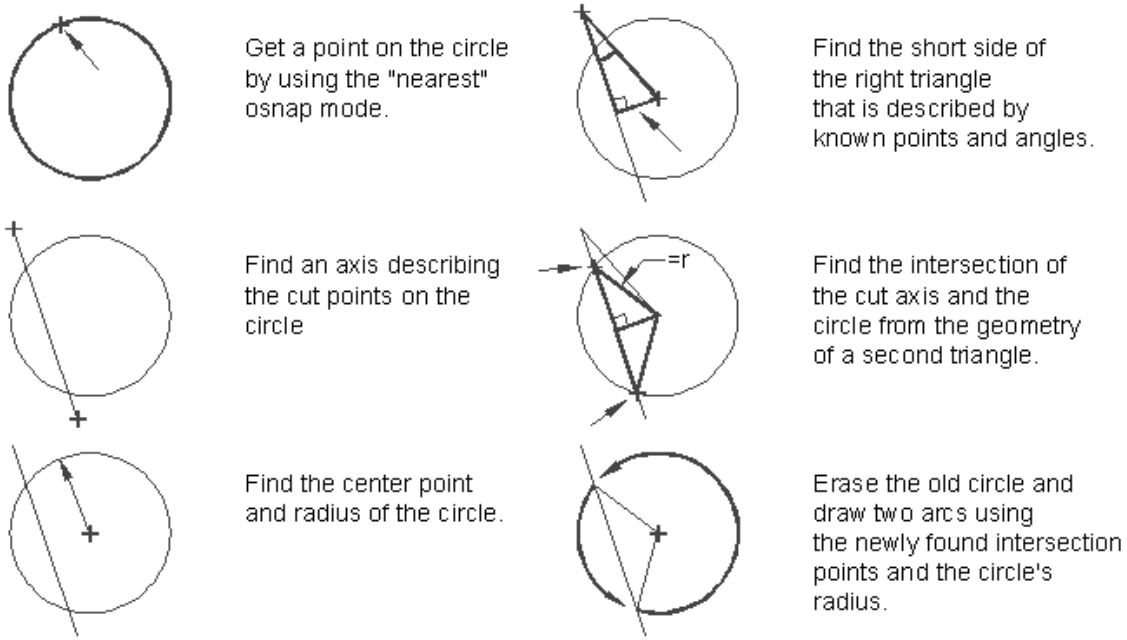


Figure 6.9: Sketch of circle cutting program.

This program makes use of the Pythagorean Theorem as well as the Sine trigonometric function as you will see in the next section.

Gathering Information

Before a problem can be solved, you must first gather all the known factors affecting the problem. The program you will explore next will give an example of how you might go about your information gathering.

Exit AutoCAD and open an AutoLISP file called Cutcr.lsp. Carefully copy the program in figure 6.10 into this file. Save and exit the Cutcr.lsp file then start AutoCAD and open the Chapt6 file again. Load the Ctcr.lsp fine then do the following:

1. Erase everything on the screen then draw a circle with its center at point 8,6 and a with a radius of 3 units (see figure 6.11).

2. Enter **cutcr** at the command prompt to start the C:CUTCR program.

3 At the prompt:

Pick circle to cut:

pick the circle you just drew.

4. At the next prompt:

Pick first point of cut line:

pick a point at coordinate 5,9.

5. At the prompt:

Pick second point:

pick a point at coordinate 8,2.

The circle is cut into two arcs along the axis represented by the two points you picked (see figure6.12).

6. Use the erase command to erase the left half of the circle. You can now see that the circle has been cut (see figure 6.13).

```
;Program to cut a circle into two arcs -- Cutcr.lsp
(defun C:CUTCR (/ cpt1 lpt1 lpt2 cent rad1 ang1
                  dst1 dst2 cord ang2 wkpt cpt2 cpt3)
  (setvar "osmode" 512)                              ;osnap to nearest
  (setq cpt1 (getpoint "\nPick circle to cut: "))    ;find point on circle
  (setvar "osmode" 0)                                ;osnap to none
  (setq lpt1 (getpoint "\nPick first point of cut line: ")) ;1st point of cut
  (setq lpt2 (getpoint lpt1 "\nPick second point: "))       ;2nd point of cut
  (setq cent (osnap cpt1 "center"))                  ;find center pt
  (setq rad1 (distance cpt1 cent))                   ;find radius of circle
  (setq ang1 (- (angle lpt1 cent)(angle lpt1 lpt2))) ;find difference of angles
  (setq dst1 (distance lpt1 cent))                   ;find dist.lpt1 to cent
  (setq dst2 (* dst1 (sin ang1)))                    ;find side of triangle
  (setq cord (sqrt(-(* rad1 rad1)(* dst2 dst2))))    ;find half cord
  (setq ang2 (- (angle lpt1 lpt2) 1.57))             ;find perpend angle
  (setq wkpt (polar cent ang2 dst2))                 ;find workpoint
  (setq cpt2 (polar wkpt (angle lpt1 lpt2) cord))    ;find first intersect
  (setq cpt3 (polar wkpt (angle lpt2 lpt1) cord))    ;find second intersect
  (command "erase" cpt1 ""                           ;erase circle
           "arc" "c" cent cpt2 cpt3                  ;draw first circle seg.
           "arc" "c" cent cpt3 cpt2                  ;draw second circle seg.
  )                                                  ;close command funct.
)                                                    ;close defun
```

Figure 6.10: The circle cut program

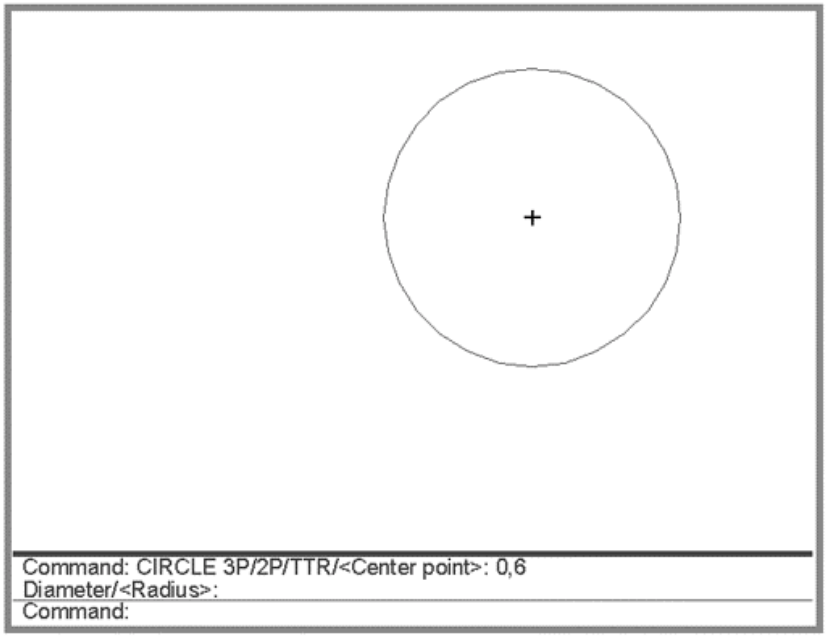


Figure 6.11: The circle drawn in AutoCAD

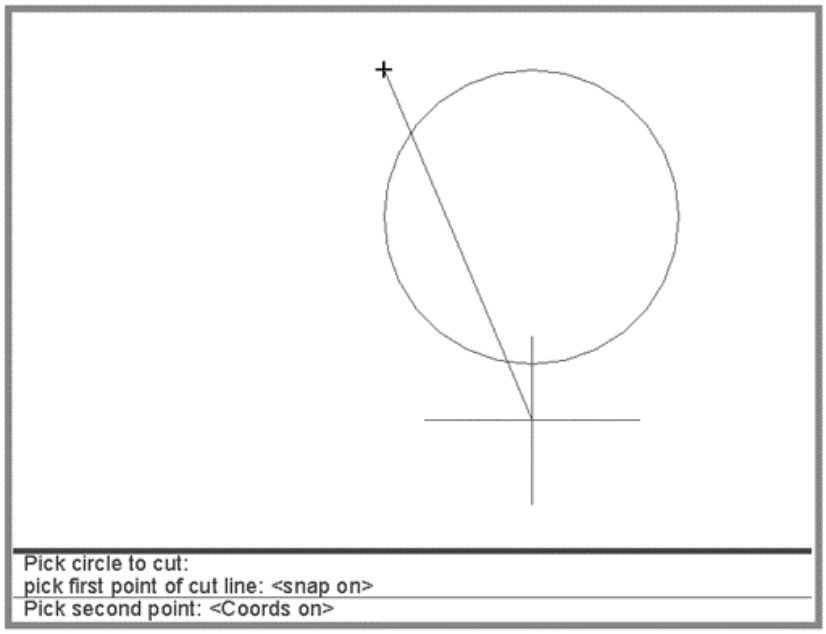


Figure 6.12: Drawing the cut axis.

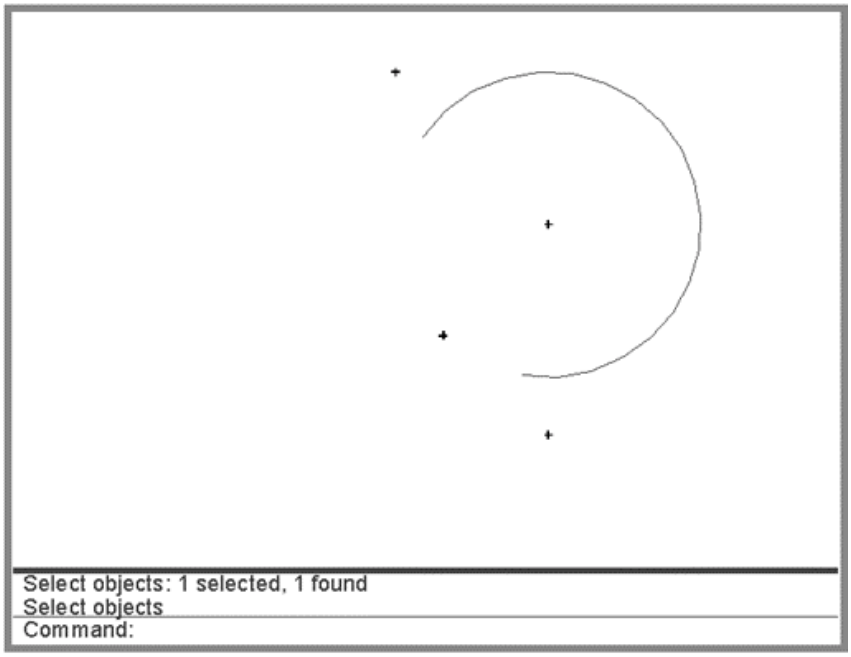


Figure 6.13: Erasing one part of the circle after it has been cut.

The first three expressions in the program after the defun functions and its arguments obtain a point on the circle:

(setvar "osmode" 512)

(setq cpt1 (getpoint "\nPick circle to cut: "))

(setvar "osmode" 0)

The setvar function sets osnap to the nearest mode then the user is prompted to pick the circle to cut. This point is stored as cpt1. The osnap mode ensures that the point picked is exactly on the circle. Later, this point will be used to both erase the circle and to find the circles center. The next function sets osnap back to none.

The next two lines prompt the user to select the points that define the axis along which the circle is to be cut:

(setq lpt1 (getpoint "\npick first point of cut line: "))

(setq lpt2 (getpoint lpt1 "\nPick second point: "))

The getpoint function is used in both these expressions to obtain the endpoints of the cut axis. These endpoints are stored as lpt1 and lpt2.

The next expression uses a new function called osnap:

(setq cent (osnap cpt1 "center"))

Here the point picked previously as the point on the circle cpt1 is used in conjunction with the osnap function to obtain the center point of the circle. The syntax for osnap is:

(osnap [point value][osnap mode])

The osnap function acts in the same way as the osnap overrides. If you use the center osnap override and pick a point on the circle, you get the center of the circle. Likewise, the osnap function takes a point value and applies an osnap mode to it to obtain a point. In this case, osnap applies the center override to the point cpt1 which is located on the circle. This gives us the center of the circle which is assigned to the symbol cent.

The next expression obtains the circle's radius

(setq rad1 (distance cpt1 cent))

The distance function is used to get the distance from cpt1, the point located on the circle, to cent, the center of the circle. This value is assigned to the symbol rad1.

Finding Points Using Trigonometry

At this point, we have all the known points we can obtain without utilizing some math. ultimately, we want to find the intersection point between the circle and the cut axis. By using the basic trigonometric functions, we can derive the relationship between the sides of triangle. In particular, we want to look for triangles that contain right angles. If we analyze the known elements to our problem, we can see that two triangles can be used to find one intersection on the circle (See figure 6.14)

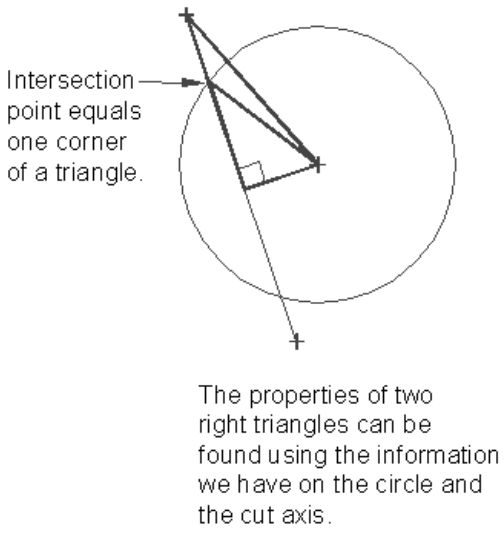


Figure 6.14: Triangles used to find the intersection

In our analysis, we see that we can find a point along the cut axis that describes the corner of a right triangle. To find this point, we only need an angle and the length of the hypotenuse of the triangle. We can then apply one of the basic trigonometric functions shown in figure 6.15 to our problem.

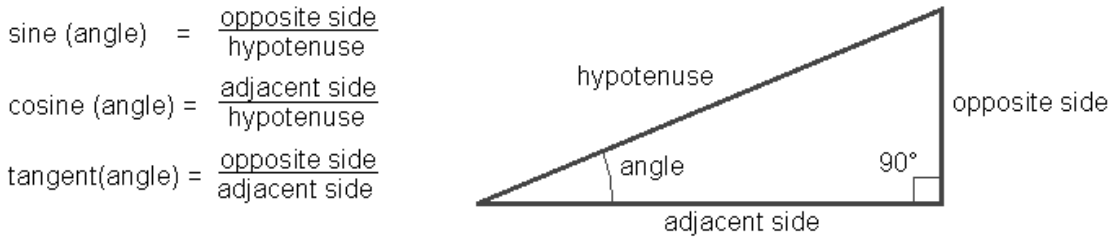


Figure 6.15: Basic trigonometric functions

The sine function is the best match to information we have.

sine(angle) = opposite side / hypotenuse

This formula has to be modified using some basic algebra to suite our needs:

opposite side = hypotenuse * sine (angle)

Before we can use the sine function, we need to find the angle formed by points lpt1, lpt2 and cent (see figure 6.16).

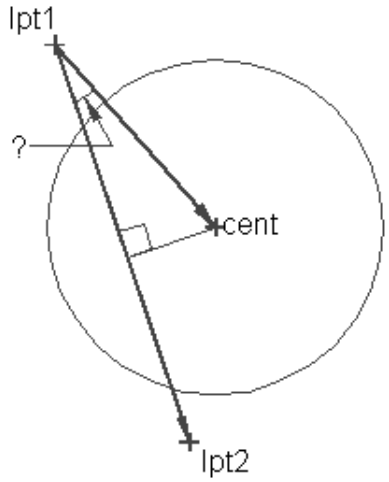


Figure 6.16: Angle needed to before the sine function can be used.

The following function does this for us:

(setq ang1 (- (angle lpt1 cent) (angle lpt1 lpt2)))

The first of these three functions finds the angle described by points lpt1 and lpt1. The second expression finds the angle from lpt1 to the center of the circle. The third line finds the difference between these two angles to give us the angle of the triangle we need (see figure 6.17).

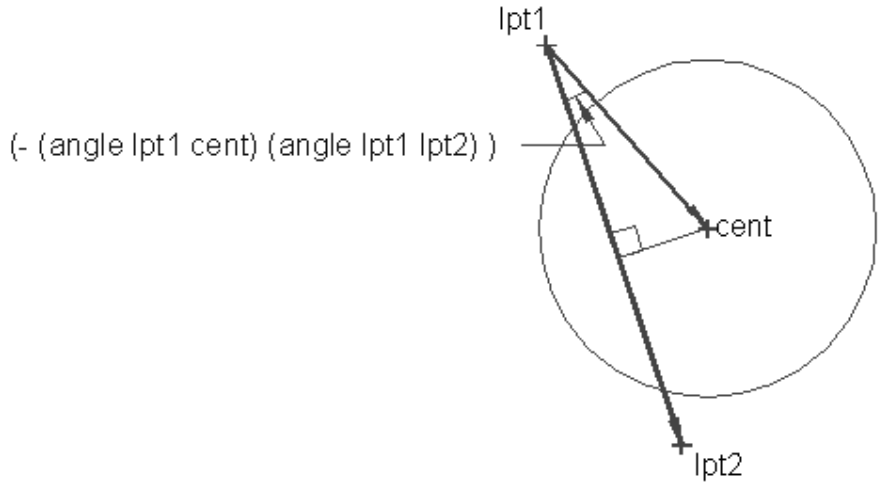


Figure 6.17: The sine expression written to accommodate the known elements.

We also need the length of the hypotenuse of the triangle. this can be gotten by finding the distance between lpt1 and the center of the triangle:

(setq dst1 (distance lpt1 cent))

The length of the hypotenuse is saved as the variable dst1. We can now apply our angle and hypotenuse to the formula:

opposite side = hypotenuse * sine (angle) becomes

(setq dst2 (* dst1 (sin ang1)))

Now we have the length of the side of the triangle but we need to know the direction of that side in order to find the corner point of the triangle. We already know that the direction is at a right angle to the cut axis. therefore, we can determine the right angle to the cut axis by adding the cut axis angle to 1.57 radians (90 degrees)(see figure 6.18). The following expression does this for us:

(setq ang2 (- (angle lpt1 lpt2) 1.57))

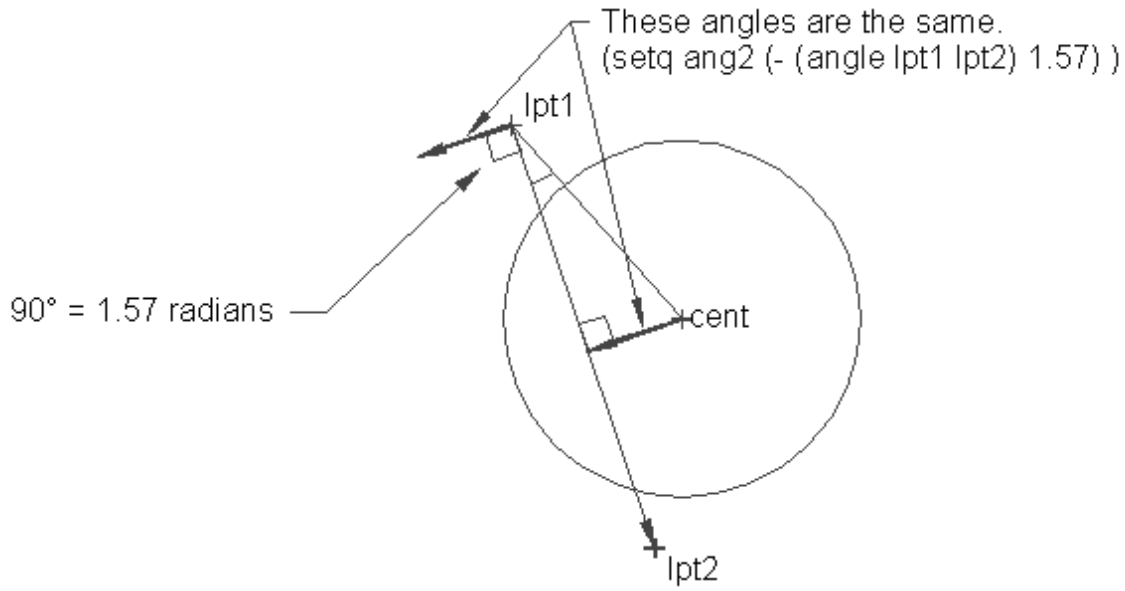


Figure 6.18: Finding the angle of the opposite side of the triangle.

We are now able to place the missing corner of the triangle using the polar function (see figure 6.19).

(setq wkpt (polar cent ang2 dst2))

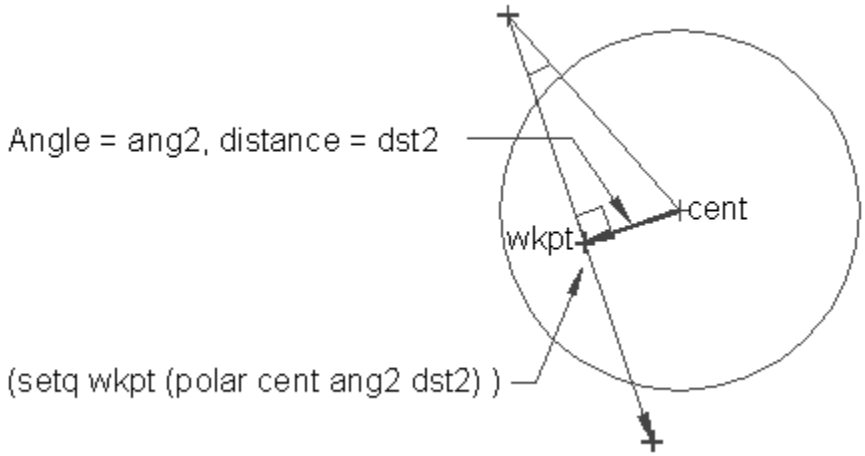


Figure 6.19: Finding the workpoint wkpt.

We assign the corner point location to the variable wkpt. We're still not finished with our little math exercise, however. We still need to find the intersection of the cut axis to the circle. Looking at our problem solving sketch, we can see that yet another triangle can be used to solve our problem. We know that the intersection lies along the cut axis. We can describe a triangle whose corner is defined by the intersection of the circle and the cut axis (see figure 6.20).

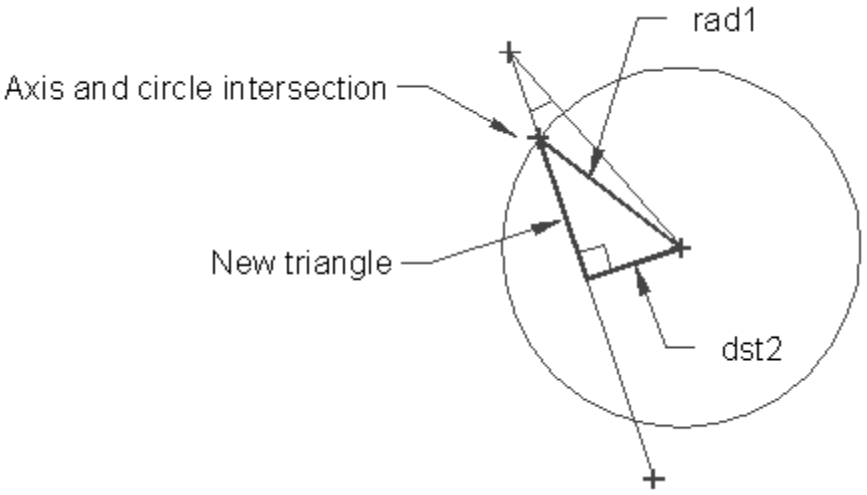


Figure 6.20: The triangle describing one intersection point.

We also already know two of the sides of this new triangle. One is the radius of the circle stored as the variable rad1. The other is the side of the triangle we used earlier stored as the variable dst2. The most direct way to find the intersection is to apply the Pythagorean Theorem shown in figure 6.21.

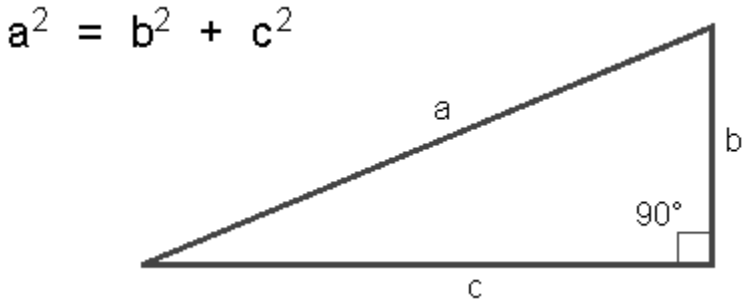


Figure 6.21: The Pythagorean Theorem

Again we must apply algebra to derive a formula to suite our needs. The formula:

$$c^2 = a^2 - b^2$$

becomes the expression:

(setq cord (sqrt(-(* rad1 rad1)(* dst2 dst2))))

We assign the distance value gotten from the Pythagorean Theorem to the variable cord. Using the Polar function, we can now find one intersection point between the circle and the cut axis:

(setq cpt2 (polar wkpt (angle lpt2 lpt1) cord))

In this expression, we find one intersection by applying the angle described by lpt1 and lpt2 and the distance described by cord to the polar functions (see figure 6.22).

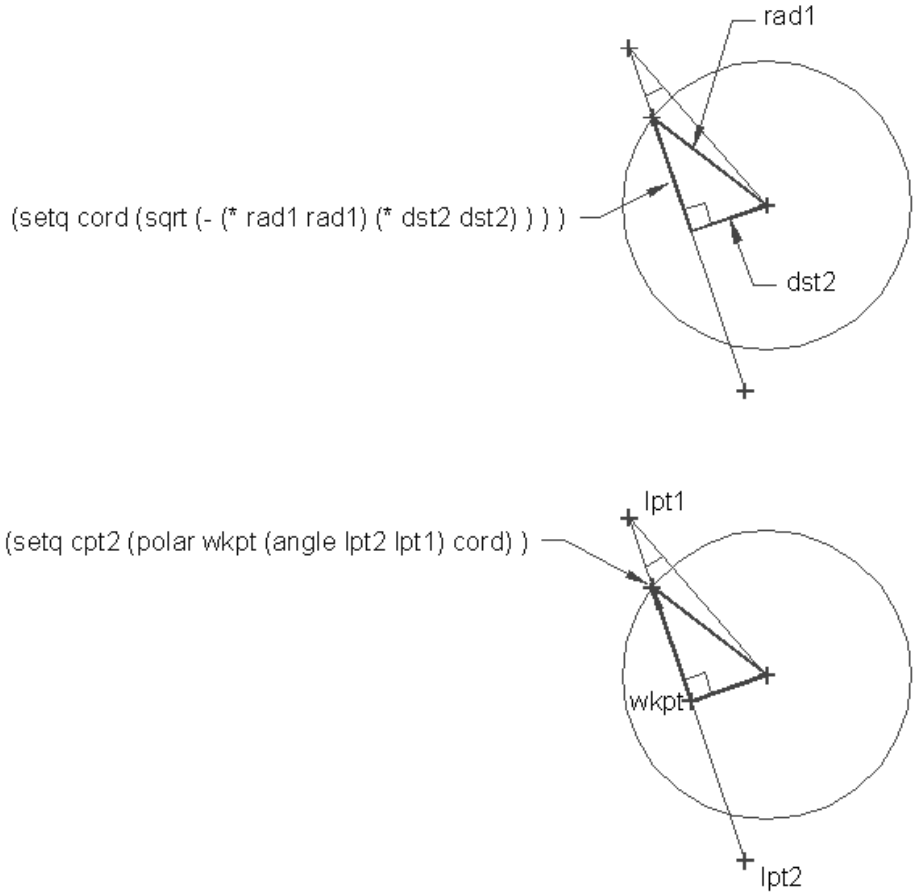


Figure 6.22: Finding the location of an intersection point.

Since the two intersection points are symmetric about point wkpt, the second intersection point is found by reversing the direction of the angle in previous expression:

(setq cpt3 (polar wkpt (angle lpt1 lpt2) cord))

Finally, we can get AutoCAD to do the actual work of cutting the circle:

(command "erase" cpt1 ""

"arc" "c" cent cpt2 cpt3

"arc" "c" cent cpt3 cpt2

)

)

Actually, we don't really cut the circle. Instead, the circle is erased entirely and replaced with two arcs (see figure 6.23).

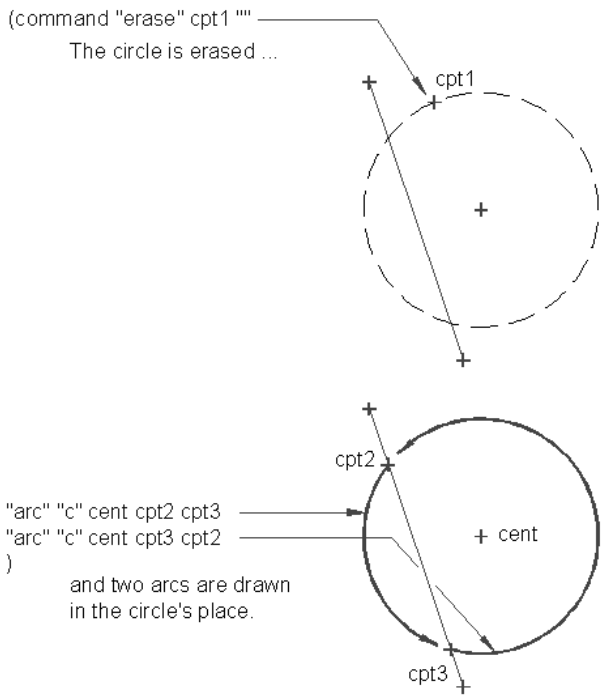


Figure 6.23: Drawing the new circle.

Functions Useful in Geometric Transformations

The majority of your graphic problems can be solved with the basic trigonometric functons used in the Cutcr program. But AutoLISP provides the tools to solve even th emost arcane trigonometric problems. This section shows the functions you are most likely to use in situations that require geometric manipulations.

Trans

Trans translates a coordinate or displacement from one user coordinate system to another. The first argument is a point of reference. The second argument is a code indicating which coordinate system the point is expressed in. The third argument is a code indicating which coordinate system the point is to be translated to. An optional fourth True/nil argument can be included. If this fourth argument evaluates to True or non-nil, then the first argument will be treated as a displacement rather than a point value. The following are the codes for the second and third arguments.

Code	Coordinate System
0	World Coordinate System
1	Current User Coordinate System
2	Coordinate system of the current view plane

Trans returns a coordinate or displacement list.

The syntax for Trans is

(trans [coordinage_list] [UCS_code] [optionalT/nil])

Atan

Atan returns the arctangent in radians of its first argument. If the argument is negative, then the value returned will be negative. If two arguments are supplied, then Atan returns a the arctangent of the first argument divided by the second argument.

