Contents

Introduction

This manual is an introduction to and reference for the language of Glyphic Codeworks: Glyphic Script. If you have not done the Glyphic Codeworks Tutorial, and have no familiarity with Glyphic Codeworks, then you should do the Tutorial before reading this document. This is because this manual will direct you to try out examples with the system.

Glyphic Script is a modern, object oriented, language. Glyphic Script has objects, message passing, inheritance and a unified object model. While all these concepts may sound complicated, they actually combine to produce a simple, easy to use and understand system. It is easier than languages like C or Basic because it is very consistent and has only a few concepts to learn.

This manual is in four sections. The first section teaches each aspect of the language in turn. In this section you will be writing and executing scripts and creating objects, so you should have a running version of the system with you as you read it. The second section explains, with examples, each of the standard objects the language provides such as numbers, strings and groups. You could read this section straight through, or come back to it as your interests and needs arise. The third section contains a number of separate advanced topics. You can read these any time after mastering the basics of the language. *[This section of the manual is not yet written].* The fourth section is a terse refernce to the language.

Typographic Conventions

Code examples in manual are set in a sans-serif font like this:

```
a := text count "e".          how many e's are in the text?
```

In Glyphic Script, formatting is important. When you see examples, enter them exactly as you see them, with bold and italics. However, anything in italics is treated as comments and ignored by the system, so you don't have to enter those. See the Procedures Manual or the Quick Reference Card for special features Glyphic Codeworks has for entering formatted text.

In some code examples, the result of executing the code is shown on the right, after an arrow. Only the text before the arrow is part of the expression that should by entered and executed:

3 + 4 $\Rightarrow$ 7

!! Paragraphs that contain important pointers or information to carefully follow are preceded by double exclamation points, like this paragraph.

» Paragraphs that contain supplementary or more in depth information are preceded by double angle brackets, like this paragraph. This information need only be read if interesting to you.

Section 1

Basic Scripting

Simple Expressions

Scripts are made up of smaller units called expressions. Expressions are small bits of code that, when run, have a value. For example, the following expressions yield these results:

Expression	Result	Remarks
3.14159	⇒ 3.14159	*a number is a simple expression*
"Hello"	⇒ "Hello"	*so is a string of characters in quotes*
3 + 4	⇒ 7	*arithmetic expression*
3 + 4 * 5	⇒ 23	*follows normal math rules*
"abc" & "def"	⇒ "abcdef"	*& is a string operator*
"abc" @ 2	⇒ "b"	*the second character of the string*
3 < 4	⇒ true	*comparison of two numbers*

Trying the Examples

You can try these expressions out for yourself using the system. We suggest that you create a Glyphic Codeworks document in your notebook for use just with this manual. After you create the document, turn to it and open a new workspace. When you are ready to take a break from the manual, simply turn back to the table of contents of the notebook. Don't close the workspace or other windows. When you turn back to the document, they will still be open.

To open a new workspace:

1. **Create a new Codeworks document in your notebook and turn to it.** Use the 'Codeworks, Folders' stationary.
2. **Tap on New Workspace… from the Utilities menu.** A new workspace appears.

To execute and expression in a workspace:

1. **Enter the expression in the text editor portion of the workspace.** Pay close attention to the formatting and punctuation of the expression: computers are very fussy.
2. **Select the expression.**
3. **Tap Save at the top of the workspace.** The result of the operation appears to the right of the 'result' entry in list at the top half of the workspace. If it is not visible, then scroll the list so it is.

Multiple Expressions

You can execute more than one expression at a time. To do this, the expressions must end periods. Each expression is executed in turn, and the result of only the last expression is the result of executing the whole group. Consider:

```
3 + 4.
1 / 5.
"Hello".
```

When all three lines are selected and executed, first the system computes the number 7, then the number 0.2, and finally the string "Hello" which is the result.

» Multiple Expressions don't have to be on separate lines, only separated by periods. The final period is optional.

Variables

A variable is a holding place for the value of an expression. There are several types of variables. They differ in how they are created and how long they last. This chapter will explore temporary and workspace variables.

Workspace Variables

Every workspace starts off with three variables. They are named a, b & c. You can see these variables in the top part of the workspace. The value that each variable is currently holding is also displayed on the right. Initially, workspace variables hold the value ??? (pronounced 'unknown').

The value of a variable can be changed by the assignment symbol ':='. If you execute the following two expressions, you'll see the value of two of the workspace variables change.

```
a := 3 + 4.
b := 1 / 5.
```

Notice that, even if you execute other expressions, unless you assign a different value to a workspace variable, it will hold onto its value.

You can use a variable in an expression. The value of the variable at the time the expression is executed is used. Executing these expressions in turn changes the values of a and b:

Expression	**Value in a**	**Value in b**
a := 3.	3	???
b := 2 * a.	3	6
a := "Hello".	"Hello"	6
a := a && "kitty".	"Hello kitty"	6
b := b * b / 2.	"Hello kitty"	18

» The results of executing expressions in a workspace appear in the list of the workspace next to the word result. Result is just another workspace variable and can be used in expressions.

Local Variables

A local variable lasts for just a short period of time. It can be used as a temporary holding place for values during the execution of a script that you won't need after the script is done. After the script is executed, local variables disappear.

You have to tell the system the names of the local variables you will use. You do this by declaring them with the '$' syntax:

```
$ x.
x := "cha".
a := x & x & x.
```

After this script is run, a will have the value "chachacha".

!! Be sure you selected all three lines of the above expression to execute. If you select and execute just the first line, the system will create the local variable x, and then destroy it. If you then select the second line and execute it, you'll get an error because the variable x no longer exists. Local variables last only as long as a single execution.

Variable Names

Local variables can have almost any name:

```
bob
q-factor
pre-1886-shipments
```

The rules are simple: it can be any combination of letters, digits, and dashes so long as it starts with a letter. Spaces and punctuation cannot be part of a variable name. Upper/Lower case distinctions do not matter; however the system will convert everything to lower case.

More than one local can be declared at a time:

```
$ the-price, the-tax.
the-price := 13.29.
the-tax := the-price * 0.07.
the-price + the-tax.
```

The result of this is 14.22. While it is running, the local variables are set to 13.29 and 0.93. Then the sum of them is returned as the result.

Unlike workspace variables, local variables don't start off with any value. When they are declared, their value is undefined. If you execute an expression with a local variable and haven't assigned the variable a value, the system will give the error "using an undefined value".

» You must declare all the local variables you are going to use before any expressions that use them. However, you can declare them with one or more declaration statements:

```
$ game-1, game-2.
```

is the same as

```
$ game-1.
$ game-2.
```

!! A common mistake is to forget the space between the $ and the variable name. If you do this you'll see get one of two error messages: either "nothing more expected" or "unused expression". The error messages are a little confusing because a $ with letters following it without a space actually means something else to the system: it is a symbol.

Message Expressions

More complicated expressions involve messages. Messages are the equivalent to functions, procedures, and subroutines of other languages.

Messages are expressions with bold-face words. For example:

```
$ text, s, n.
text := "Chattanooga".
s := text size .
n := choose 1 to s.
text from n for 3.
```

This example has three messages. The first is **size**, which when used with a string results in the number of characters in the string. The second is **choose** which picks a random number from a range. This message has an argument, **to**, which gives the end of the range. The last message is **from**, which extracts part of a string.

Notice that sometimes the message comes first (in the case of **choose**) and sometimes it comes second (in the case of **size** and **from**). While most messages read better one way or the other, don't worry if you forget which way: all messages work either way.

Parts of a Message

The first bold word is always the name of the message. The first value in the message is always the *receiver*. The receiver is the value that message works on. The other values, if any, are *arguments* to the message. These can modify how the message works.

In the example above:

Expression	Message	Receiver	Arguments
text **size** .	**size**	text	*none*
choose 1 **to** s.	**choose**	1	**to** s
text **from** n **for** 3.	**from**	text	n *and* **for** 3

Some arguments are preceded by additional bold words (other than the first one), these arguments are known as keyword arguments. Other arguments are have no bold words, they are known as positional arguments.

Since keyword arguments have an identifier, they can be written in any order. The following two messages are identical:

```
a := new group size 3 capacity 20.
b := new group capacity 20 size 3.
```

In each case, the results are the same: the message **new** is sent to the receiver group, with a **size** argument of 3 and a **capacity** argument of 20.

Positional arguments cannot, in general, be mixed up. You already know an example like this: dividing two numbers. The value of 2 / 5 is very different than 5 / 2. Similarly, the following two expressions have different results because their two positional arguments are in different orders:

```
a := new point 3, 4.
b := new point 4, 3.
```

Both of these expressions cause the message **new** to be sent to the receiver point. However, the first and second positional arguments have different values in each expression. (Note: the comma is optional between positional arguments, but the expression often reads better if you include it.)

For many messages, some or all of the arguments can be left off. In this case they are defaulted. This means that the message uses an appropriate, or default, value for those arguments.

Nesting Expressions

Without punctuation all bold words and their arguments belong to the same message. You can use parentheses to use the result of one message in another. For example:

```
$ alphabet, i.
alphabet := "abcdefghijklmnopqrstuvwxyz".
i := choose 1 to ((text size ) - 1).
(alphabet @ (i + 1)) && "comes after" && (alphabet @ i).
```

The second expression used the result of the **size** message, in an arithmetic operation (- 1), and then used the result of that as the **to** argument to the **choose** message.

Many of the parentheses in the above example are not necessary. Any expression involving only operators can be the argument to a message without parentheses. Expressions with only operators use the normal rules of arithmetic. The above example could be written:

```
$ alphabet, i.
alphabet := "abcdefghijklmnopqrstuvwxyz".
i := choose 1 to (text size ) - 1.
alphabet @ (i + 1) && "comes after" && alphabet @ i.
```

The parentheses around the **size** message are needed because it is being used inside another message. No parentheses are needed around the subtraction of one because the minus (-) operator will execute before the **choose** message. In the last expression, the only parentheses needed are those around the i + 1: they cause the result of the addition to be used as the argument to the @ operation.

If you can't remember which parts of an expression get executed when, follow this simple rule: When in doubt, use parentheses. Extra parentheses always ensure that the system does what you expect.

