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.)

x @ y <--Highest Precedence

- x

x * y x / y x % y

x + y x - y

x & y x && y

x < y x <= y x >= y x > y
x == y x != y <-- Lowest Precedence

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.
```