The syntax for Atan is

(atan [number] [optional_2nd_number])

Inters

Inters returns a coordinate list of the intersection of two vectors. The first two arguments to inters are the endpoints of one vector while the third and fourth arguments define the other vector. If an

optional fifth agrument is present and evaluates to nil, then Inters will attempt to locate an intersection point of the two vectors regardless of whether the intersection falls between the specified points or not.

The syntax for Inters is

(inters [point point point point] [optional_T/nil])

Sin

Sine returns the sine of an angle as a real number. The angle must be expressed in radians.

The syntax for Sine is

(sin [angle])

Cos

Cos returns the Cosine of an angle as a real number. The angle must be expressed in radians.

The syntax for Cos is

(cos [angle])

Conclusion

The majority of your graphic problems can be solved using the basic trigonometric functions shown in this sample program. But AutoLISP provides the tools to solve even the most arcane Trigonometric problems. If you find you need to use these math trig functions, consider making liberal use of a sketch pad or for that matter, AutoCAD itself to document your program. You may want to re-use or modify programs such as the previous example and if you don't have some graphic documentation recording how it works, you can have a difficult time understanding why you wrote your program as you did.

Chapter 7: Working with Text

Introduction

If you have ever had to edit large amounts of text in a drawing, you have encountered one of AutoCAD's most frustrating limitations. While AutoCAD's text handling capabilities is one of it's strong points, it still leaves much to be desired where editing text is concerned. Fortunately AutoLISP can be of great help where text is concerned. In this chapter, you will look at the many functions that AutoLISP offers in the way of text or string manipulation. You will also look at how textural information can be store and retrieved from a file on disk and how data can be converted to and from string data types.

Working With String Data Types

In earlier versions of AutoCAD, editing text was a tedious task. You have to use the Change command to select a text line, then press return several times before you can actually make changes to the text. Even then, you would have to re-enter the entire line of text just to change one word. The text editing features of AutoCAD have come a long way and it doesn't take the same painful effort it once did. The following program is a simple line editor which simplifies the task of editing a single line of text. It was designed for the older versions of AutoCAD before the Ddmodify and Ddedit commands were available. While its function may be a bit outdated, it will serve to demonstrate how to handle text in AutoLISP.

Searching for Strings

The program Chtxt shown in figure 7.1 is a simple line editor. It uses AutoLISPs string handling functions to locate a specific string, then it replaces that string with another one specified by the user.

Open a file called chtxt.lsp and copy the program shown in figure 7.1 into the file. Save and close this file and open a new AutoCAD file called Chapt7.

```
;function to find text string from text entity------------------------------
(defun gettxt ()
(setvar "osmode" 64)                                ;set osnap to insert
(setq pt1 (getpoint "\nPick text to edit: "))       ;get point on text
(Setvar "osmode" 0)                                 ;set osnap back to zero
(setq oldobj (entget (ssname (ssget pt1) 0) ))      ;get entity zero from prop.
(setq txtstr (assoc 1 oldobj))                      ;get list containing string
(cdr txtstr)                                        ;extract string from prop.
)
;function to update text string of text entity------------------------------
(defun revtxt ()
(setq newtxt (cons 1 newtxt))                       ;create replacement propty.
(entmod (subst newtxt txtstr oldobj))               ;update database
)
;program to edit single line of text----------------------------------------
(defun C:CHTXT (/ count oldstr newstr osleng otleng oldt old1
                  old2 newtxt pt1 oldobj txtstr oldtxt)
(setq count 0)                                      ;setup counter to zero
(setq oldtxt (gettxt))                              ;get old string from text
(setq otleng (strlen oldtxt))                       ;find length of old string
(setq oldstr (getstring T "\nEnter old string "))   ;get string to change
(Setq newstr (getstring T "\nEnter new string "))   ;get replacement string
(setq osleng (strlen oldstr))                       ;find length of substring-
  ;while string to replace is not found, do...       to be replaced
  (while (and (/= oldstr oldt)(<= count otleng))
    (setq count (1+ count))                         ;add 1 to counter
    (setq oldt (substr oldtxt count osleng))        ;get substring to compare
  );end WHILE
  ;if counting stops before end of old string is reached...
  (if (<= count otleng)
        (progn
           (setq old1 (substr oldtxt 1 (1- count))) ;get 1st half of old string
           (setq old2 (substr oldtxt (+ count osleng) otleng));get 2nd half
           (setq newtxt (strcat old1 newstr old2))  ;combine to make new string
           (revtxt)                                 ;update drawing
        )
        (princ "\nNo matching string found.")       ;else print message
  );end IF
(PRINC)
);END C:EDTXT
```

Figure 7.1: The Chtxt.lsp file

Load the Chtxt.lsp file and do the following steps:

1. Use the Dtext command and write the following line of text:

   For want of a battle, the kingdom was lost.

2. Enter Chtxt

3. At the prompt:

   Pick text to edit:

   pick the text you just entered. Note that the osnap cursor appears.

4. At the next prompt:

   Enter old string:

   enter the word battle in lower case letters.

5. At the next prompt:

   Enter new string:

   enter the word nail.

   The text changes to read:

   For want of a nail, the kingdom was lost.

In the C:CHTXT program, you are able to change a group of letters in a line of text without having to enter entire line over again. Also, you don't have to go through a series of unneeded prompts as you do with the change command. The following describes what C:CHTXT goes through to accomplish this.

The C:CHTXT program starts out with a user defined function called gettxt.

```
(defun C:EDTXT (/ count oldstr newstr osleng otleng oldt old1
old2 newtxt pt1 oldobj txtstr oldtxt)
(setq count 0)
(setq oldtxt (gettxt))
```

Gettxt prompts you to select the text to be edited. It then extracts from the drawing database the text string associated with that text. The extracted text is assigned to the symbol oldtxt. We will look at this extraction process in Chapter--- but for now, think of the gettxt function as a function for getting text.

In the next line of the Chtxt program, we see a new function strlen:

(setq otleng (strlen oldtxt))

Strlen finds the number of characters in a string.

The syntax for strlen is:

(strlen [string or string variable])

Strlen returns an integer value representing the number of characters found in its arguments. Blank spaces are counted as characters. In the above expression, the value found by strlen is assigned to the variable otleng.

The next two expressions obtain from the user the old portion of the text to be replace and the replacement text.

(setq oldstr (getstring T "\nEnter old string "))

(Setq newstr (getstring T "\nEnter new string "))

These two strings are saved as oldstr and newstr. Note that the T argument is used with getstring to allow the user to use spaces in the string.

The next set of expressions do the work of the program. First, The number of characters of the string to be replaced, oldstr, is found by using the strlen function:

(setq osleng (strlen oldstr))

This value is stored with a variable called osleng. Osleng will be used to find exactly where in the line of text the old string occurs in the line of text being edited (see figure 7.2).

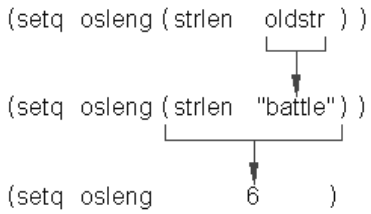


Figure 7.2: Using the strlen function

The following while function uses osleng\ to find the exact location of oldstr within oldtxt.

(while (and (/= oldstr oldt)(<= count otleng))

(setq count (1+ count))

(setq oldt (substr oldtxt count osleng))

);end WHILE

The while expression tests for two conditions. The first test is to see of the current string being read matches the string entered at the "String to be changed" prompt. The second test checks to see if the end of the text line has been reached. The **and** logical operator is used to make sure that both test conditions are met before it continues evaluating its other expressions.

We see a new function substr in this group of expressions:

(Setq oldt (substr oldtxt count osleng))

Substr extracts a sequence of characters from a string. Its syntax is:

(substr

[string or string variable]

[beginning of substring][end of substring]

)

The first argument to substr is the string within which a substring is to be extracted. By substring, we mean a group of characters that are contained within the main string. The substring can be any contiguous sequence of characters within the main string including the entire string itself. The second argument is the beginning location for the substring. This can be an integer from 1 to the total number of characters in the main string. The third argument is the ending location of the substring. This value is an integer greater than or equal to the value for the beginning of the substring and it determines the ending location of the substring (see figure 7.3).

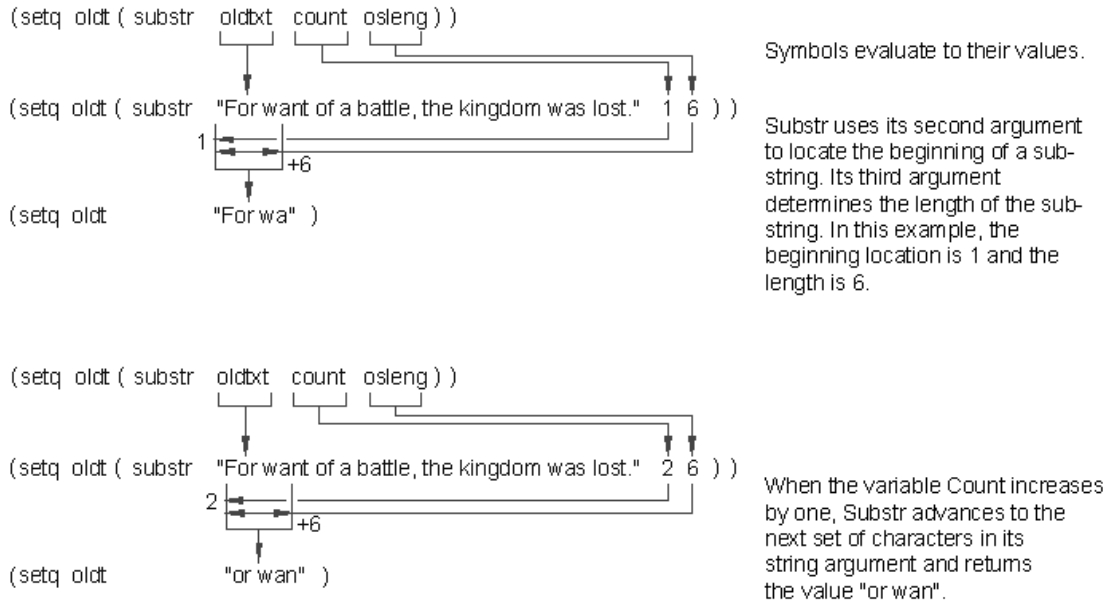
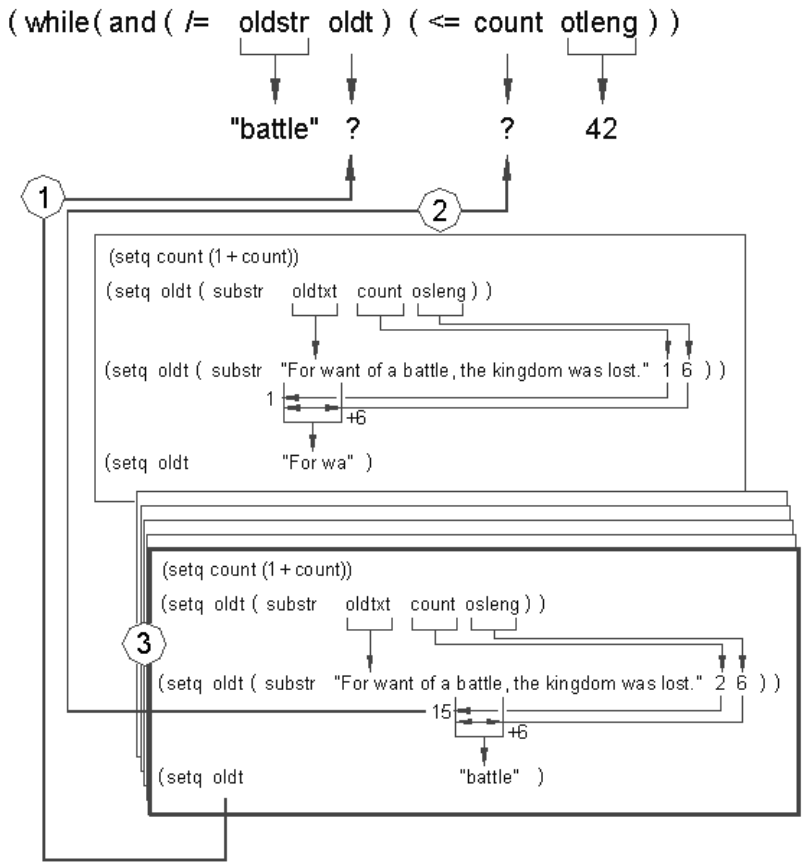


Figure 7.3: Using the substr function

The while expression extracts a group of characters from oldtxt starting at the beginning. It stores this substring in the variable oldt. Oldt is then compared with oldstr to see if they match. If the don't match, then while advances to the next group of characters in oldtxt and compares this new group to oldstr. This goes on until a match is found or the end of oldtxt is reached (See figure 7.4).



1. The While expression checks to see if the variable Oldt is equal to the string "battle". If they are equal, the While expression stops running.
2. The While expression also checks to see if the variable Count is equal to the variable Otleng. If Count is greater than Otleng, the While expression terminates.
3. The While expression continues to evaluate its set of expressions until one of the conditions is met.

Figure 7.4: Using the while expression to find a matching string.

String data types are case sensitive, This means that if you had entered "BATTLE" instead of "battle" at the string to change prompt, you would have gotten the message:

No matching string found.

The string "BATTLE" is not equal to "battle" so as edtxt tries to find a string that matches "BATTLE", it never finds it.

When the while expression is done, the next group of expressions takes the old text line apart and replaces the old string with the new. First the if function is used to test whether the oldstring was indeed found.

(if (<= count otleng)

The if expression checks to see if the variable count is equal to or less than the length of the old text line. If count is less than otleng, then the following set of expressions are evaluated:

(progn

(setq old1 (substr oldtxt 1 (1- count)))

(setq old2 (substr oldtxt (+ count osleng) otleng))

(setq newtxt (strcat old1 newstr old2))

(revtxt)

)

The progn function allows the group of expressions that follow to appear as one expression to the **if** function. The first expression of this group:

(setq old1 (substr oldtxt 1 (1- count)))

separates out the first part of the old text line just before the old string. This is done using the substr function and the count variable to find the beginning of the old string (see figure 7.5).

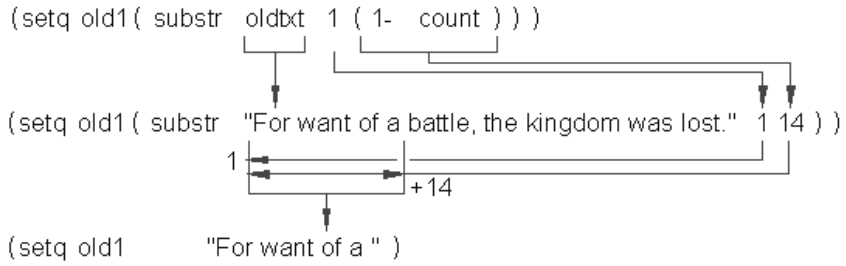


Figure 7.5: Finding the string before oldstr

The next expression:

(setq old2 (substr oldtxt (+ count osleng) otleng))

separates out the last part of the old text line starting just after the old string. Again this is done using the substr function and the count variable. This time, count is added to osleng to find the location of the end of the old string. Otleng is used for the substring length. Even though its value is greater than the length of the substring w wan, AutoLISP will read the substring to the end of Oldtxt (see figure 7.6).

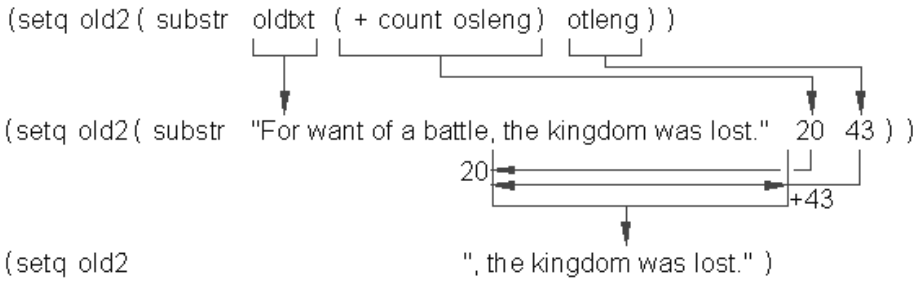


Figure 7.6: Finding the string after oldstr

Finally, the expression:

(setq newtxt (strcat old1 newstr old2))

combines the first and last part of the old text line with the new string to form the replacement text line (see figure 7.7).

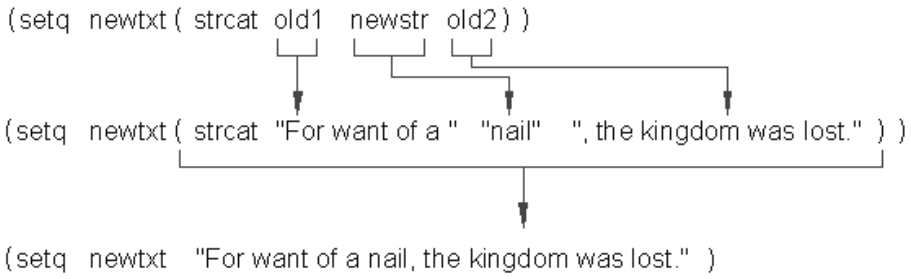


Figure 7.7: Combining the old string with the new

The last expression in this group:

(revtxt)

is a user defined function that does the work of replacing the old text with the new.

```
(defun revtxt ()

(setq newtxt (cons 1 newtxt))

(entmod (subst newtxt txtstr oldobj))

)
```

In the event that count is greater than otleng, the following expression is evaluated:

```
(princ "\nNo matching string found.")

);end IF
```

This expression prints the message:

No matching string found.

to the prompt line.

The very last expression of the program:

```
(PRINC)

);END C:EDTXT
```

seems pretty useless at first glance. Princ without any arguments prints a blank to the prompt line. If this expression were not here, however, AutoLISP would display the value of the last expression evaluated. Remember that AutoLISP constantly cycles through the read-evaluate-print loop. The generally, the value of the last expression evaluated is printed to the prompt line. While this doesn't affect the workings of the program, it may prove to be an annoyance to the user or it may confuse someone not familiar with the program. Since princ will print a blank at the prompt line when no arguments are supplied, it is often used without arguments at the end of a program simply to keep the appearance of the program clean. If you like, try deleting the Princ expression from the program and reload and run the program again. You will a value will appear in the prompt line when C:EDTXT finishes running.

How to Convert Numbers to String and Back

There are times when it is necessary to convert a string value to a number or vice versa. Suppose, for example, that you want to be able to control the spacing of numbers generated by the program in chapter 5. You may recall that this program creates a sequence of numbers equally spaced. The user is able to determine the beginning and ending numbers and the location of the beginning number but cannot determine the distance between numbers. You can use the rtos function to help obtain a distance value and include it with the program. Figure 7.8 shows the C:SEQ program from chapter 5 modified to accept distance input.

```
(defun C:SEQ (/ pt1 currnt last)
(setq pt1    (getpoint "\nPick start point: "))
(setq spc    (getdist pt1 "\nEnter number spacing: "))
(setq currnt (getint "\nEnter first number: "))
(setq last   (getint "\nEnter last number: "))
(setq stspc  (rtos spc 2 2))
(setq stspc  (strcat "@" stspc "<0" ))
(command "text" pt1 "" "" currnt)
  (repeat (- last currnt)
    (setq currnt (1+ currnt))
    (command "text" stspc "" "" currnt)
  )
)
```

Figure 7.8: The sequential number program

Converting a Number to a String

Exit AutoCAD and open the AutoLISP file seq.lsp. Make the changes shown in bold face type in figure 7.10. Save and exit the seq.lsp file and return to the file chapt7. Load the C:SEQ program and do the following:

1. Enter **seq** at the command prompt.

2. At the prompt:

   Pick start point:

   pick a point at coordinate 1,3.

3. At the prompt:

   Enter spacing:

   enter .5.

4. At the prompt:

   Enter first number:

   enter 4.

5. At the last prompt:

Enter last number:

enter **12**.

The numbers 4 through 12 will appear beginning at your selected start point and spaced at 0.5 unit intervals.

The program starts by prompting the user to pick a starting point:

(defun C:SEQ (/ pt1 pt2 currnt last)

(setq pt1 (getpoint "\nPick start point: "))

A new prompt is added that obtains the spacing for the numbers:

(setq spc (getdist pt2 "\nEnter number spacing: "))

The spacing is saved as the symbol spc. The program continues by prompting the user to enter starting and ending value:

(setq currnt (getint "\nEnter first number: "))

(setq last (getint "\nEnter last number: "))

Next, the function rtos is used to convert the value of spc to a string:

(setq stspc (rtos spc 2 2))

the syntax for rtos is:

(rtos [real or integer value][unit style code][precision])

The first argument to rtos is the number being converted. It can be a real or integer. The next argument is the unit style code. Table shows a listing of these codes and their meaning.

Code	Format
1	Scientific
2	Decimal
3	Feet and decimal inches
4	Feet and inches
5	Fractional units

This code determines the style the number will be converted to. For example, if you want a number to be converted to feet and inches, you would use the code 4. The third argument, precision, determines to how many decimal places to convert. In our example, we use the code 2 for unit style and 2 for the number of decimal places to convert to a string.

The next expression combines the converted number with the strings "@" and "<0" to form a string that can be used in with the command function:

(setq stspc (strcat "@" stspc "<0"))

The next two expressions set up the location of the beginning of the text:

(command "text" pt1 "" "" currnt)

This is done because in the next set of expressions, The string that locates the text, "@distance<0", gives a distance and direction rather than a point. The previous expressions locate a point which will cause the Text command in the next expression to place the text in the proper place:

(repeat (- last currnt)

(setq currnt (1+ currnt))

(command "text" stspc "" "" currnt)

)

)

In the last three expressions, the repeat function is used to issue the text command, enter the number and advance to the next number repeatedly until the last number is in place (see figure 7.9).

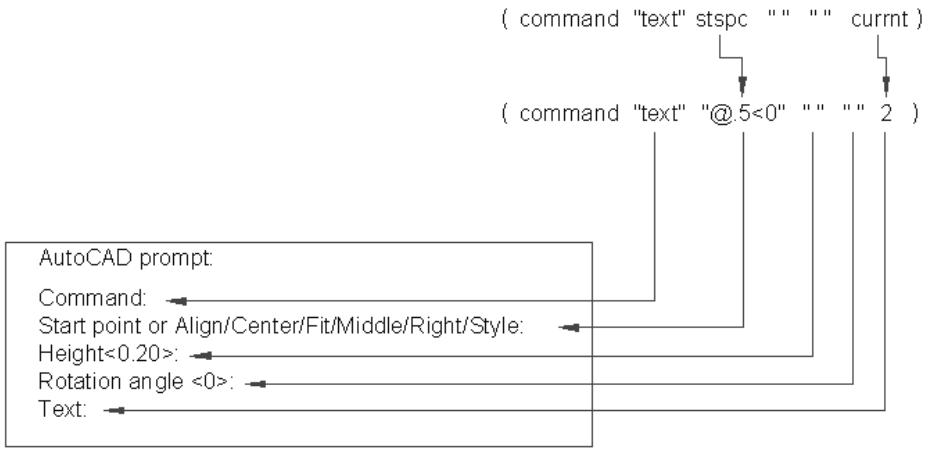


Figure 7.9: Writing the numbers into the AutoCAD file

Converting Other Data Types

Before we continue, we should briefly look at several other functions that offer data type conversion. These functions are listed in table .

Function	Uses
angtos	Converts real numbers (radians) into string values.
ascii	Converts a string into its ASCII character code.
atoi	Converts a string a string into an integer.
itoa	Converts an integer to a string.
chr	Converts an integer representing an ASCII character code into a string.

Angtos works in a similar way to rtos. It accepts a unit style code and a precision value. Its syntax is:

(angtos [angle value][unit style code][precision])

All of the other functions listed in table take a single item, the value to be converted, as their argument. For example, to convert an integer into a string, you could use the following expression:

(itoa 55)

The resulting value is "55".

The functions ascii and chr convert ASCII character codes. These are numeric values that represent letters, numbers and symbols. Figure 7.10 shows these codes and their meaning.

Code	Meaning	Code	Meaning	Code	Meaning	Code	Meaning	Code	Meaning	Code	Meaning
07	Beep	46	.	65	A	84	T	103	g	122	z
09	Tab	47	/	66	B	85	U	104	h	123	{
10	New line	48	0	67	C	86	V	105	i	124	|
13	Return	49	1	68	D	87	W	106	j	125	}
27	Escape	50	2	69	E	88	X	107	k	126	~
32	Space	51	3	70	F	89	Y	108	l		
33	!	52	4	71	G	90	Z	109	m		
34	"	53	5	72	H	91	[	110	n		
35	#	54	6	73	I	92	\	111	o		
36	$	55	7	74	J	93	]	112	p		
37	%	56	8	75	K	94	^	113	q		
38	&	57	9	76	L	95	_	114	r		
39	'	58	:	77	M	96	`	115	s		
40	(	59	;	78	N	97	a	116	t		
41	)	60	<	79	O	98	b	117	u		
42	*	61	=	80	P	99	c	118	v		
43	+	62	>	81	Q	100	d	119	w		
44	,	63	?	82	R	101	e	120	x		
45	-	64	@	83	S	102	f	121	y		

Figure 7.10: The ASCII character codes

How to read ASCII text files

There are many reasons why you may want to have a program read from and write to an ASCII file. You may store commonly used general notes in ASCII files on your hard disk which you would import into your drawing. Or you may want to store drawing information on disk for later retrieval such as layering setup or block lists.

Using a File Import Program

The program shown in figure 7.11 is a rudimentary text import program. In this section, you will use it to examine the way AutoLISP reads external ASCII files.

```
(Defun C:IMPRT (/ sp dt stl qst)
(setq nme (getstring "\nName of text file to import: "))
(setq sp  (getpoint "\nText starting point: "))
(setq txt (open nme "r"))
(setq dt  (read-line txt))
(setq lns (getdist "\nEnter line spacing in drawing units: "))
(setq ls  (rtos lns 2 2))
(setq ls  (strcat "@" ls "<-90"))
(command "text" sp "" "" dt)
   (while (/= dt nil)
      (setq dt (read-line txt))
      (command "text" ls "" "" dt)
   )
(close txt)
(command "redraw")
)
```

Figure 7.11: A text import program

Create an ASCII file containing the program in figure 7.13. Give it the name Imprt.lsp. Go back to the Chapt7 AutoCAD file and load the Imprt.lsp file. Now run the program:

1. Enter **imprt** at the Command prompt.

2. At the prompt:

   Name of text file to import:

   Enter **imprt.lsp**.

3. At the next prompt:

Text starting point:

pick a point at coordinate 2,8.

4. At the next prompt:

Enter line spacing in drawing units:

enter **.4**.

The contents of the file Imprt.lsp will be written into the drawing area using AutoCAD text.

The C:IMPRT program starts out by prompting the user to identify the file to be imported. The user is prompted to enter the name of the file to be imported:

```
(Defun C:IMPRT (/ sp dt stl qst)

(setq nme (getstring "\nName of text file to import: "))
```

The entered name is saved as the variable nme. Next, the starting point is gotten:

```
(setq sp (getpoint "\nText starting point: "))
```

This point is saved as sp. The last prompt sets up the line spacing:

```
(setq lns (getdist sp "\nEnter line spacing in drawing units: "))
```

The spacing distance is assigned to the variable lns. Note that getdist is used so the user can input a distance either through the keyboard or cursor.

In the next line, the AutoLISP function open is used to open the file to be imported:

```
(setq txt (open nme "r"))
```

In this expression, you are telling AutoLISP to open a file to be read, then assign that file to the variable txt. From this point on, you can treat the variable txt as if it were a read only version of the file itself. This variable that assumes the idobject of the file is called the file descriptor. At first, it may seem confusing that you assign the file name to a variable that is used to locate and open the file then assign the open file to a variable of a different name. But you cannot perform file reads and writes directly through a variable of the same name as the file. You can, however, assign an open file to a symbol then treat that symbol as if it were the file itself.

The syntax for open is:

(open [name of file] [read or write code])

The first argument is the name of the file to be opened. The second argument is a code that tells AutoLISP whether to allow read only, write only or appending operations on the file. The following table shows the codes and their meaning.

Code	Uses
"r"	Open a file for reading only. If the file does not exist, any attempts to read from it will result in an error message.
"w"	Open a file to write to. If the file already exists, its contents will be written over. If the file does not exist, it will be created
"a"	Open a file and append to the end if it. If the file does not exist, it will be created.

The next line uses the AutoLISP function read-line to read the first line of text from the open file:

(setq dt (read-line txt))

A line of text is read from the file represented by the symbol txt and is assigned to the variable dt. Read-line has only one argument, the file descriptor. When the file is first opened, AutoLISP goes to the beginning of the file in preparation to read it. Read-line reads the first line then AutoLISP moves to the second line waiting for further instructions to read the file. The next time the expression:

(read-line txt)

is evaluated, AutoLISP will read the next line after the previously red line then move to the following line and wait for another read-line function call.

The next two lines set up the location of the beginning of the text in the drawing editor:

(setq ls (rtos lns 2 2))

(setq ls (strcat "@" ls "<-90"))

The numeric value entered at the line spacing prompt is converted to a string then appended to the "@" and "<-90" strings to create a string that can be used in the text command that follows. The next line writes the text from the first line of the file into the AutoCAD drawing:

(command "text" sp "" "" dt)

The following while expression continually reads lines from the open file and writes them to the AutoCAD drawing editor:

(while (/= dt nil)

(setq dt (read-line txt))

(command "text" ls "" "" dt)

)

The read-line function will return nil when it reaches the end of the file. At that point, the while expression stops its recursion).

Finally, to take care of housekeeping, the open file is closed.

(close txt)

This is a very important step in the file reading process. If a file is not closed, the contents of that file can be lost.

We should mention that C:IMPRT requires the current text style to have a height value of 0. If this is not the case, then the program will not work properly. This is because in the line that actually writes the text into the drawing editor:

(command "text" ls "" "" dt)

assumes that AutoCAD will prompt for a height value for the text. The height prompt occurs only when the current text style has a height value of 0. If the current text style has a height value other than 0, the height prompt is skipped. This would cause the above expression to add one too many returns in the text command sequence (see figure 7.12).