» The above example uses several operators that aren't part of arithmetic. Like always multiplying before adding, these operators also have rules about which are done first and which are done last. These rules are called rules of precedence. In the list below, operators near the top are done before operators closer to the bottom. For operators on the same level, they are always done left to right. (Don't worry if you don't know what some of these operators do yet, you'll learn about them later.)

Highest Precedence:	x @ y
	- x
	x * y x / y x % y
	x + y x - y
	x & y x && y
	x < y x <= y x >= y x > y
Lowest Precedence:	x == y x != y

Shortcuts

There are two short cuts when writing messages. You don't need to use these shortcuts now, but you may encounter scripts in the system

that have been written using them. So it is worth briefly looking over them.

A message that has no arguments can be written using *dot-notation*. This form, as you'll see later, is generally used for accessing variables in other objects. Here it is used as a shortcut for writing small messages. For example, the last two expressions are the same message:

```
$ text.
text := "Zebra".
a := text first .
b := text.first.
```

!! Notice the embedded period between the receiver (in this case the value in the variable text) and the message (first). Two things are important: there is no space around the embedded period, and the message name is not bold.

The main advantage of this form is that it is at the top of the precedence list; they are executed before any operators or other messages. Therefore, they can be used as the argument of a normal message with out parentheses. For example, the last two expressions both choose a random vowel:

```
$ vowels.
vowels := "aeiou".
a := vowels @ (choose 1 to (vowels size )).
b := vowels @ (choose 1 to vowels.size).
```

The other shortcut is the use of the semicolon in place of the period between two expressions. This allows you to chain the messages together: the first expression is executed as normal, and the result used as the receiver for the second message. Consider:

```
choose 1 to 10; sqrt .
```

This sends the message **choose** to pick a random number. Then this random number is used as the receiver for the message **sqrt**.

A semicolon acts like a set of parentheses around everything to the left. The above expression is equivalent to:

```
( choose 1 to 10) sqrt .
```

Control Structures

Up to now we've only executed each expression in a sequence of expressions exactly once. Control structures are compound expressions that allow one or more expressions to be execute one time, many times, or not at all.

This chapter discusses three common control structures: **if**, **return**, and **for**. In a later chapter, two more will be discussed.

Control Structures - If

The simplest control structure is the if statement. An if statement tests a condition and then executes one sequence of expressions if the condition was true, or a different sequence if the it was false. For example:

```
$ name, letter, type, possibilities.
name := "John Jacob Jinglehiemer-Schmidt".
letter := name choose .
if ("aeiou" has letter)
    then [
        type := "vowels".
        possibilities := 5.
    ]
    else [
        type := "consonants".
        possibilities := 21.
    ].
letter && "is one of" && possibilities.name && "possible" && type.
```

Formatting

The two conditional sequences are between square brackets. There are several fine points about its formatting: Between the square brackets, each expression ends in a period, although the last one is optional. Outside the brackets, there is only a period after the last close bracket, as this is the end of the **if** statement. Even though this statement was

written out over nine lines, it could be written in fewer: line breaks and spaces are interchangeable.

Blocks

The bracketed statements, along with the square brackets, are called a *block*. In other languages, blocks are often wrapped in curly brackets, or the words 'begin' and 'end'. Unlike some languages, in Glyphic Script, the square brackets are always required, even if the block has only one expression inside of it.

There are two other small points of interest in the above example:

» First: The **choose** message was used in a different way. Instead of a number being the receiver, a string was the receiver. In this case, **choose** picks a random character of the string. In takes no arguments. Notice that the same message can do different (although often similar) actions when used with different receivers. Computer Science calls this *polymorphism*. It is one of the powerful features of an object-oriented, message passing language like Glyphic Script.

» Second: In the last expression, the message **name** was used with a number (the expression was 'possibilities.name'). There is a difference between the number 21 and the string "21". The number is a value that can take part in arithmetic. The string is a sequence of characters that can used to communicate with the user. We needed to turn the number into a string so that we can make it part of the final string result. **Name** is the message that results in a string which is the name of the receiver.

Nesting

If statements can be nested, like all control structures:

```
$ n, msg.
n := choose 1 to 100.
if n > 33.3 then [
    if n > 66.6
        then [ msg := "huge" ]
        else [ msg := "medium" ].
]
else [ msg := "small" ].
msg.
```

This example also demonstrates several other things: If the condition is just a simple expression (not a message) then it doesn't need to be in parentheses. If there is only one expression in the block, since it is the last expression, it doesn't need a period.

Control Structures - Return

In all the previous examples, the result of an sequence of expressions has been the value of the last expression. We have been taking advantage of a shortcut offered in workspaces: if there is no **return** statement, they return the result of the the last expression. Elsewhere in the system this isn't so: you need to have a **return** statement at the end if you want to return the last value. Furthermore, sometimes you want to return a value from the middle of a sequence of expressions.

A **return** statement causes the result of an expression to become the result of a sequence of expressions, and also stops all further execution of the whole sequence.

```
$ income, payment, monthlies.
income := 28500 / 12.
payment := 872.50.
monthlies := 200 + payment.
if monthlies / income > 0.50
    then [ return "Monthly payments are too high." ].
if payment / income > 0.30
    then [ return "Payment is too high." ].
return "This mortgage is OK".
```

If either of the blocks gets executed (because one of the conditions is true) then a string will be returned and the rest of the sequence not executed.

The word **return** can appear before any expression. Since it is not a message, **return** can appear before a message without the message having to be in parentheses:

```
if (choose 1 to 100) > 50
    then [ return "aeiou" choose ]
    else [ return "bcdfghjklmnpqrstvwxyz" choose ].
```

Return Without a Value

If you just want to stop execution without any result, you can use **return** just by itself. The expression will not have a result. In a workspace, the result will show as "No result".

```
a := choose 1 to 6.
b := choose 1 to 6.
if (a == 1 and b == 1)
    then [ return ].          snake eyes, we're done.
a := choose 1 to 6.
b := choose 1 to 6.
return .
```

Control Structures - For

A **for** statement causes a block of expressions to be execute several times. Each time through the block of expressions, a local variable is set to a different number. For example:

```
$ s.
s := 0.
for  1 to 10 do [ $ index  i.
     s := s + i.
].
return   s.                    ⇒ 55
```

This sums the numbers from 1 to 10. Each time through the loop (each *iteration*), the local variable i is set to a number from 1 to 10. Then the expression adding it to s is executed. The variable i is called the index variable.

» The declaration of the local variable i requires the bold word **index**. Actually, i is an argument (a type of local variable) to the block. Arguments are discussed later. For now, just use this kind of declaration for the local variable in a **for** loop.

For Parameters

The **for** statement can take an optional argument, **by**, that determines how much to bump the index variable each time. Before executing the next example, open the transcript window.

To open the transcript window:

- **Tap on Transcript… in the System menu.** The transcript window opens. You can move and resize this window as needed.

Execute the following example:

```
for  1900  to  2000  by  10 do [ $ index  year.
     $ is-mult-of-4,  is-century.
     is-mult-of-4 := year % 4 == 0.
     is-century := year % 100 == 0.
     if  (is-mult-of-4     and  (not  is-century))
          then  [ transcript  put  year.name && "is a leap-year" ]
          else  [ transcript  put  year.name && "is not a leap-year" ].
```

```
].
```

The block of expressions gets executed once for each decade from 1900 to 2000. The transcript will show these years. This example has several important things to notice:

- The index variable can be called anything you like. In this case it was called year.
- Inside a block, there can be additional local variables.
- The % operator computes the remainder of the division between its two arguments. 12 % 5 is 2 because the remainder of 12 divided by 5 is 2.
- The **put** message, used with transcript as the receiver, appends a string to the transcript window. This is useful for debugging, and seeing how your scripts are running.

Local Variables in Blocks

In the example above, two local variables were declared inside the block of the for loop. These variables are available only within the block. They are created fresh each time the block is executed, each time through the loop. After the block is done, the variables are no longer available. This is true of the index variable as well. For example:

```
for 10 to 20 do [ $ index i. ]      an empty block
"the final value of i is" && i.name.
```

This code won't execute. If you try to, the system will give you the error message "Unknown local or property" because the index variable i is not valid outside the block.

What happens if the name of a local variable is the same as the name of a variable outside the block? While this isn't recommended, it works the following way: Inside the block, the name refers to a new local variable; outside the block it refers to the variable it would if the block wasn't there. For example:

```
a := "Hello".                  this sets the workspace variable
for 1 to 10 do [ $ index i.
    $ a.                       declares a local variable
    a := i * i.                sets the local variable
].
```

```
a := a && "kitty".          access and sets the workspace variable
```

» A word about speed: Don't worry about the expense in time of the system creating local variables each time through the loop. Creating local variables is essentially free. In fact variables outside the block are a little more expensive, as well as harder to read. The moral: use local variables in blocks whenever possible.

Objects

If all computation took place in workspaces, then Glyphic Codeworks applications would be very boring and have no user interfaces. However, the bulk of Glyphic Codeworks programs use objects that you create and give the user a nice, graphical interface to interact with.

In this chapter you will create a cash-register object that will be used in later chapters. Please follow the instructions for creating the cash-register and save your work if you take a break.

Objects - Instance Variables

When you create an object, that object can have *instance variables*. These are variables that belong to the object. As long as the object isn't destroyed, its instance variables continue to exist and hold the last value they were set to. In this way, instance variables are much like the workspace variables discussed above. In fact, workspace variables are just instance variables of the workspace object!

Creating an Object and its Instance Variables

Create a new class called cash-register and add three fields to it named amount, sub-total, and daily-total. To do this:

1. **Tap on New Class… from the Utilities menu.** The class creator window appears.
2. **Enter the name of the class to be cash-register, and select the top-most window style icon.**
3. **Tap Create & Close.** The class creator window will close and the new cash-register object will open.
4. **For each field:**
   a. **Caret in an empty place in the grid.** A pop-up list of field types appears. Choose an empty place in the right column.
   b. **Tap Number in the pop-up menu.** A number field is added to the cash-register object.
   c. **Circle on the new field to rename it.**
   d. **Edit the name to one of amount, sub-total, or daily-total, then tap OK.**

5. **If you want, you can add labels beside the fields and edit the labels to name the fields.**

You now have created an object, named cash-register. This object has three instance variables (or properties) named amount, sub-total and daily-total. Cash-register also has a form-like view that displays these three properties in numeric write-in fields. If you toggle the mode of the window to user-mode (the aligner lines disappear), you can edit the values of the variables in the cash-register windows. If you do this, toggle the mode back to author-mode (the aligner lines appear again) before proceeding.

