World/Cortex Alpha Release

by John Niclasen
Started: 13-Jul-2011

February 18, 2016

Contents

1 Introduction

Words
Objectivity
Relations
Language
Datatypes

The World Programming Language is a stream of data characterized by datatypes and the evaluation of those. First the datatypes are recognized in the lexical analyser, then they're associated with values. Some values are given directly like numbers, strings, dates, URLs, e-mail adresses, etc., others need to be looked up using words. Words can represent variables holding any value. Words also represent operators and functions. Operators and functions are the basis of computations.

Datatypes come in three categories:

- **Atomic** datatypes
  Can't be split into smaller parts.
  Examples: integer!, real!, logic!, char!, word!, ...
- **Component** datatypes
  Have a well defined number of components.
  Examples: complex!, pair!, date!, time!, ...
- **Series** datatypes
  Have zero, one of more components.
  Examples: string!, file!, block!, binary!, image!, ...

Values cause two types of evaluation:

- **Non-computing** evaluation
  Values, that are just data.
  Examples: integer!, string!, word!, block!, ...
- **Computing** evaluation
  These will cause some further computing.
  Examples: Words representing an operator!, a function!, ...

Operators take precedence over function calls. This can be overruled using parentheses. Operators are infix and always take two arguments. A sequence of operators (with values in between) are computed from left to right. Function calls are always prefix followed by zero, one or more arguments.

World has no keywords. Operators and functions are represented by words, and they can be redefined at will.

2 Datatypes

2.1 Hierarchy

Hierarchy of datatypes and typesets:

```
any-type!
    none!
    unset!
    logic!
    scalar!
        number!
            integer!
            real!
            percent!
        complex!
        char!
        pair!
        range!
        tuple!
        vector!
        time!
    date!
    image!
    series!
        any-string!
            string!
            binary!
            file!
            email!
            url!
            tag!
        any-block!
            block!
            any-paren!
                paren!
            any-path!
                path!
                set-path!
                get-path!
                lit-path!
        list!
    lit-string!
    map!
    datatype!
    typeset!
    bitset!
    any-word!
        word!
        set-word!
        get-word!
        lit-word!
        issue!
        refinement!
    any-function!
        operator!
        function!
        routine!
        callback!
        task!
    any-object!
        context!
        error!
        port!
    task-id!
    node!
    handle!
    struct!
    library!
    comment!
    KWATZ!
```

2.2 unset!

2.3 none!

2.4 logic!

2.5 integer!

Integers are 64-bit. The apostrophe character, `'`, can be used anywhere in integers beyond the first position to separate digits. A plus, `+`, or minus, `-`, can be prefixed to indicate sign. Leading zeros are ignored.

Spec	Integer	Hex
Lowest value	-9'223'372'036'854'775'808	`#{8000 0000 0000 0000}`
Highest value	9'223'372'036'854'775'807	`#{7fff ffff ffff ffff}`

Examples:

```
0
1
-0716
+42
86'400
```

2.6 real!

Reals are 64-bit double precision floating point numbers. They comply with the IEEE 754 standard. The apostrophe character, `'`, can be used anywhere in reals beyond the first position to separate digits. A plus, `+`, or minus, `-`, can be prefixed to indicate sign. Period, `.`, or comma, `,`, can be used to indicate decimal point. Scientific notation (using `e` or `E`) can be used. Leading zeros are ignored. Trailing zeros after the decimal point are ignored.

Spec	Real	Hex
Lowest value	-1.7976931348623157e308	`#{ffef ffff ffff ffff}`
Smallest negative	-5e-324	`#{8000 0000 0000 0001}`
Smallest positive	5e-324	`#{0000 0000 0000 0001}`
Highest value	1.7976931348623157e308	`#{7fef ffff ffff ffff}`
Infinite	inf	`#{7FF0 0000 0000 0000}`
- Infinite	-inf	`#{FFF0 0000 0000 0000}`
Not a Number	nan	`#{FFF8 0000 0000 0000}`

When reals are shown, it is with the number of digits known (not including trailing zeros). This mean, the uncertainty on the last digit shown is always less than one. The one exception is the smallest number, `5e-324`, where the uncertainty also is `5e-324`.