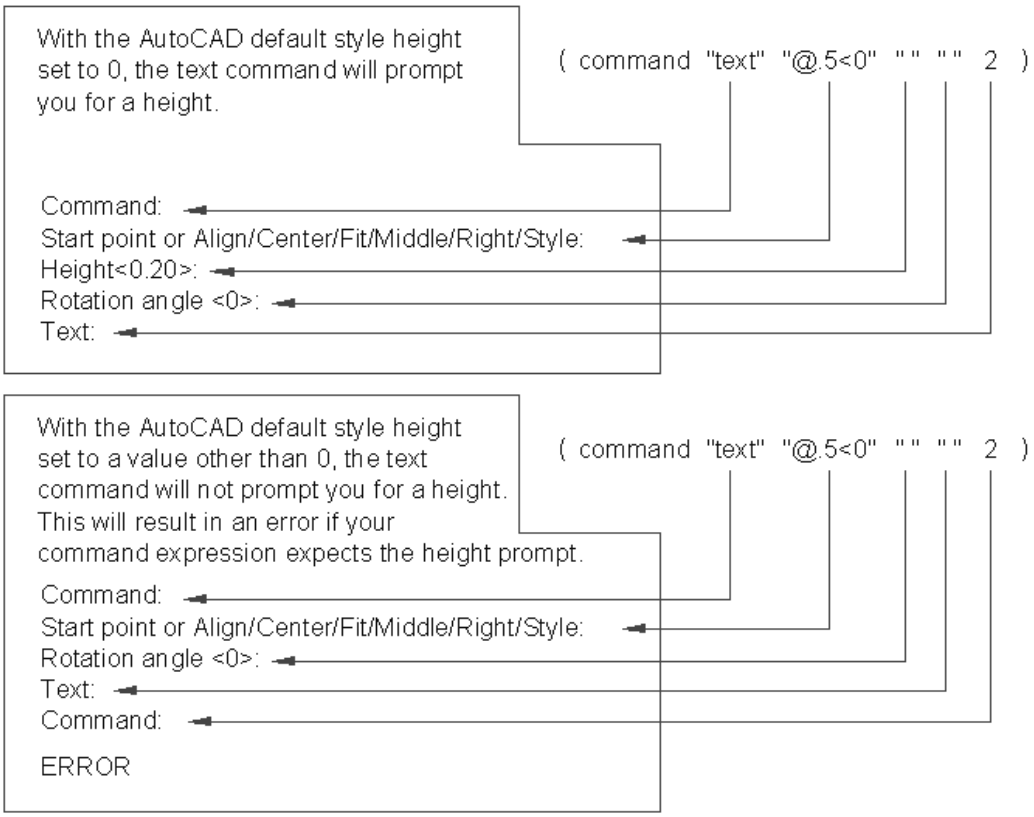


Figure 7.12: The text command expects specific input

Writing ASCII Files to Disk

AutoLISP lets you create ASCII files. You can use this capability to store different types of information ranging from the current drawing status to general notes that appear in a drawing. The program in figure 7.13 demonstrates this capability.

```
;Program to export text from AutoCAD -- Exprt.lsp
(Defun C:EXPRT (/ fname txt selset count nme oldtx)
  (setq fname (getstring "\nEnter name of file to be saved: "))
     (setq txt (open fname "w"))                      ;open file, assign symbol
     (setq selset (ssget))                            ;get selection set
     (setq count 0)                                   ;set count to zero
     (if (/= selset nil)
         (while (< count (sslength selset))           ;while count < # of lines
                (setq nme (ssname selset count))      ;extract text string
                (setq oldtx (cdr (assoc 1 (entget nme))))
                (write-line oldtx txt)                ;write string to file
                (setq count (1+ count))               ;go to next line
         );end while
     );end if
(close txt)                                           ;close file
);end C:EXPRT
```

Figure 7.13: A text export program

Using a Text Export Program

Exit AutoCAD then create a file called Exprt.lsp containing the program shown in figure 7.16. Return to AutoCAD, load the file and proceed with the following:

1. Erase the text you imported previously.

2. Use the dtext command and starting the text at the coordinate 2,8, write the following lines:

   For want of a nail, the shoe was lost;

   For want of a shoe, the horse was lost;

   For want of a horse, the rider was lost;

3. Enter **exprt** to start the text export program.

4. At the prompt:

Enter name of file to save:

Enter **test.txt**.

5. At the prompt:

Select objects:

Pick the lines of text you just entered picking each one individually from top to bottom. Press return when you are done.

The file test.txt is created containing the text you had just entered using AutoCAD's Dtext command. To make sure the file exists, enter the following at the command prompt:

type test.txt

The AutoCAD type command is the same as the DOS type command. It displays the contents of a text file. AutoCAD will switch to text mode and the contents of test.txt will be displayed on the screen.

The Exprt program starts by prompting the user to enter a name for the file to be saved to:

(Defun c:exprt (/ ts n dir)

(setq fname (getstring "\nEnter name of file to save: "))

Then, just as with the imprt program, the open function opens the file:

(setq txt (open n "w"))

In this case, the "w" code is used with the open functions since this file is to be written to. The next line obtains a group of objects for editing using the ssget function:

(setq selset (ssget))

This selection set is saved as the variable selset. Next, a variable count is give the value 0 in preparation for the while expression that follows:

(setq count 0)

The next if expression checks to see that the user has indeed picks objects for editing.

(if (/= selset nil)

If the variable selset does not return nil, then the while expression is evaluated:

(while (< count (sslength selset))

(setq entnme (ssname selset count))

(setq oldtx (cdr (assoc 1 (entget entnme))))

(write-line oldtx txt)

(setq count (1+ count))

)

This while expression uses the count variable to determine the number of times it must evaluate its set of expressions:

(while (< count (sslength selset))

The sslength function returns the number of objects contained in a selection set. In this case, it returns the number of objects recorded in the variable selset. This value is compared with the variable count to determine whether or not to evaluate the expressions found under the while expression.

The next two expressions extract the text string value from the first of the objects selected:

(setq entnme (ssname selset count))

(setq oldtx (cdr (assoc 1 (entget entnme))))

This extraction process involves several new functions which are discussed in chapter . For now, just accept that the end result is the assignment of the text value of the selected object to the variable oldtx.

Now the actual writing to the file occurs:

(write-line oldtx txt)

Here the write-line functions reads the string held by oldtx and writes it to the file represented by the variable txt.

Write-line's syntax is:

(write-line [string][file descriptor])

The first argument is the string to be written to file while the second argument is a variable assigned to the open file.

The next line increases the value of count by one:

(setq count (1+ count))

This expression counts the number of times a string of text, and therefore an object, has been processed. Since the while test expression checks to see if the value of count is less than the number of objects selected, once count reaches a value equivalent to the number of objects selected and processed, the while expression stops processing.

Finally, the all important close expression appears:

)

(close txt)

)

Just as with the C:IMPRT program, close must be used to properly close the file under DOS, otherwise its contents may be in-accessible.

Read-line and write-line are two of several file read and write functions available in AutoLISP. Table shows several other functions along with a brief description.

Function	**Description**
(prin1 *symbol/expression*)	Prints any expression to the screen prompt. If a file descriptor is included as an argument, the expression is written to the file as well.
(princ *symbol/expression*)	The same as prin1 but execute's control characters. Also, String quotation marks are dropped.
print symbol/expression)	The Same as prin1 but a new line is printed before its expression and a space is printed after.
(read-char *file_descriptor*)	Reads a single character from the keyboard. If a file descriptor is included as an argument, it reads a character from the file. The value returned is in the form of an ASCII character code.
read-line *file_descriptor*)	Reads a string from the keyboard or a line of text from an open file.
write-char *integer file_descriptor*)	Writes a single character to the screen prompt or if a file descriptor is provided, to an open file. The character argument is a number representing an ASCII character code.
write-line string *file_descriptor*)	Writes a string to the screen prompt or if a file descriptor is provided, to an open file.

The three functions prin1, princ, and print are nearly identical with some slight differences. All three use the same syntax as shown in the following:

(princ [string or string variable][optional file descriptor])

The file descriptor is a symbol that has been assigned an open file.

The main difference between these three functions is in what they produce as values. The following shows an expression using prin1 followed by the resulting value:

(prin1 "\nFor want of a nail...")

"For want of a nail..."\nFor want of a nail..."

Notice that the string argument to prin1 is printed twice to the prompt line. This is because both prin1 and AutoLISP will print something to the prompt. Prin1 prints its a literal version of its string argument. AutoLISPs read-evaluate-print loop also prints the value of the last object evaluated. The net result it the appearance of the string twice on the same line.

Princ differs from prin1 in that instead of printing a literal version of its string argument, it will act on any control characters included in the string:

(princ "\nFor want of a nail...")

For want of a nail..."\nFor want of a nail..."

In the prin1 example, the \n control character is printed without acting on it. In the princ example above, the \n character causes the AutoCAD prompt to advance one line. Also, the string is printed without the quotation marks. Again, the AutoLISP interpreter prints the value of the string directly to the prompt line after princ does it's work. Table Shows the other control characters and what they do.

Character	**Use**
\e	Escape
\n	New line
\r	Return
\t	Tab
\nnn	Character whose octal code is nnn

The print function differs from the prin1 function in that it advances the prompt one line before printing the string then it adds a space at the end of the string:

(print "\nFor want of a nail...")

"\nFor want of a nail..." "\nFor want of a nail..."

Just as with prin1, print prints a literal version of its string argument.

Conclusion

While a good deal of your effort using AutoLISP will concentrate on graphics and numeric computation, the ability to manipulate string data will be an important part of your work. You have been introduced to those functions that enable you to work with strings.

You have also seen how these functions work together to perform some simple tasks like reading and writing text files. But you don't have to limit yourself to using these functions for text editing. Since any kind of information can be stored as a string, you may find ways to further enhance your use of AutoCAD through the manipulation of string data. One of the more obvious uses that comes to mind is the importation of numeric data for graphing, charting or other types of data analysis. As long as data is stored in an ASCII format, AutoLISP can read it. And with AutoLISP's data conversion functions, numeric data can be easily translated from ASCII files.

Chapter 8: Interacting with AutoLISP

Introduction

AutoLISP offers a number of ways to interact with AutoCAD and the user. You have already seen how AutoLISP can control system variables through the setvar and getvar functions and you have seen the various ways your programs can obtain information from the user through the **get** functions. You can also control the display of menus, status and coordinate lines, graphic and text screens, and even the drawing area. By giving you control over the display, you can enhance the way your programs interact with the user.

When writing your program, consideration should be made as to how the program will act under various conditions. In this chapter, you will explore some of the ways you can exploit AutoLISP and AutoCAD features to make your programs more responsive to the user.

Reading and Writing to the Screen

There are many functions that will allow you to write prompts to the prompt line. But you are not limited to control of the prompt. Several functions are available that allow you to both read and write to other parts of the AutoCAD drawing editor. In this section, you will examine these functions.

Reading the Cursor Dynamically

In chapter 3 you used a function called RXY. This function reads the cursor location relative to a reference point and writes the relative coordinate directly to the coordinate readout. This is done dynamically as the cursor moves. The function that allows this to occur is the grread function. Grread's syntax is:

(grread [optional track argument])

Grread is a general input device reading function. It reads input from the keyboard, buttons on your pointing device, or the cursor location. Grread returns a list of two elements:

([integer][coordinate list or integer])

The first element is a code representing the type of input received. The second element is either an integer or list depending on whether a point has been input or a keyboard or pointing device button has been depressed. Table shows a list of codes for the first element of the list returned by grread:

Input Code	Meaning
(2 [ASCII code])	keyboard pressed. The second element of the list will be an integer representing the ASCII code of the key character pressed.
(3 [coordinate])	Cursor location picked. The second element of the list will be a coordinate list.
(4 [cell #])	Screen menu cell picked. The second element of the list will be an integer representing the cell number. The cells are numbered from top to bottom.
(5 [coordinate])	Dynamic cursor mode. The second element of the list will be a coordinate list. This code is returned only if a non-nil argument is supplied to the grread function.
(6 [button #])	Button on pointing device pressed. The second element of the list will be an integer representing the button number.
(7 [box #])	Tablet1 menu item selected. The second element of the list will be an integer representing the tablet item box number.
(7-10 [box #])	Tablet menu item selected. 7 equals tablet1 menu group, 8 equals tabalet2 menu group and so on. The second element of the list will be an integer representing the tablet item box number.
(11 [box #])	Aux1 menu item selected. The second element of the list will be an integer representing the tablet item box number.
(12 [coordinate])	If two grread functions are used in sequence, and the first returns a code 6, the second grread will return a code 12 and its second element will be the coordinate of the cursor at the time the pointing device button was picked.
(13 [menu cell #])	Screen menu item picked using keyboard input. The second element of the list will be the menu cell number.

Open a new AutoCAD file and turn on the snap mode. Enter the following at the AutoCAD command prompt:

(grread)

Now pick a point near the center of the screen using our mouse or digitizer puck. You will get a list similar to the following:

(3 (7.0 5.0 0.0))

The first element of the list is the integer 3. this tells us that the pick button on the pointing device was entered. The second element is a coordinate list showing the coordinate that was picked.

lets look at the expression in the RXY function that uses grread to read the cursor dynamically:

(setq pt (cadr (setq lt (grread t))))

The t argument tells grread to read the cursor location dynamically, that is, read it regardless of whether a button has been pushed or not. By using the T option, grread reads the cursor location even as it moves. The value from grread is assigned to the symbols **lt** for later processing. This value in turn is applied to cadr to obtain the coordinate list from grread. Finally, the coordinate list is assigned to the symbol pt (see figure 8.1).

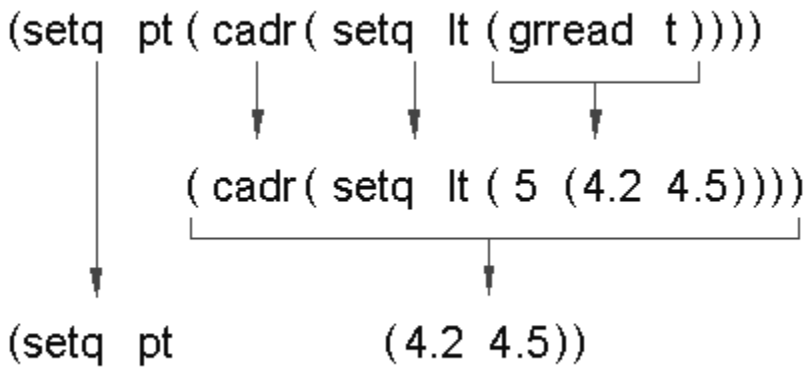


Figure 8.1: The evaluation of the grread expression

Once the coordinate is read by grread in the above expression, it is processed by the following set of expressions:

(if (= (car lt) 5)

(progn

(setq x (strcat

(rtos (- (car pt) (car lpt1))) " x "

(rtos (- (cadr pt) (cadr lpt1))) " SI= "

(rtos (*(- (car pt) (car lpt1))

(- (cadr pt) (cadr lpt1))

```
)

2 2)

)

)

(grtext -2 x)

)

)
```

This set of expressions takes the x and y components of the point pt and subtracts them from the x and y components of the reference point lpt1, which is selected earlier by the user. It then multiplies the remaining x and y values together to get the area of the rectangle formed by these two points. Finally, these values are turned into strings that can be sent to the screen.

First, the if expression tests to see if the code gotten from grread is 5. This checks to see if the coordinate was derived from the cursor in a drag mode.

```
(if (= (car lt) 5)(progn
```

Remember that **lt** is the list from the grread expression so the car of **lt** is its first element, the input code.

The next line is the outer most nest of an expression that combines a set of strings together into one string using strcat:

```
(setq x (strcat
```

This is followed by an expression that subtracts the x component of lpt1, our reference point, from the x of the current point pt:

```
(rtos (- (car pt) (car lpt1))) " x "
```

The resulting difference is converted into a string by rtos. The " x " element in this expression is the x that appears between the x and y value in the coordinate readout (see Figure 8.2).

The next expression subtracts the y component of lpt1 from the y of the current point pt then converts the resulting difference into a string:

```
(rtos (- (cadr pt) (cadr lpt1))) " SI= "
```

The " SI= " at the end of this line is the SI= that appears after the coordinate list in the coordinate readout.

The next two expressions multiplies x and y value differences to get the area of the rectangle represented by lpt1 and pt:

(rtos (*(- (car pt) (car lpt1))

(- (cadr pt) (cadr lpt1))

)

2 2)

)

)

The 2 2 in the fourth line down are the unit style and precision arguments to the rtos function.

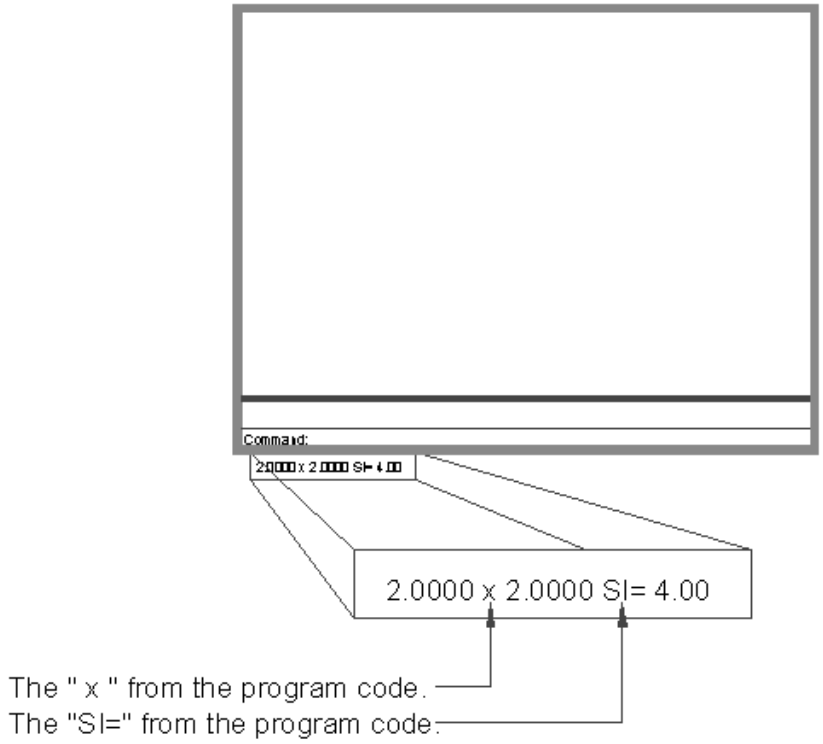


Figure 8.2: The elements of the coordinate readout

Writing Text to the Status and Menu Areas

Finally, all the values of the preceding group of expressions are concatenated then stored as the variable x which is in turn given as an argument to grtext:

(grtext -2 x)

)

);end if

Grtext writes the value of x to the coordinate readout. Grtext's syntax is:

(grtext [cell code] [string] [optional highlight code])

The cell code is the number of the screen menu cell you want to write the string to. Screen menu cells are numbered from 0 to the maximum number of cells available minus 1. If a -1 is used for the cell code, the string is written to the status area to the left of the coordinate readout. If the code is -2, as in the RXY function above, it is written to the coordinate readout area (see figure 8.3).

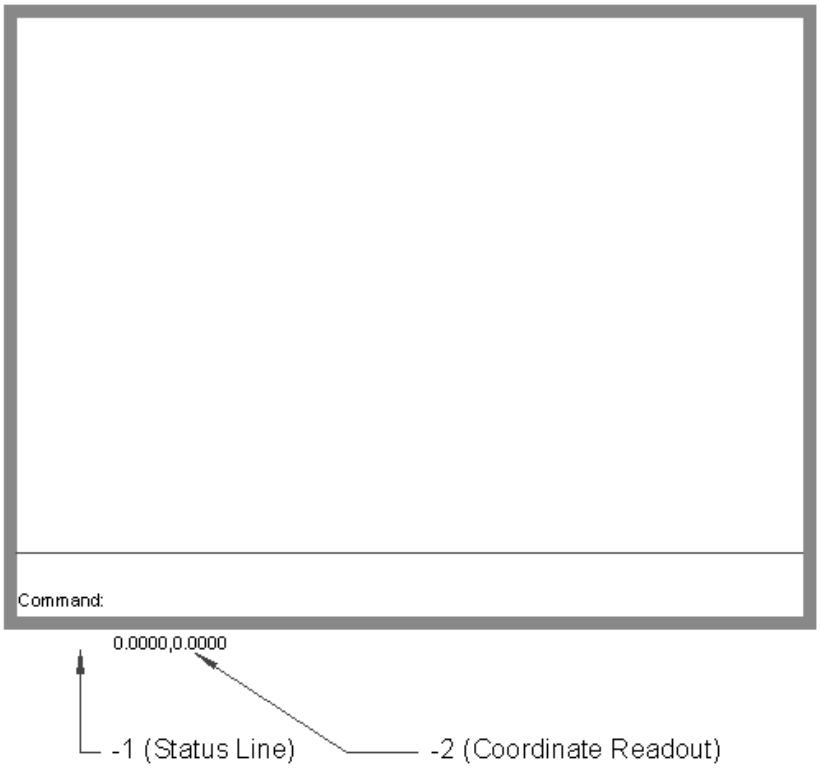


Figure 8.3: The AutoCAD screen and the corresponding screen codes

Although grtext allows you to display a string in the screen menu, such a string cannot be read by picking it with the cursor as you might think. Instead, the underlying menu option that has been written over by the grtext function is activated. If you want to override the underlying menu option, you can use grread to find the cell number picked using the using the cursor. Once you know the cell number, you can write expressions that instruct AutoCAD to perform alternate tasks.

Near the end of the RXY program, an expression sets the variable pick to T or nil depending on the input code from the variable lt:

(setq pick (= 3 (car lt)))

If the input code from lt is equal to 3, then pick is set equal to T, otherwise it is set to nil. This expression controls the while expression. If you look at the beginning of the while expression, the use of this expression becomes more clear (see figure 8.4).

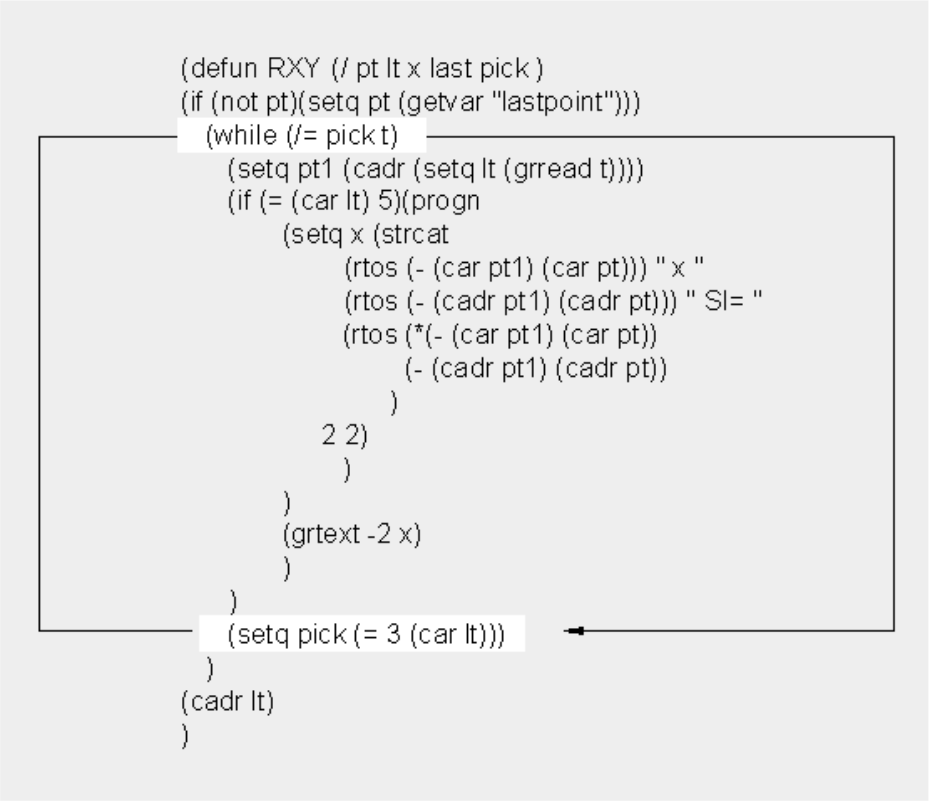


Figure 8.4: The pick variable used to control the while expression.

The while expression continually evaluates its set of expressions as long as pick is not equal to t. The last expression checks to see if the input code from lt is 3. If it is, that means that a point has been picked using the pick button. Pick is then set to T and the while expression stops running.

Calling Menus from AutoLISP

To make your program easier to use, you may want to have a particular menu appear whenever your program runs. AutoLISP provides the menucmd function for this purpose. Menucmd's syntax is:

(menucmd [menu specification])

The menu specification is a code used to call a particular menu. The code resembles the code used within the menu system to call other menus (see Appendix___ for details on menu codes). The only difference between these codes and their menu counterpart is that the menucmd code is not preceded by a dollar sign. The following table gives a brief description of these codes:

Code	Description
bn=menu name	Calls button menus. The n that follows the b is the number of the button menu group.
s=menu name	Calls screen menus
pn=menu name	Calls Pull down menus. The n that follows the p is the number of the pull down menu group.

Figure 8.5 Shows the box program from chapter 2 with the addition of the menucmd function that calls the osnap screen menu. If you were to load and run this program, the osnap screen menu would appear at the first prompt. (Note that screen menus are not a standard part of AutoCAD release 13 or 14 though they can be turned on for compatibility with older versions.)

```
(defun c:BOX ( /  pt1 pt2 pt3 pt4 )
(menucmd "s=osnapb")
(setq pt1 (getpoint "Pick first corner: "))
(setq pt3 (getcorner pt1 "Pick opposite corner: "))
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
(command "line" pt1 pt2 pt3 pt4 "c" )
)
```

Figure 8.5: The modified box program

Using menucmd with the pull down menu's is a bit more involved. Figure 8.6 shows the same program again this time making a call to the filters pull down menu. If you load and run this program, the tools pull down menu with the osnap options will pop down.

```
(defun c:BOX ( /  pt1 pt2 pt3 pt4 )
(menucmd "p1=filters")
(menucmd "p1=*")
(setq pt1 (getpoint "Pick first corner: "))
(setq pt3 (getcorner pt1 "Pick opposite corner: "))
(setq pt2 (list (car pt3) (cadr pt1)))
(setq pt4 (list (car pt1) (cadr pt3)))
(command "line" pt1 pt2 pt3 pt4 "c" )
)
```

Figure 8.6: The box program modified to call a pull down menu

Notice that a line was added in addition to the first menucmd line:

(menucmd "p1=*")

Just as you must include the $p1=* in a menu macro to display the tools pull down menu, you must also include the asterisk call in your AutoLISP program following any pull down menu call.

Drawing Temporary Images on the Drawing Area

There may be times when you will want an image drawn in the drawing area that is not part of the drawing database. Such a temporary image can be useful to help you locate points or draw temporary images to help place blocks or other objects. Figure 8.7 shows a program that performs zooms in a different way from the standard AutoCAD zoom. In this program called C:QZOOM, the screen is divided visually into four quadrants. You are prompted to pick a quadrant at which point the quadrant is enlarged to fill the screen. Optionally, you can press return at the Pick quadrant prompt and you will be prompted to pick a new view center. This option is essentially the same as a pan. Copy C:QZOOM into an AutoLISP file and load and run it.

```
(defun mid (a b)
(list (/ (+ (car a) (car b) ) 2) (/ (+ (cadr a) (cadr b) ) 2))
)
(defun C:QZOOM (/ center height ratio width
                  wseg hseg ll ur ul lr newctr)
 ;find screen position
(setq center (getvar "viewctr"))
(setq height (getvar "viewsize"))
(setq ratio  (getvar "screensize"))
(setq width  (* height (/ (car ratio)(cadr ratio))))
(setq wseg   (/ width  2.0))
(setq hseg   (/ height 2.0))
;find screen corners
(Setq ll (list (- (car center) wseg)(- (cadr center) hseg)))
(Setq ur (list (+ (car center) wseg)(+ (cadr center) hseg)))
(Setq ul (list (- (car center) wseg)(+ (cadr center) hseg)))
(Setq lr (list (+ (car center) wseg)(- (cadr center) hseg)))
;draw screen quadrants
(grdraw center (polar center pi wseg) -1 1)
(grdraw center (polar center 0  wseg) -1 1)
(grdraw center (polar center (* pi 0.5) hseg) -1 1)
(grdraw center (polar center (* pi 1.5) hseg) -1 1)
;get new center and height
(setq newctr (getpoint "\nPick quadrant/<pan>: "))
 (cond
  ( (not newctr)
      (setq newctr (getpoint "\nPick new center: "))
      (setq hseg height)
  )
  ( (and (< (car newctr)(car center))(< (cadr newctr)(cadr center)))
    (setq newctr (mid center ll))
  )
  ( (and (< (car newctr)(car center))(> (cadr newctr)(cadr center)))
    (setq newctr (mid center ul))
  )
  ( (and (> (car newctr)(car center))(< (cadr newctr)(cadr center)))
    (setq newctr (mid center lr))
  )
  ( (and (> (car newctr)(car center))(> (cadr newctr)(cadr center)))
    (setq newctr (mid center ur))
  )
 )
(command "zoom" "c" newctr hseg)
)
```

Figure 8.7: The C:QZOOM program

The program is written with comments to group parts of the program together visually. The first group establishes several variables. It finds the current view center point, the view height in drawing units, and the views height to width ratio. Based on this ratio, it finds the width of the current view in drawing units. It also finds the values for half the width and height (see figure 8.8).

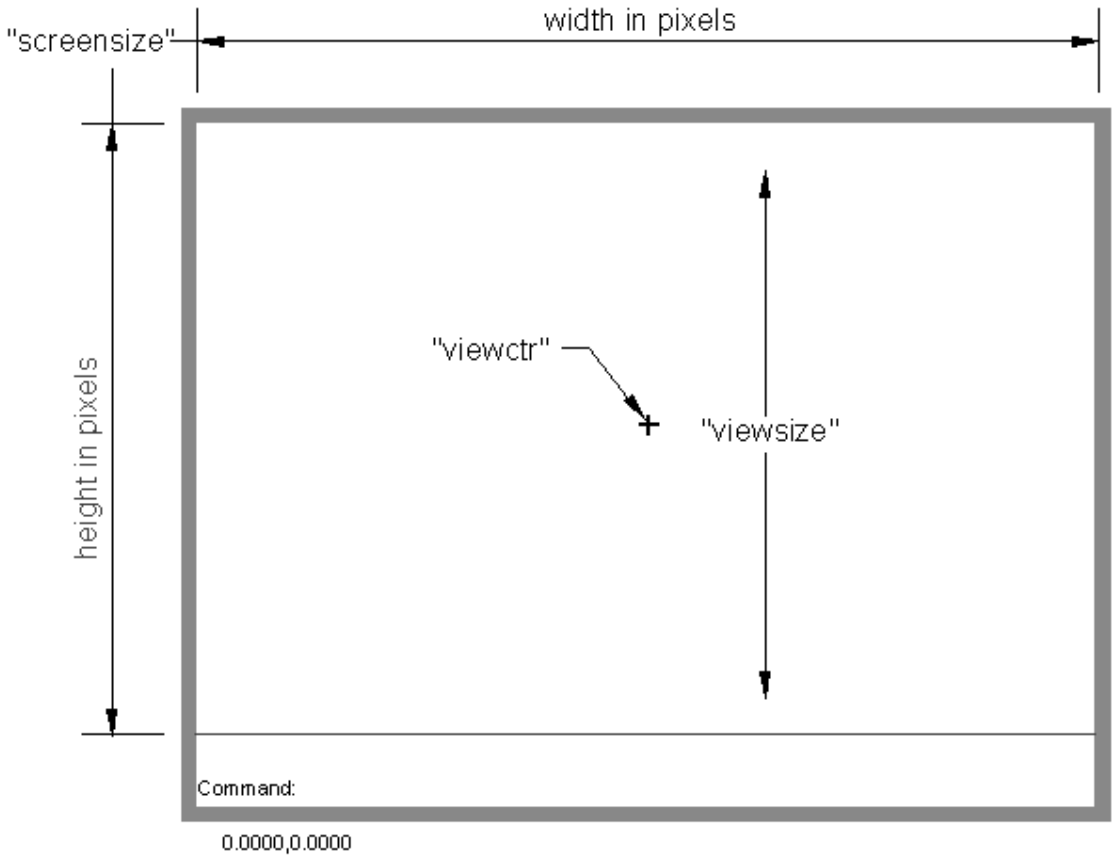


Figure 8.8: Finding the current screen properties

The next group establishes the four corner points of the current view. This is done by taking the center point of the view and adding and subtracting x and y value for each corner (see figure 8.9).