Accessing the Instance Variables

When you write scripts for an object, its instance variables can be accessed and set by name. You will write a script for cash-register that clears all three variables to zero:

1. **Add a button to cash-register.** Circle the button and rename it 'Clear'.
2. **Tap on the button to edit its script.**
3. **Replace the entire script with:**

```
$ clear-daily.
clear-daily := ask user "Clear daily-total too?".
amount := 0.
sub-total := 0.
if clear-daily then [ daily-total := 0 ].
```

4. **Tap Save.** Make sure there were no errors. A common error is to get the message "Unknown local or property". This generally means that you either misspelt the name of a property in the script, or when you named the property. Correct the misspelling in the script or of the property and save again.

When you add a button, you are adding a script property to an object. Tapping the button (in user-mode) causes the script to run. Toggle cash-register into user-mode and try it.

In the rest of the manual, you will be asked to add buttons to objects with various scripts. To save space, the detailed instructions will not be give. Just follow the procedure above.

» The Ask Message

The script above uses a new message: **ask** sent to user. User is an object that represents the real user to your scripts. The **ask** message puts up a note asking a question and offers two choices: Yes and No. If the user taps Yes, the result of the **ask** message is true, otherwise the result is false. Notice how the script stores the result of the **ask** message in a local variable and then use that result later.

It is better practice to offer the user the option of canceling an operation as destructive as clear. Fortunately, the **ask** message has just such an option. Replace the first line in the script above with these two lines:

```
clear-daily := ask user "Clear daily-total too?" with-cancel .
if (clear-daily is ???) then [ return ].
```

The optional argument **with-cancel** directs the **ask** message to include a Cancel button in the note. If the user taps Cancel instead of Yes or No, then the result of the **ask** message is ??? (unknown). The **if** statement tests for this case and executes a **return** if so, stopping all further processing of the script.

Accessing Scripts from Elsewhere

Once you have defined the Clear button above, the object cash-register will now understand the message **clear**. In a workspace you can execute:

```
clear cash-register.
```

This will act exactly as if you had tapped the Clear button. In fact, when you tap the Clear button, this is the expression the system evaluates.

» Remember that the order of the message and the receiver can be reversed. If you find it reads better, you can write the expression above as:

```
cash-register clear .
```

Accessing Instance Variable from Elsewhere

In a script of an object (like the script of the Clear button), expressions can refer to that object's instance variables simply by name. However, if you want to access the instance variables from a script that doesn't belong to the object (from a workspace, for example), then you must use a different syntax.

To refer to the instance variable of another object, you use *dot-notation*. You write the name of the object, a period, and the name of the variable. For example, in the workspace you can execute:

```
cash-register.sub-total := 22.50.
cash-register.total := cash-register.total + cash-register.sub-total.
```

!! There are no spaces around a period used in dot-notation. If you accidentally put a space in, then the system will confuse the period for the period at the end of an expression.

» Notice that dot-notation is exactly the same as the short-hand notation for messages that have no arguments. In fact, instance variables and messages with no arguments can be each be written in the same three different ways. Across each line, the three expression are equivalent and do the same thing:

Dot-Notation	Message After	Message Before
cash-register.total	cash-register **total**	**total** cash-register
cash-register.clear	cash-register **clear**	**clear** cash-register

Having three different ways of writing the same thing may be a bit confusing at first, but it allows you to choose the best way of writing an expression.

Sometimes it is convenient to set a local variable to an object. When you do this, the variable *refers* to the object. This is the same as saying the value of the variable is the object, but it makes it clear that any use of the variable will really use the object. For example, the above script could have been written as:

```
$ cr.
cr := cash-register.
cr.sub-total := 22.50.
```

```
cr.total := cr.total + cr.sub-total.
```

Writing Message Scripts

When you add a button to an object, you are adding a script for a message that takes no arguments. Often objects need to understand messages that take arguments. In this chapter you'll write scripts that take arguments and write messages to run them.

When you tap a button in an object that you created, a message is sent to your object. That message takes no arguments. Typically, the script for the button then sends the object more messages to do the bulk of the work. These messages are not associated with a button: the object understands them, but they are only sent from other scripts. This will become more clear as you read this chapter.

In this chapter, you'll need the cash-register object built in the chapter above. You should have its form view open and in author-mode. You will also need to have a browser view of the cash-register open and the workspace open.

To open the browser view of cash-register:

1. **Double-tap on the title bar of the cash-register window.** A pop-up menu appears.
2. **Tap on Open Browser in the Object Development section of the menu.** The browser appears.

Writing Scripts - Keyword Arguments

An argument lets a script use additional information. Like local variables, a script has to declare what arguments it can take. For example, consider the message:

```
cash-register ring-up-item price 3.59 quantity 2.
```

This message takes two keyword arguments: **price** and **quantity**. A script for the message **ring-up-item** would have to declare these two arguments:

```
$ price p, quantity q.
```

You can think of this as declaring two local variables, p and q, which are set by the keyword arguments **price** and **quantity**. The only difference between arguments and local variables is that the script can't set an argument, its value comes from the message expression.

The whole script for **ring-up-item** is:

```
$ price p, quantity q.
$ cost.
cost := p * q.
sub-total := sub-total + cost.
```

You should add this script to cash-register. Do this in the browser:

1. **Caret on the left side of the window.** An insert property note appears.
2. **Enter the name to be ring-up-item, and tap Script: Yes.** The script window appears. You will not see the property appear in the browser window until after the next step.
3. **Enter the script above into the window and tap Save.**

Now the cash-register can be sent the message **ring-up-item**. In the workspace, execute the expression:

```
cash-register ring-up-item price 3.59 quantity 2.
```

You should see the fields in the form view update to reflect the new purchases.

Keyword Argument Names

The names you can choose for both the keyword and variable names follow the same rules as for all variables: any combinations of letters, digits and dashes, starting with a letter. We could have chosen more explanatory names in the above expressions:

```
$ price  how-much,  quantity  how-many.
$ cost.
cost := how-much * how-many.
sub-total := sub-total + cost.
```

Often the names of the keywords are the most descriptive names for the variables. In this example price and quantity are probably even more clear than how-much and how-many. This leads you to want to write the declaration like:

```
$ price  price,  quantity  quantity.
```

You can write this, but there is a short cut: if you want the name of the variable to be the same as the name of the keyword, then you can just write the keyword. The whole script would now look like:

```
$ price  , quantity  .
$ cost.
cost := price * quantity.
sub-total := sub-total + cost.
```

» You can see that the choice of the variable name is totally up to you. It can be the same, or different than the name of the keyword. Sometimes you want them to be the same because it makes the code clear, and the keyword is the best descriptive name. Sometimes you want them to be different because you use the variable often in the script and you'd like it to be short (like p and q). Choice of variable names is a personal style.

» On the other hand, choice of keywords makes a bigger difference. Every time the message is sent, the keyword will be used identify the value. The keyword should always be descriptive. Consider the following two messages:

```
ring-up-item     cash-register  price  5 quantity  3
```

```
ring-up-item      cash-register  p 5 q 3.
```

The second is hard to read and remember. Unless you have the script open in a window, you're not likely to remember what **p** and **q** are. Choice of keyword is important and you should always give them some thought.

Writing Scripts - Defaulted Arguments

In the previous example, it might be reasonable to assume that most of the time, the **quantity** argument is going to be 1. It would be convenient if messages that had a quantity of 1 didn't even have to write it. For example:

ring-up-item cash-register **price** 17.29.

If you run the expression above, a debugger will appear with the error message "Using an undefined value" and q selected. This is because the arguments to the message didn't give a value to q, so its value is undefined. This is the same error as using a local variable before you've assigned it a value.

Scripts can handle missing values rather than just causing an error: They can default the value of the argument. In essence, we want assign 1 to q if the message didn't give it a value in the first place. There is a form of assignment, ?=, just for this case. It is called defaulting. Change the **ring-up-item** script to:

```
$ price  p, quantity  q.
$ cost.
q ?= 1.
cost := p * q.
sub-total := sub-total + cost.
```

Now, if argument **quantity** is part of the message, then q will be set by the message, otherwise it will be set to 1.

» This is actually the only way that you can assign to the variables for arguments. Argument variables, like p and q, cannot be assigned to using :=. Remember: argument variables are assigned by the message, local variables are assigned by your script.

Writing Scripts - Returning Values

Scripts in an object can return values just like the scripts you've written in a workspace. There is one difference: Unlike a blocks and scripts in a workspace, you must use a **return** statement to return a value. Scripts do not return the value of the last expression unless you write the work **return** before it.

As an example, consider the following script for the message **discount**:

```
$ on amount.
$ percent.
percent := 0.
if (amount >= 10.00) then [ percent := 0.025 ].
if (amount >= 25.00) then [ percent := 0.05 ].
if (amount >= 100.00) then [ percent := 0.075 ].
return amount * percent.
```

(Add the property **discount** to cash-register with this as its script, we'll need it later).

The last line of this script has to have the word **return**. If it did not, then this script, when executed due to the message **discount** sent to cash-register, would not return any value at all.

Writing Scripts - Self

Now that cash-register can perform a number of useful actions, you can tie them all together. Add several buttons to the cash-register form view as follows:

Add a button called buy-milk with the script:

```
ring-up-item      self price   1.59
```

Add a button called buy-eggs with the script:

```
ring-up-item      self price   0.205  quantity   12
```

Add a button called buy-cheese with the script:

```
ring-up-item      self price   5.45  quantity   0.5
    buying half a pound of cheese at 5.45/pound
```

Add a button called grand-total with the script:

```
$ amt-off, tax.
amt-off := self discount on sub-total.
tax := (sub-total - amt-off) * 0.0725.
total := sub-total - amt-off + tax.
daily-total := daily-total + total.
sub-total := 0.
```

Notice than in all these scripts, the messages were sent to something called self, not cash-register. Self is a predeclared variable that has the value of the receiver of the message. Since tapping on these buttons causes the a message to be sent with cash-register as the receiver, self has the value of cash-register.

If you replaced every occurrence of self in these scripts with cash-register, then they would work exactly the same. Self is better because if you renamed cash-register, and the scripts named cash-register they would no longer work. When the scripts use self, they operate on the correct object, no matter what you name it. The same is true if you make instances of cash-register: if the scripts use cash-register, they always operate on the class. If they use self, the work correctly.

