COWS Language

Formal Specification

COWS Version 1.3

Sean Luke

March 20, 1994

COWS is not a big language. It is, in fact, a *tiny* language. This formal specification describes practically all of it. The COWS language is not intended to become the ultimate macro language for NeXTSTEP (though I

suppose it's possible). Instead, it's a simple, easy-to-parse language that still supports full recursion and functional style, and is good to provide at least the lowest common denominator (I feel) for macro languages in NeXTSTEP.

COWS is *not* object-oriented. It is procedural. This is intentional:

- OOP would add greatly to the size of the language. COWS needs to be small.
- COWS has only one data type. OOP demands multiple types.
- OOP has a relatively steep learning curve compared to functional style, and COWS is intended for users to be able to program easily.
- I've found that integration into NeXTSTEP mostly means making method calls into an app's API, or executing events. These functions don't map as
  well into an OOP macro language as one might think.

Data Format

COWS supports a single data type allocatable on the stack (the string). Numbers are automatically converted into strings, stored as floats printed with *sprintf*. The symbols *t* and *f* have special significance as truths (the return value of functions like = or >, for example), but are likewise converted into strings. Strings, other than truths or numbers, are placed in "quotes".

Comments are placed in [brackets], and are always recognized as comments except when inside quotes. Comments are considered white space, so *hello[there]you* is interpreted as *hello you* and not *helloyou*. General flow syntax is LISP-like, so you will see functions like *(print (+ 1 2 (- 4 a)))*. Unlike

LISP, COWS does not use lists as its data type and therefore makes no use of the quote ' or LISP macros.

Functions

Like HyperTalk, COWS is event-driven. There is no main loop in COWS, just functions and global variables. Any function may be called by any other function, or by an outside entity (which starts COWS programs).

COWS functions can be either defined in COWS code (like you'd expect), or library functions called by the interpreter. In general, you can determine beforehand the number of arguments for a defined function, but not for a library functionÐit must determine if the number of arguments is wrong and

return an error code. This also means that library functions may have variable numbers of arguments but user-defined functions must have a predefined number...this is pretty much the *only* syntactically difference between library and user functions.

Delimeters

" "	*string delimeters*
[ ]	*comment delimeters*
( )	*standard delimeters*
Whitespace	*(tabs, returns, spaces)*

Keywords

these words are reserved and may not be used for

symbols

function	*defines functions*
variable	*defines local and global variables*
do	*starts a function. Also a function name (an exception)*
set	*sets variables*
if	*performs if-then-else constructs*
for	*performs for-loops*
while	*performs while-loops*

Special Symbols

these words are also reserved

t	*ᵃtrueᵒ*

f	*afalseo*
numbers	*in any format, though stored as floats*

BNF Grammar (non-terminals)

<program> :- <program-list>

<program-list> :- <function-form> <program-list> |
<global-form> <program-list> | NULL

<global-form> :- (variable <variable-list>)

Note: This means you may have several global variables in a single variable declaration. This does not preclude having multiple

declarations, of course!

<function-form> :- (function <function-name> <argument-list>
variable <local-variable-list> do <value-list>)
| (function <function-name> <argument-list>
do <value-list>)

Note: This means you can't declare a variable the same as an argument. If you define a function twice, COWS will happily redefine the global.

Note that the function form may change soon to a more
Lisp-like but less verbose syntax:
(function <function-name>
<argument-list> variable
<variable-list> <value>) or

something like that. This would eliminate the begin *keyword entirely.*

<special-form> :- <set-form> | <if-form> | <for-form> | <while-form>

<set-form> :- (set <variable-name> <value>)

<if-form> :- (if <value> <then-value> <else-value>) |
(if <value> <then-value>)

Note: then, else values are just values with the special purposes of then and else functions

<for-form> :- (for <variable-name> <start-value> <stop-value>
<step-value> <value>)

Note: start, stop, step values are just values with the special purposes of start, stop, and step in for

<while-form> :- (while <value> <do-value>) | (while <value>)
Note: while evaluates <value>. If this evaluates to t, then while evaluates <do-value> if one exists, then repeats.

<function-name> :- <symbol>

<argument-list> :- <argument> <argument-list> | NULL

<variable-list> :- <variable-name> <variable-list> | NULL

<local-variable-list> :- <variable-name> <local-variable-list> | NULL

<value-list> :- <value> <value-list> | NULL

<argument> :- <symbol>

<variable-name> :- <symbol>

<value> :- (<special-form>) | (<function-call>) | <variable-name> | <number> | <truth> | <string>

<function-call> :- (<function-name> <value-list>)

Note: value-list is the argument list to the function. Note that this means functions should be able to handle (or ignore) variable-sized argument lists from 0 up.

Terminals

SYMBOLS: non-numeric, strings without quotes, like george, <, a34, heck!

symbols also must not be t or f

NUMBERS: numeric strings without quotes, like 1.234e5, 6, -100

STRINGS: strings in quotes, like "hello", "<", "12.34", "no way!"

TRUTHS: the letters t and f without quotes (though "t" and "f" are also
generally considered truths).
NULL: Nothing.

Symbol Lookup

Function Names: Searched in the text function dictionary. If the name is not there, then searched in the library function dictionary.
Variable Names: Searched in the topmost variable dictionary in the stack. If the name is not there, then the global dictionary is searched.

Startup Procedure

To start up a program, the program is read and its global variables and function text are broken up and stuck in respective dictionaries. Then an application is free to call any function or set/read any global variable. If the program is changed, it must be reset and re-read.