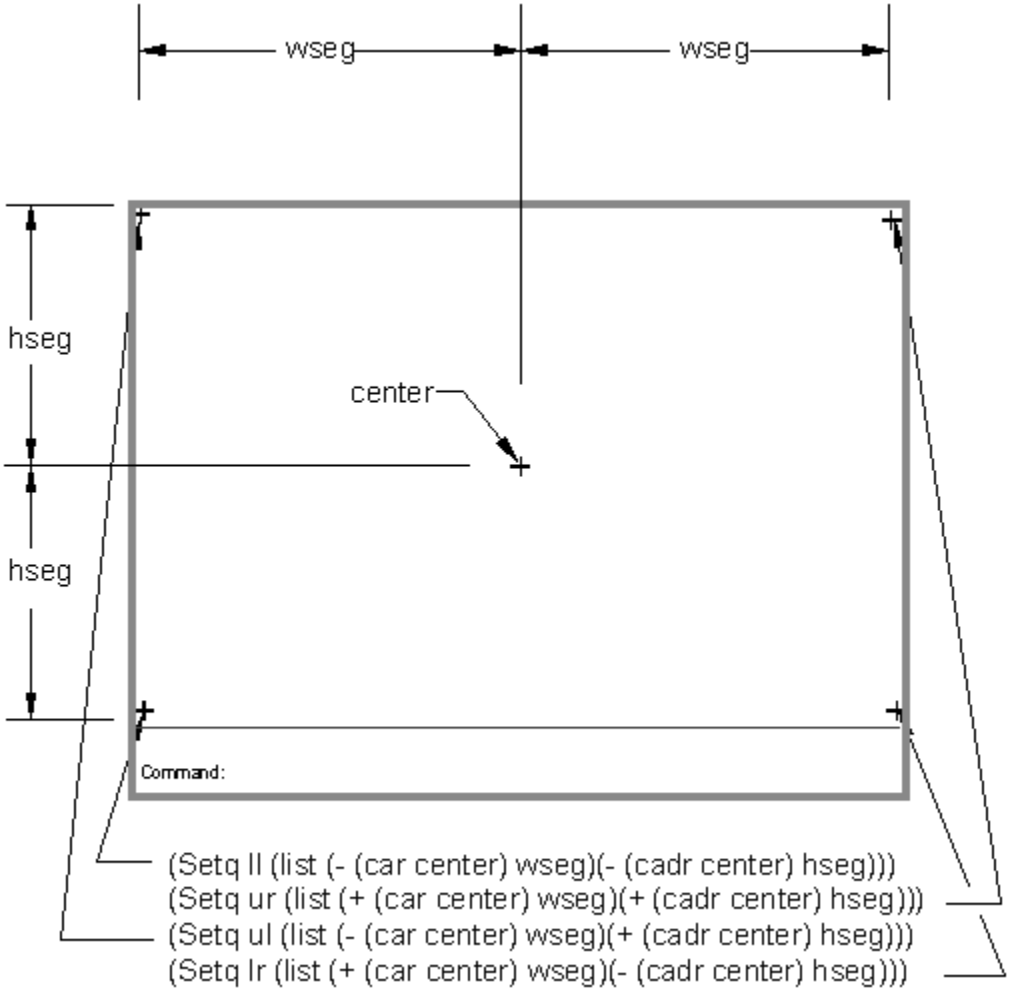


Figure 8.9: Finding the current displays four corner coordinates.

The next group draws the lines that divides the screen into quadrants (see Figure 8.10). Looking a the first line, you can see the grdraw function:

(grdraw center (polar center pi wseg) -1 1)

The syntax for grdraw is:

(grdraw [from point][to point][color code][optional highlight code])

The lines grdraw draws act like blips. As soon as you issue any command that changes the display, including

redraws, the lines disappear. The first and second arguments to grdraw determine the endpoints of the temporary line. The third argument is the color code. This code is an integer value that represents the color you want the line to have. If the color code is -1, then the line will be drawn in a color that compliments its background. This ensures that the line will be seen. The fourth argument is optional. Whenever it is a integer other than 0 the line will be drawn highlighted. This usually means in a dotted pattern similar to that shown in a crossing window.

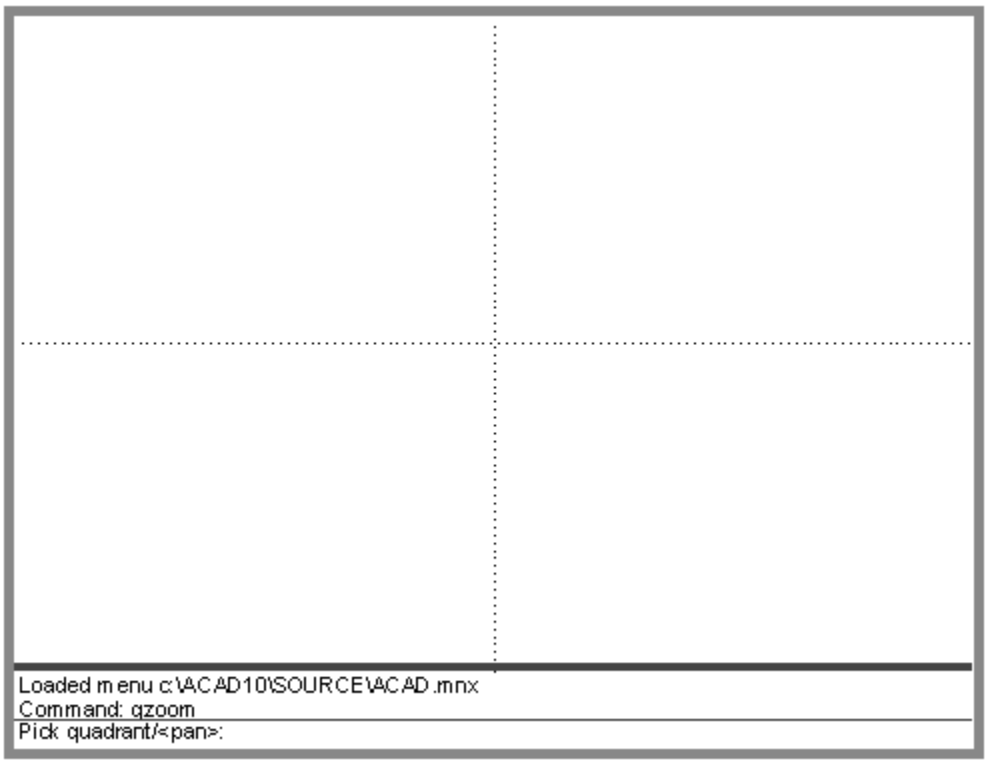


Figure 8.10: The quadrants drawn on the screen

In the C:QZOOM program, grdraw draws lines from the center of the current view out toward the four sides of the screen. The -1 color code is used so the color of the lines is opposite to the background color. Finally, the highlight option is used by supplying a 1 as a fourth argument.

The last group of expressions does the work of reading the pick point from the user and determining which quadrant to enlarge.

Using Defaults in a Program

Virtually every AutoCAD command offers a default value. For example, the line command will continue a line from the last point selected if no point is selected at the First point prompt. Defaults can be a great time saver especially when the user is in a hurry. In this section, you will see first hand how you can add defaults to your own programs.

Adding Default Responses to your Program

In the C:QZOOM program, a default response was added. If the user presses return without picking a point, the program goes into a pan mode allowing the use to select a new view center. By giving the user the pan option in this way, the program becomes easier to use and more flexible. Other AutoCAD commands also provide default values for options. For example, the offset command will offer the last offset distance as a default value for the current offset distance. If the user decides he or she can use that value, he or she only needs to press return to go on to the next part of the command.

You can incorporate similar functionality into your programs by using global variables. Figure 8.11 shows the sequential number program created in chapter 5 with code added to include a default value for the number spacing.

```
(defun C:SEQ (/ pt1 currnt last spc)
(if (not *seqpt)(setq *seqpt 2.0))                ;setup global default
(setq pt1    (getpoint "\nPick start point: "))       ;get start point
(princ "\nEnter number spacing <")                ;first part of prompt
(princ *seqpt)                                      ;print default part of prompt
(setq spc    (getdist pt1 ">: "))                   ;finish prompt - get spac'g
(setq currnt (getint "\nEnter first number: "))     ;get first number
(setq last   (getint "\nEnter last number: "))      ;get second number
(if (not spc)(setq spc *seqpt)(setq *seqpt spc));set global variable
(setq stspc  (rtos spc 2 2))                        ;convert spacing to string
(setq stspc  (strcat "@" stspc "<0" ))              ;create spacing string
(command "text" pt1 "" "" currnt)                   ;place first number
  (repeat (- last currnt)                           ;start repeat 'till last
    (setq currnt (1+ currnt))                       ;add 1 to current number
    (command "text" stspc "" "" currnt)             ;place text
  )                                                 ;end repeat
)                       ;end defun
```

Figure 8.11: The modified C:SEQ program

Make the changes to your copy of the C:SEQ program so it looks like figure 8.11. Open a new file in AutoCAD,

load the newly modified C:SEQ program, and then run it. The program will run as it has before but it now offers a default value of 2.0 at the Enter number spacing prompt:

Enter number spacing <2.0>:

Press return at this prompt. The default value of 2.0 will be applied to the number spacing. If you enter a different value, .5 for example, this new value becomes the default. The next time you run the program, .5 will appear as the default value for the number spacing:

Enter number spacing <0.5>:

There are actually several expressions that are added to make this default option possible. First is a conditional expression that test to see if a global variable called *seqpt is non-nil:

```
(defun C:SEQ (/ pt1 currnt last spc)

(if (not *seqpt)(setq *seqpt 2.0))
```

It its' value is nil, it is given the value of 2.0. This is just an arbitrary value. You can make it anything you like.

Next, the user is prompted to pick a point just as in the previous version of the program:

```
(setq pt1 (getpoint "\nPick start point: "))
```

The next set of expressions does the work of displaying the default value to the AutoCAD prompt.

```
(princ "\nEnter number spacing <")

(princ *seqpt)

(setq spc (getdist pt1 ">: "))
```

The prompt is broken into three parts. The first expression above prints everything before the default value. The second expression prints the default value. The third expression uses the getdist function to obtain a new distance value from the user. The end of the prompt is included as the prompt string to the getdist function. The net result is a single line appearing at the AutoCAD prompt (see figure 8.12).

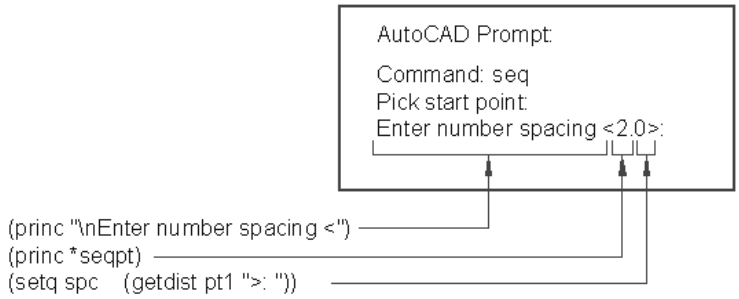


Figure 8.12: Using princ to construct a prompt

The next two lines are unchanged from the earlier version of the program:

```
(setq currnt (getint "\nEnter first number: "))

(setq last (getint "\nEnter last number: "))
```

The next line is a new conditional expression that tests to see if a value was entered at the Enter number spacing prompt:

```
(if (not spc)(setq spc *seqpt)(setq *seqpt spc))
```

This expression test the variable spc to see if it value is non-nil. If it is nil, indicating the user pressed return without entering a value, spc is assigned the value of the global variable *seqpt. This is the default value that appears in the Enter number spacing prompt. If spc does have a value, then its value is assigned to *seqpt thus making the value of spc the new default value.

The rest of the program is unchanged:

```
(setq stspc (rtos spc 2 2))

(setq stspc (strcat "@" stspc "<0" ))

(command "text" pt1 "" "" currnt)

(repeat (- last currnt)

(setq currnt (1+ currnt))

(command "text" stspc "" "" currnt)

)

)
```

You may wonder why the global variable *seqpt starts with an asterisk. Names given to global variables don't have to be different from other symbol but you may want to set them off by preceding them with an asterisk. This is a convention used in Common LISP which we have carried over to AutoLISP.

Creating a Function to Handle Defaults

If you find that many of your programs utilize defaults, it may be worthwhile to create a function that creates the default prompts for you. Figure 8.13 Shows the C:SEQ program with a function added to handle default prompts.

```
(defun deflt (str1 def)
(strcat str1 " <" (rtos def 2 4) ">: ")
)


(defun C:SEQ (/ pt1 currnt last spc)
(if (not *seqpt)(setq *seqpt 2.0))              ;setup global default
(setq pt1    (getpoint "\nPick start point: "))
(setq spc    (getdist (deflt "\nEnter spacing" *seqpt)))
(setq currnt (getint "\nEnter first number: "))
(setq last   (getint "\nEnter last number: "))
(if (not spc)(setq spc *seqpt)(setq *seqpt spc)) ;set global variable
(setq stspc  (rtos spc 2 2))
(setq stspc  (strcat "@" stspc "<0" ))
(command "text" pt1 "" "" currnt)
  (repeat (- last currnt)
    (setq currnt (1+ currnt))
    (command "text" stspc "" "" currnt)
  )
)
```

Figure 8.13: The C:SEQ program with a default handling function.

The function deflt takes two arguments. The first is the beginning text of the prompt and the second is the default value.

(defun deflt (str1 def / lunts)

(setq lunts (getvar "lunits"))

(strcat str1 " <" **(rtos def lunts 4)** ">: ")

)

The arguments are concatenated to form a single string which is the value returned by the function. Since the default value is a real data type, it is converted to a string using the rtos function. The getvar expression at the beginning of the function finds the current unit style which is used in the rtos function to control the unit style created by rtos.

The C:SEQ function uses this function in the expression:

(setq spc (getdist (deflt "\nEnter spacing" *seqpt)))

Here, the function is place where the prompt string normally appears in a getdist expression. When deflt is evaluated, it returns the string:

"\nEnter spacing <2.0000>: "

This string is then supplied to the getdist function as the prompt string argument.

The C:SEQ program still requires the two conditional expressions that were added earlier:

```
(if (not *seqpt)(setq *seqpt 2.0))
.
.
.
(if (not spc)(setq spc *seqpt)(setq *seqpt spc)) ;set global
```

But without increasing the amount of code, we are able to make a simpler and more flexible system to add prompts to our programs. An added benefit is a more readable program.

Dealing with Aborted Functions

If you are writing programs for yourself, you may not be too concerned with how the program looks or behaves. But if you start to write programs for others to use, you have to start thinking about ways of making your programs more error proof. You should provide ways of easily exiting your program without creating problems for the unfamiliar user. Error handling, as it is often called, is writing your program to include code that anticipates any possible input errors the user might come up with. Fortunately, most of AutoLISPs get functions have some error handling capabilities built in. If you enter a string when a get function expects a point, you will get a message telling you that a point is expected.

But the most common error handling problem you will encounter is the aborted program. We can get an idea of how an aborted program can affect a users work by using the C:BREAK2 program from chapter 6.

Open an AutoCAD file and load the C:BREAK2 program. Start C:BREAK2 by entering **break2** at the command prompt. At the prompt:

```
Select object:
```

press the Ctrl-C key combination to abort the program. Now any time you use your cursor to select a point or object, you will get the "nearest" osnap override. This because you aborted the program before it was able to set the osnap mode back to none (see figure 8.14).

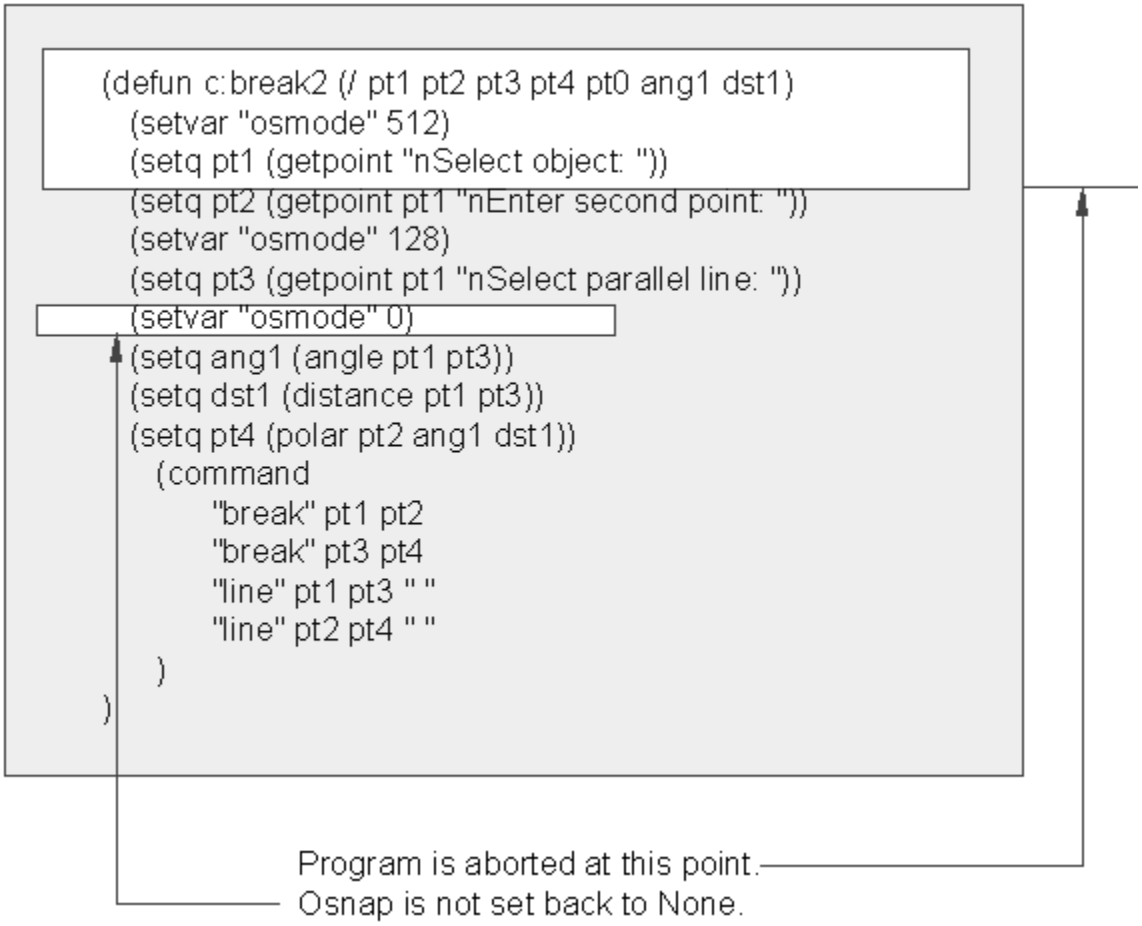


Figure 8.14: What happens when C:BREAK2 is aborted

Using the *error* Function

To deal with problems like this, you can use a special AutoLISP function called *error*. If a function is created using *error* as its name, it is evaluated whenever an error occurs. Figure 8.15 shows the Break2.lsp file with the addition if an *error* function. Open the Break2.lsp file and add the *error* function shown in figure 8.16.

```
(defun *error* (msg)
(setvar "osmode" 0)
(princ msg)
(princ)
)


(defun c:break2 (/ pt1 pt2 pt3 pt4 pt0 ang1 dst1)
   (setvar "osmode" 512)                                ;near osnap mode
   (setq pt1 (getpoint "\nSelect object: "))            ;get first break point
   (setq pt2 (getpoint pt1 "\nEnter second point: "))  ;get second break point
   (setvar "osmode" 128)                                ;perpend osnap mode
   (Setq pt3 (getpoint pt1 "\nSelect parallel line: "));get 2nd line
   (Setvar "osmode" 0)                                  ;no osnap mode
   (setq ang1 (angle pt1 pt3))                          ;find angle btwn lines
   (setq dst1 (distance pt1 pt3))                       ;find dist. btwn lines
   (setq pt4 (polar pt2 ang1 dst1))                     ;derive pt4 on 2nd line
      (command
            "break" pt1 pt2                             ;break 1st line
            "break" pt3 pt4                             ;break 2nd line
            "line" pt1 pt3 ""                           ;close ends of lines
            "line" pt2 pt4 ""
      )
)
```

Figure 8.15: The Break2.lsp file with an error checking function added.

Save the file go back to the AutoCAD file. Be sure that osnap is set to "none". Load and start the C:Break2 program. Again, at the **Select object** prompt, enter a Ctrl-C.

Now, instead of leaving the osnap mode in the "nearest" setting, the *error* function returns the osnap setting back to "none". It also prints the message:

Function cancelled

Lets look at this function to see exactly how it works. The first line looks like a typical defun expression:

(defun *error* (msg)

The argument list contains the symbol msg. *error* accepts as an argument, an error message. This error message is the one that appears normally without the *error* function. In the next line:

(setvar "osmode" 0)

the osnap mode is set back to 0. Next, the error message supplied by AutoLISP is printed to the AutoCAD prompt:

(princ msg)

The last princ prevents the error message from appearing twice in the prompt.

This works very nicely assuming that you always have osnap set to "none". But suppose your osnap setting varies during your editing session and you want your function to return to whatever the current setting is at the time a program is issued. Figure 6.16 shows the Break2 program again with some additional code that helps restore the osnap setting to its previous setting regardless of what it may have been.

```
(defun *error* (msg)
(setvar "osmode" *osnap)
(princ msg)
(princ)
)


(defun c:break2 (/ pt1 pt2 pt3 pt4 pt0 ang1 dst1)
   (setq *osnap (getvar "osmode"))
   (setvar "osmode" 512)                              ;near osnap mode
   (setq pt1 (getpoint "\nSelect object: "))          ;get first break point
   (setq pt2 (getpoint pt1 "\nEnter second point: "))  ;get second break point
   (setvar "osmode" 128)                              ;perpend osnap mode
   (Setq pt3 (getpoint pt1 "\nSelect parallel line: "));get 2nd line
   (Setvar "osmode" *osnap)                           ;no osnap mode
   (setq ang1 (angle pt1 pt3))                        ;find angle btwn lines
   (setq dst1 (distance pt1 pt3))                     ;find dist. btwn lines
   (setq pt4 (polar pt2 ang1 dst1))                   ;derive pt4 on 2nd line
      (command
            "break" pt1 pt2                         ;break 1st line
            "break" pt3 pt4                         ;break 2nd line
            "line" pt1 pt3 ""                       ;close ends of lines
            "line" pt2 pt4 ""
      )
)
```

Figure 8.16: The C:BREAK2 program modified to handle any osnap setting

The line:

(setq *osnap (getvar "osmode"))

is added to the beginning of the program. This creates a global variable *osnap which holds the osnap code that determines the current osnap setting. The expression that returns the osnap mode to "none":

(setvar "osmode" 0)

is replaced by one that sets the osnap mode to whatever was saved as *osnap:

(setvar "osmode" *osnap)

This same expression appears in the *error* function so in the event of a cancellation by the user, the osnap mode is set back to its previous setting.

Organizing Code to Reduce Errors

The error handling function shown here as an example could be incorporated into your Acad.lsp file so it is available for any AutoLISP error that may occur. You can also enlarge it to include other settings or variables that may require resetting. But the way a program is organized can affect the impact an error has. For example, we could have written the C:SEQ program in a slightly different way. Figure 8.17 shows the program with its expressions in a slightly different order.

```
(defun deflt (str1 def)
(strcat str1 " <" (rtos def 2 4) ">: ")
)


(defun C:SEQ (/ pt1 currnt last spc)
(if (not *seqpt)(setq *seqpt 2.0))              ;setup global default
(setq pt1    (getpoint "\nPick start point: "))
(setq spc    (getdist (deflt "\nEnter spacing" *seqpt)))
(if (not spc)(setq spc *seqpt)(setq *seqpt spc)) ;set global variable
(setq currnt (getint "\nEnter first number: "))
(setq last   (getint "\nEnter last number: "))
(setq stspc  (rtos spc 2 2))
(setq stspc  (strcat "@" stspc "<0" ))
(command "text" pt1 "" "" currnt)
  (repeat (- last currnt)
    (setq currnt (1+ currnt))
    (command "text" stspc "" "" currnt)
  )
)
```

Figure 8.17: The C:SEQ program in a different order

The conditional expression:

(if (not spc)(setq spc *seqpt)(setq *seqpt spc))

immediately follows the expression that prompts for the spacing distance. This placement seems to be a natural place for this expression since it immediately sets the variable spc or *seqpt to a value once the value of spc is obtained. But what happens if the user decides to cancel the program after this expression is evaluated. If the user inputs a new value for the number spacing, then the global variable *seqpt holds that new value even though the program has been canceled. The next time the user uses the C:SEQ program, the value that was entered previously is the new default value. Even though the program was cancelled the value entered for the number spacing became the new default.

This may or may not be a problem but for many, issuing a cancel means canceling the affects of any data entry made during the command. So to avoid having the global variable *seqpt changed when the program is cancelled, the conditional expression is moved to a position after all the prompts are issued. This way, the user can cancel the program with no affect to the *seqpt variable.

Debugging Programs

While we are on the subject of errors, We should discuss the debugging of your programs. As you begin to write programs on your own, you will probably not get them right the first time. Chances are, you will write a program then run it only to find some error message appear. Then you must review your program to try and find the offending expression.

Common Programming Errors

Most of the time, errors will be due to the wrong number of parentheses or the wrong placement of parentheses within your program. If this is the case, you usually get the error message:

error: malformed list

or

error: extra right paren

There aren't any simple solutions to this problem other than checking your program very carefully for number and placement of parentheses. Perhaps the best thing to do is to print out your program. It is often easier to spot errors on paper than it is to spot them on your computer screen.

Since a misplaces paren can cause a variety of problems, printing out your program and checking the parentheses placement is the best start.

Another common error is to mis-spelled symbols. This is especially a problem with confusing lower case l's with 1's and zeros with o's. The full range of typos is possible and often hard to detect. Again, the best solution is to print out your program and take a careful look.

If you get the message:

error: Insufficient string space

chances are, you didn't provide a closing double quote in a string value as in the following:

(menucmd "p1=*)

Also, prompt strings cannot exceed 100 characters.

Finally, it is common to try to apply a wrong data type to a function. We have mentioned that one common error is to give a variable a string value which happens to be a number:

(setq str1 "1")

Later, you might attempt to use this string as an integer in another function:

(1+ str1)

This results in a bad argument type error.

For your convenience, we have included appendix B that contains the AutoLISP error messages and their meaning. You may want to refer to it as you debug your programs.

Using Variables as Debugging Tools

AutoLISP helps you find errors by printing to the screen the offending expression along with the error message. But sometimes this is not enough. If you find you are having problems with a program, you can check the variables in the program using the exclamation point to see what values they have obtained before the program aborted. If you have an argument list, you may want to keep it empty until you finish debugging your program. That way, you can check the value of the programs's variables. Otherwise, the values of the variables will be lost before you have a chance to check them.

If you have a particularly lengthy program, you can use the princ function to print variables to the prompt line as th program runs. By placing the princ function in strategic locations within your program, you can see dynamically what your variables are doing as the program runs. You can also have the princ function print messages telling you where within your program it is printing from.

Conclusion

As you begin to write you own program, many of the issues brought to light in this chapter will confront you. We hope that by introducing these topics now, you will have a better grasp of what is required in a program design. By knowing what is possible within AutoCAD, you can develop programs that simplify the users efforts.

Chapter 9: Using Lists to store data

Introduction

We mentioned that there are actually two classes of lists, those meant to be evaluated, which are called expressions or forms, and lists that are repositories for data such as a coordinate list. No matter what the type of list you are dealing with, you can manipulate lists to suite the needs of your program. In this section we will look at the general subject of lists and review some of the functions that allow you to manipulate them.

There are a several functions provided to access and manipulate lists in a variety of ways. You have already seen car and cdr. Table 9.1 shows a list of other functions with a brief description of their uses.

Function	Use
(mapcar function list list...)	Apply elements of lists as arguments, to a function. Each element in the list is processed until the end of the list is reached
(apply function list)	Apply the entire contents of a list to a function.
(foreach symbol list expression)	Sets individual elements of a list to symbol then evaluates an expression containing that symbol. Each element in the list is processed until the end of the list is reached.
(reverse list)	reverses the order of elements in a list.
(Cons element list)	Adds a new first element to a list. The element can be any legal data type.
(append list list ...)	Takes any number of lists and combines their elements into one list.
(last list)	Finds the last element of a list.
(length list)	Finds the number of elements in a list.
(member element list)	Finds the remainder of a list starting with element.
(nth integer list)	Finds the element of a list where integer is the number of the desired element within the list. The first item in a list is number 0.

So far, we have concentrated on the use of lists as a means of structuring and building your programs. But lists can also be use as repositories for data. You have already seen how coordinate list are used to store the x and y coordinate values of a point. Lists used for storing data can be much larger than the coordinate example. Consider the mdist program you saw in chapter 5. This program uses the append function to constantly add values to a list. This list is then evaluated to obtain the sum of its contents (see figure 9.1).

```
(Defun C:MDIST (/ dstlst dst)
(setq dstlst '(+ 0))
  (while (setq dst (getdist "\nPick point or Return to exit: "))
         (Setq dstlst (append dstlst (list dst)))
         (princ (Eval dstlst))
  )
)
```

Figure 9.1: The Mdist program

We have an example here of a list being both a repository of data and a form or a list that can be evaluated. This is accomplished by starting the list with a functions, in this case, the plus function. Each time a value is appended to

the list, it is evaluated to get the sum of the numeric elements of that list.