You cannot use self when you send messages to the cash-register from the workspace. This is because for scripts in the workspace, the value of self is the workspace, and the workspace doesn't understand messages like **ring-up-item** or **discount**.

Control Structures - Part II

This chapter discusses three new control structures, **while**, **until** and **switch**, and a common variant of **for**.

Control Structures - While & Until

While and **until** statements execute a block over and over again until a condition is satisfied. For example:

```
$ dice, rolls.
dice := 0.
rolls := 0.
until [ dice == 7 ] do [
    dice := (choose 1 to 6) + ( choose 1 to 6).
    rolls := rolls + 1.
].
"It took" && rolls.name && "rolls to roll a 7".
```

When the **until** statement is executed, the expression dice == 7 is evaluated. If it isn't true, then the block is executed and the whole process repeated. When the condition becomes true, then the **until** statement finishes.

!! Note that the condition is in square brackets. Actually, it is a small block. The condition must be in square brackets because it is executed multiple times.

The **while** statement works the same way, only it keeps executing the block while the condition is true and stops executing when the condition is false. For example:

```
$ n.
n := 1.
while [ (n fib ) < 100 ] do [
    n := n + 1.
].
(n fib ).name && "is the first Fibonacci number greater than 100"
```

Infinite Loops

Be careful to ensure that the condition will eventually cause the loop to stop. If you accidentally execute a statement that is never going to end (or if it's just going to take too long), you can interrupt it.

To interrupt a running script:

1. **Press the pen down anywhere near the lower left hand corner of the screen.** You should press within an inch of the corner. You will have to hold the pen until the User Interrupt note appears. This may take a few seconds.
2. **Tap Halt in the User Interrupt note.** After a few seconds, a debugger will open. You can simply close it.

Control Structures - Switch

The **switch** statement lets you choose one of several different blocks of code to execute depending on some value. For example:

```
$ rank, suit.
rank := choose 1 to 13.
switch rank;
    case 1 do [ rank := "Ace" ];
    case 11 do [ rank := "Jack" ];
    case 12 do [ rank := "Queen" ];
    case 13 do [ rank := "King" ];
    default do [ rank := rank name ].
switch (choose 1 to 4);
    case 1 do [ suit := "Clubs" ];
    case 2 do [ suit := "Hearts" ];
    case 3 do [ suit := "Diamonds" ];
    case 4 do [ suit := "Spades" ].
return rank && "of" && suit.
```

Consider the first **switch** statement in this example: The **switch** statement is actually a consists of a **switch** expression, four **case** expressions and a **default** expression. The **switch** expression specifies the value of rank as the test value used in the following case expressions.

Each **case** expression specifies a value to match against. If it equals the test value, then the block of the **case** expression is executed, and the **switch** statement is finished.

After all the **case** expressions, if none of them matched, then the block of the **default** expression is executed. The **default** expression is optional, as shown in the second **switch** statement. If it is missing, and no **case** expression matched the test value, then the **switch** statement does nothing.

!! The punctuation of the **switch** statement must be followed carefully. Pay particular attention to the semicolons: There are semicolons between each part of the statement. There is a period only at the end of the whole **switch** statement.

Control Structures - For

There is another form of the **for** statement. Instead of executing a block once for each number in a sequence, it executes the block once for each element of a group. Up to now the only examples of a group that has been discussed are strings. A string is a group of characters. Groups are actually a more generalized object that will be discussed in a later chapter. Everything here will apply to those groups as well.

In a numeric **for** statement you specified the parameters of the numeric sequence with the receiver, **to** and (optionally) the **by** arguments. In this **for** statement you specify the group of elements to iterate over, in this case a string:

```
$ text, num-vowels.
text := "John Jacob Jinglehiemer-Schmidt".
num-vowels := 0.
for text do [ $ element e.
    if ("aeiou" has e)
        then [ num-vowels := num-vowels + 1 ].
].
return text.name && "has" && num-vowels.name && "vowels".
```

Another difference from previous **for** statements is that instead of the index variable, there is a different variable in the iteration. In this example, the local e will be set to each letter (or element) of the string in turn. The index variable is also available if you want it:

```
$ text, last-vowel
text := "John Jacob Jinglehiemer-Schmidt".
last-vowel := ???.
for text do [ $ element e, index i.
    if ("aeiou" has e)
        then [ last-vowel := i ].
].
if (last-vowel is-not ???)
    then [ return "The last vowel is in position" && last-vowel.name ]
    else [ return "There were no vowels" ].
```

» By now, you know enough about scripts to recognize that declaration expression in the **for** statement's block looks exactly like argument declarations for a script. In fact they are argument declarations. The

for statement passes arguments to the block, and **element** and **index** are block arguments.

Reverse For

This form of **for** statement has an optional argument **reverse**. If you specify **reverse**, then the elements (and the index) are iterated from the end to the start:

```
$ text, rev-text.
text := "college".
rev-text := "".
for text reverse do [ $ element e.
    rev-text := rev-text & e.
].
return rev-text.name && "is" && text.name && "spelt backwards".
```

» Notice that the **reverse** argument doesn't take any value, you just add it to the statement. This is an example of a flag argument, which is discussed in more detail later.

» Numeric **for** statements don't use **reverse**. Instead you just use a negative **by** argument:

```
for 10 to 1 by -1 do [ $ index t.
    transcript put "T minus" && t.name & ", and counting…".
].
transcript put "Liftoff!".
```

More Arguments

This chapter builds on the work in the chapter Writing Message Scripts. It will explain more details about different types of arguments and techniques for handling them. Make sure you have the cash-register object open.

Arguments - Positional

Sometimes it makes no sense to come up with a keyword for an argument. Perhaps for convenience or perhaps because it makes the message read better. In these cases you can have an argument that doesn't correspond to a keyword. They are called *positional arguments*, because their position in the message makes a difference. (Recall that keyword arguments can be written in any order).

Suppose we want a new message for cash-register to read:

```
cash-register  price-item  "eggs".
```

One could argue that this reads better than adding a keyword before the "eggs" argument, like:

```
cash-register  price-item  name  "eggs"
```

To use an argument that has no keyword, the script declares the argument with a dash in place of the keyword. (Think of the dash meaning that the keyword is omitted.) The script for the **price-item** message is:

```
$ - item-name.
switch  item-name;
    case  "milk"  do [ return  1.59 ];
    case  "eggs"do [ return  0.205 ];
    case  "cheese"do [ return  5.45 ].
return  ???.
```

Positional Arguments and Style

Only use positional arguments when there is no other way to make a message read nicely. While several predefined messages use positional arguments, we have found it rare that a user created message needs to use them. In fact, even the message above could have been renamed so as to read better with a keyword argument:

```
cash-register price item "eggs"
```

Note the subtle difference: the name of message has been split and the last word used as keyword for the argument. Unless there is a conflict with using the 'price' as a message name, this is a better solution. A common mistake of style is to include too many words in the message name. Programmers familiar with other languages are likely to make this error, as other languages don't have keywords and all descriptive text must become part of the name of the function or procedure.

» The above script returns the value ??? (unknown) when it doesn't know what the answer is. This is better programming style than not returning anything in this case. If the script that sent this message can't handle the ??? value, an error will occur. But if the script wants handle this condition, then it is a simple matter for it to test for the value ???.

Arguments - Flags

Some arguments don't need values. They are just indications to the script to do things one way or another. The **reverse** argument in the **for** statement is an example: it doesn't take a value, it just indicates whether or not to iterate backwards. These types of arguments are *flag arguments*. Flag arguments are very similar to keyword arguments and all of what you have learned about keyword arguments applies.

We will rewrite the **discount** script of cash-register to accept a flag argument, **double**. If the **double** argument is present, then the script will compute twice the normal discount.

```
$ on amount, double .
$ percent.
double ?= false.
percent := 0.
if (amount >= 10.00) then [ percent := 0.025 ].
if (amount >= 25.00) then [ percent := 0.05 ].
if (amount >= 100.00) then [ percent := 0.075 ].
if double then [ percent := percent * 2 ].
return amount * percent.
```

The declaration line now additionally declares an argument **double**. Notice that we've used the short cut where the name of the variable is the same as the keyword. This is common for flag arguments, although not required.

The third line defaults the value of the flag to false. When a message includes a flag argument, such as in the expression:

```
cash-register discount on 23.99 double
```

the system sets the flag argument in the script to true. If the message omits the flag argument, as in:

```
cash-register discount on 23.99
```

then like all omitted arguments, the system doesn't give the argument in the script any value at all. In this case, the default assignment expression in the script:

```
double ?= false.
```

will give the argument the value of false. After the default assignment expression has executed, you can be sure that the argument is either true or false.

Conditionally using Flags

Sometimes you don't know when you use a message that takes a flag if you want the flag or not: it may depend on values of variables when the script is run. For example, suppose after we sold five hundred dollars in a day, we want to start giving out double discounts. We could rewrite the script for **grand-total** to read:

```
$ amt-off, tax, be-generous.
be-generous := daily-total >= 500.
amt-off := self discount on sub-total double be-generous.
tax := sub-total * 0.0725.
total := sub-total - amt-off + tax.
daily-total := daily-total + total.
sub-total := 0.
```

The local variable be-generous will be true if we've sold over five hundred dollars today, otherwise it will be false. Then, when the **discount** message uses the **double** flag argument, it specifies a value, the value of be-generous. In the script for **discount** the flag argument **double** will be set to either true or false and will apply the double discount if directed.

!! If you give a flag argument a value, be sure to give it only true or false. The script that understands the flag variable will only expect these two values and will probably generate an error otherwise.

Section 2

Fundamental Objects

About This Section

This section discusses the various standard objects in the system. These are common objects that you will use often in your programs. Within each chapter, the fundamentals of how the object works is discussed, followed by messages with examples. You may choose to jump around the section, reading only the fundamentals, and leave the message details for later. Or you can read each chapter in full as you need those objects.

The objects discussed in this section are:

Numbers	Numbers represent quantities. They can be integers or floating point values. They understand arithmetic and comparison operations. There are two kinds of numbers: standard and extended precision.
Strings	Strings are collections of characters. They can be combined and divided up.
Groups	Groups are collections of objects: such as a set, or a sequence. Groups are a powerful way of handling multiple objects.
Misc. Objects	This chapter discusses the objects transcript, user, folder and time.
Special Objects	This chapter discusses the objects ???, nil, false, true.
Object	The messages that object understand are inherited by all objects. There are several common messages and messages for object introspection.

