Nisus® Macro Programming Dialect Version 3

NISUS

The Amazing Word Processor for the Apple Macintosh®

☞ This booklet contains technical information for programmers. Nisus users do not need to read this in order to use any of the built-in features of Nisus.

Table of Contents

MACROS INTRODUCTION

The Nisus Macro Programming Language

The Nisus Macro Programming Language consists of two separate dialects: The Menu Command Dialect and the Programming Dialect. The Menu Command Dialect is designed to be easy to use and understand. It relies on the menu commands for all its commands. Combined with the PowerSearch and PowerSearch+ text description capabilities, the Menu Command Dialect lets you create very powerful custom features within Nisus.

The Menu Command Dialect is described and documented in your manual in Chapter 14, Working With Macros. This pamphlet describes the Programming Dialect, which is not for the beginner. Unless you have used and feel comfortable with the easier Menu Command Dialect in Nisus, you should not try using the Programming Dialect. This powerful dialect is provided for two reasons:

1. To give those who have reached the practical limits of the Menu Command Dialect tools to extend their ability to create new features.
2. To allow an easy way for us to create custom features at your request. If you are regularly faced with a time-consuming document processing job, it may be time and cost effective for you to contract with Paragon to write custom features. We will gladly give you a free estimate of the cost of a specific custom feature.

In previous versions of Nisus some users of the Menu Command Dialect requested a conditional (the "If" statement). In most cases we found that the user could perform the required task using the existing Menu Command Dialect, and perform it much more elegantly and efficiently than using the If () statement. Using PowerSearch and PowerSearch+ usually dispenses with the need to use a conditional.

For example, suppose you want to find all occurrences of the equal sign

(=) which are not preceded or followed by a space, and then make sure that there is a space on each side. Those who are used to thinking in languages like Basic or C would want to split the task into several steps something like:

1. Search until "=" is found.
2. Check if the previous character is a space; if not, insert a space.
3. Check if the following character is a space; if not, insert a space.
4. Repeat until finished.

Using Power Search+, the same task becomes a two line macro or two **Replace All** commands (anything which follows // is a comment):

```
// Replace all "=" not preceded by a space, with a space followed by the same "="
Find/Replace ":<[^\s]=" "\s&" "gta"
// Replace all "=" not followed by a space, with the same "=" followed by a space
Find/Replace "=:>[^\s]" "&\s" "gta"
```

Such is the power of the "PowerSearch+" language that tasks which would take many lines of code in a programming language like Basic, with the attendant errors, take only two very simple statements. In the Programming Dialect you can, although you do not have to, write macro programs following the steps in the example above; in the Menu Command Dialect it is almost impossible to write such macros. Our advice is not to go to the Programming Dialect language without first trying to use the Menu Command Dialect. The effort will usually be worthwhile.

The Programming Dialect is a full, BASIC-like programming language, though in many ways it is much more powerful. This implementation of the Programming Dialect lacks some constructs which make the creation of long and complicated programs easier. For example, in this version, all looping must be done with the **If()** statement and the **GoTo** command; there are no "while" or "until" loops. The conditional "If()" statement does not have an "Else()" to follow. Of course, neither of these constructs are necessary, and their absence does not reduce the functionality of the language in any way.

In Nisus you have a very large number of features available, each of which is accessible as a macro command. Therefore, you should rarely need to write long and complicated macros. When writing programs in the Programming Dialect you need to be aware that you can also use the commands from the Menu Command Dialect, though you cannot mix commands from the two dialects in one line.

The following are the rules for writing macros:

All reserved words are not case sensitive (except **DOUBLE, LONG, SelectWithRuler, DoubleLock,** and **ReportErrors,** which are reserved variables and are case sensitive).

Statements in the Programming Dialect are separated by carriage returns or semicolons. Any non-reserved word can be used as a variable. Variable names are case sensitive and can only consist of the standard alphanumeric symbols. The first letter of a variable name must be an alphabetic or underline, all subsequent letters can be either alphabetic, underline or numeric. The extended ASCII characters, greater than ASCII 127, are not allowed in variable names. The maximum number of characters in a variable name is 255, all of which are significant. All variables must be one word; that is, they cannot contain spaces, though they may contain the underline character. No declaration of variables is necessary. The first statement assigning a value to a variable (the first time a variable is used followed by an equal sign) creates the variable and at the same time defines its type. All integers in Programming Dialect are Long; that is, they are 32 bit signed integers. The maximum value of an integer is 2,147,483,647, and its minimum value is -2,147,483,648. For example, the statement myData = 10 assigns 10 to the variable myData which becomes an integer variable of type Long. A subsequent assignment statement such as myData = 6.5 will truncate the 6.5 and store the whole number 6 in myData.

Any non-reserved word (except a number, which has a special meaning in the Menu Command Dialect) on a line immediately followed by a colon (:), is identified as a label. Labels can be used in GoTo statements to give you complete control of program execution, including looping and conditional branching. The Programming Dialect is an interpreted language. Compilation of the code is not supported at this time. All macro commands are first checked for their meaning in the Programming Dialect, and, if no meaning is found for a command, interpreting is passed to the Menu Command Dialect interpreter for processing. After processing and execution of each Programming Dialect line, control is once more passed to the Programming Dialect interpreter and the process repeats.

Variables, Data Types and Expressions

Variable related commands are listed in the following table:

free FreeAll FreeGlobal	FreeLocal global	local type

☞ *For more information on variable functions refer to the "**Macros Command Listing Glossary**" on p. 36.*

One of the most important additions to Nisus macros is the use of variables and the ability to construct algebraic, string and boolean expressions using variables. For example, it is possible to construct all of the following expressions with the new macro language (functions appear in boldface):

Math Expressions:

```
myVar = 3.1416 + 2 * ((-85 + -10E-35) / 2.45)
myVar = 5 * 5 + 3^(2./4) + 5 * (6.21 - Tan(ACos(-1)))
myVar = √3
myVar = 5 * 5
```

```
myVar = myVar + 36.5 - 55
currentTime = time
oneHour = 60 * 60
quittingTime = oneHour * (12 + 5)
xMasEve90 = MakeDate (24, 12, 1990)
```

Boolean Expressions:

```
myTest = 5 > 6^31 - 3 * time
if (myVar->type != "UNKNOWN") myVar = myVar + 1
if (myVar) clipboard = myVar
if ( !myVar) clipboard = "myVar is zero"
if ( (date >= xMasEve90 && date < xMasEve90 +
  oneHour * 24 ) && (time > xMasEve90 + quittingTime) )
  skiAngleFire = TRUE
if ("This" > "That") goto doSwap
```

String Expressions:

```
myString = "Hi ya bub!"
myString = "The time is " + time + ", and the date
  is " +date+ "."
noCommas = "9,123,456,789" / ","
noHiYa = "Hi ya bub!" - "Hi ya "
theBub = left(right("Hi ya bub!", 4), 3)
myString = "π/2 = " + (ACos(-1) / 2)
// Note: ACos(-1) / 2 must be parenthesized in order to be evaluated before conversion
   to string.
• currentClipboard = clipboard
```

As you can see, Nisus supports several data types. They are the following:

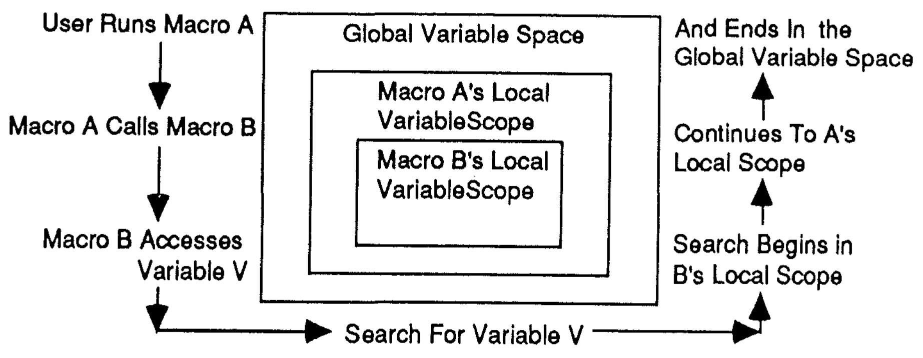
- long integer: *I*, in the range -2,147,483,648 ≤ *I* ≤ 2,147,483,647
- double precision floating point: *F*, with 18 digits of accuracy past the decimal point with a maximum value of 1.18973E+4932.
- boolean: *B*, with a value of either **true** or **false**.
- string: *S*, a pure text string, no character attributes like font, size etc., up to 2K - 1 = 2047 characters in length; e.g., "this is a string".