Suppose your have a list that does not contain a function, but you want to apply some function to it. The following sections discusses ways you can use the functions listed in table 9.1 perform computations on lists.

Getting Data from a List

In the mdist program, a function was applied to a list to get the total of all the numbers in that list. Functions like plus, minus, multiply and divide accept multiple numeric values for arguments. But what if you want to apply a list to a function that will only take single atoms for arguments.

Using Simple Lists for Data Storage

Mapcar is used where you want to use a list as a queue for arguments to a function. It allows you to perform a recursive function on a list of items. For example, suppose you want the sequential numbering program from chapter 5 to place the numbers at points you manually select rather than in a straight line. Figure 9.2 shows a program that does this:

```
;Program to write sequential numbers -- Seqrand.lsp
(defun C:SEQRAND (/ rand currnt ptlst)
(setvar "cmdecho" 0)                              ;no echo to prompt
(setq rand T)                                     ;set up rand
(setq currnt (getint "\nEnter first number in sequence: "))
(while rand                                       ;while point is picked
    (setq rand (getpoint "\nSelect points in sequence: " ));get point
    (setq ptlst (append ptlst (list rand) ))      ;add point to list
)
 (mapcar
   '(lambda (rand)                                ;define lambda expression
      (if rand                                    ;if point (rand) exists
       (progn
       (command "text" rand "" "" currnt)         ;place number at point rand
       (setq currnt (1+ currnt))                  ;get next number
       )
      )
    )
  ptlst                                           ;list supplied to lambda
 )
(setvar "cmdecho" 1)                              ;echo to prompt on
(princ)
)
```

Figure 9.2: Program to place sequential number in random locations

In the C:SEQRAND program, the following while expression is used to allow the user to pick random point locations for the numbered sequence:

```
(while (not (not rand))
(setq rand (getpoint "\nSelect points in sequence: " ))
(setq ptlst (append ptlst (list rand) ))
)
```

This while expression creates the list ptlst comprised of points entered by the user. The user see the prompt:

Select points in sequence:

each time he or she selects a point. Once the user is done, the mapcar expression reads the list of points and applies them to a function that enters the sequence of numbers at those points:

```
(mapcar
'(lambda (rand)
(if (not (not rand))
(progn
(command "text" rand "" "" currnt)
(setq currnt (1+ currnt))
)
)
)
ptlst
)
```

In this set of expressions, mapcar applies the elements of the list ptlst to a lambda expression. You may recall that a lambda expression is like a function created using defun. The difference being that lambda expressions have no name. The lambda expression above uses the single argument rand and, using the AutoCAD text command, writes the variable currnt to the drawing using rand as a coordinate to place the text. The lambda expression also adds 1 to the currnt variable increasing the number being added to the drawing by 1.

Mapcar takes the list of points ptlst and, one at a time, applies each element of the list to the variable rand in the lambda expression until all the elements of the list have been applied. The if conditional expression is added to the lambda expression to check for the end of ptlst.

Mapcar can apply more than one lists to a function as shown in the following expression:

(mapcar 'setvar

'("cmdecho" "blipmode" "osnap" "expert")

'(0 0 512 1)

)

Here mapcar applies several AutoCAD system variables to the setvar function. One element is taken from each list and applied to setvar. "Cmdecho" is set to 0, "blipmode is set to 1, "osmode" is set to 512, the nearest mode, and "expert" is set to 1.

Evaluating Data from an Entire List at Once

Apply is similar to mapcar in that it allows you to supply a list as an argument to a function. But rather than metering out each item in the list one by one, apply gives the entire contents of a list as an argument all at once. As an example, you could use apply in the Mdist function to add distances.

```
;Program to measure non-sequential distances
(defun MDIST (/ dstlst dst)
  ;while loop to obtain list of points-------------------------------------
  (while (setq dst (getdist "\nPick distance or Return to exit: "))
         (Setq dstlst (append dstlst (list dst)))   ;append new point to list
         (princ (apply '+ dstlst))                  ;print current total
  );end while;------------------------------------------------------------
);end MDIST
```

Figure 9.3: The Mdist function using the apply function.

In this example, apply is given the list of distances dstlst which it applies to the plus function. If you load and run this program, it works no differently from the earlier version of MDIST.

Using Complex Lists to Store Data

In chapter 8, you created an error function that reset the osmode system variable when an error occurred. The problem with that error function was that it was too specific. It only worked for certain conditions namely, to restore the osmode system variable to its previous setting. But you can create a function that will help you handle system variable settings in a more general way using the length function.

The length function allows you to find the length of a list. This function is often used in conjunction with the repeat function to process a list. The following function converts a list of system variables to a list containing both the variable and its current setting:

```
(defun GETMODE (mod1)
(setq *mod2 '())
(repeat (length mod1)
(setq *mod2
(append *mod2
(list (list (car mod1) (getvar (car mod1))))
)
)
(setq mod1 (cdr mod1)))
)
```

Using this function, you can store the current system variable settings in a list. Figure 9.4 shows the C:BREAK2 program from chapter 8 with modifications to add the getmode function.

```
;function to save system variable settings---------------------------------
(defun GETMODE (mod1)
   (setq *mod2 '())                ;create global variable to store settings
   (repeat (length mod1)           ;find length of variable list and repeat
      (setq *mod2                  ;build *mod2 list
            (append *mod2
                (list (list (car mod1) (getvar (car mod1))))
            )
      )
      (setq mod1 (cdr mod1))       ;go to next element in list
   );end repeat
)

;function to restore system variable settings-----------------------------
(defun SETMODE (mod1)
   (repeat (length mod1)                   ;find length of list and repeat
      (setvar (caar mod1) (cadar mod1))    ;extract setting info and reset
      (setq mod1 (cdr mod1))               ;go to next element in list
   );end repeat
)
;function for error trap --------------------------------------------------
(defun *error* (msg)
  (setmode *mod2)                          ;reset system variables
  (princ msg)                              ;print error message
  (princ)
)

;program to break circle into two arcs-------------------------------------
(defun C:BREAK2 (/ pt1 pt2 pt3 pt4 pt0 ang1 dst1)
   (getmode '("osmode" "orthomode" "cmdecho"))        ;saves system vars.
   (mapcar 'setvar '("osmode" "orthomode" "cmdecho")  ;set vars. for funct.
           '(512 0 0)
   )
   (setq pt1 (getpoint "\nSelect object: "))          ;get first break point
   (setq pt2 (getpoint pt1 "\nEnter second point: ")) ;get second break point
   (setvar "osmode" 128)                              ;perpend osnap mode
   (Setq pt3 (getpoint pt1 "\nSelect parallel line: "));get 2nd line
   (setq ang1 (angle pt1 pt3))                        ;find angle btwn lines
   (setq dst1 (distance pt1 pt3))                     ;find dist. btwn lines
   (setq pt4 (polar pt2 ang1 dst1))                   ;derive pt4 on 2nd line
      (command
            "break" pt1 pt2                         ;break 1st line
            "break" pt3 pt4                         ;break 2nd line
            "line" pt1 pt3 ""                       ;close ends of lines
            "line" pt2 pt4 ""
      )
   (setmode *mod2)                                  ;reset system vars.
)
```

Figure 9.4: Revised BREAK2 program

The following expression has been added to the BREAK2 program:

(getmode '("osmode" "orthomode" "cmdecho"))

Here, a list of system variables is supplied as an argument to the getmode function. The following explains what getmode does with this list.

The first expression in getmod creates a list to be appended to. The second expression is a recursive one using the repeat function:

(setq *mod2 '())

(repeat (length mod1)

Repeat uses an integer argument to determine the number of time it is to repeat the evaluation of its other arguments. Here, length is used to find the length of mod1 which is a local variable that has the list of system variables passed to it. Length finds the number of elements in the list mod1, which in our example is 3, and passes that value to the repeat function (see Figure 9.5).

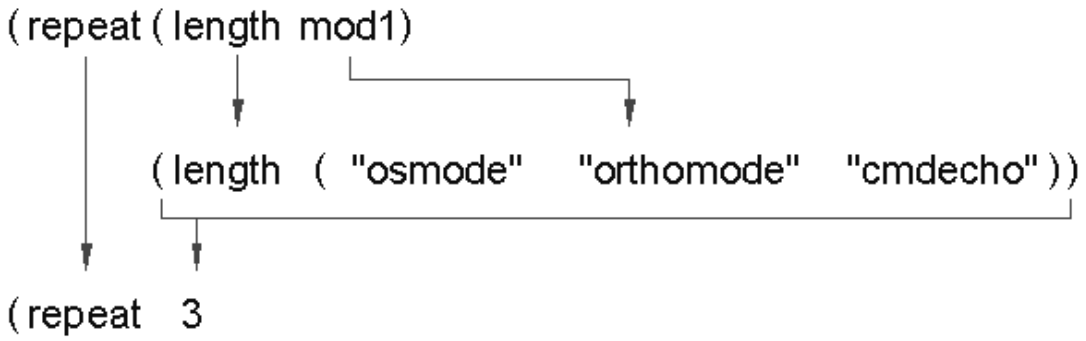


Figure 9.5: Using length and repeat in the getmode function

Repeat then processes the following set of expressions 3 times:

(setq *mod2

(append *mod2 (list (list (car mod1) (getvar (car mod1)))))

)

(setq mod1 (cdr mod1)))

This set of expressions takes an element of the list mod1 and finds the current setting for that element.

(getvar (car mod1))

Then the setting value is combined with the setting name to form a two element list:

(list (car mod1) (getvar (car mod1)))

This new list is then appended to the list *mod2 and *mod2 is assigned the value of the new appended list:

(setq *mod2

(append *mod2 (list (list (car mod1)(getvar car mod1)))))

)

Finally the first element of mod1 is removed in preparation for the next iteration:

(setq mod1 (cdr mod1)))

Remember that cdr returns a copy of a list with its first element removed. The whole process is then repeated again.

When the getmode is done, a global variable called *mod2 is created. Mod2 might look like this:

(("osmode" 0)("orthomode" 1)("cmdecho" 1))

In this list, each element is another list that contains the mode as its first element and its current setting as its second element.

Once the desired settings are saved, you can go on to change the settings to suite your program. In the case of the C:BREAK2 program, "osmode" is changed to 512, the nearest setting, "orthomode" is set to 0, which turns off the orthomode, and "cmdecho" is set to 0 which controls command echoing to the prompt line.

When BREAK2 has done its work, you need a way to restore your saved settings. The setmode function in figure 9.4 will restore the settings saved by the Getmodes function:

(Defun setmode (mod1)

(repeat (length mod1)

(setvar (caar mod1)(cadar mod1))

(setq mod1 (cdr mod1))

)

)

This function also uses length and repeat to perform a recursion. In this case, the recursive function consist of two expressions:

(setvar (caar mod1) (cadar mod1))

(setq mod1 (cdr mod1))

The first of these expressions takes the first element of the list of saved settings then applies its two elements to setvar. In our example, the result of this combination looks like this:

(setvar (caar mod1)(cadar mod1)) = (setvar "osmode" 0)

The result is an expression that sets "osmode" to 0. The next expression removes the first element from the list of settings then the processes is repeated.

The setmode function is placed at the end of the C:BREAK2 program to reset the variables. It is also placed in the *error* function. As long as getmode is used in a function to save system variables, The *error* function shown in figure 9.4 will work to restore system variables in the event of a canceled AutoLISP program. With the addition of the getmode and setmode functions, you have a general system for maintaining system variables.

Using Lists for Comparisons

Like mapcar, foreach applies individual elements of a list to a function. But foreach only accepts one list. Foreach is often used to test elements of a list for a particular condition. For example, you could test a list of coordinates to sort out those above another datum point:

(foreach n

'((4.00 1.00) (10.09 1.01) (11.96 6.80)

(7.03 10.38) (2.11 6.79) (4.00 1.00))

(if (> (cadr n) 2)(Setq newlist (append newlist (list n))))

)

This function simply checks the y value of each coordinate against the value 2. If y is greater than 2, the coordinate is added to a new list that contains only coordinates whose y value is greater than 2.

Locating Elements in a List

You won't always want to use all the elements of a list in your programs. In many instances, you will want to obtain a specific element from a list. There are two functions, member and nth, that can help you find specific elements in a list.

Note:

In the following discussion, an obsolete component of AutoLISP called Atomlist will be discussed. You can think of Atomlist as a very big list. Just think of Atomlist as any list in which you want to place a marker that you can later refer to..

Searching Through Lists

To see how these two functions work, look at a program called Clean. Clean is an older program intended for earlier versions of AutoCAD that had limited memory resources. The purpose of Clean was to remove unused user-defined functions in order to free-up memory.

```
;program to clean symbols and functions from atomlist and close open files
;-------------------------------------------------------------------------
(defun C:CLEAN (/ i item)
 (setq i 0)                                              ;set up counter
   ;while not at the end of atomlist do...
   (while (not (equal (setq item (nth i atomlist)) nil))
     (if (= (type (eval item)) 'FILE)                    ;if item is a file
       (close (eval item))                               ;close the file
     );end IF
   (setq i (1+ i) )                                      ;add 1 t counter
   );end WHILE
 (setq atomlist (member 'C:CLEAN atomlist))              ;redefine atomlist
'DONE                                                    ;without symbols
)                                                        ;previous to C:CLEAN
                                                         ;and print DONE
```

Figure 9.6: The CLEAN program.

Before we analyze this program, we must first discuss the atomlist. Atomlist was a special list used to store data in earlier versions of AutoCAD. It contains the names all of the built-in AutoLISP functions. It was also used to store any user defined functions and symbols. If you are using Release 11 or earlier, you can view its contents by first flipping the screen to text mode, then entering:

!atomlist

You will get a listing of all the user defined and built in AutoLISP functions currently loaded into AutoCAD.

The list contains all of the AutoLISP functions we have discussed so far plus a few we have not. If you are using

Acad.lsp to load some of your own functions, they will also appear at the top of the list. Whenever you create a new function or symbol, it is added to the beginning of atomlist. If you have an older version of AutoCAD, try entering the following:

(setq myfunc "mufunc")

If you enter **!atomlist**. you will see that myfunc is added to the list. The more functions you add, the larger atomlist gets and more memory is used to store symbols and functions.

If the C:CLEAN function is included in your Acad.lsp file, it is also added to the atomlist at startup time. C:CLEAN's purpose is twofold. First, it closes any files that may have been opened and inadvertently left open. This might occur if the open function was used to open an ASCII file and due to a function being canceled, the file was never closed. As we mentioned, this can cause loss of data to the open file. Second, C:CLEAN clears the atomlist of any function that is added after it thereby recapturing memory space.

In order to find and close any open files, C:CLEAN uses the **while** function in conjunction with the **if** and **type** functions. First, a counter is set to 0. the symbol i is used as a counting device:

(setq i 0)

Next, a **while** expression checks to see if an item of the atomlist is equal to nil. The nth function is used to read each element of atomlist.

(nth i atomlist)

Nths syntax is:

(nth [integer][list])

where integer is the numeric position of an element in list. In C:CLEAN, nth returns the element whose position within atomlist is represented by i. The variable i is a counter to which 1 is added each time the while function loops through the expressions it contains. The net result is that each element of the list atomlist is compared to nil. While continues to loop through its expressions until such a condition is met (see Figure 9.7).

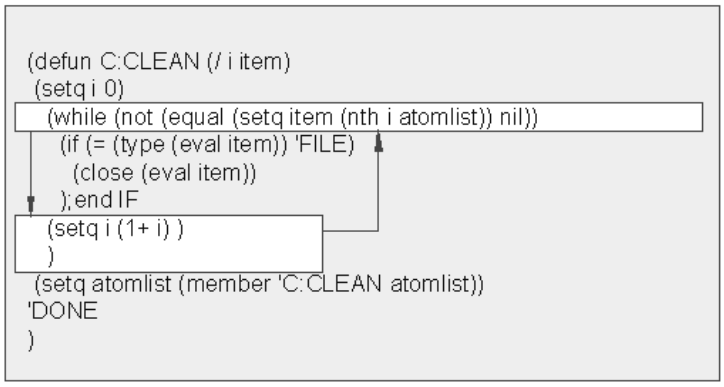


Figure 9.7: The while loop in C:CLEAN

Notice that the element returned by nth is assigned to the variable item. This allows the next if conditional function to test the element to see if it is a file descriptor:

(if (= (type (eval item)) 'FILE)

The eval function is used to force an evaluation of the variable item to extract its value. Eval can be used to find the value of a nested symbol, that is, a symbol whose value is also a symbol. Eval is the basic mechanism by which AutoLISP evaluates expressions. It is always being applied to expressions by AutoLISP. You could think of eval as the opposite of quote.

The type function above returns the data type of the value of item. If eval is not used, type would return the data type of the symbol item rather than the data type of items value (see figure 9.8).

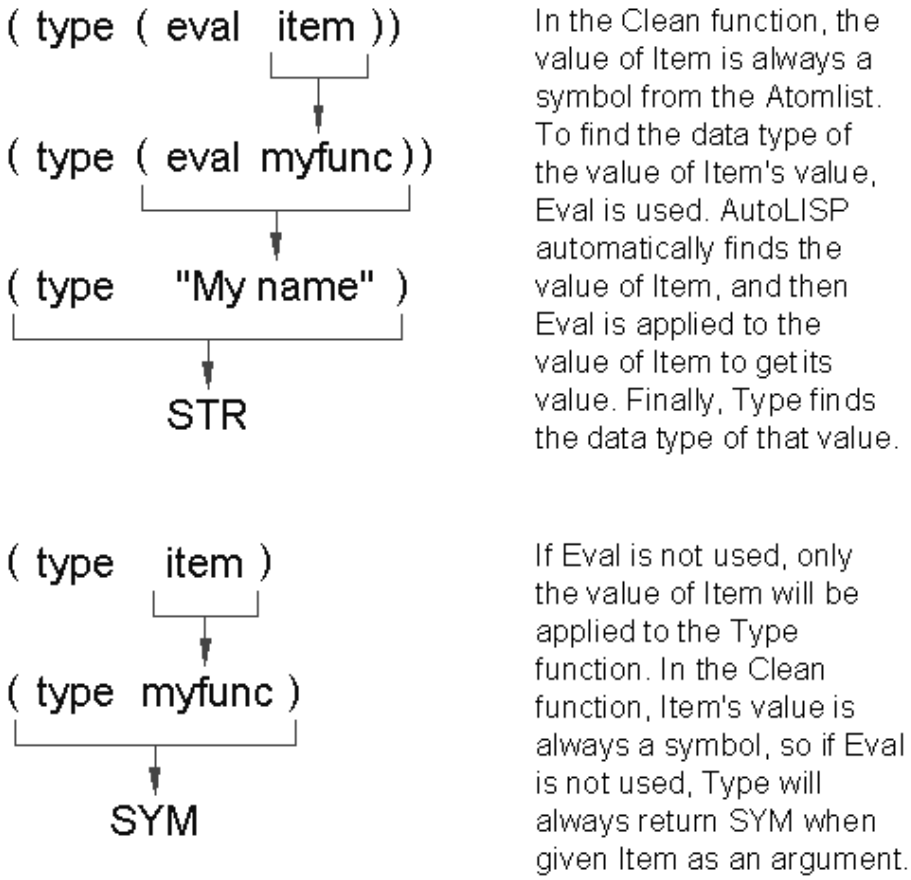


Figure 9.8: Using eval to force one level of evaluation

The following table gives a list of the values returned by type and their meaning:

Value	Meaning
REAL	real number
FILE	file descriptor
STR	string
INT	integer
SYM	symbol
LIST	list and user defined functions
SUBR	AutoLISP function
PICKSET	selection set
ENAME	object name
PAGETB	function paging table

If item turns out to be equal to a file descriptor or FILE, then the next line closes that file:

(close (eval item))

Using an Element of a List as a Marker

Once C:CLEAN has finished closing any open files, then it proceeds to redefine atomlist:

(setq atomlist (member 'C:CLEAN atomlist)

Here, the function member is used to find a list whose elements are all the elements of atomlist beginning with C:CLEAN. The expression then assigns that list to atomlist. The symbol C:CLEAN acts like a marker within the list telling the member function where to begin the list. The net affect is that of clearing atomlist of all the symbols and functions that were added to atomlist after C:CLEAN.

Members syntax is:

(member [element][list])

Member returns a list whose elements are the same as those of list starting with element.

Finding the Properties of AutoCAD Objects

One of the most powerful features of AutoLISP is its ability to access the properties of drawing objects. You can find properties such as the endpoint coordinates of lines, their layer, color and linetypes, and the string value of text. You can also directly modify these properties.

Object properties are accessed using two AutoLISP data types, object names and selection sets. Object names are similar to symbols in that they are a symbolic representation of an object. An object name is actually a device used to point to a record in a drawing database. This database record holds all the information regarding the particular object. Once you knows an object name, you can access the information stored in the object's record.

Using Selection Sets and Object Names

A selection set is a collection of object names. Selection sets can contain just one object name or several. Each name in the selection set has a unique number assigned to it from zero to 1 minus the number of names in the set.

To find out how you access this object information, lets look at the C:EDTXT PROGRAM used in chapter 7.

As you may recall, this program allows you to edit a line of text without having to enter the entire line. The fourth line in C:EDTXT does the work of actually extracting the text string from the database:

```
(setq oldtxt (gettxt))
```

Here, the program makes a call to a user defined function called gettxt:

```
(defun gettxt ()

(setvar "osmode" 64)

(setq pt1 (getpoint "\nPick text to edit: "))

(Setvar "osmode" 0)

(setq oldobj (entget (ssname (ssget pt1) 0) ))

(setq txtstr (assoc 1 oldobj))

(cdr txtstr)  )
```

Gettxt first finds a single point pt1 that locates the text to be edited:

```
(setvar "osmode" 64)

(setq pt1 (getpoint "\nPick text to edit: "))

(Setvar "osmode" 0)
```

Next, the real work of finding the object is done. The next line uses several functions to extract the object name from the drawing database:

(setq oldobj (entget (ssname (ssget pt1) 0)))

This innocent looking set of expressions does a lot of work. It first creates a selection set of one object:

(ssget pt1)

```
;function to find text string from text entity------------------------------
(defun gettxt ()
(setvar "osmode" 64)                                ;set osnap to insert
(setq pt1 (getpoint "\nPick text to edit: "))       ;get point on text
(Setvar "osmode" 0)                                 ;set osnap back to zero
(setq oldobj (entget (ssname (ssget pt1) 0) ))      ;get entity zero from prop.
(setq txtstr (assoc 1 oldobj))                      ;get list containing string
(cdr txtstr)                                        ;extract string from prop.
)
;function to update text string of text entity------------------------------
(defun revtxt ()
(setq newtxt (cons 1 newtxt))                       ;create replacement propty.
(entmod (subst newtxt txtstr oldobj))               ;update database
)
;program to edit single line of text----------------------------------------
(defun C:CHTXT (/ count oldstr newstr osleng otleng oldt old1
                  old2 newtxt pt1 oldobj txtstr oldtxt)
(setq count 0)                                      ;setup counter to zero
(setq oldtxt (gettxt))                              ;get old string from text
(setq otleng (strlen oldtxt))                       ;find length of old string
(setq oldstr (getstring T "\nEnter old string "))   ;get string to change
(Setq newstr (getstring T "\nEnter new string "))   ;get replacement string
(setq osleng (strlen oldstr))                       ;find length of substring-
  ;while string to replace is not found, do...       to be replaced
  (while (and (/= oldstr oldt)(<= count otleng))
    (setq count (1+ count))                         ;add 1 to counter
    (setq oldt (substr oldtxt count osleng))        ;get substring to compare
  );end WHILE
  ;if counting stops before end of old string is reached...
  (if (<= count otleng)
        (progn
           (setq old1 (substr oldtxt 1 (1- count))) ;get 1st half of old string
           (setq old2 (substr oldtxt (+ count osleng) otleng));get 2nd half
           (setq newtxt (strcat old1 newstr old2))  ;combine to make new string
           (revtxt)                                 ;update drawing
        )
        (princ "\nNo matching string found.")       ;else print message
  );end IF
(PRINC)
);END C:EDTXT
```

Figure 9.9: The C:EDTXT program

You may recall from chapter 4 that ssget accepts a point location, to find objects for a selection set. If you entered the expression above at the command prompt, and pt1 has been previously defined as a point nearest an object, you would get the name of a selection set. Try the following exercise:

1. Open an AutoCAD file and place the following text in the drawing:

   For want of a battle, the kingdom was lost.

2. Enter the following expression:

   (setq pt1 (getpoint "\nPick the text: "))

3. Use the insert osnap override option from either the side or pull down menu and pick the text.

4. Enter the following expression:

   (ssget pt1)

   You will get a message that looks similar to the following:

   <Selection set: 1>

   This is a selection set. The number following the colon in the above example would be different depending on whether previous selections sets have been created.

5. Once a selection set has been created, ssname is used to find the object name. Enter the following:

   (ssname (ssget pt1) 0)

Ssname will return the object name of a single object in the selection set. If you enter the expression above, you get an object name that looks similar to the following:

<Object name: 600000c8>

Just as with selection sets, the number that follows the colon will different depending on the editing session.

The syntax for ssname is:

(ssname [selection set][integer])

The selection set can be gotten from a symbol representing the selection set or directly from the ssget function as in our example above. The integer argument tells ssname which object to select within the selection set. In our example, there is only one object, so we use the integer 0 which represents the first object in a selection set. If there were several objects in the selection set, say 4, we could use an integer from 0 to 3.

At the next level, the function entget does the actual database extraction. Enter the following:

(setq oldobj (entget (ssname (ssget pt1) 0)))

You will get a list revealing the properties of the text.

((-1 . <Entity name: 20a0598>) (0 . "TEXT") (5 . "33") (100 .

"AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDbText") (10 -0.147023 2.84992 0.0)

(40 . 0.2) (1 . "For want of a battle, the kingdom was lost") (50 . 0.0) (41 .

1.0) (51 . 0.0) (7 . "STANDARD") (71 . 0) (72 . 0) (11 0.0 0.0 0.0) (210 0.0

0.0 1.0) (100 . "AcDbText") (73 . 0))

This list obtained using entget is called a property list. Entgets' syntax is:

(entget [object name])

Entget returns a list containing the object's properties. Property lists consists of other lists whose first element is an integer code. The code represents a particular property like an objects layer, color, linetype or object type. Property lists are a class of list called association lists.

You may recall that earlier in this chapter, you constructed a list of system variables. that list looked like the following:

(("osmode" 0)("orthomode" 1)("cmdecho" 1))

This is also an association list. each element of the list is a list of two elements, the first of which can be considered a keyword.

Each list within an object's property list starts with an integer code. That integer code is the key-value to that list otherwise known as the group code. The group code is associated with a particular property. For example, the group code 1 in associated with the string value of a text object. The 10 group code is associated with the insertion point of the text. Table 9.2 shows the group codes for text and their meaning.

Code	Meaning
-1	Entity name
0	Entity type ("TEXT", "LINE", "ARC", etc.
7	Text style
8	Layer
10	Insertion point
11	Center alignment point (for centered text)
21	Right alignmenet point (for right-justified text)
31	Second alignment point (for fit or aligned text)
40	Text height
41	X scale factor
50	Rotation angle
51	Oblique angle
71	Text mirror code (2, mirrored in x axis; 4, mirrored in y axis)
72	Text alignment code (0, left justified; 1, center at baseline;2, right justified; 3, text uses align option; 4, centered at middle; 5, text uses fit option)
210	3-D extrusion amount in x, y, or z direction

If you are familiar with the AutoCAD DXF file format and coding system, then these group codes should be familiar. Appendix C gives a detailed listing of these codes if you want to know more.

Now that our expression has retrieved the property list, we need a way to pull the information out of the list. A function for this purpose is found in the next line. Enter the following:

(setq txtstr (assoc 1 oldobj))

In the previous expression, the property list is assigned to the variable oldobj. In the above expression, we see a new function called assoc:

(assoc 1 oldobj)

This expression returns the list:

(1 . "For want of a battle, the kingdom was lost")

Remember that oldobj is the variable for the property list of the text you selected earlier.

The Syntax for Assoc is:

(assoc [key-value] [association list])

Assoc looks through an association list and finds the list whose first value is the key-value. It then returns the list containing the key-value.

In the case of our property list example, assoc looks through the property list oldobj and finds the list whose first element is the group code 1 then it returns that list. The list returned by assoc is assigned to the symbol txtstr. Finally, cdr is applied to txtsrt to obtain the string value of the selected text. Enter the following:

(cdr txtstr)

The string value associated with the group code 1 is retrieved. Figure 9.10 diagrams the entire operation.

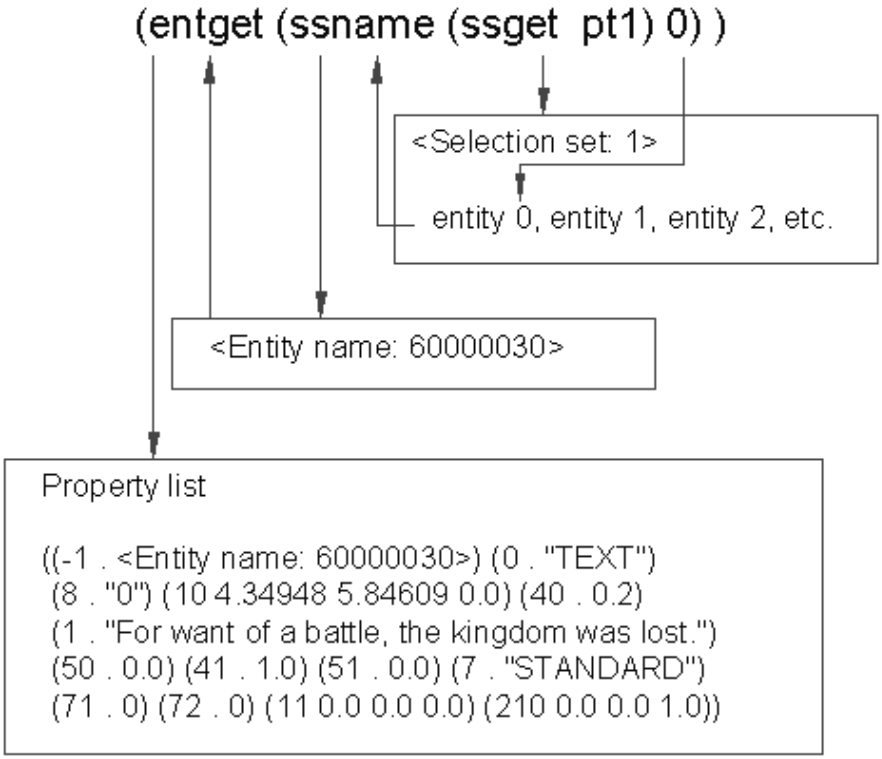


Figure 9.10: Diagram of property list extraction

In summary, to find a particular property of an object, you must first create a selection set containing that object using ssget, then extract the object name from the selection set using ssname, then extract the property list using the object name through the function entget. Once you have gotten the property list, you can apply assoc to it to get the specific property you want using group codes. Finally, cdr can be applied to the singled out propert to get the value of the property.