Numbers

Numbers represent numerical quantities. There are two kinds of numbers: standard and extended. Standard numbers have six digits of precision and a range of $\pm 10^{\pm 30}$ and are very efficient in both time and memory. Integers in the range ±32,000 are handled very efficiently and quickly. Standard numbers are what you get by default.

Extended numbers have fifteen digits of accuracy and a range of $\pm 10^{\pm 32,000}$. However, they are about two to three times slower than standard numbers and take much more space.

Both kinds of number represent base-ten quantities exactly with no rounding errors.

In this chapter's examples, x, y, and z represent any numeric values, n and b integer values, and s a string.

Arithmetic

Numbers can perform the standard arithmetic operations and mathematical functions:

x + y	*add*
x - y	*subtract*
- x	*negate*
x * y	*multiply*
x / y	*divide, dividing by zero results in ±infinity*
x % y	*remainder*
x **div** y	*integer division, integer result, rounded towards zero*
x.abs	*absolute value*
x.sqrt	*square root*
x.sin	*trigonometric sine*
x.cos	*trigonometric cosine*
x.tan	*trigonometric tangent*
x **min** y	*minimum of x and y*
x **max** y	*maximum of x and y*
n.fib	*the n-th Fibonacci number*

Comparison

Numbers can be compared with the normal comparison operations. These comparisons all return either true or false.

x == y	*equality*
x != y	*inequality*
x < y	*x less than y*
x <= y	*x less than or equal to y*
x > y	*x greater than y*
x >= y	*x greater than or equal to y*
x between y and z	*y <= x, and x <= z*

Precedence

The mathematical operators have precedence. This means that a sequence of operations are evaluated as you would evaluate them yourself. The precise rule is based on the concept of precedence. Each operator has a precedence. Operators with high precedence are evaluated before any operators of lower precedence. Operators of the same precedence are evaluated left to right. Of course, you can use parentheses to override this: expressions in parentheses are always evaluated first.

Highest Precedence:	- x
	x * y x / y x % y
	x + y x - y
	x < y x <= y x >= y x > y
Lowest Precedence:	x == y x != y

Entering and Displaying

Numbers can be entered in scripts. They follow the standard convention for entering numbers in computers. In addition you can enter integers in base 2, 8 or 16. Some examples are:

165	*an integer*
3.14159	*a number with a decimal part*
0.2	*just a decimal part*
.2	*the leading zero is optional (but recommended)*
-10	*a negative number*
5e2	*exponent notation: this equals 5 x 10^2, or 500*
0.625e-9	*this equals 0.625 x 10-9, or 0.000000000625*
0xA5	*hexadecimal (base 16) number, this equals 165*
0o245	*octal (base 8) number, this equals 165*
0b10100101	*binary (base 2) number, this equal 165*

When numbers are displayed, they are always displayed in base 10. In general they are limited to three decimal places and only used exponent notation if needed.

n.name *a string that is the default display format of a number*
format n **using** s *a string of the number formatted according to s*

The format string follows rules similar used in most spread sheets: '9's indicated required digit positions, '#'s indicate optional. Most other characters are simply copied. Note that it is possible to format numbers in ways that they can't be entered. This is the same message that number fields use to format their values for display. The number format option of number fields lets you specify the format string.

Some example formats and how they format the numbers: .314159, -23.4, and 181282:

"$#,##9.99"	$0.31	*the dollar sign is put in front*
	-$23.40	*forced to two decimal places*
	$181,282.00	*commas added*
"9.9###"	0.3142	*rounded to four places*
	-23.4	*only needed one decimal place*
	181282.0	*always adds places to the left if needed*
".999e"	.314e0	*forced exponent notation*
	-.234e2	
	.181e6	
"#9.9e?"	0.3	*exponent notation only if needed*
	-23.4	
	1.8e5	*too big to fit, so uses exponent*
"9%"	31%	*percent multiplies the value by 100*
	-2340%	
	18128200%	
"#,##9.99 cr; #,##9.99 db"		*semicolon separates two formats:*
	0.31 cr	*positive numbers use first*
	23.40 db	*negative numbers use second*
	181,282.00 cr	

number **scan** s	*convert a string to a number*
number **scan-integer** s **base** b	*convert a string to n integer*

These two messages allow you to convert a string to a number. **Scan** converts decimal numbers with optional decimal points, exponent notation, percent signs and minus signs. It will also interpret a number in parentheses as a negative number (accounting style). **Scan-integer** will convert only integer strings, but will do so in a number of bases: If the **base** argument is supplied, then it is used as the base of the number. The base can be anywhere from 2 to 16. If the base isn't specified, then it is assumed to be decimal, unless the string starts with "0x", "0o", or "0b", in which case it is hexadecimal, octal or binary respectively.

!! Notice that these to messages are sent to the object number, not to any particular number.

Rounding and Truncation

Numbers can be rounded and truncated.

round x **to** n	*round x to the n-th power of 10*
truncate x **to** n	*truncate x to the n-th power of 10*

N defaults to 0, which rounds or truncates to integers. If n is -2, then this gives two decimal places, if n is 3 then this rounds or truncates to thousands:

round 2.345 **to** -2	⇒ 2.35
truncate 2.345 **to** -2	⇒ 2.34
round 4590 **to** 3	⇒ 5000
truncate 4590 **to** 3	⇒ 4000

x.integer	*integer portion of x (truncates toward zero)*
x.fraction	*fractional part of x*
	x is equal to x.integer plus x.fraction
x.ceiling	*nearest integer >= to x (truncates toward +∞)*
x.floor	*nearest integer <= to x (truncates toward -∞)*
x.mantissa	*mantissa of x, always a number between -1 and 1*
x.exponent	*exponent of x*
	x is equal to x.mantissa times 10 raised to x.exponent

Extended Numbers & Conversion

Any numeric operation involving an extended number, results in an extended number. Since extended numbers are less efficient in both time and memory, you will want to use them only when you need the precision, and convert back to standard when you can. (**Integer**, **ceiling**, **floor**, and **exponent** are exceptions: they produce standard numbers if possible even if their argument is extended).

x.extended *returns an extended number equal to x*
x.standard *returns a standard number equal to x*

These return an extended or standard version of x respectively. If the number is already in the right format, it just returns it immediately. When converting to standard from extended, the value is rounded as needed.

x.precision *returns the number of decimal digits of precision.*

If x is a standard number, then 6. If x is an extended number, then 16. *[This is a bug, it should return 15.]*

Random Numbers

choose n *returns a random integer between 1 and n, inclusive.*
choose n **to** m *returns a random integer between n and m, inclusive.*

number.seed := x. *reset the random seed to x*

This resets the random number generator to a particular point, x. X should be a number between zero and one. If you reset the random number generator to the same value, then the same numbers will be returned from the choose message. This is useful if you are debugging a program that uses random numbers, because you can force the system to repeat exactly the same set of choices again.

Strings

Strings contain sequences of characters. The character set is ASCII. A string can have zero, one, or more characters, up to several thousand.

» Unlike other systems, there is no special syntax or object type for characters. A character in Glyphic Script is just a string with one element. When you get a character out of a string, you get a string with just that one character in it.

» The grand-parent of string is group, so strings can understand many of the messages that groups do. See the chapter on groups later in this section.

String Operations

s & t	*concatenate s and t*
s && t	*concatenates s and t with a space between*

These operators produce a new, third string that has all the characters of the first, followed by all the characters of the second. The && operator does the same thing, but places a space character between the two strings in the final result. For example:

```
"cat" & "apult"      ⇒ "catapult"
"cat" && "bird"      ⇒ "cat bird"
```

s @ n	*returns the n-th character of s*
s @ n := c	*sets the n-th character of s to be c*

In the both cases, the index n must be between 1 and the size of the string s. The first character of s is numbered 1. In the second example, the character c is any one character string. Generally you don't set characters in a string, you just build new strings.

```
"abc" @ 2.           ⇒ "b"
a := "def".
a @ 3 := "g".
a.                   ⇒ "deg"
```

s **from** n *the characters from position n to the end of the s*
s **from** n **to** m *the characters from position n to position m of s*
s **from** n **for** m *the m characters of s starting at position n*

These return a new string which is a copy of part of the string s. The copy always starts at the n-th position. If the **to** argument is given, then the copy stops at the m-th position. If the **for** argument is given, then it continues for m characters. If no other arguments, then the copy continues to the end of the string.

"the dog is blue" **from** 12 ⇒ "blue"
"the dog is blue" **from** 2 **to** 7 ⇒ "he dog"
"the dog is blue" **from** 7 **for** 5 ⇒ "g is b"

s **has** c *returns true if the string s has the character c*

The test looks for any occurrence of the character c (a one character string) in s:

"aeiou" **has** "e" ⇒ true
"aeiou" **has** "q" ⇒ false

Comparison

Strings can be compared with the normal comparison operators. Strings compare in the order that they would appear in the dictionary: "cat" comes after "california" and before "catatonic". When strings are compared, case makes a difference, and all uppercase letters come before all lowercase ones.

s == t *equality*
s != t *inequality*
s < t *s comes before t*
s <= t *s comes before or is same as t*
s > t *s comes after t*
s >= t *s comes after or is same as t*

Examples:

"cat" < "dog" ⇒ true
"cat" < "california" ⇒ false
"cat" >= "catatonic" ⇒ false

Entering and Displaying

Strings can be directly typed in a script by enclosing the string between double quotes. When typed in a script, strings follow slightly different rules than the rest of the text of the script:

- Case distinctions matter: the string "abc" is different than "ABC".
- Formatting is ignored: the string "abc" is the same as "**abc**". This is because ASCII has no way to encode formatting.
- The string cannot have any line breaks in it. If you want to have a line break so the script formats nicely, enter two strings and use the & operator. If you want to have a line break character in the string, use the escape sequence '\r' (see below).
- The string must be less than 250 characters. If you need to have a string which is longer than 250 characters, enter two strings and use the & operator.