Examples:

```
w> 0.
== 0.0
w> .1
== 0.1
w> tau
== 6.283185307179586
w> e: 2.718281828459045
== 2.718281828459045
w> sqrt 2
== 1.414213562373095
w> 2 ** 53
== 9.007199254740992e+15
w> -1.12321e-320
== -1.123e-320
w> 1 / 0
== inf
w> -1 / 0
== -inf
w> 0 / 0
== nan
```

2.7 percent!

2.8 complex!

2.9 char!

If a char is constructed as an escape sequence starting with a caret, `^`, the result is mapped depending on what follows the caret:

ASCII Code	Character	Maps to	Definition
33	`#"^!"`	30	control code
45	`#"^-"`	9	tab
47	`#"^/"`	10	newline
64 - 93	`#"^@"` - `#"^]"`	0 - 29	control codes
95	`#"^_"`	31	control code
97 - 122	`#"^a"` - `#"^z"`	1 - 26	control codes
126	`#"^~"`	127	del

For the rest, the caret has no effect. Notice how to specify these special characters:

Character	Definition
`#"^^"`	caret character
`#"^""`	quotation mark

Characters can also be specified using hex form:

```
#"^(00)" - #"^(FF)"
#"^(00)" - #"^(ff)"
```

2.10 pair!

2.11 range!

2.12 tuple!

2.13 vector!

A vector is a fixed-size array of C-like datatypes.

Specification

```
vector-name: make vector! [
    C-datatype size
[
    Block of initial values
]]
```

The block of initial values is optional. The C-datatype can be one of:

Argument type	Description
integer!	Sigend 64-bit integer
real!	64-bit floating point
uint8	Unsigned 8-bit integer
sint8	Signed 8-bit integer
uint16	Unsigned 16-bit integer
sint16	Signed 16-bit integer
uint32	Unsigned 32-bit integer
sint32	Signed 32-bit integer
uint64	Unsigned 64-bit integer
sint64	Signed 64-bit integer
float	32-bit floating point
double	64-bit floating point
uchar	Unsigned char
schar	Signed char
ushort	Unsigned short
sshort	Signed short
uint	Unsigned integer
sint	Signed integer
ulong	Unsigned long
slong	Signed long
pointer	A pointer

Creation

Example:

```
my-vector: make vector! [uint16 4 [1 2 3 4]]
```

Vectors can (beside using **make**) also be created with **to** and coerced from a C pointer with **as**. This is useful when interfacing with code written in other languages, for example via callbacks. Examples:

```
my-vector1: to [vector! float] #{0000803f00000040}
my-vector2: as [vector! sint32 16] my-struct/c-array-pointer
        ; array holds 16 x 32-bit data
```

Type and size

A vector's type and size can be found with

```
pick my-vector 'type
pick my-vector 'size
```

2.14 time!

2.15 date!

2.16 image!

2.17 string!

Character	**Definition**
`^^`	caret character
`^"`	quotation mark
`^{`	opening brace (curly begin)
`^}`	closing brace (curly end)

2.18 binary!

Example:

```
w> #{0102}
== #{0102}
```

2.19 file!

2.20 email!

2.21 url!

2.22 tag!

2.23 block!

2.24 paren!

2.25 path!

2.26 set-path!

2.27 get-path!

2.28 lit-path!

2.29 list!

2.30 lit-string!

2.31 map!

2.32 datatype!

2.33 typeset!

2.34 bitset!

2.35 word!

2.36 set-word!

2.37 get-word!

2.38 lit-word!

2.39 issue!

2.40 refinement!

2.41 operator!

2.42 function!

2.43 routine!

Interface Specification

```
routine-name: make routine! [
    "routine description"
    [special attributes]
    library "routine-name" [
        argument1 [arg1-world-type] arg1-type
        "argument1 description"
        argument2 [arg2-world-type] arg2-type
        "argument2 description"
        ...
    ]
    return-type return-world-type
]
```

Special attributes can be:

typecheck Will check and eventually convert datatypes

The following fields are optional:

- Routine description (string!)
- Special attributes (block!)
- Argument block, in case there are no arguments (block!)
- Argument names (argument1 and argument2 in the above) (word!)
- Argument World types (datatype! or word!), if typecheck isn't specified
- Argument description (string!)
- Return World type (datatype! or word!)

