**COWS Language**

# Concepts

**COWS Version 1.3**

**Sean Luke**

**March 20, 1994**

This document provides a general overview of the COWS language. For a more formal description of the COWS language, see *COWS Language Formal Specification.*

## Basic Format

COWS looks like LISP. It uses the exact same overall syntax and parenthesis mess. But to the the more awake programmer, COWS works more like HyperTalk in that it has no main loop, is event driven, and uses strings for its sole data structure (at least for the while).

Like LISP, COWS refers to *symbols* for its function and variable names. To look up or store those symbols, COWS uses several *dictionaries* stored as hash tables. The *function dictionary* stores your user-defined functions. The *library dictionary* stores the standard library functions and any library functions defined by the application to which the COWS interpreter is attached. The *global dictionary* stores global variables. The *current dictionaries* of local variables within a function are stored on a stack along

with evaluated values and unfinished symbols.

When COWS is in a function, it pushes a current dictionary onto the stack and stores function arguments and local variables there. When it exits the function, COWS pops the dictionary off the stack. Hence, COWS can handle recursion quite nicely.

## Communicating with The Outside World

COWS functions are called by sending a method to the interpreter, along with a bunch of strings for the arguments to the method. COWS functions return a single string.

But COWS can also call Objective-C methods as it runs. These are stored in the function dictionary along with their selectors. This enables COWS programs to act as macro facilities for NeXT programs.

## COWS Data Types

COWS has only one data type: The string. All numbers are stored as floating-point numbers encoded in a string. All truth values are stored in a string.

| | |
|---|---|
| *Strings* | When referenced, strings must be written with quotes: **"hello"**. |
| *Numbers* | Numbers are stored with precision as such: **"1.231e43"**<br>When referenced in COWS, they can just be written **1.231e43**. |
| *Truths* | Truth values are either **"t"** or **"f"**, and can just be written **t** or **f**. |

## COWS Values

In COWS, a *value* is anything that returns a value, be it a special form, a function, a string constant, a number, a truth, or a variable. All values are stored internally as strings.

COWS thrives on valuesÐthey are the embodiment of the COWS data type. Values are passed as arguments to other functions or special forms. Each function or special form returns a single value.

## COWS Functions

Functions are declared globally, and can be declared in any order. Function arguments follow the function-name, and are optional. Local variables are declared using the optional *variable* keyword, and are initialized to an empty string. You define a COWS function in the format:

**(function** *function-name*
*argument-name argument-name ...* [optional]
**variable** *variable-name variable-name ...* [optional]
**do** *function-body***)**

An example:

⇒ **(function print-x-to-y x y**
**variable z**
**do (if (< x y) (for z x y 1 (print z))))** *prints the numbers from x to y, given that x < y.*

During the course of action, COWS functions may call any other COWS function, be it a user-supplied function or a library function. This includes a function calling itself (recursion). There is no specified limit to recusion depth.

## COWS Global Variables

Global variables are declared globally just like functions, and can be declared in any order. Variables are initialized to an empty string. Be careful not to declare a global variable twice! You define a global variable in the format:

**(variable** *variable-name variable-name ...* **)**

An example:

⇒ **(variable a hello junk)** *allocates globals* a, hello, *and* junk.

## COWS Special Forms

COWS has four *special forms*. Special forms look like functions, but they're not quite, in that they may take other functions or variables as arguments, not just values. Special forms are complex to write interpreters for and add considerably to the syntax of a language, so COWS has just four of them:

*set* Sets variables.
Returns: *value*

Format: **(set** *variable value***)**

Examples:

⇒ **(set a 100)**
*sets a to 100*

⇒ **(set b "hello")**
*sets b to* hello

*if* Provides if-then-else control. *test-value* must return a *t* or *f*.
Returns: *result of either then-function or else-function.*

Format: **(if** *test-value then-function***)**
Format: **(if** *test-value then-function else-function***)**

Examples:

⇒ **(if (> 100 a) (print "okay"))**
*if 100 > a, print* okay, *otherwise do nothing.*

⇒ **(if (= 1 a) (print "yes!") (+ a 100))**
*if a is 1, then print* yes!*, otherwise add 100 to a.*

*while* Provides while-do control. *test-value must return a t or f.*
Repeatedly
calls *test-value* and optionally *function* until *test-value* returns *f.*
Returns: *result of test-value.*

Format: **(while** *test-value***)**
Format: **(while** *test-value function***)**

Examples

⇒ **(while (< 100 (set a (+ a 1))))**
*adds 1 to a until it equals or is higher than 100*
⇒ **(while (> 100 (set a (- a 10))) (print a))**
*subtracts 10 from a and prints a until a is less than or equal to 100*

*for* Provides for-next looping.
Returns: *variable's value*

Format: **(for** *variable start-value stop-value step-value loop-function***)**

Examples
⇒ **(for a 1 100 1 (print a))**
*prints numbers 1 to 100.*
⇒ **(for a 100 0 -2 (print a))**
*prints even numbers from 100 down to 0*

## COWS Standard Function Library

COWS comes with a very small *Standard Function Library* intended to provide the minimal tools a user might need to communicate with an application. Other libraries will follow to add power to the language, but are not mandatory. These libraries might include array-manipulation, inter-process communication, string manipulation, and better math control.

In the Standard Function Library hese definitions are sometimes given with items like *value+,* which means one or more values, or *value*,* which means zero or more values.

***Control***

*do* Performs each value, then returns the last.

**(do** *value* value***)**

*do-first* Performs each value, then returns the first.

**(do-first** *value value**

***Response***

*print* Prints values out to stdout, with a newline after each. Returns the first value.

**(print** *value**

*error* Flags as an error message *value.*

**(error** *value***)**

***General Predicates***

*is* Determines if several string values are the same. Returns t or f.

**(is** *value*+**)**

***Truth Tables***

*and* ANDs several values, which should be truths (else they're taken as f).

**(and** *value*+**)**

*or* ORs several values, which should be truths (else they're taken as f).

**(and** *value*+**)**

*not* NOTs a value, which should be a truth (else it's taken as f).

**(not** *value***)**

***Strings***

*concatenate* Concatenates several string values, returning the result.

**(concatenate** *value**)

*quote* Returns a double-quote ". Because quotes are delimeters, a quote can't, well, be quoted (put in a string constant). Quote would be used as in

⇒ **(concatenate "Call me " (quote) "John" (quote) ".")**

*which would return:* **Call me "John".**

**(quote)**

***Math***

= Determines if several numerical values are the same. Returns t or f.

**(=** *value+***)**

< Does what you'd expect. Non-numbers are 0. Returns t or f.

**(<** *value value***)**

> Does what you'd expect. Non-numbers are 0. Returns t or f.

**(>** *value value***)**

+ Adds several values. Non-numbers are 0. Returns the result.

**(+** *value+***)**

* Multiplies several values. Non-numbers are 0. Returns the result.

**(*** *value+***)**

- Subtracts several values from a first value. Non-numbers are 0. Returns the result.

**(-** *value value**)**

/ Divides several values into a value. Mathematically: (((*v1/v2)/v3)/v4)…
Non-numbers are 0. Returns the result. Flags an error if divide-by-zero.

**(/** *value value**)**