Some characters that you may want to have in a string are difficult to enter. For example, if you just enter a double quote, it will end the string rather than include a double quote in the string. Certain characters can be entered by writing and *escape sequence*. This is a two character sequence starting with a backslash that represents one character in the string. The escape sequences are:

\n	a new line (control-j in ASCII)
\r	a carriage return (control-m in ASCII)
\t	a tab character
\\	a backslash
\"	a double quote character

» PenPoint's text editor interprets \r as a new break character and ignores \n. Other systems (including MS-DOS, Macintosh, and Unix) interpret these characters differently from PenPoint and from each other. If you are working with strings from or destined for these other systems, you'll have to learn how they handle these characters.

Groups

Groups are powerful objects. With them you can do things that would be much harder in other languages. A group is a collection of other objects. Any object whatsoever can be in a group: a number, a string, the value true, a cash-register, or any object you create. Even a group can be part of another group. The items in a group are its *elements*.

There are many different ways of working with groups, because there are many different ways of working with groups of objects. Sometimes you want to treat them simply as a collection of related objects. Sometimes the objects are kept in a specific order. In other languages there are stacks, queues, lists, and arrays. In Glyphic Script, there is only group, which serves them all.

Basic Operations

g **add** e	*add e to the group g*
g **remove** e	*remove the first occurrence of e, if it's there, from g*
g.size	*return the number of objects in g*

These are the basic messages for handling a group. Use them when you don't particularly care about the order of items in a group. You can follow the effect of the following expressions by opening up a workspace, executing the first expression, then opening up the workspace variable a to view it. Then, as you execute each expression in turn, you can see the effect on the group:

Expression	Contents of a	Comments
a := **new** group.	*empty*	*creates a new, empty group*
a **add** "cat".	cat	*adds a string*
a **add** "bird".	cat, bird	*adds another*
a **add** 123.	cat, bird, 123	*adds a number*
a **remove** "bird".	cat, 123	*removes a string, gap closes*
a **add**"cat".	cat, 123, cat	*adds a second copy of a string*
a **remove** "cat".	123, cat	*removes the first one it finds*
a **remove** 456.	123, cat	*no effect, as 456 isn't there*
a.size.		*resulting size is 2*

List Operations

A group can be treated as an ordered list. Each element in the list has an index. The index runs from one to the size of the group. If you insert or remove a given item, all the items shift over.

g **insert** e **after** i
g **insert** e **before** i *insert e into the group after or before index i*
g **delete** i *delete i-th item of the group*
g **place** e **after** f
g **place** e **before** f
insert e into the group after or before the first occurrence of f
g @ i *returns the i-th item of the group*
g @ i := e *replaces the i-th item of the group with e*
g **index** **of** e
returns the index of the first occurrence of e, if there aren't any then ???

Expression	Contents of a	
a := **new** group.	*empty*	
a **insert** "cat" **before** 1.	cat	
a **insert** "bird" **after** 1.	cat, bird	
a **place** 123 **before** "bird".	cat, 123, bird	*bird is now item 3*
a **delete** 1.	123, bird	*bird is back to item 2*
a **place** "cat" **after** 123.	123, cat, bird	
a **delete** 3.	123, cat	
a @ 2 := "dog".	123, dog	*replaces cat with dog*
a @ 1.	123, dog	*results in 123*
a **index** **of** "dog"	123, dog	*results in 2*
a **index** **of** "cat"	123, dog	*results in ???*

Queue Operations

Many tasks involve keeping track of objects in a queue. A queue is a computer science term for lists where items are added or removed only from the ends. For example: a cash-register tape is a kind of queue: new entries are only added to the end. A movie ticket line is a kind of queue: new people get on the end of the queue, and as tickets are sold, people leave the front.

While you could use the list operations above, there is a set of queue messages to make queue management easier. Queues have two ends, the front and back. The object at the front of the queue is first, and the

object at the back is last. Pushing an object onto a queue means to add it from an end. Popping it means to remove it from an end. When you pop an object, the result of the operation is the object just removed.

g **push** e **first**	*add e to the front of the queue*
g **push** e **last**	*add e to the end of the queue*
g **pop first**	*remove and return the front of the queue*
g **pop last**	*remove and return the end of the queue*
g.first	*return the front of the queue (without removing it)*
g.last	*return the end of the queue (without removing it)*

Example Sequence:

Expression	Contents of a	
a := **new** group.	*empty*	
a **push** "sue" **last** .	sue	*add sue*
a **push** "bob".	sue, bob	*last is the default*
a **push** "jill" **first** .	jill, sue, bob	*jill gets ahead in line*
a.first	jill, sue, bob	⇒ *jill*
a.last	jill, sue, bob	⇒ *bob*
a **pop last** .	jill, sue	⇒ *bob, take from end*
a **pop** .	jill	⇒ *sue, last is default*
a **pop first** .	*empty*	⇒ *jill*

Set operations

In all the above operations, a group could hold an object more than once. If you add "cat" to a group three times, there will be three "cats" in the group. Similarly, if there were two numbers in the group both 100, then removing 100, would only remove one of them. Sometimes it is desirable to only add an object to a group if it is not there, and remove every copy of it if it is. These messages do this:

g **include** e	*add e to the group if it isn't there*
g **exclude** e	*remove e from the group, multiple times if needed*

If a group doesn't have any element more than once, then the group can be thought of as a set. You can perform set operations on two groups, producing a third group from them:

g **union** f	*a new group with every element in g and f*
g **intersect** f	*a new group with only elements that are in both g and f*
g **difference** f	*a new group with every element in g that is not in f*

Example Sequence:

Expression	Contents of a/b/result	
a := **new** group.	*empty*	
a **add** "sue".	sue	*start off with a group*
a **add** "bob".	sue, bob	
a **add** "sue".	sue, bob, sue	
a **include** "bob".	sue, bob, sue	*no change, already has bob*
a **exclude** "sue".	bob	*removes both sues*
a **include** "jill".	bob, jill	*adds jill*
b := **new** group.	*empty*	
b **include** "bob".	bob	
b **include** "sue".	bob, sue	
a **union** b	⇒ bob, jill, sue	*a new group*
a **intersect** b	⇒ bob	*only item in both*
a **difference** b	⇒ jill	*item in a, not in b*

General Operations

These messages are applicable to groups in almost all contexts.

g.empty *returns true if g is empty*

g **from** i
g **from** i **to** j
g **from** i **for** j

returns a new group that contains the elements of g starting at index i. With no extra arguments, it takes all the elements to the end. With the to argument, it takes elements up to index j. With the for argument, it takes the next j elements.

g & f

returns a new group which has all the elements of g followed by all the elements of f

g **has** e *returns true if g has the element e*

g **count** e *returns the number of times e is in g*

choose g *returns a random element of g*

These messages take blocks of code to control their operation.

g **transform** **by** [$ **element** e, **index** i. …]

Returns a new group that contains the results of executing the block for each element in g.

g **reject** **by** [$ - e. …]

Executes the block for each element in g and removes the element if the result of the block is true.

g **sort**
g **sort** **by** [$ - a, - b. …]
g **sort** **by-value** [$ - a. …]

These messages sort the group. In the first form, the elements are compared with the < operator. This works for numbers and strings, but most other objects don't understand the < operator. In these cases, you have two choices: the first form, with the **by** argument, you supply a block that compares the two block arguments and returns true if the first should sort before the second. The second form, with the **by-value** argument, you supply a block that returns for each element a value that can be compared with the < operator. The first form is more efficient than the second.

for g **do** [$ **element** e, **index** i. …]
for g **reverse** **do** [$ **element** e, **index** i. …]
This is just the **for** statement discussed earlier in this manual.

Miscellaneous Objects

This chapter discusses four unrelated objects: user, transcript, folder and time. *[the last two are still not discussed]*

User

The user object has several messages for interacting with the user via modal notes. Each of these messages displays a note to the user. The system will not proceed until the user answers the note by tapping on one of its buttons.

ask user msg **yes-button** y **no-button** n **with-cancel**

This message displays a note with two buttons at the bottom. The text displayed in the note is the string msg. By default the two buttons are labeled 'Yes' and 'No', but the labels can be changed with the **yes-button** and **no-button** arguments. The note is modal and the user cannot continue until a button is tapped. When the user taps a button, the note is removed and the result of the **ask** message is true if the user tapped the 'Yes' button, and false if they tapped the 'No' button.

If the optional argument **with-cancel** is present, then a third button is added labeled 'Cancel'. You can't change the text of this button. If the user taps this button instead of the other two, then the note is removed and the result of the **ask** message is ???.

Examples:

```
a := choose 1 to 10.
until [ ask user "Do you like the number" && a.name & "?" ]
            do [ a := choose 1 to 10 ].

$ answer.
a := 23.
answer := ask user "Change this number?"
            yes-button "Double It"
            no-button "Halve It"
            with-cancel .
if (answer is-not ???) then [
   if answer then [ a := a * 2 ]
   else [ a := a / 2 ].
].
```

tell user msg **ok-button** o **with-cancel**
This message is very similar to the **ask** message, except that it only puts up one button with a default label of 'OK'. You can change the label of the button with the **ok-button** argument.

This message is often used to tell the user something has been done, or to confirm an action.

Examples:

```
tell user "Time for dinner!"

$ confirm.
confirm := tell user "Really remove bob from the list?"
                ok-button "Remove"
                with-cancel .
if (confirm is ???) then [ return ].      user canceled
santas-list remove "bob".
```

query user msg **default-text** d **ok-button** o **with-cancel**
This message displays a note with a write in field. The write in field starts off with the text supplied by the **default-text** argument (or blank by default). In addition, the note has an 'OK' button and an optional 'Cancel' button. The text of the 'OK' button can be changed with the **ok-button** argument. The user can edit the text in the write in field. If the user taps the 'OK' button, then the text in the write in field is returned. If the user taps the 'Cancel' button (if any) then ??? is returned.

Examples:

```
$ n.
n := query user "What is your name?".
tell user "Hello" && n.

$ new-item.
a := "apples".
new-item := query user "Enter a fruit:"
                default-text a with-cancel .
if (new-item is ???) then [ return ].
a := new-item.
```

user **choose from** g
This message displays a pop-up menu of the items in the group g. If the user taps on one of the items, the menu is removed and that item is

returned as the result of the **choose** message. If the user taps outside the menu, then the menu is removed and ??? is returned.