- date/time: *D,* technically is the number of seconds since Midnight, January 1, 1904. When a date/time type is used in a string the result is in the date format that you have set in your document. When a date/time type is used in a comparison or assigned to a numerical variable, a double float representing the number of seconds since Midnight, January 1, 1904, is used in the compare or assignment.

Variables and functions in Nisus are unique among programming languages for several reasons. The type of a variable is not explicitly declared; the type of a variable is derived from its use in an assignment statement. Data types can be mixed in any assignment statement or expression. All functions are designed to accept any data type, automatically casting their arguments to an appropriate type before they are used.

Variables are scoped locally and globally. Since the nature of scoping allows for many variables with the same name, it is important to understand how it works. When variables are autoinstantiated, they are implicitly defined to have a local scope. Locally scoped variables can be accessed only by the macro in which they were created as well as macros that are called by the creating macro. Local variables are automatically deleted when the creating macro exits; global variables remain between macro runs and are not deleted unless specified by you with the **FreeGlobal** command.

When a macro accesses a particular variable name, a process begins that tries to locate the variable. The search for the variable begins in the current scope, which is the scope of the currently running macro. If the variable is not found in the current scope, the search continues up the chain of macro calls made, looking in each of the preceding macros' respective scope. If the variable is not found in any of the running macros' scopes, the final place that is searched is the global variable space. The following diagram illustrates this process:


User Runs Macro A
Macro A Calls Macro B
Macro B Accesses Variable V
Global Variable Space
Macro A's Local VariableScope
Macro B's Local VariableScope
And Ends In the Global Variable Space
Continues To A's Local Scope
Search Begins in B's Local Scope
Search For Variable V

If the variable is found during a search, the search stops and that variable is returned to the macro that initiated the search. If the variable is not found, it will be autoinstantiated in the current macros' local scope, its data type derived when data is assigned to it.

There are two commands at your disposal that allow you to control a variable's placement in local and global space. They are the **Local** and **Global** declaration commands. When a variable is defined and you have not used the **Local** or **Global** command; by default, a variable is placed in the defining macros local scope upon instantiation. Once a variable has been defined, no new instance of that variable name will be created unless you use the **Local** command to force an instantiation in the current scope. The following sections will illustrate the different cases of local and global instantiation.

Default Case, Implicit Local

- Macro *A* creates variable *V,* with the statement *V* = 1. Once *V* is defined, it can be accessed anywhere within *A's* scope.
- Macro *A* then calls macro *B.* The variable *V,* which was created in macro *A,* is accessible to macro *B,* because macro *A's* scope encompasses macro *B's* scope. The advantage here is that you do not have to pass *A's* variables to macro *B.*

- Macro *B* can do anything it wants to variable *V*. The changes to *V* remain after macro *B* exits, returning control to macro *A*. This is an important point, because this is a unique treatment of local variables and is different from other popular programming languages (e.g., if *B* sets *V* equal to 2, *V* = 2, then upon return to *A*, *V* will still be equal to 2, not 1).

The default case has some advantages, but this may not be what you want. One reason why you may not want macro *B* to access macro *A*'s variable *V* is that the variable *V* may be a generic name for a counter in a loop in *A*, and changing *A's* looping counter in macro *B* would have drastic effects. To get around the problem, use the **local** command.

Local Declaration

In the following discussion, the subscript to variable *V* specifies the macro in which it was created and defined; i.e., its scope.

- Macro *A* creates variable *V*, which we will call V_A, with the statement $V_A = 1$. Once V_A is defined, it can be accessed anywhere within *A's* scope.
- Macro *A* then calls macro *B*. The variable V_A, which was created in macro *A*, is accessible to macro *B*, but we want macro *B* to have access to its own variable V_B so as not to effect V_A. We accomplish this with the statement **local** *V*, V_B now exist in *B's* local scope.
- Macro *B* can do anything it wants to variable V_B, and V_A will not be affected; in fact, the existence of V_B means you can not even access V_A from inside of macro *B*. After macro *B* exits, returning control to macro *A*, V_A contains the original value (e.g., if *B* sets V_B equal to 2, $V_B = 2$, then upon return to *A*, V_A will still be equal to 1, not 2).

Now we come to the final case, global variables. Global variables remain between runs of macros, they are not automatically removed from global variable space.

Global Declaration

- Macro *A* declares variable *V*, which we will call V_G, to be global, using the command **global** *V*. Once V_G is defined, it can be accessed anywhere within *A's* scope or within any other macro, for that matter.
- Macro *A* exits and returns control to the user. V_G is global and is not deleted. The user then runs macro *B*. Macro *B* also has access to V_G. V_G will have the same value that macro *A* last assigned to it.

If you want to hide access of a global variable to a macro, or temporarily override a variables value—like the value for the reserved variable **DOUBLE**—you can use the **local** command to declare a new instance of the variable.

☞ *For more information on variables, refer to the three sections in "Macros Technical Information" in this booklet, titled "Variables," p. 25, "Autoinstantiation," p. 29, and "Autotypecasting," p. 30.*

Numerical & Math Functions

Nisus has a number of math functions. All of the math functions typecast (convert) their arguments to the double precision floating point data type, and return a double precision number for the result and are therefore accurate to 18 digits. The following table lists all of the math functions currently available in Nisus:

acos	cos	exp	ints	modf
asin	cosh	fabs	log	pow
atan	div	floats	log10	random
atan2	DOUBLE	floor	LONG	rrandom
ceil	DoubleLock	fmod	mod	Seed

☞ *For more information on math functions refer to the "Macros Command Listing Glossary," p. 36.*

Since most of the calculations done with math functions use double precision numbers it is important to know how to format them to output or text. When printing out floating point numbers, you should be aware of the reserved variable **DOUBLE**, that is used to specify the formatting of floating point numbers when they are converted to text. The default value for **DOUBLE** is "%lG". If you have ever programmed in the C language, you are probably familiar with the formatting specifications for the printf() function. Nisus macros use printf() to format all numerical data for output or text. The following are some examples of formatting double precision numbers:

Default Format of "%lG"

- 3.141592653589793239 formats to 3.14159
- -21.123e-26 formats to -2.1123E-25

Format of "%.18lf"

- 3.141592653589793239 formats to 3.141592653589793239
- -21.123e-26 formats to 0.000000000000000000

Format of "%.18lg"

- 3.141592653589793239 formats to 3.14159265358979324
- -21.123e-26 formats to -2.1123e-25

☞ *By changing the 18 in the above two examples you can control the number of digits past the decimal point that will be printed. For more information on printf() formatting commands, refer to the section in "Macros Technical Information," titled "Formatting Specifications," p. 32.*

☞ *Any number outside of the range of a long integer, any number that has a fractional part or any number that has a decimal point in it, is considered a double precision floating point number. Placing a decimal point in a number is one way to guarantee that a double precision number will be created. The other way to guarantee that a double precision number will be created is to set the variable* ***DoubleLock*** *equal to* ***true.***

String Functions

All of the boolean operators and the arithmetic operators add, subtract and divide (+, -, /), allow strings as operands. In the case of boolean operators, like grater than (>), an ASCII comparison results returning a boolean value; e.g. "this" > "that" is equal to **true.** For the arithmetic operators refer to the Macro Command Listing Glossary, p. 36.