Understanding the structure of Property Lists

The first thing you might have notice about the property list in the example above is that most of the sublists were two element lists with a period separating the elements. This type of list is called a dotted pair. It is not a list in the true sense of the term because many of the functions used to manipulate lists will not work on dotted pairs. For this reason, dotted pairs are usually considered a data type in themselves.

You can use car and cdr on dotted pairs just as you would on lists. for example, you could enter the following:

(car '(A . B))

the symbol A is returned. You can also enter:

(cdr '(A . B))

and the symbol B is returned. Dotted pairs act slightly differently from regular lists. If cdr is applied to a normal list, as the following:

(cdr '(A B))

a list, (B), is returned. But in the case of a dotted pair, the second element of the dotted pair is returned by itself, not as part of a list.

You can create a dotted pair using the cons function. Normally, cons must have two arguments. The first is the element to be added to the beginning of a list and the second is the list to be added to. Enter the following:

(cons 'A '(B))

The list (A B) is returned. You could think of cons as the opposite of cdr. But if the second argument to cons is not a list, then a dotted pair is created. Enter the following:

(cons 'A 'B)

The dotted pair (A . B) is returned.

Cons and dotted pairs reflect the inner workings of AutoLISP and to explain these AutoLISP items thoroughly is beyond the scope of this book. At the end of chapter 10, we mention a few sources for more information on the general subject of LISP which can shed light on Cons and dotted pairs. For now, lets continue by looking at a function that allows us to directly modify the AutoCAD drawing database.

Changing the properties of AutoCAD objects

Now that you have seen how object properties are found, it is a short step to actually modifying properties. To update an object record in the drawing database, you redefine the objects property list then use the function entmod to update the drawing database. Looking at the edtxt.lsp file again, we find the revtxt function:

(defun revtxt ()

(setq newtxt (cons 1 newtxt))

(entmod (subst newtxt txtstr oldobj))

)

The first thing revtxt does is use the cons function to create a dotted pair using the integer 1 for the first element and the string value held by newtxt for the second. Newtxt is a string value representing the new text that is to replace the old text oldobj. As we mentioned earlier, cons creates a dotted pair when both its arguments are atoms. The new dotted pair looks like this:

(1 . "For want of a nail, the kingdom was lost.")

Notice that the structure of this list is identical to the list oldobj which was retrieved from the text property list earlier.

The last line of the revtxt function does two things. first it uses the function subst to substitute the value of newtxt for the value of txtstr in the property list oldobj:

(subst newtxt txtstr oldobj)

Substr requires three arguments. the first is the replacement item, the second is the item to be replaced, and the third is the list in which the item to be replaced is found. Substr's syntax is as follows:

(subst [replacing item][item to be replaced][list containing item])

Subst returns a list with the substitution made.

Next, the function entmod updates the drawing database. It looks at the object name of the list that is passed to it as an argument. This list must be in the form of a property list. It then looks in the drawing database for the object name that corresponds to the one in the list. When it finds the corresponding object in the drawing database, it replaces that database record with the information in entmod's property list argument. The user sees the result as a new line of text.

Getting an Object Name and Coordinate Together

In the gettxt function, a function could have been used that doesn't require the you obtain a selection set. Entsel will find a single object name directly without having to use ssget to create a selection set. Since Entsel only allows the user to pick a single object, it is best suited where a program or function doesn't require multiple selections of objects.

Here is gettxt using entsel:

```
(defun gettxt ()

(setq oldobj (entget (car (entsel "\nSelect object: "))))

(setq txtstr (assoc 1 oldobj))

(cdr txtstr)

)
```

Entsel acts like the get functions by allowing you to provide a prompt string. Instead of returning a number, string, or point, entsel returns a list of two elements. The first element is an object name, and the second is a list of coordinates specifying the point picked to select the object:

```
(<Object name: 60000012> (4.0 3.0 0.0))
```

Above is a sample of what is returned by entsel. Since our Gettxt function is only concerned with the object name, car is used on the value returned from entsel:

```
(car (entsel "\nSelect object: "))
```

This expression replaces the ssget and ssname function used previously:

```
(ssname (ssget pt1) 0)
```

Also, since entsel pauses to allow the user to select an object, the getpoint expression can be eliminated along with the setvar function.

Conclusion

You have seen how lists can be used to manipulate data both as forms and as simple lists of information. AutoLISP makes no distinction between a list that is an expression to be evaluated and a list that is used to store data. Once you understand the methods for manipulating list, you can begin to develop some powerful programs.

In the next chapter, you will look in more detail at how you can access information directly from AutoCAD. You will also look at how to access the property of complex objects such as blocks an polylines.

Chapter 10: Editing AutoCAD objects

Introduction

In the last chapter you were introduced to the ssname and entget functions that, together with ssget, allowed you to extract information about an object from the drawing database. In this chapter, you will learn how to perform operations on several objects at once. Also, you will look at how to obtain information regarding a drawing's table information which consists of layers and their settings, viewport, UCSs and other system options.

There are actually several functions that allow you to access the AutoCAD drawing database directly. Table Lists the functions and gives a brief description of each.

Function	Description
(entnext [object name])	If used with no argument, entnext will return the object name of the first object in the database. If an object name is given as an argument, entnext returns the first sub-object of object name. A sub-object is an object contained in a complex object such as a polyline vertex or a block attribute.
(entlast)	Returns the object name of the last object added to the drawing database.
(entsel [prompt])	Prompts the user to select an object then returns a list whose first element is the object's name and whose second element is the pick point used to select the object. A prompt can be optionally added. If no prompt is used, the prompt "Select object:" is given automatically.
(handent handle)	Returns an object name given an object's handle.
(entdel object name)	Deletes object name. If object has previously been deleted in the current editing session, then the object named will be restored.
(entget object name)	Returns the property list of object name
(entmod property list)	Updates the drawing database record of the object whose object name appears in the property list. The object name is the -1 group code sublist of the property list.
(entupd object name)	Updates the display of polyline vertices and block attributes that have been modified using entmod.

You have already seen first hand how a few of these functions work. In this and the following chapter, you will explore the use of several more of these very powerful editing tools.

Editing Multiple objects

The Edtxt program you looked at in the last chapter used ssget to obtain a single object. However, ssget is really better suited to obtaining multiple sets of objects. You can use groups of objects collected together as selection sets to perform some operation on them all at once.

To examine methods for editing multiple objects, we will look at a program that offers an alternate to the Extend command. Though Extend allows you to extend several objects, you have to pick each object individually. Picking objects individually allows for greater flexibility in the type of object you can extend and the location of the extension. However, there are times when you will want to perform a multiple extend operations like extending several lines to a another line.

You can use AutoLISPs' ssget function to help you simplify the processing of multiple objects. Figure 10.1 shows a sketch of how a multiple extend program might work manually and Figure 10.2 shows the actual program derived from that sketch.

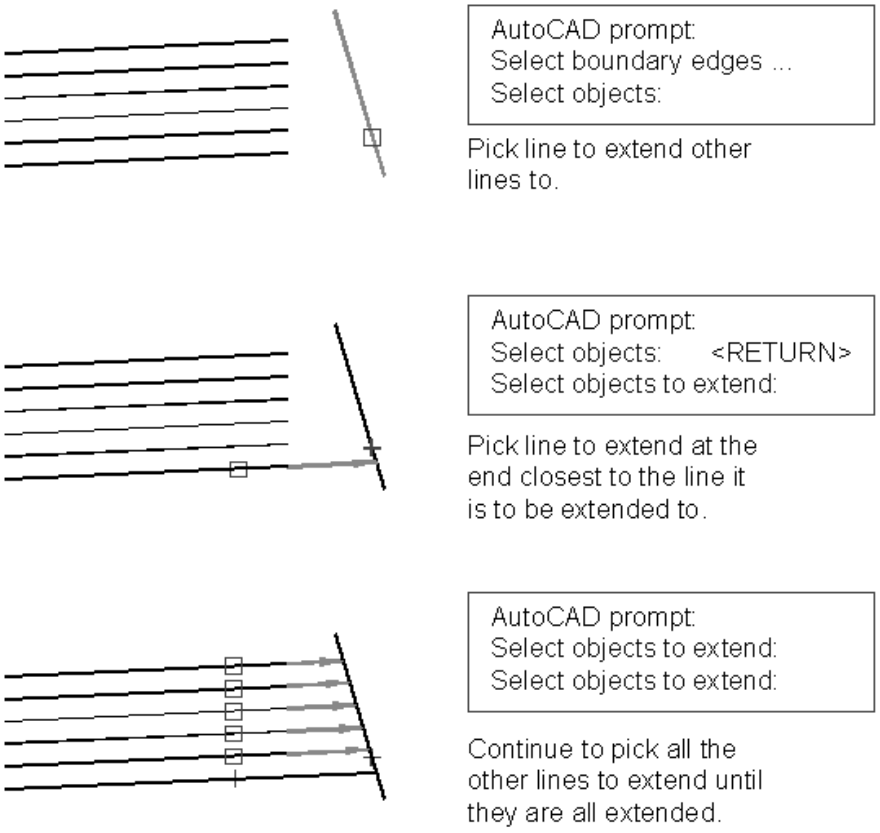


Figure 10.1: Sketch of process using extend.

```
;Program to extend multiple lines - Mlext.lsp  ------------------------------

(defun c:MLEXT (/ x y sset1 count pt1 pt2 int obj elst)
(graphscr)                                          ;shift to graphics
(princ "\nSelect Boundary edge...")                 ;print prompt
(setq obj (car (entsel)))                           ;Get entity name
(setq x (getpoint "\nPick axis crossing lines to extend: "))
(setq y (getpoint x "\nPick endpoint: "))           ;get axis crossing lines
          (setq sset1 (ssget "c" x y))              ;get entities to extend
          (setq count 0)                            ;set counter
             (if (/= sset1 nil)                     ;test for selection set
                 (while (< count (sslength sset1))  ;while still select. set
                     (setq elst (entget (ssname sset1 count))  ;get entity name
                           pt1  (cdr (assoc 10 elst))  ;get one endpoint
                           pt2  (cdr (assoc 11 elst))  ;get other endpoint
                           int  (inters x y pt1 pt2)   ;find intersection
                     );end setq                          of axis and line
                     (command "extend" obj "" int "")  ;command to extend line
                     (setq count (1+ count))           ;go to next line count
                 );end while
             );end if
);end defun
```

Figure 10.2: The Mlext.lsp file

1. Open a file called Mlext.lsp and copy the program from figure 10.2 into your file. Start AutoCAD and open a new file called chapt10. Remember to add the equal sign at the end of the file name.

2. Draw the drawing shown in figure 10.3. The figure indicates the coordinate location of the endpoints so you can duplicate the drawing exactly. Do not include the text indicating the coordinates.

3. Load the Mlext.lsp file and enter **mlext** at the command prompt. When you see the prompt

Select boundary edges...

Select object:

pick the line labeled boundary edge in figure 10.3. At the next prompt

Pick axis crossing lines to extend:

4. A rubber banding line representing the crossing axis appears. Pick a point at coordinate 8,2. You can use on the snap mode and dynamic coordinate readout to help you locate this point.

5. At the next prompt

Pick endpoint:

pick a point at coordinate 8,9. The lines that are crossed by the axis are extended to the boundary edge line. Lets look at how this program works.

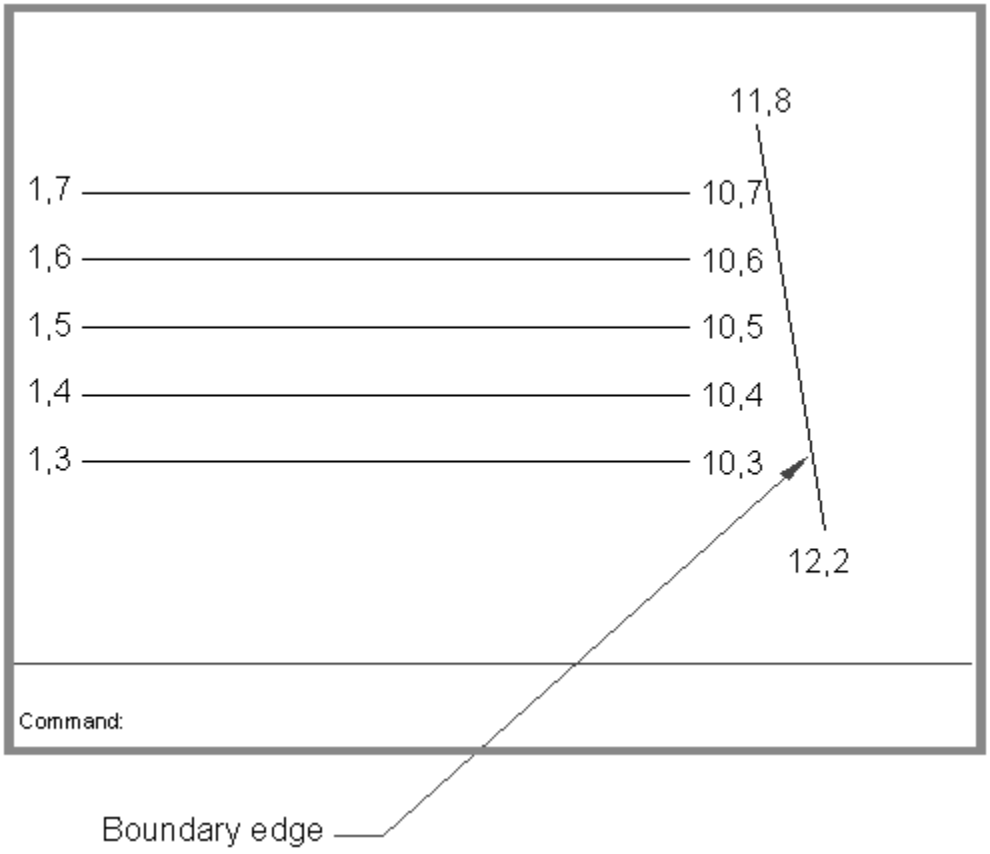


Figure 10.3: Lines drawn in the chapt10 file

Let's look at how this program works. The first expression in the program is:

(graphscr)

This simply flips the display to the graphics screen in case the user is currently in Text mode. Another function

called Textscr does just the opposite. It flips the display into text mode if the current screen is in a graphics mode.

Next, the Mlext finds the line used to extend to:

```
(princ "\nSelect boundary edge...")

(setq obj (car (entsel)))
```

Here we use entsel to obtain the object name. Since entsel returns a two element list containing the name and pick coordinate, we use car to extract the object name from the list. Also note that we use a prompt similar to the one used by the extend command.

The next several lines creates a selection set of lines to extend:

```
(setq x (getpoint "\nPick axis crossing lines to extend: "))

(setq y (getpoint x "\nPick endpoint: "))

(setq sset1 (ssget "c" x y))
```

Here two points are obtained indicating an axis along which the lines to be extended lie. These points will be used later to help find other pick points. Then ssget is used with the crossing option to create a selection set. The two points defining the axis are used as the two corner points of the crossing window.

The next two lines do some setup work:

```
(setq count 0)

(if (/= sset1 nil)
```

The first of these two lines set a counting variable to zero. The next line checks to make sure a set of objects has indeed been selected and a selection set created.

Once these things have been established, the actual work is done:

```
(while (< count (sslength sset1))

(setq elst (entget (ssname sset1 count))

(setq pt1 (cdr (assoc 10 elst)))

(setq pt2 (cdr (assoc 11 elst)))

(setq int (inters x y pt1 pt2))

(command "extend" obj "" int "")

(setq count (1+ count))

);end while
```

This while expression is evaluated until the counter variable **count** reaches the total number of objects in the selection set **sset1**. Each time the expression is evaluated, **count** is increased by one until count equals the length of the selection set. Lets look at what each iteration of the while expression does.

Finding the Number of Objects in a Selection Set

First, the while expression checks to see if the counting variable **count** is less than the total number of elements in the selection set sset1. This is done through the sslength function:

(while)< count (sslength ssget1))

Sslength simply returns the number of objects in its selection set argument. The argument can be the actual selections set or a symbol representing it. In our example, the symbol ssget1 is used. If count is less than the number of objects, we know that we haven't processed all the objects in the selection set, in which case, the expressions that follow are evaluated.

Next, the variable elst is given the object name of an object in the selection set:

(setq elst (entget (ssname sset1 count)))

The count variable is used by ssname to determine which object in the selection set sset1 is to be examined. Entget then extracts the property list for that object and this list is assigned to the variable elst.

Next, the two endpoints of the objects are extracted:

(setq pt1 (cdr (assoc 10 elst)))

(setq pt2 (cdr (assoc 11 elst)))

Here the Assoc function is used to extract the two endpoint coordinates from the property list. The group codes 10 and 11 are used by Assoc to locate the sublist in elst containing the coordinates in question (see appendix C for a full list of group codes and their meaning). These coordinates are assigned to variables pt1 and pt2.

Once the endpoint's coordinates are found, a function called inters is used to find the intersection point of the current object being examined and the object crossing axis derived in the beginning of the function:

(setq int (inters x y pt1 pt2))

Inters is a function finds the intersecting point of two pairs of coordinates. Inters syntax is:

(inters x_1 y_1 x_2 x_2)

where x_1 and y_1 are the x and y coordinates of one axis and x_2 and y_2 are the coordinates of the second axis. Inters returns a list containing the coordinates of the intersection of the two axes. In the Mlext program, this list is assigned to the variable int.

Finally, the command function is used to invoke the extend command and extends the current object:

(command "extend" obj "" int "")

The first thing that the extend command asks for is the line to extend to. Here, obj is used to indicate that line. A return is issued to end the selection process then a point value is entered to indicate both the line to be extended and the location of the extend side. In this case, the intersection point of the line and the extend axis is used for this purpose. Finally, a return is issued to end the extend command.

The last line in the while expression increases the value of count by one in preparation to get the next object in the selection set.

(setq count (1+ count))

If count is still less than the number of objects in the selection set, the process repeats itself.

Since the Extend and Trim commands work in a nearly identical way, you can create a program that performs both multiple extends or trims on lines by changing just a few elements in the Mlext program. Figure 10.4 shows such a program called Etline. The elements that are changed from Mlext are indicated in the comment.

```
;program to extend or trim multiple lines --Etline.lsp-----------------------

(defun c:ETLINE (/ x y u sset1 count pt1 pt2 int obj)
(graphscr)                                          ;shift to graphics
(initget "Extend Trim")                            ;ADDED set keywords
(setq EorT (getkword "\Extend or <Trim>: "))       ;ADDED select operation
(if (equal EorT "")(setq EorT "Trim"))             ;ADDED test operation choice
(princ "\nSelect boundary edge...")                 ;print prompt
(setq obj (car (entsel)))                           ;Get entity name
(setq x (getpoint "\nPick axis crossing lines to edit: "))
(setq y (getpoint x "\nPick endpoint: "))           ;get axis crossing lines
          (setq sset1 (ssget "c" x y))              ;get entities to extend
          (setq count 0)                            ;set counter
             (if (/= sset1 nil)                     ;test for selection set
                 (while (< count (sslength sset1))  ;while still select. set
                     (setq elst (entget (ssname sset1 count))  ;get entity name
                           pt1  (cdr (assoc 10 elst))  ;get one endpoint
                           pt2  (cdr (assoc 11 elst))  ;get other endpoint
                           int  (inters x y pt1 pt2)   ;find intersection
                     );end setq                         of axis and line
                     (if (equal EorT "Extend")   ;ADDED Test for extend choice
                      (command "extend" obj "" int "") ;extend line or...
                      (command "trim" obj "" int "");ADDED trim line
                     );end if
                     (setq count (1+ count))           ;go to next line count
                 );end while
             );end if
);end progn
```

Figure 10.4: Program to perform both extend an trim functions

Improving Processing Speed

When you begin to write program that act on several objects in a recursive fashion, speed begins to be an issue. There are two things you can do to improve the speed of such recursive program. The first is to simply set the cmdecho system variable to 0. the second is to modify the drawing database directly rather than rely on AutoCAD commands to make the changes for your. In this section, you will look at both options first hand.

Using Cmdecho to Speed up Your Program

You may have noticed that when you ran the mlext program, the commands and responses of each line edit appeared in the command prompt. The program is actually slowed by having to wait for AutoCAD to print its' responses to the prompt line. You can actually double the speed of the Mlext program by simply adding the following expression at the beginning of the program:

(setvar "cmdecho" 0)

Cmdecho is an AutoCAD system variable that controls the echo of prompts to the command prompt. When set to zero, it will suppress any AutoCAD command prompts that would normally occur when AutoLISP invokes an AutoCAD command.

Open the Mlext.lsp file and add the above line to the program. Also include the following line at the end of your program to set the cmdecho variable back to 1.

(setvar "cmdecho" 1)

Your file should look like figure 10.5. This figure shows the Mlext program with the changes indicated by comments.

```
;Program to extend multiple lines - Mlext.lsp  ------------------------------

(defun c:MLEXT (/ x y u sset1 count pt1 pt2 int obj)
(graphscr)                                          ;shift to graphics
(setvar "cmdecho" 0)                          ;ADDED   echo to prompt off
(princ "\nSelect Boundary edge...")                 ;print prompt
(setq obj (car (entsel)))                           ;Get entity name
(setq x (getpoint "\nPick axis crossing lines to extend: "))
(setq y (getpoint x "\nPick endpoint: "))           ;get axis crossing lines
          (setq sset1 (ssget "c" x y))              ;get entities to extend
          (setq count 0)                            ;set counter
             (if (/= sset1 nil)                     ;test for selection set
                 (while (< count (sslength sset1))  ;while still select. set
                     (setq elst (entget (ssname sset1 count))  ;get entity name
                           pt1  (cdr (assoc 10 elst))  ;get one endpoint
                           pt2  (cdr (assoc 11 elst))  ;get other endpoint
                           int  (inters x y pt1 pt2)   ;find intersection
                     );end setq                         of axis and line
                     (command "extend" obj "" int "")  ;command to extend line
                     (setq count (1+ count))           ;go to next line count
                 );end while
             );end if
(setvar "cmdecho" 1)                          ;ADDED   echo to prompt back on
);end defun
```

Figure 10.5: The Mlext.lsp file with the additions made.

Next, go back to the Chapt10 drawing and re-create the drawing in figure 10.3. Load and run the Mlext program as you did previously. Notice that it runs much faster and that the extend command prompts no longer appear at the command prompt. Setting Cmdecho to zero will improve the speed of any program that executes AutoCAD commands recursively.

Improving Speed Through Direct Database Access

Another method for improving speed is to make your program modify the drawing database directly instead of going through an AutoCAD command. Figure 10.6 shows a modified version of the Mlext program that does this.

```
;Function to find closest of two points-------------------------------------
(defun far ( fx fy dlxf / dst1 dst2 intx)
    (setq dst1 (distance dlxf fx))                ;find distnce to one pt
    (setq dst2 (distance dlxf fy))                ;find distnce to other pt
    ;If 1st pt.is farther than 2nd pt then eval 1st pt........
    (if (> dst1 dst2) fx fy )
)


;Proram to extend multiple lines -- Mlext2.lsp
;-----------------------------------------------------------------------------
(defun c:MLEXT2 (/ sset1 count pt1 pt2 int OBJ objx objy
                   elst int far1 sub1 sub2)
 (graphscr)
 ;Get entity list of line to be extended to then find endpoints..........
 (princ "\nSelect boundary edge...")                        ;print prompt
 (Setq obj (entget (car (entsel)))                         ;get boundary
       objx (cdr (assoc 10 obj))                           ;get 1st endpoint
       objy (cdr (assoc 11 obj))                           ;get 2nd endpoint
       sset1 (ssget)                                       ;get lines to trim
       count 0
 )                                                         ;set count to zero
    ;IF lines have been picked.........
    (if (/= sset1 nil)
        ;As long as count is less than number of objects in selection set...
        (while (< count (sslength sset1))
               ;Get intersect of two lines and find farthest endpt of line ...
               (setq elst (entget (ssname sset1 count))      ;get entity list
                     pt1  (cdr (setq sub1 (assoc 10 elst)))  ;get 1st endpoint
                     pt2  (cdr (setq sub2 (assoc 11 elst)))  ;get 2nd endpoint
                     int  (inters objx objy pt1 pt2 nil)     ;find interects
                     far1 (far pt1 pt2 int)                  ;find far point
               )
               ;IF pt1 equals point farthest from intersect..........
               (if (= far1 pt1)
                   (entmod (subst (cons 11 int) sub2 elst))  ;update pt2
                   (entmod (subst (cons 10 int) sub1 elst))  ;else update pt1
               );end IF 2
               (setq count (1+ count))                     ;add one to count
        );end WHILE
    );end IF 1
);END of defun
```

Figure 10.6: The Mlext2 program that directly modifies the drawing database.

In this section you'll enter the modified program, and then use the Chapter 10 drawing to see how the program works.

1. Exit the Chapt10 drawing and open an AutoLISP file called Mlext2.lsp.

2. Copy the program in figure 10.6 into the file. Save and exit Mlext2.lsp.

3. Return to the Chapt10 drawing. Once again, reconstruct the drawing shown in figure 10.3.

4. Load and run Mlext2.lsp.

5. At the first prompt:

   Select boundary edges...

   Select object:

6. Pick the boundary edge line indicated in figure 10-3. At the next prompt:

   Select objects:

   enter a **C** to use a crossing window.

   7. Pick the two points indicated in figure for the corners of the crossing window. The lines will extend to the boundary edge line.

Notice that the extension operation occurred much faster than before. Since the program doesn't have to go through an extra level of processing, namely the AutoCAD Extend command, the operation occurs much faster. Lets look at how the program was changed to accomplish the speed gain.

You might first notice the function far added to the program file. We will look at this function a bit later. The beginning of the program shows some immediate changes.

```
(defun c:MLEXT2 (/ sset1 count pt1 pt2 int OBJ objx objy

elst int far1 sub1 sub2)

(graphscr)

(princ "\nSelect boundary edge...")

(Setq obj (entget (car (entsel)))

(setq objx (cdr (assoc 10 obj)))

(setq objy (cdr (assoc 11 obj)))
```

Instead of simply obtaining the object name of the boundary edge line, we extract the endpoint coordinates of that line and set the coordinates to the variables objx and objy. These endpoints are used later in conjunction with the

inters function to find the exact point to which a line must be extended.

Next, we obtain a selection set of the lines to be changed using ssget without any arguments:

(setq sset1 (ssget))

Remember that when you use ssget in this way, the user is allowed to select the method of selection just as any select object prompt would. The user can use a standard or crossing window, pick objects individually, or selective remove or add objects to the selection set. In our exercise, you were asked to enter a **C** for a crossing window to select the lines.

This is followed by an if conditional expression to test if objects have been selected.

(if (/= sset1 nil)

The following while expression then does the work of updating the drawing database for each line that was selected. Just as with the Mlext program, the while expression checks to see if the value of count is less than the number of objects in the selection set. It then finds the object list for one of the lines and derives the two endpoints of that line.

(while (< count (sslength sset1))

(setq elst (entget (ssname sset1 count)))

(setq pt1 (cdr (setq sub1 (assoc 10 elst))))

(setq pt2 (cdr (setq sub2 (assoc 11 elst))))

This part is no different from Mlext. But the next line is slightly different from its corresponding line in Mlext.

(setq int (inters objx objy pt1 pt2 nil))

Here, inters is used to find the intersection between the line currently being examined and the boundary edge line. We see the two variables objx and objy used as the first two arguments to inters. These are the two endpoints of the boundary edge line derived earlier in the program. The variables pt1 and pt2 are the endpoints of the line currently being examined. A fifth argument, nil is added to the inters expression. When this fifth argument is present in an inters expression and is nil, then inters will find the intersection of the two pairs of coordinates even if they don't actually cross (see figure 10-7).

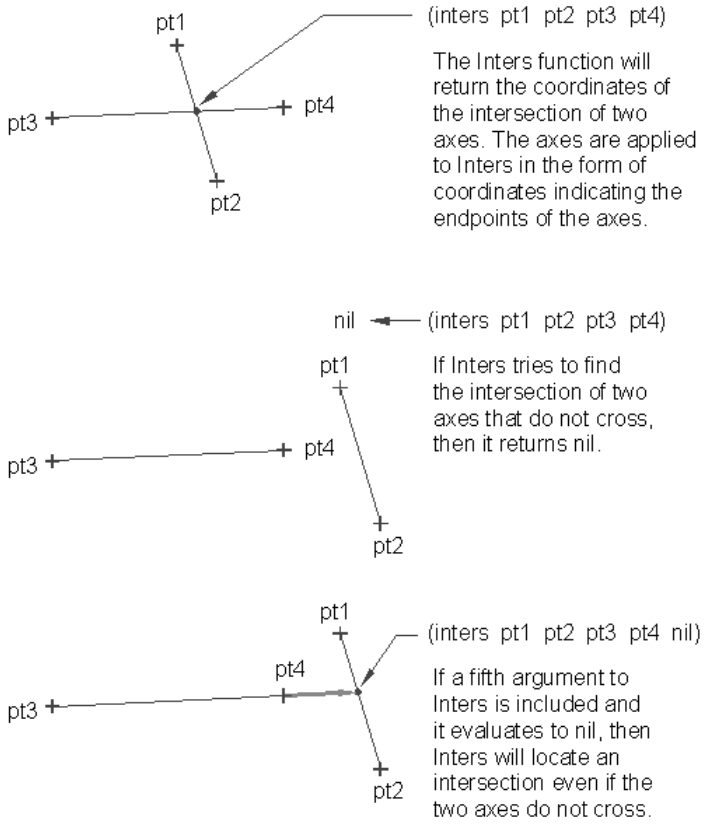


Figure 10-7: How the Inters function works

Inters treats the two lines as if they extended infinitely in both directions in order to find a point common to both lines. This feature is needed since the line being edited and the boundary edge line don't actually cross.

The next expression calls the user defined function far:

(setq far1 (far pt1 pt2 int))

This function finds which of two points is closest to a third point. The first two arguments to far are the points to be compared against the third argument which is the reference point. Far then returns the point that is farthest from the reference point. The result is that far finds the endpoint of the line that is the farthest from the intersection of the line and the boundary edge line. We will look at how far works later. For now lets continue with the main program.