Example:

```
$ animals, new-pet.
animals := new group.
animals add "bird".
animals add "cat".
animals add "dog".
animals add "zebra".
new-pet := user choose from animals.
if (new-pet is ???) then [ return ].
return "Eric the" && new-pet.
```

Transcript

The transcript is a window that displays text messages. It is used only for debugging: the window will not exist if you give your completed application to someone without Glyphic Codeworks. You can have your scripts write text onto the end of the transcript at any time. This can sometimes help you understand and debug your programs. The transcript is not infinitely long: it only holds the last several thousand characters. You can open the transcript from the System menu.

transcript **put** msg

This is the only message transcript understands. It appends the string msg to the end of the transcript on a new line.

Example:

```
$ n, root-n, old-guess.
n := 2.
old-guess := 0.
root-n := n.extended.
until [ (root-n - old-guess abs ) < 1e-15 ] do [
  old-guess := root-n.
  root-n := (n / old-guess + old-guess) / 2.
  transcript put "root-n is now" &&
      ( format root-n using "9.###############").
]
return root-n
```

Section 3

Language Reference

Objects

Objects are the fundamental way that all information is stored and manipulated in the system. An object has information and operations (properties) that describe some entity that a program works with. For example a receipt might be an object in a retail sales program. The receipt object would have information such as the date, which items were bought, and the sales price. The receipt object would also describe operations such as calculating the tax, updating inventory, and voiding the sale.

All information in the language is contained in objects. This includes ordinary data like numbers and lists, as well as complex data such as a sales receipt. Programming in this system consists of building objects with information and operations.

!! An object is an entity in the system which can have state, receive messages and interact with other objects. Objects are created in response to user and program actions such as **new** and **copy** . Objects are destroyed automatically when they are no longer referenced by any other object.

Properties

An object can have one or more properties. A property is a named value or script in the object (like instance variables and methods in other object systems).

The name of a property is any valid symbol according to the following rules:

- Any string of alpha-numeric and hyphen characters starting with a letter, case is disregarded

  `line-width` `speed` `the-1st-appointment` `xjs5000`

- Any string of the graphic characters: `!#$%&*+-/:<=>?@\^|~` (except the sequences `:=`, `?=`, `?`, `!`, and `???`)

  `+` `==` `>=` `&&`

Inheritance

Each object can inherit properties from another object. These two objects form a parent/child relationship. This can be repeated: the child object can itself be the parent to other objects, and the parent object can be the child of yet some other object (cycles are not allowed). These

parent/child relationships form a tree (also know as a directed, acyclic graph). At the root of the tree is an object that has no parent. Typically this object is the object called object[1,2].

When something in the system attempts to reference a property of a given object, in general, first the object is checked to see if it has the property. If not then its parent is checked, then the parent's parent and so on until either the property is found, or there are no more parents to check. This process is called 'lookup'. If the property is found in one of the parents, then it is said to be an inherited property. There are some variations on this process, see property scopes, and self sends below.

Every object has a property called parent. The value of the parent property is the parent of the object. If the object doesn't have a parent (for example object doesn't) then the value of this property is nil .

Creating New Objects

New objects are created in response to user and program actions. There are two basic ways that an object can be created: coping and newing. All other ways in which cause an object to be created are simply variants of these two operations. Copying an object creates a new object that is like the original in all respects, except that it is new and distinct object. Newing an object creates a new object which is a child of the original. In this case the new object is generally like the original because it inherits all the properties of it. However, the new object has its own copies of each 'instance' property.

1. "Rose is a Rose is a Rose" — G. Stein
2. "A rose by any other name would smell as sweet." — W. Shakespeare

Property Scopes

Each property also has a scope. The scope determines how the property should be handled during the lookup process and during spawning. There are three scopes:

- **Instance** properties are copied for each child during newing. Thus child objects don't need to inherit them as each have their own copy. (Instance properties are inherited in the rare cases where a child doesn't have its own copy.)

- **Class** properties are accessible to all operations. Child objects inherit these properties. They have no special effect on newing.

- **Private** properties are accessible only for this object. Child objects do not inherit these properties. They have no special effect on newing.

Values

A property of an object may either name a script or a value. When it names a value, this value be any other object, such as the number 17, the value true , or a sales receipt. This value is actually a reference to the other object, and thus two or more properties in any number of objects can all have the same value.

Garbage Collection

While objects are explicitly created in response to various user and program actions, objects are not explicitly destroyed. Instead, the system figures out when an object is no longer needed (because no property in no object refers to it anymore), and then destroys it. This process is called garbage collection and takes place in the background.

Scripts & Messages

Objects are manipulated by sending them messages. A message is a request for a property. When a message is sent to an object (called the receiver of the message), the property named in the message is sought in the object via lookup. Assuming that the object has or inherits the property, either the value of the property is referenced (if it contains a value) or property's script is run (if it contains an executable script).

A script is a sequence of messages. When a script is run, each message in the script is sent in turn. Scripts can take arguments. Arguments are additional information that can be specified in the message to modify the action of the script.

Some messages return a value back as a result. If the property in the receiver was a value then the result is always the value. If the property contained an executable script, then the script can decide what value, if any, to return.

Messages

There are several forms that a message may take. Each form is designed for ease of use in a particular situation. However, they are all equivalent and interchangeable. A message must specify two pieces of information: the name of the property to reference and the receiving object. Optionally, a message may supply arguments.

The four forms of messages are shown below. Note that in the code examples some words are bold and some are not. This is important. When you write code in the system, you need to enter code with this formatting. There are various accelerators in Glyphic Codeworks to make this easy.

Message first	**new** group **capacity** 100.
property	new
receiver	group
arguments	**capacity** 100
usage	the property name is in bold and comes first, the receiver second. This form is commonly used for messages where the property name is an imperative verb and causes action to occur.

Message second	management**add** susan.		
property	add		
receiver	management		
arguments	susan		
usage	the property name is in bold and comes second, the receiver first. This form is commonly used for messages where the property causes local change to the object itself. Note that this form and message first form are completely interchangeable: you can simply swap the first two words.		

Operator	30 * 40 + 50.		
property	*	+	(there are two messages)
receiver	30	120	(120 is the result of the first message)
arguments	40	50	
usage	the property name is the first operator symbol, the receiver the first object mentioned. This form is commonly used for arithmetic and other very common operations because it is concise. Notice that unlike the first two forms, subsequent operator symbols form new messages, not arguments. These messages follow the rules of arithmetic precedence. (Parenthesis can be used if needed.)		

Dot Notation	susan.pet.breed.		
property	pet	breed	(there are 2 messages)
receiver	susan	woofer	(susan has a dog named ‘woofer’)
arguments	-	-	(there are no arguments)
usage	the property name follows the receiver after a period (or dot). The property name needn’t be in bold here. This form is commonly used to reference properties that have values (rather than scripts). This form cannot pass arguments. These messages are always read left to right.		

Scripts

When a script is run several things may happen: Arguments may be passed into the script. The script may send messages to various objects. The script may return a value to the script or user who sent the message that caused this script to run. There are two major areas of scripts to consider: the objects a script can manipulate, and the ways it can send messages to those objects.

Locals, Arguments, and Globals

Within a script (or any block), you can define variables for the scripts local use. Local variables are defined with a dollar sign ($, the definition character), and then the names of the locals:

```
$ biggest-block, x, y.      defines three locals
```

The locals can have any object, or result of any message assigned to them and can be used later as either the receiver of a message or as an argument to a message. For example:

```
$ sum, average.
sum := bill.salary.
sum := sum + hillary.salary.
sum := sum + socks.salary.
average := sum / 3.
```

Some messages have arguments. These arguments can be used by a script for additional information. Most arguments are preceded by a keyword, a symbol in bold, that indicates what the argument is to represent. For a script to use an argument, it must declare the keywords it is expecting, and a name for the value of the argument it will use internally. For example if the following message were run:

```
send a-box to "1600 Pennsylvania Avenue" via federal-express.
```

The send property of a-box might contain a script like to following:

```
$ to address, via carrier.
$ envelope, postage-due.
envelope := carrier get-container .
put self into envelope.
postage-due := carrier cost of envelope.weight to address.
carrier accept envelope.
return postage-due.
```

All scripts have a set of objects that they can refer to known as globals. These objects are determined by the development environment. At present, they encompass all the objects in the home project of the script, and the exported objects. For example, the objects object and

group are globals that can be used by name at any time.

Undefined Values

When an argument isn't passed and before a local has been assigned a value, their values are said to be undefined. In general, using an undefined value is an error and causes a run-time exception.

However, it is useful to allow messages to leave off arguments and get default values. There are two general ways of doing this: defaulting and testing. Assume that a bank object has a withdraw-cash property, and a cash-account property. We'd like the script for withdraw-cash to, by default, withdraw $100 from the cash-account, and optionally change which account is the cash-account now and in the future. The script might look like:

```
$ - amt, from acct.                    — the two arguments
$ cash.
amt ?= 100.                            — default amt to 100 dollars
if (acct?) then [
    cash-account := acct               — if given, change the account
].
if (cash-account.balance > amt)
then [                                 — if we have enough, take it out
    cash := amt.
cash-account.balance := cash-account.balance - amt.
] else [                               — otherwise, take as much as we have
    cash := cash-account.balance.
cash-account.balance := 0.
].
return cash.
```

Notice that the defaulting of amt looks like an assignment to a variable (amt ?= 100). It is, it is an assignment to the argument if and only if the argument wasn't passed in (this is the only type of assignment allowed for arguments). The test to see if the acct argument was passed (amt?) is returns true if it was, and false if wasn't. With the above property we could then run the messages:

```
my-bank withdraw 400 from my-checking-account.
my-bank withdraw 200.
my-bank withdraw 200 from my-holiday-account.
my-bank withdraw .
```

After these messages, the checking account would be out 600 dollars (first two messages), and the holiday account 300 (last two messages).

» Although only occasionally useful, you can use the default form of assignment (?=) and the is defined test (?) with local variables as well.

» The undefined value can be assigned from locals and arguments to other locals, as well as passed on to messages. In other words, if a script declares an argument, and simply uses it as an argument to another message, then if it was undefined in the script, it will be as if the argument was left off in the message called. This behavior can be bypassed by following the argument or local with an exclamation point (!) which will cause a run-time error for undefined values even in these allowed cases.