Nisus supports several string related functions and commands. They are the following:

CharToNum clipboard left LowerRoman length	MacroCopy MacroPaste mid NumToChar offset	right strings UpperRoman

☞ *For more information on string functions refer to the "Macros Command Listing Glossary," p. 36.*

The following example illustrates a few of the string functions.

```
// This macro takes a date in short format and makes sure that all numbers are two
   digits in length; e.g., 8/3/90 reformats to 08/03/90. This is useful because you can
   then correctly sort on the reformatted date.
shortMonth=""
shortMonth = makeDate(day,month,year);
num1 = 0; num2 = 0; num3 = 0;      // Get the month, store in num1.
monthLen = length(shortMonth);
offset1 = Offset("/", shortMonth);
num1 = Left(shortMonth, offset1 - 1);
// Get the day, store in num2.
```

```
offset2=offset1+Offset("/",Mid(shortMonth, offset1+1,
  monthLen));
num2 = Mid(shortMonth, offset1+1, offset2 - offset1);
// Store the year in num3.
num3 = right(shortMonth, monthLen - offset2);
// Setting LONG to this format guarantees a field width of two.
local LONG; LONG = "%021d";
// Reconstruct the date, with the correct field width.
shortMonth = num1 + "/" + num2 + "/" + num3
```

Document Functions

Document functions are listed in the following table:

CharNum DateCreated DocName DocPath EndCharNum ends	LastDateSaved LineCharNum LineNum NextRulerStart PageLineNum PageNum	RulerName RulerStart SelectEnd SelectRuler SelectStart SelectWithRuler	SetSelect SetSelectMore StartEnds starts TimeModified

For more information on document commands and functions refer to the "Macros Command Listing Glossary," p. 36.

Here is an example using a few of the document functions. For more examples, examine the Outlining Macros that are shipped with Nisus.

```
// This macro copies all of the noncontiguous selections, with their governing
   ruler into a new document and then prints this new document. The micro
   rulers must be displayed before this macro is run in order for it to work
   correctly.
// If no selections, exit.
if (selectStart == selectEnd) exit
// Save all of the selections start and end points.
s->qPush(startends);if(s->error)goto reportError
// Create a new file to place the selections. printFile is the name of the new file.
9
```

```
printFile = "PrintFile @ " + (time / ":"); clipboard
  = printFile
0
New '\C9'
Toggle Front 2
oRulerStart = -1
// loop until no more selections.
printLoop:
  if (!s -> size) goto doPrint
  //     Select one section of text at a time. Verify the ruler hasn't changed.
  SetSelect(s[1],s[2]);
  if(rulerStart==oRulerStart)goto addSelection
  // This selection has a different governing ruler, copy and paste it.
  SelectRuler(s[1])
  Copy
  Toggle Front 2
  SetSelect(endCharNum, endCharNum)
  Paste
  Toggle Front 2
// Copy and paste the selection into the new document, loop back for next sel.
addSelection:
  SetSelect(s[1], s[2]); oRulerStart = rulerStart
  find/replace ":e" '&\r' "gs" '\C9'
  s -> pop; s -> pop; goto printLoop
// Finish up by printing the new document.
doPrint:
```

Looping Constructs

Looping constructs are listed in the following table:

error exit false	if GoTo *label:*	NumFound stop true

For more information on looping commands and functions refer to the "Macros Command Listing Glossary," p. 36.

Nisus provides you with very simple looping constructs that consist of an **if ( )** statement, a **GoTo** statement, a **labelName:** statement, an **exit** statement and a **stop** statement. The **if** statement evaluates any combination of algebraic and logical expressions between a pair of parentheses. If the expression between parentheses **( )** evaluates to **true** or a non zero numeric value or a non-empty string, then *all* of the statements immediately following the right parentheses **)** to the end of line will be executed.

Looping Example:

```
// This macro numbers the beginning of every paragraph with an uppercase roman
   number, starting with roman numeral romanStart. If you want a different start then
   change the assignment to romanStart below.

romanStart = 1

// Select All paragraph starts and push them on the stack
Find All "^:." "goT"
s -> push (Starts); if (s -> error) GoTo reportError
counter = NumFound + romanStart - 1

// Loop over all paragraph starts, in reverse order, so changes to the document do not
   affect the starts stack, pasting in the new roman number.
loop:
   if (! s -> size) exit
   SetSelect(s -> first, s -> pop)
   clipboard = UpperRoman(counter)
   paste "" '\CC'
   counter = counter - 1
GoTo loop
```

☞ *Currently, no short circuiting occurs in the if( ) statement. This means that every expression between the if statement's left and right parentheses must be evaluated before the final boolean result can be determined.*

☞ *Label names are immediately followed by a colon; e.g., myLabel:.*

☞ *Labels are not really statements. They must appear on a line by themselves. They cannot appear at the beginning, middle, or end of multiple statements on a single line.*

Data Storage & Manipulation Functions

Data storage and manipulation functions are listed in the following table:

create	ints	qpush
ends	last	shuffle
error	pop	size
first	push	StartEnds
floats	qcreate	starts

☞ *For more information on data storage commands and functions refer to the "Macros Command Listing Glossary," p. 36.*

One of the first data storage types you learn about in programming is the array. A simple array is a sequentially numbered list of data elements from which you can store and retrieve data. One analogy, for example, is a row of houses on a street, each having a mail box with a unique address. You can send mail (store) to any one of the mailboxes and the residents can pick up (retrieve) their corresponding mail. As you advance in programming you may learn about different types of storage. Nisus has combined three different types of data storage into one. The storage class in Nisus has the qualities of a single dimensioned array, a stack and a queue all merged into one.

The Nisus storage class gives you the capability to push data onto a stack and then access the data on the stack in array fashion. Its main advantage is that you do not have to worry about maintaining the size of the storage. Array indexing variables can be kept to a minimum with the use of stack and queue operations.

Additionally, one final difference between Nisus and classical programming languages is that the Nisus storage class is nonhomogeneous. In other words, you can mix the types of data you store. You can store strings along with doubles and longs. In this sense it is like a *record* in Pascal, or *struct* in C.

Accessing Storage as an Array

There are two ways of accessing a storage element as an array. The first is the traditional array accessing method using an index value between a pair of angle brackets. The following example demonstrates such an access:

```
myData -> create(3)
myData[3] = " is the answer"
myData[1] = 123.5
clipboard = myData[1] + 50 + ": " + myData[3]
//  clipboard is now equal to "173.5: is the answer".
```

☞ *Array index values will be autotypecast to a long integer when necessary.*

☞ *Autoinstantiation does not automatically create myData's storage elements 1 through 3, as in the example above. You must use the **create** or **push** commands to accomplish this.*

☞ *Arrays are one based; i.e., the first element of an array has the index 1 (e.g., the first element of myData in the example above, is myData[1]).*

The second possible type of array access is a relative array access. The access will be relative to the previous access of storage. The index value used in a relative access can be positive, negative, or zero. A relative access is signified when the index value appears between double angle brackets.

The following example illustrates the syntax and use of relative accessing, assuming that the active document has some noncontiguous selections:

```
myData->push(starts)      //  Stack push the selection's start for the set
                          //  of noncontiguous selections.
//  Set insertion point to beginning of the first selection.
SetSelect(myData[1], myData[1])
loop:
//  Terminate the loop when all data is used up, determined by a bad access.
if (myData -> error) exit
```

```
Paste "" "•"  // paste a bullet at each saved start.
//  Note, both of the following relative accesses point to the same data.
SetSelect(myData[[1]], myData[[0]])
GoTo loop
```

Relative access, to move to the next selection's start.