Once the program finds the farthest of the two endpoints, the next three lines actually make the changes to the database.

```
(if (= far1 pt1)

(entmod (subst (cons 11 int) sub2 elst)) (entmod (subst (cons 10 int) sub1 elst))

);end IF 2
```

The conditional if expression checks to see if the farthest endpoint of the current line is equal to pt1. This test is done to determine which endpoint of the current object should be modified. We want the program to modify the endpoint closest to the intersection of the line and the boundary edge line so this test finds which endpoint is the one to change. If pt1 happens to be the equal to far1, therefore being farthest endpoint, then the sublist representing pt2 is modified. If pt1 proves not to be the farthest endpoint, then the sublist associated with it is modified.

Remember that subst replaces one list for another within an association list. Then endmod updates the drawing database record to reflect the new property list that is passes to it as an argument. The net result is the extension of a line to the boundary edge line.

The rest of the program adds one to the counter variable and the whole process is repeated until all the objects in the selection set have been processed. Since this program circumvents the AutoCAD Extend command and directly modifies the drawing database, it executes the changes to the lines objects much faster. However, to accomplish this extra speed, you must do some additional programming.

Now, lets briefly look at the far function. It is a fairly simple function that first obtains the distance between a reference point and two other points, then depending on which point yields the greater distance, the points value is returned. The value of far's three arguments are passed to the variables fx fy and dsfx. Fx and fy are the points in question and dsfx is the reference point:

```
(defun far ( fx fy dlxf / dst1 dst2 intx)
```

The function then finds the distance between fx and dlfx and assigns the value to dst1:

```
(setq dst1 (distance dlxf fx))
```

The same procedure is applied to fy:

```
(setq dst2 (distance dlxf fy))
```

Finally, the conditional if expression tests to see which distance is greater and returns a point value depending on the outcome:

```
(if (> dst1 dst2) fx fy )
```

Filtering Objects for Specific Properties

There are a number of other functions available that allow you to manipulate selection sets Table lists them and gives a brief description of what they do:

Function	Description
(ssadd [ent. name][s. set])	Creates a selection set. If used with no arguments, a selection set is created with no objects. If only an object name given as an argument, then a selection set is created that contains that object. If an object name and a selection set name is given, then the object is added to the selection set.
(ssdel [ent. name][s. set])	Deletes an object from a selection set. ssdel then returns the name of the selection set. IF the object is not a member of the selection set, then ssdel returns nil.
(sslength [s. set])	Returns the number of objects in a selection set.
(ssmemb [ent. name][s. set])	Checks to see if an object is a member of a selection set. If it is, then ssmemb returns the name of the selection set. if not, then ssmemb returns nil.
(ssname [s. set][nth object])	Returns the object name of a single object in a selection set. The second argument to ssname corresponds to the object's number within the selection set. The object numbers begin with zero and go to one minus the total number of objects in the selection set.

You have already seen two of these functions, sslength and ssname, used in previous examples. Lets see how we can use ssadd to filter out object selections.

Filtering a Selection Set

Figure 10-8 shows a function that is intended to filter out objects in a selection set based on layers. This function is useful where a group of objects are so close together that they are difficult to select, or in situations where several objects of different layers lie on top of each other and you want to select just the object on a specific layer. It returns a selection or object name depending on whether the filtered selection set contains only one item.

```
;function to filter entities by layer -- Lfilter.lsp-------------------------
(defun LFILTER (/ lay sset count ent newent)
    (setq lay (cons 8 (strcase (getstring "\nEnter layer name: "))))
    (setq sset (ssget))                              ;get entities
    (setq count 0)                                   ;set counter to zero
    (while (< count (sslength sset))                 ;while still select. set
      (setq lay2 (assoc 8 (entget(setq ent(ssname sset count))))) ;get layer
            (if (equal lay lay2)                     ;if layer matches entity
                (if (not newent)                     ;if new not select. set
                    (setq newent (ssadd ent))        ;make new select. set
                    (setq newent (ssadd ent newent)) ;else add to select. set
                );end if
            );end if
      (setq count (1+ count))
    );end while
   (if (= 1 (sslength newent))(ssname newent 0) newent) ;return select. set or
);end defun                                           entity name
```

Figure 10.8: The layer filtering program

Let's see first hand how this function works.

1. Open a file call Lfilter.lsp and copy the Lfilter program in figure 10.8. Save and exit the file.

2. Return to the AutoCAD Chapt10 drawing and erase any objects in the file.

3. Create the layers listed in table 10.1. Be sure to assign the line types indicated for each layer.

4. Draw the lines shown in figure 10.9. and assign each line to the layer shown directly to the right of each line. Use the coordinate and spacing information indicated in the drawing to place the lines.

5. Load the Lfilter program, then issue the erase command.

6. At the Select object prompt, enter:

(lfilter)

You will get the prompt:

Enter layer name:

8. Enter **hidden**. You will get the next prompt:

Select objects:

9. Enter C to use a crossing window and pick the points 2,1.25 for the lower left corner of the window and coordinate 8,8.25 for the upper right. The lines that pass through the crossing window will ghost. Press return after picking points. The lines will un-ghost then the prompt will return the number of objects found. The objects in the selection you just made with the crossing window that are on the layer **hidden** will ghost.

10. Press return. You have just erased only the items in your crossing window that were on the layer hidden.

11. Enter the Oops command in preparation for the next section.

Layer Name	**Linetype**
hidden	hidden
center	center
dashed	dashed

Table 10.1: Linetypes for drawing in figure 10.8

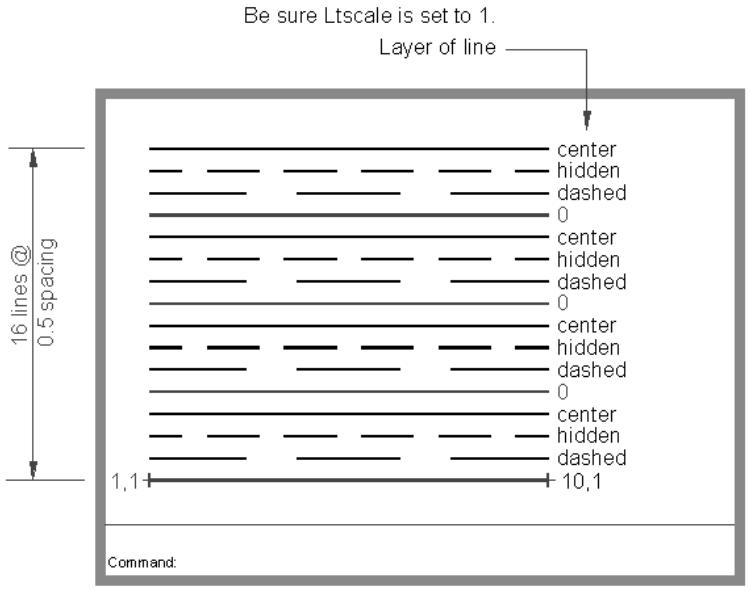


Figure 10.9: Test drawing for Lfilter.lsp

Now that you know what Lfilter does, let's look at how it works. First, it creates a dotted pair representing the layer sublist in an object property list:

```
(defun LFILTER (/ lay sset count ent newent)

(setq lay (cons 8

(strcase (getstring "\nEnter layer name: "))))
```

When you are prompted for a layer, the name you enter is first converted to all upper case using the Strcase function. This is done because the value associated with the layer group code in a property list is always in upper case.

In order to make a comparison of data, we must make sure that the values we use in the comparison are in the same format. Since AutoLISP is case sensitive when it comes to string values, we must make sure that the value the user enters matches the case of the layer name in the property list. The strcase function will convert a string to either all upper case or lower case depending on whether a third argument is present and is not nil.

Next, the cons function creates a dotted pair using 8 as the first element. 8 is the group code for layer names. The dotted pair looks like this:

```
(8 . "Hidden")
```

Finally, the newly created dotted pair is assigned to the variable **lay** which will later be used as a filtering value.

The next line obtains the selection set to be filtered:

```
(setq sset (ssget))
```

Here, ssget is used without any argument thus allowing the user to select objects using any of the usual AutoCAD selection options. This selection set is then assigned to the variable sset.

Next we come to the while expression.

```
(setq count 0)

(while (< count (sslength sset))

(setq lay2 (assoc 8

(entget(setq ent(ssname sset count)))))

(if (equal lay lay2)

(if (not newent)

(setq newent (ssadd ent))

(setq newent (ssadd ent newent))
```

```
);end if

);end if

(setq count (1+ count))

);end while
```

This while expression compares the 8 group code sublist of each object in the selection against the variable lay. If it finds a match, the object is added to a new selection set **newent**. Let's look at this while expression in detail.

First, the counter is set to zero and the conditional test is set up:

```
(setq count 0)

(while (< count (sslength sset))
```

Then the 8 group code sublist from the first object in the selection set is extracted and assigned to the variable lay2:

```
(setq lay2 (assoc 8

(entget(setq ent(ssname sset count)))))
```

Next, the variables lay lay2 are compared:

```
(if (equal lay lay2)
```

If there is a match signifying that the object is on layer "HIDDEN", then the program checks to see if the selection set newent exists:

```
(if (not newent)

(setq newent (ssadd ent))

(setq newent (ssadd ent newent))
```

If newent does not exist, then ssadd creates a new selection set containing the object whose layer group code sublist matches (8 . "HIDDEN") then assigns that selection set to the variable newent. If newent does exist, ssadd adds the object to the selection set newent and redefines newent. This last conditional **if** is required since ssadd must be given different arguments depending on whether it is to create a new selection set or just add an object to an existing selection set.

Finally, the counter in increased by one and the while conditional loop repeats itself:

```
);end if

);end if

(setq count (1+ count))

);end while
```

Once the new selection set containing the filtered objects is complete, the last expression returns either the selection set of objects, or if there is only one object in the selection set, the object name.

(if (= 1 (sslength newent))(ssname newent 0) newent)

);end defun

The function sslength is used with the = predicate to see if newent contains only one element. If it does, then ssname is used to extract the object name of the element from the selection set newent. Otherwise, the entire selection set newent is returned. This last step is added to allow the user to use lfilter where only one item is accepted for input such as the offset or fillet commands.

In this sample program, layers are used to filter objects, but you can use any object property as a filter. You can filter objects by linetype, color, or any property available from the property list. Consult the list of group codes in Appendix C for a list of group codes and their associated properties.

Selecting Objects Based on Properties

Another method for filtering can be found built into the ssget function. Ssget allows you to select objects based on a filter list. This filter list is an association list much like a property list. The Getlayer function shown in figure 10.10 simply selects the entire contents of a layer that the user specifies.

```
;function to select all entities on a layer---------------------------------

(defun GETLAYER (/ lay)
 (setq lay (list (cons 8
     (strcase (getstring "\nEnter layer name: "))))
 (ssget "X" lay)
)
```

Figure 10.10: The Getlayer function.

Let's see what it does:

1. Save and exit the chapt10 file.

2. Open a file called Getlayer.lsp and copy figure 10.10 into the file. Save and exit the file.

3. Return to the chapt10 drawing file then load Getlayer.lsp.

4. Issue the Erase command. At the Select object prompt, Enter:

(getlayer)

5. At the prompt

Enter layer name:

Enter **center**. All the objects on layer Center will ghost.

6. Press return. All the objects on layer Center are erased.

Getlayer does its work by using the "X" argument to ssget. This argument allows ssget to create a selection set based on an association list of properties. In the case of Getlayer, the list is one element long.

First, Getlayer prompts the user for a layer:

(defun GETLAYER (/ lay)

(setq lay (list (cons 8

(strcase (getstring "\nEnter layer name: "))))

The layer name is used to construct an object property dotted pair much like the one in the flayer function. This dotted pair is further included in a list using the list function. The result is a list containing a single dotted pair:

((8 . "CENTER"))

This list is assigned to the variable lay which is in turn applied to ssget to create the selection set:

(ssget "X" lay)

)

Here we see the "X" argument used with ssget to tell ssget that a filter list is to be used to create the selection set. Ssget then searches the drawing database to find all the objects that have properties that matches the filter list.

Getlayer could have been simplified to one expression thereby eliminating the need for the lay variable:

(ssget "X" (list (cons 8

(strcase (getstring "\nEnter layer name: "))))

We include the lay variable to help explain how this function works.

A filter list can have more than one property sublist element much like an objects property list. But ssget will only accept certain group codes in the filter list. Table shows those group codes and their associated properties:

Group code	Meaning
0	Object type
2	Block name
6	Linetype name
7	Text style name
8	Layer name
38	Elevation
39	Thickness
62	Color number; 0=byblock, 256=bylayer
66	Attributes-follow flag for blocks
210	3D extrusion direction vector

Accessing AutoCAD's System Tables

The tblnext and tblsearch functions are provided to help you gather information about layers, linetypes, views, text styles, blocks, UCSs and viewports. Each one of these AutoCAD tools is represented in a table that contains the tools status. Tblnext and tblsearch return this table information in the form of association lists similar to object property lists. But unlike object property lists, you cannot modify lists returned from Tblnext and tblsearch. However, you can modify the settings associated with a tblnext or tblsearch listing using the standard AutoCAD commands.

To use tblnext, enter the following at the command prompt:

(tblnext "layer")

You will get an association list similar to the following:

((0 . "LAYER") (2 . "0") (70 . 0)(62 . 7) (6 . "CONTINUOUS"))

The individual dotted pairs can be extracted from this list using Assoc just as with any other association list or property list. Enter the tblnext expression above again and you will get an association list of the next layer. Each time tblnext is used, it advances to the next table setting until it reaches the last item in the particular table you are searching. Once it reaches the end, tblnext returns nil. To reset tblnext to read from the beginning of the table again, you include a second argument that evaluates to non-nil as in the following:

(tblnext "layer" T)

If this expression is entered, you will get the same list as the one you got the first time you used tblnext. We used T as the second argument but it could be any expression that evaluates to non-nil. Once you get a list, you can manipulate in the same way as any other association list.

Tblsearch works slightly differently. Instead of stepping through each table item, tblsearch will go to a specific table item which you name. Enter the following:

(tblsearch "layer" "hidden")

You will get the association list pertaining to the layer "hidden".

((0 . "LAYER") (2 . "HIDDEN") (70 . 0)(62 . 7) (6 . "HIDDEN"))

If you include a non-nil third argument to tblsearch, then the next time tblnext is used, it will start from the next item after the one obtained from tblsearch.

Though we used layer settings as an example for tblnext and tblsearch, any of the table settings mentioned at the beginning of this section can be used.

Figure 10.11 shows a program that uses tblnext to store layer settings in an external file. This program can be useful if you use a single multi-layered drawing for several types of output. For example, an architect might have a drawing that serves as both an electrical layout plan and a mechanical floor plan with different layers turned on or off depending on which type of plan you want to edit or print. You can store your different layer settings for the electrical and mechanical plans then restore one or the other group of settings depending on which plan you indent to work on.

```
;Program to save layer settings in a file -- Lrecord.lsp
;-----------------------------------------------------------------------------
(defun c:lrecord (/ fname lafile record)
 (setq fname (getstring  "\nEnter name of layer file: "));get name of file
 (setq lafile(open fname "w"))                           ;open file, file desc.
 (setq record (tblnext "layer" T))                       ;get first layer set.
  (while record                                          ;while record not nil
   (prin1 record lafile)                                 ;print record to file
   (princ "\n" lafile)                                   ;print to next line
   (setq record (tblnext "layer"))                       ;get next layer
  );end while
 (close lafile)                                          ;close layer file
);end defun

;Program to restore layer settings saved by lrecord
;-----------------------------------------------------------------------------
(defun c:lrestore (/ clayer fname lafile flayer lname oldcset)
 (setvar "cmdecho" 0)                                    ;turn off prompt echo
 (setvar "regenmode" 0)                                  ;turn off autoregen
 (Setq clayer (getvar "clayer"))                         ;find current layer
 (setq fname (getstring "\nEnter name of layer file: ")) ;get layer file name
 (setq lafile(open fname "r"))                           ;open layer file
 (setq flayer (read (read-line lafile)))                 ;read first line
 (while flayer                                           ;while lines to read
    (setq lname (cdr (assoc 2 flayer)))                  ;get layer name
    (setq oldcset (assoc 62 flayer))                     ;get color setting
    (if (and (< (cdr oldcset) 0) (equal lname clayer))   ;if col. is off/currnt
        (command "layer" "C" (cdr oldcset) lname "Y" "") ;insert "Y" response
        (command "layer" "C" (cdr oldcset) lname "")     ;else normal
    );end if
    (Setq oldcset (assoc 70 flayer))                     ;find if frozen
    (if (= (cdr oldcset) 65)                             ;if frozen then...
        (if (equal lname clayer)                         ;if current layer
            (command"layer" "freeze" lname "y" "")       ;insert "y" response
            (command"layer" "freeze" lname "")           ;else normal
        );end if
        (command "layer" "thaw" lname "")                ;else thaw layer

    );end if
    (setq flayer (read-line lafile))                     ;read next in file
    (if flayer (setq flayer (read flayer)))              ;strip quotes
 );end while
 (close lafile)                                          ;close file
 (setvar "regenmode" 1)                                  ;reset autoregen on
   );end defun
```

Figure 10.11: A program to store layer settings

The Lrecord program show at the top of the figure 10.11 simply creates a file then copies each layer association list into the file. Lrecord uses the prin1 function to perform the copying because prin1 does not affect the list in any way. If princ were used, the strings data types within the list would be striped of their quotation marks. Also, Write-line is not used since it expects a string argument. Both princ and prin1 will write any data type to a file.

The Lrestore program simply reads each line back from a file created by Lrecord. Since the Read-line function returns a string, the read function is used to strip the outermost level of quotation marks from the string to return the association list.

```
(setq flayer (read (read-line lafile)))
```

The Layer data is then extracted from this list and applied to the layer command which sets the layer back to the saved settings. The **if** conditional test is used to see if the layer to be restored is the current layer. A different command expression is evaluated depending on whether the layer in question is current or not. This is done since the layer command will issue an extra prompt if the current layer is to be turned off or frozen.

```
(if (and (< (cdr oldcset) 0) (equal lname clayer))

(command "layer" "C" (cdr oldcset) lname "Y" "")

(command "layer" "C" (cdr oldcset) lname "")

)
```

Conclusion

You have seen a variety of ways to select, edit, and manipulate AutoCAD objects. Selection sets and object filters can provide a powerful means to automating AutoCAD. Tasks that would normally take several minutes to perform manually can be reduced to a few seconds with the proper application of selection sets and recursive expressions.

You have also seen how changes in the way you write your program can affect your programs speed. Though the speed of your programs may not be an issue to you now, as your experience with AutoLISP expands, your need for speed will also expand.

In the next chapter, we will continue the discussion of object access by looking at how polylines and attributes can be edited with AutoLISP.

Chapter 11: Accessing Complex Objects

Introduction

In this the final chapter, you will look at several programs that not only introduce you to some new AutoLISP functions, but also review many of the functions you learned about from earlier chapters. In the process, you will learn how to access complex object types. You will also look at ways to store data as a permanent record within a drawing.

Accessing Polyline Vertices

Polylines are complex objects that are a composite of many objects. There are a variety of polylines from straight lines, to three dimensional Bezier-spline polylines. But even the most complex polyline can be broken into three basic components, Vertices, lines and arcs. AutoCAD stores polylines as a kind of compound object made up of several layers of information. To help you get a grasp of this idea, you can think this storage structure as an onion; as you peal off one layer, another layer is revealed. The first layer, the one accessed by entget, gives general information about the polyline. In this section, you will look at ways to access deeper layers of information using AutoLISP.

It will help our investigation if we first take a look at typical polyline property list first hand.

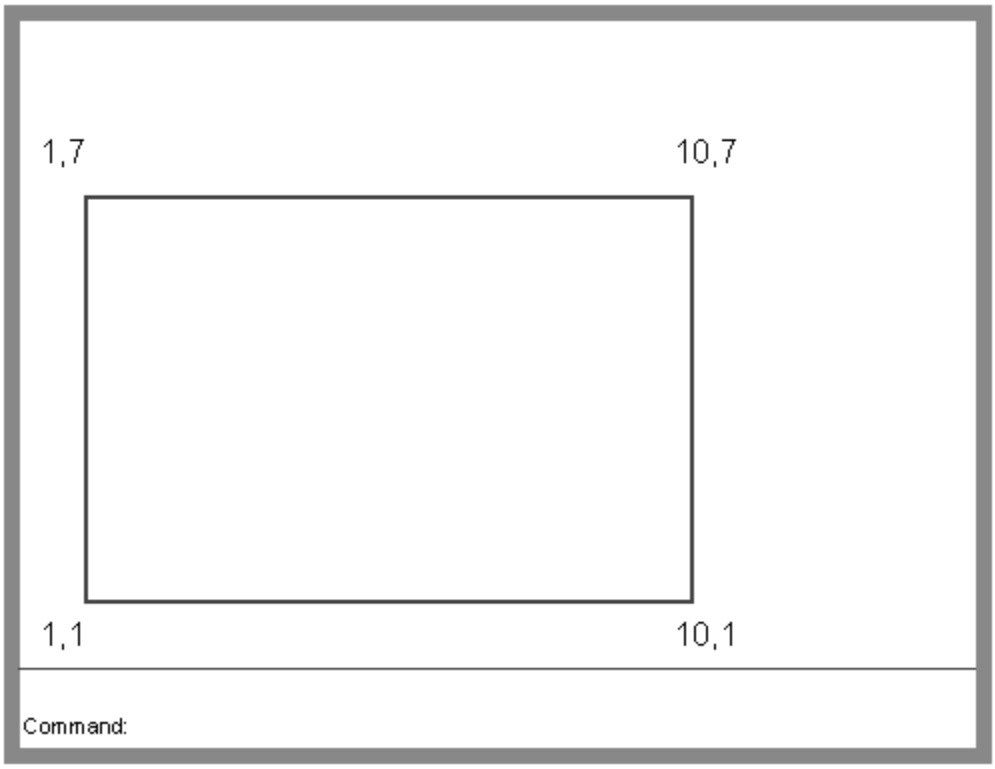


Figure 11.1: Test drawing for exercise

Open a drawing file called Chapt11 and draw the polyline rectangle shown in figure 11.1. Use the coordinates shown in the figure to locate the corners.

1. Enter the following expression at the command prompt:

(entget (car (entsel)))

2. The selection cursor box appears and you get the select object prompt. Pick the polyline you just drew. The following listing appears.

((-1 . <Object name: 60000120>)

(0 . "POLYLINE")

(8 . "TEXT")

(66 . 1) (10 0.0 0.0 0.0)

(70 . 1) (40 . 0)

(41 . 0)

(210 0.0 0.0 1.0)

(71 . 0)

(72 . 0)

(73 . 0)

(74 . 0)

(75 . 0))

We have list the polyline property list vertically for clarity though you will see the list shown as a continuous string on your text screen. We see the object name, object type and layer listed but where the point group code 10 should indicate some coordinate location, only zeros are shown. Also, we would expect to see a list of at least four coordinate values. Instead, we see some somewhat unfamiliar codes from the 40 and 70 group codes, all showing zeros.

This doesn't mean that we are unable to access more specific information about polylines. We just need to dig a little deeper. The tool we use for digging is the entnext function.

We mentioned that it helps to think of the polyline data as being stored in layers like those of a onion. The listing above represents the outermost layer. To get to the next layer, you need to use the entnext function.

1. Enter the next expression:

   (entget (entnext (car (entsel))))

2. You will get a listing like the following:

   ((-1 . <Object name: 60000120>)

   (0 . "VERTEX")

   (8 . "0")

   (10 1.0 1.0 0.0)

   (40 . 0)

   (41 . 0)

   (42 . 0)

(70 . 0)

(50 . 0))

This new property list gives us some new information. The object name is different from the previous list. Also, instead of showing "POLYLINE" as an object type, we get "VERTEX". The layer information is nearly the same but for the first point value, group code 10, we see the coordinate for the first corner of the box, 1.0 1.0 0.0. For purposes of our discussion, we'll call this vertex object a polyline sub-object or just sub-object.

Now, we know how to get more detailed information about a polyline by introducing entnext into our expression to extract an object's sub-object information. We have "peeled off" the first layer of our onion to reveal the next layer of information. But we only got the first vertex of the polyline. To get the others, you just continue adding more entnext functions.

1. Enter the following:

(entget (entnext (entnext (car (entsel)))))

2. When the Select object prompt appears, pick the polyline box. Another new property list appears.

((-1 . <Object name: 60000120>)

(0 . "VERTEX")

(8 . "0")

(10 10.0 1.0 0.0)

(40 . 0)

(41 . 0)

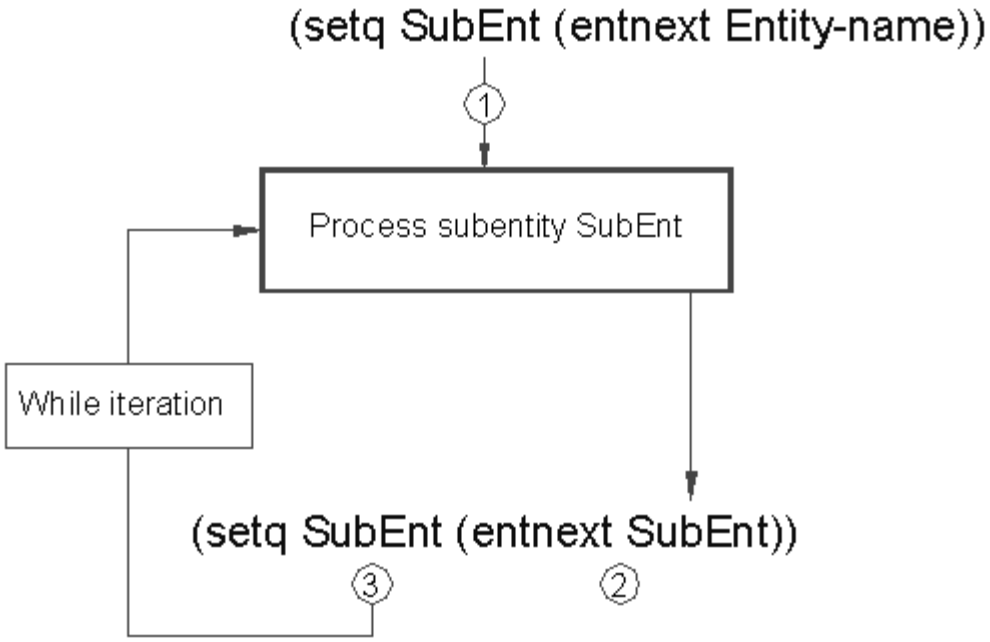
(42 . 0)

(70 . 4)

(50 . 0))

Now we see a list that describes the second vertex of the polyline. Note the group code 10 value shows a coordinate 10.0 1.0 0.0 which is the next vertex in the polyline.

It would be rather awkward to have to keep expanding our expression by adding more entnext functions to extract each sub-object's property lists. If we have a polyline that has 40 vertices or more, we would end up with a enormous program in order to extract all the vertices. Fortunately, we can use a recursive function to get the property list of each vertex. By using the same variable name to which we assign each vertex object name, we eliminate the need to add continual nests of the entnext function. Figure 11.2 shows a diagram of how this recursive function works and figure 11.3: shows a function that uses this recursion to extract a list of polyline vectors. The Function at the top of the figure, Getvec, is the one we will look at first.



① The subentity is found using Entnext. It is then assigned to a variable SubEnt and processed.

② Then SubEnt is applied to the Entnext function to get the next subentity name.

③ The variable SubEnt is assigned the new entity name and the process is repeated.

Figure 11.2: Using recursion to extract subobject information

```
;Function to create list of polyline vertices---------------------------------

(defun getver (EntNme / SubEnt VerLst vertex)
 (setq SubEnt (entnext EntNme))                    ;get first vertex
 (setq VerLst '())                                 ;setup vertex list
 (while SubEnt
   (setq vertex (cdr (assoc 10 (entget SubEnt))))  ;get first vertex point
   (setq VerLst (append VerLst (list vertex)))     ;add vertex to verlst
   (setq SubEnt (entnext SubEnt))                  ;go to next vertex
 )
 VerLst                                            ;return vertex list
)

;Function to check if point lies between endpoints of line--------------------
 (defun btwn (a b c)
(setq ang1 (angle a b))                            ;find vertex to point ang.
(setq ang2 (angle a c))                            ;find vertex to vertex ang.
(if (EQUAL (RTOS ang1 2 2) (RTOS ang2 2 2)) b)     ;if equal return point.
)

;Program to insert Vertex in simple polyline
;-------------------------------------------------------------------------------
(defun C:ADDVERT (/ pEnt VerLst Newpt int NewVer ptyp)
  (setq pEnt (entsel "Pick vertex location: "))    ;Get new vertex and pline
  (setq VerLst (getver (car pEnt)))                ;extract vertices
  (setq ptyp (assoc 70 (entget (car pEnt))))
  (setq Newpt  (osnap (cadr pEnt) "nearest"))      ;Get new vertex location
      (while (cadr VerLst)
          (setq NewVer                             ;add vertex to new NewVer
            (append NewVer (list (car VerLst)))
          )
          (setq int                                ;Check for between-ness
            (btwn (car VerLst) newpt (Cadr VerLst))
          )
          (if int                                  ;if between, add to NewVer
            (setq NewVer (append NewVer (list int)))
          )
          (setq VerLst (cdr VerLst))               ;Remove vertx. from list
      );end while
  (setq NewVer (append NewVer (list (car VerLst)))) ;add last vertx. to NewVer
  (command "erase" (car pEnt) "")                  ;erase old pline
  (command "pline")                                ;start pline command
  (foreach n NewVer (command n))                   ;insert points from NewVer
  (if (= (cdr ptyp) 1)
       (command "close")
       (command "")
  )                                                ;end pline command
)
```

Figure 11.3: Function that implements diagram in figure 11.2

Lets examine this function in detail.

The function takes an object name as an argument. Presumably the object in question is a polyline. It proceeds to obtain the first vertex object name from that object.

```
(defun getvec (EntNme / SubEnt VecLst vector)
(setq SubEnt (entnext EntNme))
```

This vertex object name is assigned to the symbol subEnt. Next, a list is created to hold the vector coordinates:

```
(setq VecLst '())
```

The following while expression then goes to each vector property list and extracts the coordinate value. It first extract the coordinate from the current vertex object:

```
(while SubEnt
(setq vector (cdr (assoc 10 (entget SubEnt))))
```

Next, it appends that coordinate value to the vertex list VecLst.

```
(setq VecLst (append VecLst (list vector)))
```

finally, the variable SubEnt is assigned the vertex object name of the next vertex and the process is repeated:

```
(setq SubEnt (entnext SubEnt))
)
```

When all the vertices have been obtained, the list of vertices is returned:

```
VecLst
)
```

Now that we know what entnext is capable of, lets look at a practical application. Figure 11.3 includes a program called Addvect that adds a vertex to a polyline. If you have ever had to add a vertex to a polyline using the AutoCAD pedit command, you know it can be a trying effort. This program simplifies the operation to one step. Figure 11.4 gives a graphic description of how this program works.