Self and Its Properties

A script can also manipulate the object that was sent the message and all that objects properties. The name self refers to the object that received the message that caused a script to run. Furthermore, the value of property (including inherited ones) can be accessed or set by name within the script. For example, if a box object had two properties: height, width, then the following could be the script for a make-square property:

```
$ average.
average := (width + height) / 2.
width := average.
height := average.
```

» Getting and setting the properties of self are actually message sends. If the property is a script, that script will be run with a single, optional positional parameter. While this isn't generally important, it allows you to override an instance variable with a script.

There are two variants of self that are sometimes important. The name this refers to the object in which the script was found. Remember that a message was sent to an object, which caused a property lookup, which caused a script to run. That property may have been inherited, in which case, the place where the script was found is not the same was the object the message was sent to.

The name super works like self, in that you can use it send messages to the object to which the original message was sent, except that it causes

lookup to start at the parent of the object where the script was found. This is useful to adding incremental behavior to a property's script. For example, if a bird object wants to sing the same song as its parent does with some additions before and after, its script for the sing property might look like:

```
self tweet .        make some sounds
self chirp  .
super sing .         sing what my parent sings
self chirp  .       sing some more sounds
self chirp  .
```

Literals

There are several kinds of objects directly that you can use within a script. These are known as literals, as you write them literally in a script.

- **Numbers** `42` `-24.5` `0x1AC` `5.2e-6`
- **Strings** `"San Francisco"` `"billiards"`
- **Symbols** `$name` `$red` `$+`
- **Specials** `true` `false` `nil` `???`

Numbers can be positive or negative, have a decimal point and a fractional part, and have a scientific exponent. However, you can write integers in hexadecimal, octal, or binary (bases 16, 8, or 2) by starting the number with `0x`, `0o`, or `0b` respectively.

Strings are sequences of characters enclosed in double quotes. Unlike the rest of the language, within a string both case and spaces make a difference and are retained in the string precisely as they are written. Within a string, several special characters can be encoded using the backslash character. The backslash and the character following it are converted into a single character in the string as follows:

- **`\n`** newline, line separator
- **`\r`** carriage return, line separator in some older systems
- **`\t`** tab character
- **`\"`** quote character, allows you to have s quote in a string
- **`\\`** backslash, allows you to have a backslash in a string

Symbols are used when you need to manipulate the name of a property. They are written by a dollar sign followed by what every characters would make up a legal property name. The are rarely used.

Specials name objects that are always available. There are four of them: true and false are the results of comparison operations, nil is used to mean no-object (for example in the parent slot of object, since it has no parent), and ??? (pronounced unknown) is used whenever no value makes sense.

Blocks

Blocks are portions of a script that can be treated separately. They may be conditionally and repeatedly invoked (see if, for, and while), they may be passed as arguments (for error conditions). A block is a series of messages in square brackets. The beginning of the block may declare local variables for itself and arguments as needed. In fact, a whole script is really a block without the surrounding brackets. For example:

```
s := 0.
for 1 to 50 do [ $ index i, i-squared.
    i-squared := i * i.
    s := s + i-squared.
].
```

The code between square brackets is a block. The for statement (» actually a message to the number 1) executes the block once for each number from one to fifty. The block has an argument named index that is set for each execution of the block, and a local variable.

» Variables are lexicaly scoped and shadowed.

» There are two differences between whole scripts and blocks: 1) By default, scripts have no value whereas the value of executing a block is the value of the last expression. 2) Scripts will generate an error if called with an argument they don't declare. On the other hand, blocks generate an error if they are called without all their arguments.

Return Values

Scripts can return a value. If the last expression of a script or a block starts with the word **return** , then the value of that expression is returned as the value of the script. Even if the return appears in a block, the whole script is finished and returns the value. For example, the following script returns the first client who needs a sales call:

```
for my-clients do [ $ element c.
    if (c.needs-call)
        then [ return c ].      returns, stopping the for loop
```

```
].
return    my-family.mother.      otherwise,might as well call mom…
```

The value for a return is actually optional. A return without a value will still cause the script to finish, and the script with return without a value.

» The value of a script that returns without a value is undefined, just like local variables and arguments that haven't been assigned. In the same way, it will cause a run-time error if you try to use this value. This is why the script above returns a value in the case where it couldn't find a client. Often the value ??? (unknown) is returned in these situations (rather than one's mother). It is a legal object and can be tested for by the script that sent the message.

Formatting Messages

There are several aspects of formatting message sends in scripts collected here.

Scripts are written in formatted text. As seen in the examples above, bold and non-bold distinctions play a part in the language. Specifically: bold words are always the names of messages or arguments. Other formatting is important in Glyphic Script as well. Italic passages are treated as comments, and strike-through passages are treated as code not to be executed. Any text in either (or both) formats is skipped when the system looks at a script. The reason for having both styles is so that parts of a script with comments can be disabled without loosing track of the comments. This example has comments for both lines of the script, but the second line has been disabled and won't be executed:

```
x := y * y.                    x gets y squared
x := x + (y / 10)              add in a little bit more
```

Many messages return a useful value. Often it is convenient to use that value in another message without first storing the value in a local variable. There are two ways to do this: Parenthesis group a message that is to be sent and message used in its place. They have been used without explanation in the examples above, and operate like they do in most languages and everyday math. The second is the use use of a semi-colon which parenthesizes the message send to the left. This is best shown by example. In the following groups of messages, the first

expression is equivalent to the code fragments that follow it:

a) **schedule** work-order **for** (shop **next-free-time** **duration** "3:00").
b) $ t.
t := shop **next-free-time** **duration** "3:00".
schedule work-order **for** t.

a) date-book **find-appointment** **with** "bill";
cancel **because** "band rehearsal".
b) (date-book **find-appointment** **with** "bill")
cancel **because** "band rehearsal".
c) $ appt.
appt := date-book **find-appointment** **with** "bill".
appt **cancel** **because** "band rehearsal".

a) payment := bill.total * discount * tax + shipping-charges; **round** **to** 2.
b) payment := (((bill.total * discount) * tax) + shipping-charges)
round to 2.
c) $ a, b, c.
a := bill.total * discount.
b := a * tax.
c := b + shipping-charges.
payment := c **round to** 2.

» The semi-colon syntax is different than the Smalltalk-80 continuation syntax that it resembles.

Primitives

» This section is only of interest to primitive writers. If you aren't writing primitives, this won't make much sense.

Each primitive must be matched with a script. The script declares the arguments that the takes. Then the script has a line that declares the primitive group and name. When the script is invoked, the primitive is called and the result of the primitive, if any, is returned. For example, the declaration of the round property for numbers is:

$ **to** places.
primitive math, round.

Grammar

This is a grammar of the language presented in an enhanced BNF format:

sym	::= *sequence-a*	— a production of sym with two alternatives
	::= *sequence-b*	
	{ *sequence* }	— an optional sequence
	{ *sequence* }*	— optional sequence zero or more times
	ε	— the empty sequence

Short semantic descriptions are in italics. Where ambiguous, the first of several alternatives is favored.

[This grammer is not totally acurate. It doesn't handle operator precedence.]

Language Elements

script ::= block
::= { decl }* primExp
A script is either a block of code, or a primitive.

block ::= { decl }* { expression . }* { return } { expression } { . }
a block consists of optional declarations followed by a sequence of expressions, which are executed in turn. The block may end in a return in which case the script (not just the block) is exited. The return may have an expression whose value is returned as the value of script. In absence of a return, the value of a block is the value of the last expression, and a script will return no value.

decl ::= $ formals .

formals ::= formal { , formal }*
::= ε

formal	::= functor word	*named argument*
	::= - word	*positional argument*
	::= functor	*named argument with same local name*
	::= word	*local variable*

primExp ::= primitive { word , } message { . }
::= primitive { word , } number { . }

The optional word is the name of the primitive set, and either the message or number is key to which primitive.

expression ::= expHead { expTail }*

The base expression is executed, then the tail expressions left to right. Finally head expressions, right to left.

expHead ::= message value argsSet
::= value { message argsSet }
::= valSet

in the first two forms, the message is sent to the value with arguments; in the third form the result is the value

expTail ::= ; message argsSet
::= ops argsSet

the message (or operator) is sent to the value from the right with arguments

argsSet ::= args { := expression }

If there is an assignment then the whole message that these args are for is an assignment message; the value of the expression forms the last positional parameter to the message.

args ::= arg { { , } arg }*
::= ε

arg	::= keyword value	*named argument with value*
	::= keyword	*'flag' argument, implied value of true*
	::= value	*positional argument*

valSet	::= word := expression	*(1)*
	::= word ?= expression	*(3)*
	::= value := expression	*(2)*

The value of the expression to the right of the assignment symbol (the := or ?=) is assigned to the variable or value. The value of this assignment is the value of the expression.
1) word may be a local or a property of self
2) conditional assignment is made to either an argument or local (assignment is only made if the argument wasn't passed in, or if the local hasn't been assigned to yet
3) the property named by the value is assigned to (for example foo.x or foo @ 3).

value ::= primary { member }* { ! }

::= primary { member }* ops
::= operator value ops
the members are evaluated left to right, each names a property that is fetched from the value to the left; If there is an exclamation, then the value is checked for undefined and a run-time error is generated if it is

opsSet ::= ops := expression

ops ::= operator value { ops }

primary ::= word { ? } *(1)*
::= self
::= number
::= string
::= symbol
::= special
::= (expression) *value is the value of the expression*
::= [block] *value is a block object*
1) the word may be either a local, an argument, a global, or a property of self whose value is fetched. In the case of locals or arguments, the optional question mark results in a value of true or false, depending of if the value is defined or not.

Lexical Elements

. , ; ( ) [ ] ? ! := ?= $ -
the literal punctuation marks and symbols

return *the bold literal:* **return**

self *any of the literals:* self this super

number *a decimal number with optional fractional part and exponent, or a binary, octal, or hexadecimal number; may have a leading minus sign*

string *a string of characters between double quotes; must appear on one line; can contain quoted special characters using the backslash notation*

symbol *a dollar sign followed directly by any legal word or operator*

special *any of the literals:* true false nil ???

word	*any non-bold sequence of alphabetic, numeric, or dash characters that begins with an alphabetic*
functor	*a bold version of word*
operator	*any sequence of the operator characters:* ! # $ % & * + - / : < = > ? @ \ ^ | ~ *except the sequences:* $:= ?= ? !
member	*a word preceded by a period*