☞ *When a relative access goes beyond the bounds of the number of elements stored, an error flag will be set in the storage variable. This flag can be tested as in the above **if** statement. The storage error flag will also be set if there are any other kinds of storage errors, namely memory errors.*

Accessing Storage as a Stack or Queue

If you are not familiar with stacks and queues here is a brief description: Stacks and queues are typically taught as LIFO and FIFO arrays. LIFO is an acronym for Last In First Out and FIFO is an acronym for First In First Out. An easy analogy for queues is as follows. If you are the *first* one in line at the airport waiting to board an airplane you will be the *first* one from that line to get your seat. This analogy demonstrates FIFO, or queue operations. In LIFO, or a stack operation, exactly the opposite is true. In our analogy, let's change the reality of waiting in line to be analogous to a stack operation. Now, if you are the *last* one in line you are going to be the *first* one to get your seat. Using stack and queue operations in Nisus can be thought of as placing people in the front of or at the end of the line, respectively. Nisus always removes people from the front of the line (**pop**). So, if you add people to the end of the line (**qpush**), you have created a queue. If you add people to the beginning of the line (**push**), you have created a stack.

All stack and queue operations in Nisus use a method call from the storage variable to invoke the operation. Object oriented programming (OOP) defines methods as functions that are associated with a specific set of data. A collection of data and the methods that operate on that data are defined as a *class* in C++. This is why we have called the Nisus storage type a storage class.

There are currently 11 methods associated with the storage class. The synta .or accessing a method is the following:

```
myVar -> method
```

"myVar" is the storage variable name, "->" signifies a method call, and "method" is the name of the method you are calling. Some storage class methods accept an argument that you pass between parentheses. In this case the syntax looks like this:

```
myVar -> method(arg)
```

All storage class methods set an error flag in the storage variable when they are unable to accomplish their operation. This error flag is accessed using the **error** method described below. The following is a list of the methods associated with the storage class and their corresponding description:

push*(E)*	Pushes the result of the expression *E* onto the front of the storage variable, storing the result in a newly created element. The new element will then be the *first* element of the storage variable. *E* can be any variable name or constant, any data type excluding storage variables. **Push** does not return any value. Please read below about the optional keywords which can be passed as alternative arguments to **push**.
qpush*(E)*	Pushes the result of the expression *E* onto the end of the storage variable, storing the result in a newly created element. The new element will then be the *last* element of the storage variable. *E* can be any variable name or constant, any data type excluding storage variables. **Qpush** does not return any value. Please read below about the optional keywords which can be passed as alternative arguments to **qpush**.
pop	Removes the first element from the storage variable, shrinking the storage variable size by 1. The value returned by this method is the contents of the element that was removed from the storage variable.

create*(I)*	Adds *I* number of empty elements to the front of the storage variable. You are required to create the elements of a storage variable before you can access them. Remember, **push** and **qpush** create elements for you automatically. **Create** does not return any value.
qcreate*(I)*	Adds *I* number of empty elements to the end of the storage variable. You are required to create the elements of a storage variable before you can access them. Remember, **push** and **qpush** create elements for you automatically. **Qcreate** does not return any value.
first	Returns the contents of the first element. This method will not effect the size of the storage variable.
last	Returns the contents of the last array element. This method will not effect the size of the storage variable.
shuffle	Causes the contents of every element to be swapped with the contents of another randomly selected element, mixing up the contents of the storage variable.
size	Will return the number of elements in the storage variable.
swap(I_1,I_2)	Swaps the contents of element I_1 with the contents of element I_2.
error	If any of the above methods, as well as all indexed accesses, have encountered an error, then an error flag will be set for that storage variable. This method returns a boolean value indicating the state of that flag.

☞ *It is a syntax error to use a storage class variable in any context that does not include a call to a method or an indexed array access.*

Optional Keyword Arguments for Push and Qpush

For the purpose of selecting groups of similar types of data, there are reserv keywords that you can use as arguments to **push** and **qpush**. The keywords are mainly used in conjunction with a noncontiguous selection. (You can create a noncontiguous selection with the rectangular selection tool (Option-select), or **Find All** from the Find/Replace dialog or modified (hold down Option) Tools menu.)

The syntax for using the keywords is as follows:

```
    myVar -> push(keyword)  or  myVar -> qpush(keyword)
```

The following table lists the keywords and their descriptions:

strings	Stores all of the elements of the noncontiguous selection in the storage variable as individual text strings.
ints	Stores all of the elements of the noncontiguous selection as long integer numbers in the storage variable. This is accomplished by converting the text of the noncontiguous selection to a number.
floats	Stores all of the elements of the noncontiguous selection as double precision floating point numbers in the storage variable. This is accomplished by converting the text of the noncontiguous selection to a number.
starts	Saves all of the selections' starts of the noncontiguous selection in the storage variable.
ends	Saves all of the selections' ends of the noncontiguous selection in the storage variable.
StartEnds	Saves all of the selections' starts and ends of the noncontiguous selection in the storage variable. The order in which they are saved is selection's start followed by selection's end. This order is the same for both **push** and **qpush**.

☞ *Even though noncontiguous selection was used in the above keyword descriptions, it does not mean you have to have a noncontiguous selection. Any selection will work.*

Example of Storage Use:

```
// This routine sums all of the values in a noncontiguous selection. The selection is
   currently made on a column of numbers using the rectangular selection tool
   (option-select).
sum = 0.0; s -> qPush (floats)   // push all selection as floats
if (s->error) GoTo reportError   // make sure we
                                    got all of the selections.

loop:
  sum = sum + s -> pop
// add to the total
if (s->size) GoTo loop
// while data is in queue.
Clipboard = sum
// place result in clipboard
paste
exit
// We had an error with the storage, probably low on memory.
reportError:
paste "" '\rERROR: Unable to collect all numbers in
  selection.'
```

Da s and Times

The three functions **MakeDate**, **Time** and **Date** return a date/time data type. This data type can be used in mathematical calculations like any other number. But, when the date/time data type is used in string calculations, or typecast to a string, it converts to a readable text string. The Hong Kong time zone conversion macro below illustrates this principle.

Dates and times functions are listed in the following table:

date day DayOfWeek	hour MakeDate minute	month second time	year WhatDay

☞ *For more information on date and time functions refer to the "Macros Command Listing Glossary," p. 36.*

An example piece of code from the Create Calendar Macro:

```
// This macro excerpt finds the day of week a user supplied month begins on and also
   the total numbers of days in that month.
// Get the month and year they want a calendar for.
theMonth = 0; theYear = 0
clipboard = month
:1 "What Month (1-12) ?" '\CC'
eval 'theMonth = \CC'
if (theMonth <1 || theMonth > 12) GoTo reportInputError
clipboard = year
:1 "What Year (19XX) ? " '\CC'
eval 'theYear = \CC'
if (theYear < 1904 || theYear > 2039) GoTo
  reportInputError
// Calculate the start of the next month.
endMonth = theMonth + 1; endYear = theYear
if(endMonth == 12)endMonth = 1;endYear = endYear + 1
// Then calculate the total number of days in this month.
totalDays = 0.0;
```

```
startDay = MakeDate(1, theMonth, theYear);
endDay = MakeDate(1, endMonth, endYear);
totalDays = (endDay - startDay) / (3600 * 24)
// Find out what day the month starts on, 1-7 respectively equals Sunday through
   Saturday.
calandarStart = WhatDay(startDay) - 1
```

An example of a timer:

```
myTime = 0.0; myTime = time
:1 "Click on OK when you want to stop timing."
myTime = time  - myTime; clipboard = myTime + "
  total secs."
:1 '\CC'
```

Report current date and time example:

```
clipboard = "Today is " + date + " and the time is
  " + time
// Will display as: Today is 8/5/90 and the time is 3:14:03 PM
:1 '\CC'
```

Report current date and time in Hong Kong if you are in the Pacific time zone (no daylight savings) example:

```
// Hong Kong is 16 hours ahead.
clipboard = "Hong Kong date and time: " + (date +
  16 * 3600) + " and the time is " + (time + 16 *
  3600)
// Will display as: Hong Kong date and time: 8/6/90 and the time is 7:14:54 AM
:1 '\CC'
```

As you can see in the above example the date/time data types returned by the **date** and **time** functions can be used in arithmetic as well as string operations. The comment // *Will display as..* shows you the final result.