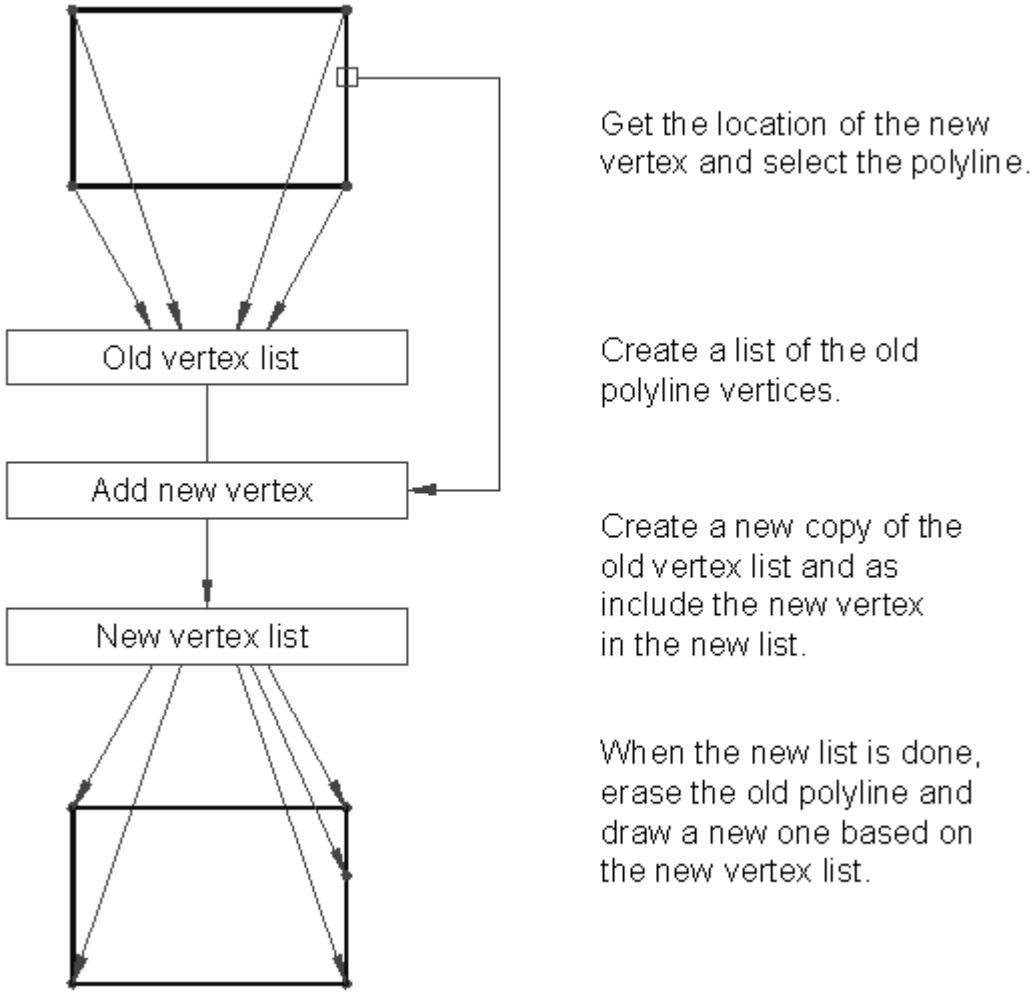


Figure 11.4: How Addvect work conceptually

Lets see first hand how it works.

1. Save and exit the Chapt11 file.
2. Open an AutoLISP file called Addvert.lsp then copy figure 11.3 into the file. Save and exit Addvert.lsp
3. Return to the Chapt11 drawing file and load addvert.lsp.
4. Enter Addvert at the command prompt. At the prompt:

Pick vertex location:

Pick the square polyline at the coordinate 10,6. The box disappears and a new box is drawn with an additional vertex at the point you picked.

This program reduces into one step a process than normally takes seven steps through the Pedit command. Lets see how it works in detail.

First, an object and a point are gotten using the entsel function:

```
(defun C:ADDVECT (/ pEnt VecLst Newpt int NewVec type)

(setq pEnt (entsel "Pick vector location: "))
```

As you may recall, entsel pauses the program's processing and prompts the user to select an object. Once a user responds, a list of two elements is returned with the object name and the coordinate used to pick the object.

The next line uses a user defined function, getvec, to create a new list containing only the vertex coordinates from the polyline object picked. This list of vertices is assigned to the variable VecLst.

```
(setq VecLst (getvec (car pEnt)))
```

We saw earlier how getvec works. It returns a list of polyline vertices. In the above expression, the list of vertices is assigned to the VecLst variable.

The next line obtains the associated value of the 70 group code from the polyline. The 70 group code identifies the type of polyline it is, whether it is closed, curve-fit, spline curved, etc. See Appendix for a full list of the 70 group code options.

```
(setq ptyp (assoc 70 (entget (car pEnt))))
```

this information will be used at the end of the program to determine how the polyline is redrawn.

The next line used the osnap function to establish a point exactly on the polyline.

```
(setq Newpt (osnap (cadr pEnt) "nearest"))
```

Here, the coordinate from the entsel function used earlier is applied to the osnap "nearest" function to obtain a new point. This new point is exactly on the polyline.

Defining a New Polyline

The while expression that follows builds a new list of vertices from which a new polyline will be drawn. This list is actually a copy of the list created by our user defined function Getvec with the new point added in the appropriate place.

The test expression in the while expression tests to see if the end of the list VecLst has been reached:

(while (cadr VecLst)

Next, the first element of Veclst is added to a list called NewVec:

(setq NewVec

(append NewVec (list (car VecLst)))

)

This expression is basically just copies the first element of the original vertex list to a new list NewVec.

The next set of expressions tests to see if our new vertex newpt lies between the first two point of the vertex list:

(setq int

(btwn (car VecLst) newpt (Cadr VecLst))

)

Another user-defined function is used to actually perform the test. This function is called btwn and it tests to see if one coordinate lies between two others. If Btwn does find that Newpt lies between the first and second point of VecLst, then Btwn returns the value of Newpt. Otherwise Btwn returns nil.

If the btwn test function returns a coordinate, the next expression adds the new vertex to the NewVec list.

(if int

(setq NewVec (append NewVec (list int)))

)

Finally, the first element of the vertex list is removed and the whole process is repeated.

(setq VecLst (cdr VecLst))

);end while

Once the while loop is done, VecLst is a list of one element. That last element is added to the NewVec list:

(setq NewVec (append NewVec (list (car VecLst))))

Drawing the new Polyline

The last several lines erase the old polyline and redraw it using the new vertex list. First the old line is erased:

(command "erase" (car pEnt) "")

Then the pline command is issued:

```
(command "pline")
```

Next, the foreach function is used to input the vertices from the NewVec list to the Pline command:

```
(foreach n NewVec (command n))
```

You may recall that foreach is a function that reads each element from a list and applies that element to a variable. That variable is then used in an expression. The expression is evaluated until all the elements of the list have been evaluated in the expression. In this case, each vertex from the NewVec list is applied to a command function which supplies the vertex coordinate to the pline command issued in the previous expression.

Once foreach has completed evaluating every element of the NewVec list, the last expression ends the Pline command:

```
(if (= (cdr Ptyp) 1)

(command "close")

(command "")

)

)
```

The if conditional expression tests to see if the polyline is closed or not. If it is, then it enter the word "close" to close the polyline. If not, then an enter is issued. You may recall that in the first part of the program, the 70 group code sublist was extracted from the polyline property list. This sublist was assigned to the variable ptyp. Here, ptyp is tested to see if its code value is 1. If it is 1, this means that the polyline is closed thereby causing the if expression to evaluate the (command "close") expression. If this expression is left off, the new polyline box would have only three sides.

Testing for Polyline Types

In the last expression above, you got a glimpse of a special concern when dealing with polylines. There are really several types of polylines which must all be handled differently. The Addvect program will only function properly when used on simple polylines made up of line segments. Polylines that are curve-fitted or splined will contain extra vertices that are not actually part of the drawn polyline. these extra vertices are used as control points in defining curves and splines.

Fortunately, the 70 group codes enable you to determine what type of vertex you are dealing with. In the program above, we used the 70 group code only to determine whether the polyline is closed or not but other conditions can be tested for. You could include a test for a spline vertex by comparing the 70 group code value of a vertex to the value 16. If it is 16, the value for a spline frame control point, then you know not to include the vertex in your vertex list. We won't try to give you examples here as we our space is limited. However, you may want to refer to Appendix for more details on the group codes.

How Arcs are Described in Polylines

Arcs in polylines are described using a bulge factor and two vertices. You can think of the bulge factor as the tangent of the angle described by chord of the arc and a line drawn from one end of the arc to the arc's midpoint (see figure 11.5).

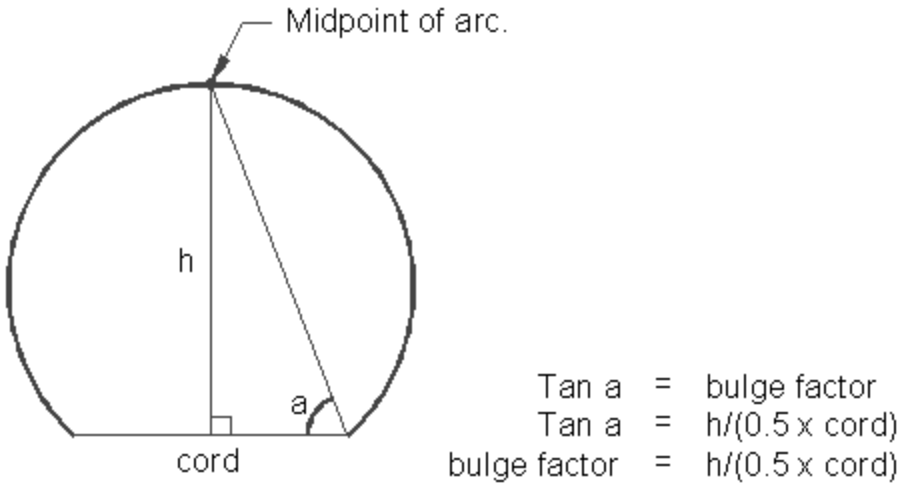


Figure 11.5: The arc bulge factor

From this relationship, the following formula is derived:

bulge = h / 0.5 cord = 2h/cord

We can derive the geometry of the arc from these simple relationships. Figure 11.6 shows how we can derive the arcs angle from the bulge factor. We don't give an example of a program to edit arcs in a polyline. Such a task could take a chapter in itself since it can be quite involved. Also, you may not find a need for such a capability at this point. However, should you find a need, we have given you the basics to build your own program to accomplish the task.

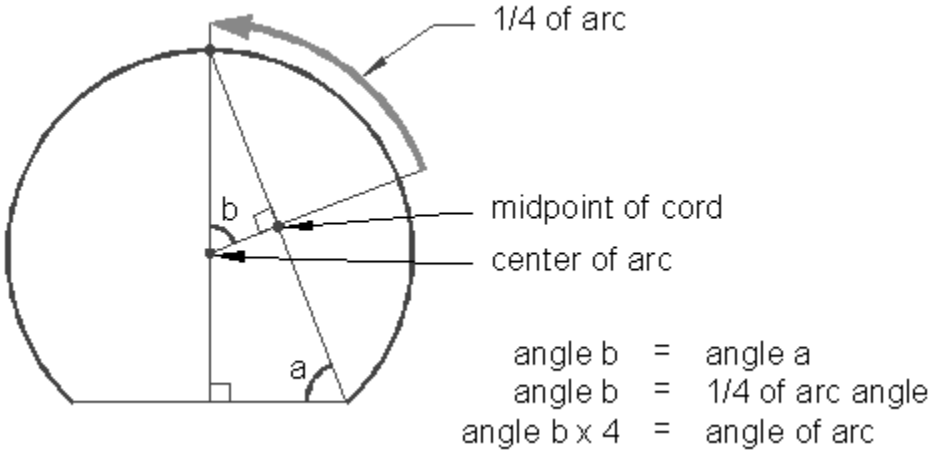


Figure 11.6: Finding the angle of an arc

Accessing Object Handles and Block Attributes

In the last section, you saw how to extract information from a complex object. You can use the same method to extract information from block attributes. The program you are about to examine uses block attributes as well as object handles and external files to help assign and store names to objects in your drawing. We will look at how entnext can be used to get attribute information from a block and how you can use attributes to permanently store information. You will also explore one possible way of using object handles which are permanent names AutoCAD can give to objects in a drawing.

Using Object Handles

We know that every object is given an object name. This name is the key to accessing the object's record in the drawing database. Unfortunately, this object name changes from editing session to editing session. This means that during one editing session, an object will have the object name <Object name: 600000d4> while in another session the same object will have the name <Object name: 60000012>. For the most part, this may not be of concern to you. But if you want to have a way of permanently identifying objects from one editing session to another, you may find this fact disturbing.

Fortunately, starting with release 10, you can add what is called an object handle to each and every object in your AutoCAD drawing. Handles are added to the object's in a drawing by AutoCAD whenever you turn on the handles function using the handles command. You do not have control over the naming of Objects, AutoCAD automatically assigns an alpha-numeric name to every object in the drawing.

The handent function in conjunction with other functions can be used to obtain an objects handle from the drawing

database. The handles are added to an object's property list as a group code 5 property sublist. To get an object's handle, you use the usual assoc-entget function combination to extract the sublist.

1. Return to the Chapt11 drawing and enter the following at the command prompt:

   handles

2. At the prompt:

   Handles are disables.

   ON/DESTROY:

   enter **ON.** Next enter

   (assoc 5(entget(car (entsel))))

3. At the select object prompt, pick the polyline box. You will get a list similar to the following:

   (5 . "29")

The second element of the group 5 property is the object handle. Note that the handle is a numeric value in quotes, so it is really a string data type even though it is a number.

Using handles, you could write a simple routine to display an object's handle which you could record somewhere. Then, you could have another program to retrieve an object based on this handle. We've taken this idea a step further and have written a program that allows you to assign any name you like to an object and later select that object by entering the name you have assigned to it. Figure 11.7 shows this program and Figure 11.8 shows a diagram of how it works.

```
;Function to turn a list into a string--------------------------------------

(defun ltos (lst / gfile strname)
(setq gfile (open "acad.grp" "w"))                      ;open a file on disk
(prin1 lst gfile)                                       ;print list to file
(close gfile)                                           ;close file
(setq gfile (open "acad.grp" "r"))                      ;open file
(setq strname (read-line gfile))                        ;read list from file
(close gfile)                                           ;close file
strname                                                 ;return converted list
)

;Function to obtain name list stored in attribute---------------------------
 (defun getatt (/ nament)
(setq nament (ssname(ssget "X" '((2 . "NAMESTOR")))0))  ;get attribute block
(read (cdr (assoc 1(entget (entnext nament)))))         ;get attribute value
)

;Function to clear stored names---------------------------------------------
(defun attclr ()
(setq nament (ssname (ssget "X" '((2 . "NAMESTOR")))0)) ;get attrib. block
(setq namevl (entget (entnext nament)))                 ;get attrib. ent. list
(setq namelt (assoc 1 namevl))                          ;get attrib. value
(entmod (subst (cons 1 "()") namelt namevl))            ;add list to attrib
)

;Program to assign a name to an entity
;-----------------------------------------------------------------------------
(defun C:NAMER (/ group gname ename sname namevl namelt)
(setq ename  (cdr (assoc 5 (entget (car (entsel "\nPick object: "))))))
(setq gname  (list (strcase (getstring "\nEnter name of object: "))))
(setq group  (getatt))                                  ;get names from attrib.
(setq gname  (append gname (list ename)))               ;new name + ent. name
(setq group  (append group (list gname)))               ;add names to list
(setq sname  (ltos group))                              ;convert list to strng
(setq namevl (entget (entnext (ssname (ssget "X" '((2 . "NAMESTOR")))0))))
(setq namelt (assoc 1 namevl))                          ;get attrib. value
(entmod (subst (cons 1 sname) namelt namevl))           ;add list to attrib
(entupd (cdr (assoc -1 namevl)))
(princ)
)

;Function to select an entity by its name-----------------------------------
 (defun GETNAME (/ group gname )
(setq gname  (strcase (getstring "\nEnter name of entity: ")))
(setq group  (getatt))                                  ;get names from attrib.
(handent (cadr (assoc gname group)))
)
```

Figure 11.7: Program to give names to objects

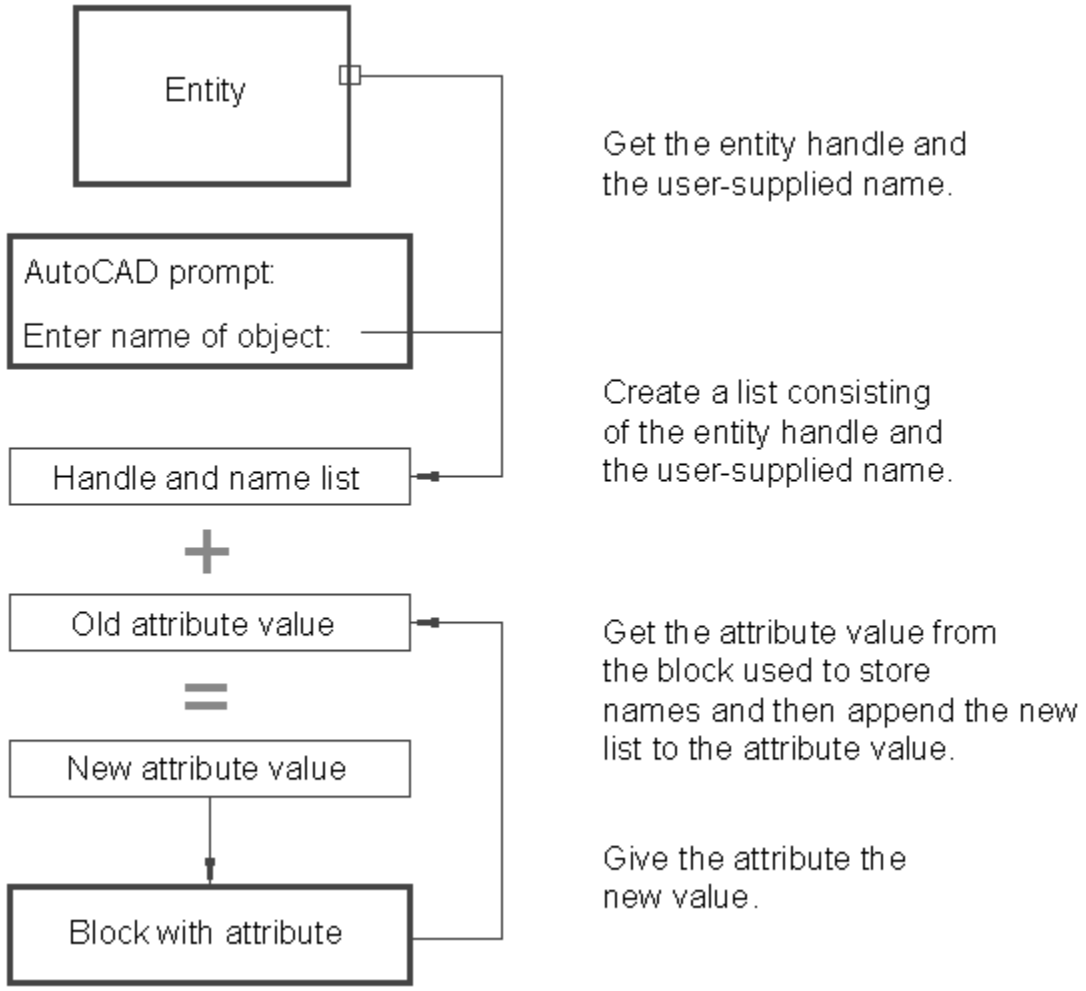


Figure 11.8: Diagram of a program to name objects

This program makes use of a block attribute as a storage medium for the names you assign to objects. Lets take a closer look.

First, you need to define the attribute used for storage.

1. Exit the Chapt11 file and open an AutoLISP file called Namer.lsp. Copy the program shown in figure 11.7 into your file then save and exit the file.

2. Return to the Chapt11 file then load Namer.lsp.

3. Enter **attdef** to start the attribute definition command.

4. Enter the following responses to the attdef prompts:

Attribute modes -- Invisible:N Constant:N Verify:N Preset:N

Enter (ICVP) to change, RETURN when done:**~CR**

Attribute tag: **name**

Attribute name: **name**

Default attribute value: **()**

Start point or Align/Center/Fit/Middle/Right/Style: **2,9**

Height (2.0): **~CR**

Rotation angle <0>: **~CR**

5. The word **name** will appear in at coordinate 2,9. Now issue the block command and enter the following responses to the block prompts:

Block name (or ?): **namestor**

Insert base point: **2,9**

select objects: [**pick attribute defined in previous step.**]

6. Issue the insert command and enter the following responses to the insert prompts:

Block name (or ?): **namestor**

Insertion point: **2,9**

X scale factor <1> / Corner / XYZ: **~CR**

Y scale factor (default=X): **~CR**

Rotation angle <0>: **~CR**

Enter attribute values

name <()>: **~CR**

You have just defined the attribute within which the program **namer** will store your object names. Now you are ready to use the program.

1. Enter **namer** at the command prompt. At the prompt:

    Pick object:

pick the polyline box.

2. At the next prompt

    Enter name of object:

Enter **square**. The computer will pause for a moment then the command prompt will return. Also, the value of the attribute you inserted earlier will change to a list containing the name SQUARE and the object handle associated with the name. The Namer program uses the attribute as a storage device to store the name you give the object with its handle.

3. To see that the name square remain associated with the box, exit the chapt11 file using the end command then open the file again.

4. Load the Namer.lsp file again.

5. Now issue the copy command. At the Select object prompt, enter

    (getname)

You will get the prompt

    Enter name of object:

Enter **square**. The box will highlight indicating that it has been selected.

6. At the Base point prompt, pick a point at coordinate 2,2.

7. At the Second point prompt, pick a point at coordinate 3,3.

The box is copied at the displacement 1,1.

The attribute used to store the name could have been made invisible so it doesn't intrude on the drawing. We intentionally left it visible so you could actively see what is going on.

Namer works by first extracting the object handle of the object selected then creating an association list of the handle and the name entered by the user. This association list is permanently stored as the value of an attribute. The attribute value is altered using the entmod function you saw used in the last chapter. Let's take a detailed look at how namer and getname work.

Using Object Handles

Namer starts by obtaining the object handle of the object the user picks:

(defun C:NAMER (/ group gname ename

sname nament namevl namelt)

(setq ename

(cdr (assoc 5 (entget (car (entsel "\nPick object: "))))

)

Here, entsel is used to get the object name of a single object. The car function extracts the name from the value returned from entsel then entget get the actual object name. At the next level, the assoc function is used to extract the 5 group code sublist from the object. The actual object handle is extracted from the group code using the cdr function. This value is assigned to the variable ename.

In the next expression, a list is created containing the name given to the object by the user:

(setq gname

(list (strcase (getstring "\nEnter name of object: ")))

)

The user is prompted to enter a name. this name is converted to all upper case letters using the strcase function. Then it is converted into a list using the list function. finally, the list is assigned to the variable gname.

The next expression calls a user defined function called getatt:

(setq group (getatt))

This function extracts the attribute value from a the block named namestor. You may recall that the default attribute value of namestor was "()". Getatt extracts this value and the above expression assigns the value to the variable group. We'll look at how getatt works a little later.

Next, the object handle is appended to the list containing the name the user entered as the name for the object. This appended list is then appended to the list named **group** which was obtained from the attribute.

(setq gname (append gname (list ename)))

(setq group (append group (list gname)))

The variable **group** is the association list to which user defined object name are stored. It is the same list you see in the block attribute you inserted earlier.

The next line converts the list group into a string data type using a user defined function called ltos:

(setq sname (ltos group))

Ltos simply write the list represented by the symbol group to an external file then reads it back. The net affect is the conversion of a list into a string. This is done so the value of **group** can be used to replace the current value of the attribute in the namestor block. This is a situation where data type consideration is important.

Attribute values cannot be anything other than strings so if our program were to try to substitute a list in place of a string, an error would occur.

Extracting Attribute Data

The next several lines obtain the property list of the **namestor** block attribute and its attribute value in preparation for entmod:

(setq namevl (entget

(entnext (ssname (ssget "X" '((2 . "NAMESTOR")))0)))

)

Here, the ssget "X" filter is used to select a specific object, namely the block named "NAMESTOR". In this situation, since the name of the block is a fixed value, we include the name as a permanent of the expression:

(ssget "X" '((2 . "NAMESTORE")))

This helps us keep track of what block we are using and also reduces the number of variables we need to use.

Once the block is found by ssget, ssname gets the blocks' object name and entnext extracts the attributes' object name. Entget extracts the attributes property list which is assigned to the namevl variable. The line that follows uses the assoc function to extract the actual attribute value.

(setq namelt (assoc 1 namevl))

Figure 11.9 shows how this works.

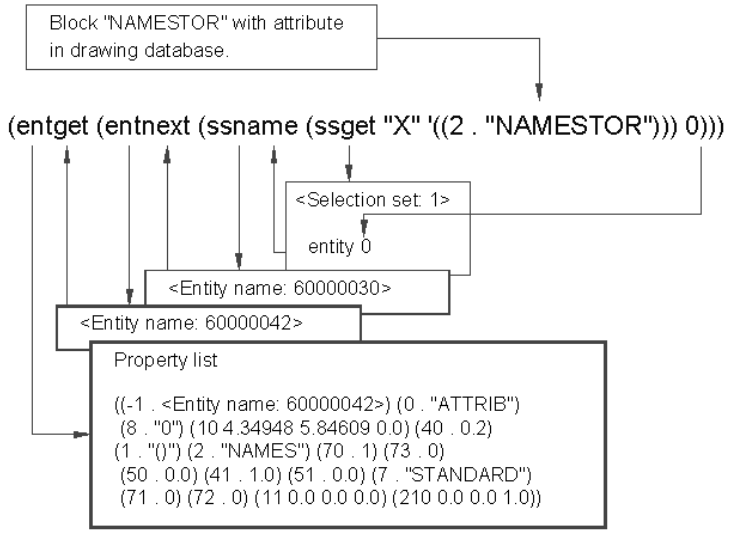


Figure 11.9: Extracting an attribute value

Just as we used entnext to extract the vertices of a polyline, you can use entnext to obtain the attribute information from a block. If there is more than one attribute in a block, you step through the attributes the same way you step through the vertices of a polyline. The Getatt function works in a similar way to these expressions you have just examined.

Finally, entmod is used to update the attribute to store the association list of the object name and object handle:

```
(entmod (subst (cons 1 sname) namelt namevl))

(entupd (cdr (assoc -1 namevl)))

(princ)

)
```

The subst function is used to substitute the newly appended name with the old attribute value in the attributes property list. Then entmod updates the drawing database with the updated property list. The entupd function is used to update the display of the attribute in the block. Entupd is only needed where attributes and curve-fitted polylines are being edited and you don't want to regenerate the entire drawing to update the display. You could think of it as a regen for specific objects.

The getname function is actually quite simple compared with namer.

```
(defun GETNAME (/ group gname getname handl nament newent)

(setq gname (strcase (getstring "\nEnter name of object: ")))

(setq group (getatt))

(handent (cadr (assoc gname group)))

)
```

Getname prompts the user for the name of the object to be selected. It then obtains the association list of name from the storing attribute using the user defined Getatt function. Finally, getname extracts the object handle from the association list using the name entered by the use as the key-value. The handent function returns the object name of the object whose handle it receives as an argument.

Namer and getname are fairly crude program as they have very little in the way of error checking. For example, if while using the getname function, you enter a name that does not exit, you get an AutoLISP error message. Also, if you attempt to save more than dozen names, you will get the out of string space error message. This is due to the 100 character limit AutoLISP places on string data types. There is also no facility to check for duplicate user supplied names. We wanted to keep the program simple so you won't get too confused by extra code. You may want to try adding some error checking features yourself, or if you feel confident, you can try to find a way to overcome the 100 character limit.

Conclusion

Programming can be the most frustration experience you have ever encountered as well as an enormous time waster. But it is also one of the most rewarding experiences using a computer can offer. And once you master AutoLISP, you will actually begin to save time in your daily use of AutoCAD. But to get to that point, you must practice and become as familiar as possible with AutoLISP. The more familiar you are with it, the easier it will be to use and the quicker you will be able to write programs.

We hope this tutorial has been of value to your programming efforts and it will continue to be helpful to you as a reference when you are stuck with a problem. Though we didn't cover every AutoLISP function in detail, In particular, we did not cover binary operations and a few other math functions. We did cover the major functions and you were able to see how those functions are used within programs solving real world problems. You were introduced to the program in a natural progression from entering your first expression through the keyboard to designing and debugging programs and finally to accessing the AutoCAD drawing database.

Appendix A: Menu Primer

You can find comprehensive information on AutoCAD Menus in the AutoCAD help system. Here are instructions for finding the section devoted to Menus.

1. In AutoCAD, choose Help > Developer Help from the menubar
2. Click on the Contents tab in the Help Topics dialog box, then expand the Customization Guide listing
3. Expand the Custom Menus listing. You will see the list of topics. Click on the topic of your choice.

You can also refer to Chapter 29 of Mastering AutoCAD for tutorials on menu customization.

Appendix B: Error Messages

Error codes have been completely retooled in AutoCAD 2002. AutoCAD 2002 now offers Visual LISP which is a programming environment designed for AutoLISP. There, you'll find an environment that helps you find errors quickly, You can open Visual LISP by choosing Tools > AutoLISP > Visual LISP Editor. The Visual LISP Editor offers many of the tools you expect to find in a modern programming environment.

A discussion of Visual LISP is not currently available in this document. You can find a tutorial and reference guide built into Visual LISP by choosing Help > Developer List. Click on the Contents tab to see a listing of topics including Visual LISP Developer’s Guide and Visual LISP Tutorial.

Visual LISP is an advanced tool that expects the user to have a working knowledge of AutoLISP. You may want to work with the ABC's of AutoLISP until you feel comfortable with AutoLISP.

Appendix C: Group Codes

You can find a comprehensive listing of group codes in the AutoCAD help system. Here are instructions for finding the group codes.

1. In AutoCAD, choose Help > Developer Help from the menubar

2. Click on the Contents tab of the Help Topics dialog box, then expand the DXF Reference listing

You can then expand the other options under DXF Reference for group codes pertaining to the selected topic.

Appendix D: System and Dimension Variables

You can find a comprehensive listing of dimension variables in the AutoCAD help system. Here are instructions for finding the dimension variables information.

1. In AutoCAD, choose Help > Help from the menubar

2. Click on the Contents tab in the Help dialog box then expand the Command Reference listing

3. Expand the System Variables listing

You will see a listing of system variables with a brief description to the right. You can click on the system variable name to get a more detailed description.