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.
```