You may have noticed the number 3600 used a few times in the above examples. This is because when dealing with a date/time data in a in o hour and is used in the calculations for unit conversions.

☞ *The date format is the same as the front document. Use the **Set Date & Page# Formats** command to change the date format.*

MACROS TECHNICAL INFORMATION

Boundary Between Menu Command Dialect and Programming Dialect

There is a definite boundary between the Menu Command Dialect and the new Programming Dialect. You absolutely cannot mix commands from the old and new macros on a single line. Data passing is limited to the clipboards and the result of GREP matches between \(and \); i.e., \1, \2, ..., \9. The reserved word **clipboard** and the **eval** function are used to facilitate the passing of information from one side to the other.

You should not find this a severe limitation. You will probably inadvertently mix the two dialects and wonder why things do not work. For this reason, you should study the macros that are shipped with Nisus.

Variables

Variable names must begin with an alphabetic (a-z or A-Z) or the underscore (_) character and thereafter may contain the numbers 0-9. The names of variables, unlike function commands and reserved constants, are case sensitive; e.g., dog and Dog are not the same variable. Variable name lengths are limited to 255 characters.

Nisus provides global as well as locally scoped variables.

Local Variables

Local variables can only be accessed in the macro where they were created, as well as all the macros that the creating macro calls. Local variables are deleted upon exit of the variables creating macro.

Global Variabl

Global variables can be accessed by any macro (unless a local variable with the same name overrides the global's scope). Global variables remain between macro runs and are only destroyed when you explicitly do so with one of the free commands. Unless explicitly stated, variables will always be created in a local scope. You can use the **local** and **global** commands to force a variables scope.

☞ *The macro facility does not yet fully support recursive types of procedures; macros are currently limited to a nesting depth of 20 macros. A macro may call itself, creating no new scope, and this is not considered a recursive call. This feature may change in the future, so it is advised that you do not write macros that depend on it.*

☞ *For more information about how variables are treated in Nisus, read the section in this booklet titled "Autoinstantiation," p. 29.*

Comments

The new macro language supports the // (two forward slashes) comment style of C++. Anything to the right of and including //, to the end of the line, will be replaced by a space, effectively ignoring the comment.

Statements

The new macro language is executed in statements. The parser for the new macros can only execute statements one line at a time. Statements cannot cross multiple lines; i.e., they cannot cross lines separated by carriage returns.

Multiple statements can be placed on a single line when the statements are separated by semicolons (;). The last statement on the line does not require a semicolon.

xample of multiple statements on the same macro line:

```
if (numfound) i = i + numfound; clipboard = ""; GoTo
  finishUp
```

☞ *The commands that make up statements, unlike label names and variable names, are not case sensitive. This means that the statement if ( ) is equivalent to If ( ).*

☞ *You do not need a semicolon following the last statement in any line.*

☞ *Currently, the longest statement line that can be executed is 2047 characters long.*

☞ *If a statement does not execute in either the menu command dialect or the programming dialect, then a warning dialog will be presented containing the offending statement. If you do not want the warning dialog to appear, set the variable **ReportErrors** equal to **false**.*

The "eval" Statement

Eval, short for evaluate, is used to pass information from the clipboards, and found expressions from PowerSearch+ (GREP), to the programming dialect. If you have used the **Execute Selection** menu command (located under the Tools menu in the Macros submenu) in a macro, then you are already halfway toward understanding what **eval** accomplishes for you.

Eval allows you to create a statement line in the new macro language at run time. **Eval** takes an argument string between single quotes ('). **Eval** will expand the metacharacters between two single quotes and then pass the newly expanded string onto the parser to be executed.

Example:

```
// Prompt the user, get the name of the search key and place it in the current clipboard.
:1 "Input the name of the search key." ""

// The metacharacter for the current clipboard is \CC use it in conjunction with eval to
   assign it to our own variable, myString.
eval 'myString = \CC'
```

☞ *There can only be one eval statement on a single line, and it must be the first statement.*

☞ *Currently, the longest statement line you can create and execute with eval is 2047 characters long.*

☞ *Unlike the menu command Execute Selection, you cannot recursively nest eval statements.*

Expressions

Expressions are calculations that are evaluated to some value, including algebraic and logical; e.g., 3 + 5 is an expression that evaluates to 8. They are also any function that returns a value. Expressions appear on the right hand side of an assignment (=) statement. Expressions appear between the pair of parentheses in an **if ( )** statement. Finally, expressions can appear between the parentheses of any function that requires arguments—the result of the expression becomes the argument that is passed to the function.

☞ *Functions, unlike label names and variable names, are not case sensitive. This means that a function like **numtochar**(25) is equivalent to **NumToChar**(25).*

Operators and their Order of Precedence

Algebraic and logical operators, as well as functions, are evaluated from left to right and are evaluated in order of precedence, from highest to lowest, as shown in the following table:

Highest	**(), ->**
↓	**Functions including ^ (power) and √ (square root).**
	!
	***, /, mod, div**
	+, -
	<, <=, >, >=
	==, !=
	&&, ||
Lowest	**=**

☞ *All operators have left to right associativity except for √, ! and = which* have right to left associativity.

Operator Overloading (Polymorphism)

The +, - and / operators are overloaded in function, meaning that they have a different operation when one or both of the operands is a string. In the case of +, the strings are concatenated. In the case of -, the first occurrence of the string on the right of - is removed from the string on the left of -. And finally, in the case of / (divide), all occurrences of the string on the right of / will be removed from the string on the left of /.

☞ *As a result of operator overloading and autotypecasting, described in "Autotypecasting," p. 30, a nonassociative algebra results; e.g., "hi ya" + 3 + 5 is equal to "hi ya 35" and is not the same as "hi ya" + (3 + 5) that is equal to "hi ya 8"*

Autoinstantiation

In most programming languages, you have to define the variable's data type before you can use it. In Nisus, variables are autoinstantiating. This means that you do not have to define the type of variable before its use. The type of a variable is derived from its use in an assignment statement. For example, in the following assignment statement, the variable "tmp" will be created and it will be of type double:

```
tmp = 3.1415
```

Autoinstantiation is satisfactory in most cases, though in some cases you may want to force a variable to be of a certain type. If you need a variable to be of a certain type, you must use a "dummy" assignment statement (variable = data) to a constant of the correct type before the variable's actual use.

Autoinstantiation will not occur if the variable already exists in the local variable space or the global variable space. You can create a new instance in the outermost local scope by using the ***local*** *command.*

A special case of autoinstantiation exists with the storage class. To autoinstantiate a storage class variable, you must use one of the storage functions.

Autotypecasting

In most programming languages, you cannot mix up different types of data in expressions or as arguments to functions. In Nisus, data is autotypecast when necessary to complete an operation.

In algebraic and boolean expressions, data is typecast to the highest type used in an operation. The typecast hierarchy from highest to lowest appears in the following table:

Highest	**String**
	Boolean
↓	**Date/Time**
	Double
Lowest	**Long**

Operations where this kind of typecasting occurs are in the following table:

*	/
+	-
<	<=
>	>=
==	!=

* operator is not defined for string data types.

Autotypecasting does not occur if the operands are of the same type.

There are three boolean functions that always cast their arguments to boolean before the operations occurs. They are the logical **not, and** and **or** (!, &&, || respectively) functions.

Most functions expect some kind of input from you in the form of function arguments. A math function like **cos( )** (trigonometric cosine of an angle in radians) only operates on double precision data. It is not necessary for you to explicitly pass a double precision number in the **cos** functions argument list. Like all functions in Nisus, **cos** will typecast the data you pass to the data type it needs. In the case of **cos**, the data will end up being typecast to a double. For example, passing a string to **cos, cos** ("3.141592654"), will work and return a value of -1.

All math functions cast their arguments to doubles. Functions that work on strings always cast the first argument to a string and additional arguments to longs, except the **offset** function which casts both of its arguments to strings. The majority of the remaining functions cast their arguments to longs. Finally, the storage class typecasts array indices to longs before the storage access occurs.

Refer to the section called ***"Macros Command Listing Glossary,"*** *p. 36, to see exactly what data type a function is going to typecast its arguments into.*

Autotypecasting does not occur here when there is no need for it; i.e., the type of data you pass to a function is exactly what the function expects.

Sometimes you do not want the result obtained from autotypecasting. You can force a cast from any type to any other type with an assignment statement.

Example:

```
myLong = 123456              // init myLong
myString = "hello there"     // init myString
myString = 500               // "500" is now a string.
myLong = myString            // 500 is now a long.
```

You may need to do this, for example, if you are doing typical math calculations; i.e., you will not be able to use strings, like **clipboard,** in a mathematical calculation without first casting the string to a number. (Remember, **clipboard** is a predefined string variable representing the current clipboard.)

Formatting Specifications

Date and Time Formats

The format of date can be changed by using the **Set Date & Page# Formats** menu command. The time can be changed to a 24 hour format and back again via the Control Panel Desk Accessory in the Apple Menu.

Double and Long Formats

When a Double or Long data type is converted to text, its format is under your control. The information stored in each type variable will be formatted according to some standard default format, unless you specify another format before conversion to text. The specification is done in terms of a simple assignment statement.

Each of the type names can be assigned a string which is used to specify the format of any subsequent variable of that type. For example, with the assignment statement **DOUBLE** = "%4.11lf", all variables storing numbers in double precision (i.e., all non-integer numbers) and accessed after this statement in the macro, will be formatted according to the %4.11lf specification.

The meaning of the codes in the specification string are the same as those in the standard ANSI C-Language Printf() function[1]. The various codes are quite extensive, though for most purposes, only a very small subset is required. The listing of the codes is given next. The examples below will illustrate the most frequently used formats. Notice that the formatting variables DOUBLE and LONG are in all caps. These variables are case sensitive, so all caps are required.

[1] For more information on printf() refer to the following:

Kernighan, Brian W. and Dennis M. Richie, The C Programming Language: Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

Table 7-1 Basic Printf Conversions (p 154)
Table B-1 Printf Conversions (p 244) [Appendix B]

The default format for double precision floating point numbers is "%lG" (that's percent ell gee). There are several conversion options available for doubles with "%lg" or "%lG" being the most preferable.

DOUBLE = "%6.18lG"	Field width of six, eighteen decimal digits.

The default format for long integers is "%ld" (that's percent ell dee). "ld" is the only conversion option available to you for formatting long integers. The only choices that you have is with flags and changing the fieldWidth.

LONG = "%8ld"	Field width of eight.
LONG = "%-10ld"	Left justified with a field width of ten.

All formatting specifications use the same arrangement as in the following table. Square brackets ([]) indicate options Everything else is required:

startSymbol	flags	fieldSize	precision	argSize	conversion
%	[{-,+,0}]	[<int numb>]	[.<int numb>]	l	{d,e,E,f,g,G}

startSymbol (required)
The start symbol is always a percent symbol % and is required.

flags
This uses any combination of the following:

-	— Left justifies the text in the field.
+	— Always prints a + or - symbol in front of the field indicating the sign of the number.
0	— Pads the field with leading zeros.

fieldSize
This is an option that sets the field size. It uses the integer value for the size of the field where data will be placed. Spaces are used to pad the field when the data is smaller than the field size. The field will be expanded if the data is larger than the field size.

precision

This is an option that sets the number of decimal digits. With doubles, a decimal point followed by an integer value is the number of digits past the decimal point that will be printed. The last digit will be rounded.

argSize (required)

With doubles and long ints, argSize will always be "l" (that is, ell) representing a long or double data type. It is required and can not be omitted.

conversion

only one of the following (required):

- Long Integers: Long will always have a "d" conversion.
- Double Floats: Doubles have several choices.

e, E	— Give you a signed decimal floating point number of the form [-]m.ddd(eE)+/-xx.
f	— Gives you a signed decimal fixed-point number of the form [-]mmm.ddd.
g, G	— Will use the smaller, in character length, of "e" or "E", and "f".

☞ *The format is always a concatenation of startSymbol, flags, fieldSize, precision, argSize and conversion, forming a single word. There can be no spaces or tabs between any character in the formatting specification.*

DATA TYPES OUTLINE

I. Data Types

A. String – "this is a string". All strings appear between double quotes. Currently strings can be 2K -1, or 2047, characters long. Strings do not contain any formatting attributes. They are pure text. When pasted into a document, a string assumes the surrounding text attributes.

B. Long Integers – long integers are numbers that do not contain a fractional amount following a decimal point. Long integers are in the range -2147483648 to 2147483647.

C. Double – double precision floating point numbers can contain a fractional amount following a decimal point. Double precision numbers in Nisus are accurate to 18 digits. The largest floating point number you can represent is 1.18973E+4932. The smallest fraction that can be represented is 1.08420E-19.

D. Boolean – the boolean type represents the result of a conditional expression, e.g., 5 greater than 6 (5 > 6). It is either a yes or a no (true or false, respectively) and in this case is false.

E. Date and Time – the date and time types have characteristics of both the double type and the string type. When these types are used in comparisons they act as if they are a double float. If you were to assign them to a string they would assign as a string like "24 Nov., 1990".

II. Constant Data – user defined constants.

A. String – string data can be any text that appears between double quotes; e.g., "this is a string, 12345". Use **NumToChar**(34) to enter a quote character into a string.

B. Long Integers – long integer constants are created when a number is entered without a decimal point; e.g., 500136. A long integer is in the range -2147483648 to 2147483647. If a number is outside of this range, or **DoubleLock** is equal to **true**, then the number is automatically cast to a Double data type.

C. Double – when a number has a decimal point in it, or outside the range of a long integer, it is considered a floating point number (e.g., 3.1415926 is a floating point number because it contains a decimal point. All floating point numbers in Nisus are double precision).

D. Boolean – there are two boolean constants which are the reserved words **true** and **false**.

MACROS COMMAND LISTING GLOSSARY

!	Logical Not operator. Used in logical expressions (equivalent to **Not**).
!=	Not equal operator. Used in logical expressions (equivalent to <>).
"	String delimiter. Used to create String (text) data.
%	Modulus operator. Used in algebraic expressions. Returns the remainder of F_1 divided by F_2; e.g., 22 % 7 is equal to 1 (equivalent to 22 **mod** 7).
&&	Logical And operator. Used in logical expressions (equivalent to **And**).
(	Left Parenthesis. Used to nest expressions or in function calls
)	Right Parenthesis. Used to nest expressions or in function calls.
*	Multiply operator. Used in algebraic expressions.
+	Plus operator. Used in algebraic expressions, or S_1 + S_2 concatenates the string S_2 to the end of string S_1.
,	Parameter separator. Used in function calls.
-	Minus or Negate operator. Used in algebraic expressions, or S_1 - S_2 removes the first occurrence of string S_2 from string S_1.
->	Method operator. Call to a variables method.
/	Divide operator. Used in algebraic expressions, or S_1 / S_2 removes every occurrence of string S_2 from string S_1.
//	Comment Marks. Comment follow the two forward slashes to the end of line.
:	End Of Label Mark. Immediately follows a label identifier.
;	Statement separator. Used to delimit two statements on the same line.
<	Less Than operator. Used in logical expressions.
<=	Less Than Or Equal To operator. Used in logical expressions.
<>	Not Equal operator. Used in logical expressions (equivalent to !=).
=	Assignment operator. Set a variable equal to an expression.
==	Equal To operator. Used in logical expressions.
>	Greater Than operator. Used in logical expressions.
>=	Greater Than Or Equal To operator. Used in logical expressions
[	Left Angle Bracket. Used to access a storage variable as an array.
[[	Double Left Angle Bracket. Used in relative position array accesses to a storage variable.
]	Right Angle Bracket. Used to access a storage variable as an array.
]]	Double Right Angle Bracket. Used in relative position array accesses to a storage variable.
^	Exponentiation (Power) operator. Used in algebraic expressions; e.g., 2 ^ 3 is equal to 8 (equivalent to **pow** (2, 3)).
||	Logical Or operator. Used in logical expressions (equivalent to **Or**).
√	Square Root math function. Returns the square root of an expression; e.g., √4 is equal to 2 (equivalent to **sqrt** (4)).
acos *(F)*	Arc Cosine trigonometric function. Returns 0 to π radians; *F* is in the range -1 to +1.
and	Logical And operator. Used in logical expressions (equivalent to &&).
asin *(F)*	Arc Sine trigonometric function. Returns -π/2 to π/2 radians; *F* is in the range -1 to +1.
atan *(F)*	Arc Tangent trigonometric function. Returns -π/2 to π/2 radians.
atan2 *(F_1, F_2)*	- Arc Tangent trigonometric function. Returns -π to π radians, the arc tangent of F_1 divided by F_2. **Atan2** uses the sign of both arguments to determine the correct quadrant (sign) of the return value.

ceil *(F)*	Ceiling math function. Returns the smallest integer not less than *F* as a floating point number; e.g., ceil (7.3) is equal to 8.0 and ceil (6.0) is equal to 6.0.
CharNum	Character Number document function. Returns the number of characters from the beginning of the document to the insertion point.
CharToNum *(S)*	Character To ASCII Number string function. Returns the ASCII number that is equivalent to the first character in the string *S*.
clipboard	Current Clipboard string variable. Clipboard is used to assign text to or from the current clipboard.
cos *(F)*	Cosine trigonometric function. Returns the cosine of *F* in radians.
cosh *(F)*	Hyperbolic Cosine trigonometric function. Returns the hyperbolic cosine of *F*.
create *(I)*	Stack Create Elements storage method. Used to append *I* empty elements to the beginning of a storage variable.
date	Current Date date/time function. Returns a date data type representing the current date.
DateCreated	File or Folder Creation Date document function. In a document returns the creation date of the document. In the catalog window, **DateCreated** returns the creation date of the currently displayed folder.
day	Current Day Enumerated date/time function. Returns 1 through 7 representing Sunday through Saturday, respectively.
DayOfWeek	Current Day date/time function. Returns the current day as a string; e.g., "Sunday".
div	Integer Division algebraic function. Returns the integral part of F_1 divided by F_2; e.g., 22.5 **div** 3 is equal to 7.0.
DocName	Document's Name document function. Returns the front document's Name.
DocPath	Document's File Path and Name document function. Returns the complete path of a document including the document name.

DOUBLE	Double Precision Floating Point Formatting Specification format variable. **DOUBLE** is the formatting specification that the ANSII C language printf() function uses to format double precision floating point numbers when they are converted to text. The default value for **DOUBLE** is "%lG" (that is, percent ell gee). **DOUBLE** will override the default when you assign a new formatting specification to it. **DOUBLE** must always be in uppercase because **DOUBLE** is a variable that is case sensitive, like all variables.
DoubleLock	Double Precision Conversion flag variable. **DoubleLock** is a boolean flag that, when set to **true,** causes every constant number to be converted to a double precision floating point number; e.g., the integer constant 4, containing no decimal point, will be converted to 4.0. If **DoubleLock** has not been defined by you, it will default to a value of **false**. Since **DoubleLock** is a variable, it is case sensitive, meaning that you must enter it in the same upper and lowercase configuration as seen here.
EndCharNum	End Character Number document function. Returns the character offset from the beginning of the document for the last character in the document; i.e., the number of characters in a document.
ends	Selection End Points **push** or **qpush** keyword operator. Stores the selection end points, for all of the selections in a document, into a storage variable.
error	Error miscellaneous function. Returns a boolean data type representing the error state of the last command executed. A value of **true** indicates an error, usually a sign of insufficient memory.
error	Error storage method. Returns a boolean data type representing the error state of the last operation on a storage variable. A value of **true** indicates an error.

eval	Evaluate As Macro Line special purpose function. Evaluates and then executes the following text, between single quotes, as a line from the extended macros. Evaluation causes all metacharacters to be expanded; e.g., **eval** 'clipboard = \1\r' (\1 is a metacharacter indicating the a found GREP expression between \(and \) and \r is the metacharacter for return). Similar to the Execute Selection menu command, but does not cause macro nesting and a new scope. The **eval** command must appear on a line by itself. You cannot nest **eval** commands. For more information refer to the section in **"Macros Technical Information"** titled "The 'eval' Statement," p. 27
exit	Exit command. Causes the current macro to exit and return control to the caller of the macro.
exp ***(F)***	Base Of Natural Log Exponentiation math function. Returns e^F, the base of natural logarithms to the *F* power.
fabs ***(F)***	Absolute Value math function. Returns the absolute value of *F;* i.e., $\vert F \vert$.
false	Logical False predefined constant. Used in logical expressions. Returns a boolean data type set equal to the value of false; i.e., the value of **not true**.
first	First Element storage method. Returns the contents of the first element of a storage variable (equivalent to myStorage[1]).
floats	Floating Number **push** or **qpush** keyword operator. Stores every selection as a floating point number into a storage variable.
floor ***(F)***	Floor math function. Returns the largest integer not greater than *F* as a floating point number; e.g., **floor** (2.675) is equal to 2.0 and the **floor** (3.0) is equal to 3.0.
fmod ***(F_1, F_2)***	Floating Modulus math function. Returns the remainder of F_1 divided by F_2 with the same sign as F_1.
free ***Id***	Delete Local Or Global Variable miscellaneous function. Deletes the first occurrence of *Id* found by first searching in the local variable space and then in the global variable space.
FreeAll	Delete All Local And Global Variables miscellaneous function. Deletes every local and global variable.
FreeGlobal	Delete All Global Variables miscellaneous function. Deletes every global variable.
FreeLocal	Delete All Local Variables miscellaneous function. Deletes every local variable.
global ***Id***	Global variable declaration. Causes the next use of *Id* to be stored in global variable space.
GoTo ***Id***	Unconditional Branch. Causes the macro to start executing statements following the label *Id*. You cannot branch control.
hour	Hour Of Day date/time function. Returns 0 through 23 representing the 24 hours of the current day.
if ***(E)***	Conditional Test. If the expression *E* evaluates to **true** then the statement(s) on the same line immediately following **if** *(E)* will be executed.
ints	Integer Number **push** or **qpush** keyword operator. Stores every selection as an integer number into a storage variable.
last	Last Element storage method. Returns the contents of the last element of a storage variable (equivalent to myStorage [myStorage->size]).
LastDateSaved	Modification Date document date/time function. Returns a date data type that is equal to the modification time stamp on disk for the front document. When the front window is the Catalog window, **LastDateSaved** returns the modification date for the currently displayed folder.
left ***(S,I)***	Left Sub String string function. Returns *I* left characters from the string *S*; e.g., **left** ("hi ya bub", 5) is equal to "hi ya".
length ***(S)***	String Length string function. Returns the number of characters in the string *S*.
LineCharNum	Line Character Number document function. Returns the number of characters from the beginning of the line to the insertion point.
LineNum	Line Number document function. Returns the line number from the beginning of the document to the insertion point.

***label*:**	Label declaration. The name ***label***, you choose the name, is a location in a macro that can be used to branch control to using the **GoTo** command. The correct syntax of a label requires that a colon immediately follows the label name. A label must appear on a macro line by itself.
local *Id*	Local variable declaration. Causes the next use of *Id* to be stored in the current scope's local variable space. You must do this if you are to hide all previous declarations of a particular *Id* from the scope of the current macro.
log *(F)*	Natural Logarithm math function. Returns ln*(F)*, the natural logarithm of *F; F* is greater than zero.
log10 *(F)*	Logarithm Base 10 math function. Returns log*(F)*, the log base 10 of *F; F* is greater than zero.
LONG	Long Integer Formatting Specification format variable. **LONG** is the formatting specification that the ANSII C language printf() function uses to format long integer numbers when they are converted to text. The default value for **LONG** is "%ld" (that's percent ell dee). **LONG** will override the default when you assign a new formatting specification to it. **LONG** must always be in uppercase because **LONG** is a variable that is case sensitive, like all variables.
LowerRoman *(I)*	Lower Case Roman Number miscellaneous function. Returns an all lowercase string representing the roman number for the integer *I*.
MacroCopy	Copy Selection document function. Copy the selection into the current clipboard. This only works in document windows.
MacroPaste	Paste Current Clipboard document function. Paste the current clipboard into the document. This only works in document windows.
MakeDate (I_1,I_2,I_3)	Create Date Data Type date/time function. Returns a date data type representing the I_1 day, of the I_2 month, of the I_3 year.
mid (S, I_1, I_2)	Middle Sub String string function. Returns I_2 characters of string *S* beginning at character position I_1 ; e.g., mid("hi ya bub", 7, 3) is equal to "bub".
minute	Minute Of Hour date/time function. Returns 0 through 59 representing the 60 minutes of the current hour.

mod	Modulus operator. Used in algebraic expressions. Returns the remainder of F_1 divided by F_2; e.g., 22 mod 7 is equal to 1 (equivalent to 22 % 7).
modf *(F)*	Modf math function. Returns the signed fractional part of *F;* e.g., **modf** (3.1415) is equal to 0.1415.
month	- Current Month date/time function. Returns the current month as a string; e.g., "January".
NextRulerStart	Next Governing Ruler Position document function. Returns the character offset from the beginning of the document for the next governing ruler following the insertion point. NextRulerStart will return 2,147,483,647 (2^31-1) when the next ruler does not exist.
not	Logical Not operator. Used in logical expressions (equivalent to !).
NumFound	Number Of Find Or Find/Replacements miscellaneous function. Returns the number of items found or replaced in the previous Find/Replace.
NumToChar *(I)*	ASCII Number To Character string function. Returns a string containing the character equivalent of the ASCII number *I*.
offset (S_1, S_2)	Position Start Of String string function. Returns the character offset of the first occurrence of S_1 in S_2.; e.g., **offset** ("bub", "hi ya bub") is equal to 7.
or	Logical Or operator. Used in logical expressions (equivalent to ||).
PageLineNum	Page Line Number document function. Returns the line number from the top of the page to the insertion point.
PageNum	Page Number document function. Returns the page number from the beginning of the document to the insertion point.
pop	Pop storage method. Removes the *first* element of a storage variable and returns its contents.
pow (F_1, F_2)	Exponentiation (Power) operator. Used in algebraic expressions; e.g., **pow** (2, 3) is equal to 8 (equivalent to 2^3)).
push *(E)*	Stack Push storage method. Creates and appends a new *first* element for a storage variable and stores the result of expression *E* in the new elements contents.

qcreate *(I)*	Queue Create Elements storage method. Used to append *I* empty elements to the end of a storage variable.
qpush *(E)*	Queue Push storage method. Creates and appends a new *last* element for a storage variable and stores the result of expression *E* in its contents.
random	Random miscellaneous function. Returns a random integer in the range -32767 to 32767.
ReportErrors	Report Errors flag variables. **ReportErrors** is a boolean flag that, when set to **false**, causes the syntax warning dialog not to appear. This allows you to check for errors yourself and report them in some other way. If **ReportErrors** has not been defined by you, it will default to a value of **true**, causing syntax errors to be reported. Since **ReportErrors** is a variable, it is case sensitive, meaning that you must enter it in the same upper and lowercase configuration as seen here.
right *(S, I)*	Right Sub String string function. Returns *I* right characters from the string *S*; e.g., **right** ("hi ya bub", 3) is equal to "bub".
rrandom *(F_1, F_2)*	Random Number With Range miscellaneous function. Returns a random floating point number, F_R, in the range $F_1 \leq F_R \leq F_2$.
RulerName	Ruler Name document function. Returns the name for the governing ruler of the insertion point.
RulerStart	Ruler Start document function. Returns the offset in characters from the beginning of the document for the governing ruler preceding the insertion point.
second	Second date/time function. Returns 0 to 59 representing the 60 seconds of the current minute.
seed *(I)*	Seed miscellaneous function. Sets the starting *seed* value of the random function to *I*.
SelectEnd	Selection End Point document function. Returns the end of selection offset, of the last selection, in characters from the beginning of the document.
SelectRuler *(I)*	Select Governing Ruler Icon document function. Selects the governing ruler of the character offset *I*.
SelectStart	Selection Beginning Point document function. Returns the beginning selection offset, of the last selection, in characters from the beginning of the document.
SelectWithRuler	Selection Includes Beginning Ruler flag variable. **SelectWithRuler** is a boolean flag that, when set to **true**, and used in conjunction with either the **SetSelect** or **SetSelectMore** commands, causes the ruler immediately to the left of the selection to be selected along with the text. If **SelectWithRuler** has not been defined by you, it will default to a value of **false**. Since **SelectWithRuler** is a variable, it is case sensitive, meaning that you must enter it in the same upper and lowercase configuration as seen here.
SetSelect *(I_1, I_2)*	-Select Text document function. Selects text from character offset I_1 to character offset I_2.
SetSelectMore *(I_1,I_2)*	Noncontiguous Select Text document function. Adds, to the previous **SetSelectMore**, another selection of text from character offset I_1 to character offset I_2. The very first selection created with **SetSelectMore** is ignored. To work around this always begin your **SetSelectMore**'s with a non-consequential statment, **SetSelectMore**(0,0).
shuffle	Shuffle storage method. Randomly swaps the contents of the elements of a storage variable.
sin *(F)*	Sine trigonometric function. Returns the sine of angle *F* which must be in radians.
sinh *(F)*	Hyperbolic Sine trigonometric function. Returns the hyperbolic sine of *F*.
size	Size storage method. Return the number of elements in a storage variable.
sqrt *(F)*	Square Root Function. Returns the square root of an expression; e.g., **sqrt** (4) is equal to 2 (equivalent to $\sqrt{4}$).
StartEnds	Selection Start and End Points **push** or **qpush** keyword operator. Stores the selection start and selection end points, for all of the selections in a document, into a storage variable.
starts	Selection Start Points **push** or **qpush** keyword operator. Stores the selection start points, for all of the selections in a document, into a storage variable.
stop	Stop command. Causes all running macros to **exit** returning control to the user.

strings	String **push** or **qpush** keyword operator. Stores the text, for all of the selections in a document, into a storage variable.
swap *(I_1,I_2)*	Swap Contents storage method. Swap the contents of element I_1 with the contents of element I_2 of a storage variable.
tan *(F)*	Tangent trigonometric function. Returns the tangent of *F* in radians.
tanh *(F)*	Hyperbolic Tangent trigonometric function. Returns the hyperbolic tangent of *F*.
time	Current Time date/time function. Returns a time data type representing the current time.
TimeModified	Document Last Modified Time date/time function. Returns a time data type representing the last time a document was modified while editing.
true	Logical True predefined constant. Used in logical expressions. Returns a boolean data type set equal to the value of true; i.e., the value of **not false**.
type	Type Name Of Variable miscellaneous method. Returns a string representing the data type of a variable; e.g., myVar=3.5, myVar->**type** is equal to "DOUBLE".
UpperRoman *(I)*	Upper Case Roman Number miscellaneous function. Returns an all uppercase string representing the roman number for the integer *I*.
WhatDay *(D)*	Day Corresponding To Date date/time function. Given *D;* a date or time data type; or the number of seconds since midnight, January 1, 1904; **WhatDay** (*D*) will return 1 through 7 representing Sunday through Saturday, respectively, the day of week that *D* falls on.
year	Current Year date/time function. Returns 1904 through 2040 representing the current year.