Typical combinations of World types and argument types:

World type	**Argument type**	**Description**
integer!	uint8	Unsigned 8-bit integer
integer!	sint8	Signed 8-bit integer
integer!	uint16	Unsigned 16-bit integer
integer!	sint16	Signed 16-bit integer
integer!	uint32	Unsigned 32-bit integer
integer!	sint32	Signed 32-bit integer
integer!	uint64	Unsigned 64-bit integer
integer!	sint64	Signed 64-bit integer
real!	float	32-bit floating point
real!	double	64-bit floating point
char!	uchar	Unsigned char
char!	schar	Signed char
integer!	ushort	Unsigned short
integer!	sshort	Signed short
integer!	uint	Unsigned integer
integer!	sint	Signed integer
integer!	ulong	Unsigned long
integer!	slong	Signed long
string!	pointer	A string
binary!	pointer	A binary
struct!	pointer	A structure as in C
handle!	pointer	A handle

When argument type is pointer, World type can be string!, binary!, struct! or handle!. When calling the routine, a pointer argument can also be of type none!, which is treated like the NULL pointer in C. Some routines take a pointer to a handle as argument, and the routine will then update the handle. To achieve this, argument type should be set to

pointer-adr

Next when the handle's value is used in new routines, argument type should be pointer.

*Remember to set **typecheck**, when using handles.*

2.44 callback!

Callback Specification

A callback is specified by calling **make** with the callback! datatype and a block. Within the block is a specification block and a body block.

```
callback-name: make callback! [[
    "callback description"
    args [
        "struct description"
        C-datatype argument1 "argument1 description"
        C-datatype argument2 "argument2 description"
        ...
    ]
    /local
    local1 "local1 description"
    ...
][
    ...
]]
```

All values in the specification block are optional.

2.45 task!

Task Specification

A task is specified by calling **make** with the task! datatype and a block. Within the block is a specification block and a body block.

```
task-name: make task! [[
    "task description"
    name: task-name
    argument1 [optional-type]
    "argument1 description"
    argument2 [optional-type]
    "argument2 description"
    ...
    /refinement
    "refinement description"
    refinement-argument1 [optional-type]
    "refinement-argument1 description"
    ...
][
    ...
]]
```

All values in the specification block are optional. If *task-name* is set within the specification block using *name:* (set-word!), this *task-name* will be shown in task lists using the **tasks** function.

A task is spawned by calling it. Calling the same task several times will spawn several separate tasks.

Yield example

This is a simple ping-pong example, where each task yields by calling **wait 0** to let other tasks run:

```
ping: task [
    "Ping"
][
    while [true] [
        prin "ping "
        wait 0
    ]
]

pong: task [
    "Pong"
][
    while [true] [
        prin "pong "
        wait 0
    ]
]

ping
pong
```

Use <Ctrl>-C to stop World. Tasks can also be killed:

```
id1: ping
id2: pong
kill id1
kill id2
```

Preemptive example

By not using **wait**, the tasks will be interrupted after a number of instructions. This example prints "ping" and "pong" 100 times. A task kills itself, when it reaches end of task or an **exit**:

```
ping: task [
    "Ping"
][
    loop 100 [
        prin "ping "
    ]
]

pong: task [
    "Pong"
][
    loop 100 [
        prin "pong "
    ]
]

ping
pong
```

It is seen, that each task prints several times before being interrupted. Number of instructions can be set by the /tick refinement in the **tasks** function:

```
tasks/tick 200
```

Setting ticks to zero till turn off preemptive multitasking, and each task will run as long as it can, until it reaches a wait or ends.

Message example

In this example, simple messages are sent between the two tasks to let them know, they can run. The messages are a simple **true** value. Any value can be sent as a message.

```
ping: task [
][
    loop 10 [
        wait 'message
        receive
        prin "ping "
        send id2 true
    ]
]
pong: task [
][
    loop 10 [
        send id1 true
        wait 'message
        receive
        prin "pong "
    ]
]

id1: ping
id2: pong
```

The two tasks can use the variables, id1 and id2, which belong to the global context, because they can see the global context. If a task creates new variables, they are local to that task.

A task can also send messages to itself. The main task has id zero, and to send and receive a message can be done as:

```
w> send 0 42
== 0
w> receive
== 42
```

2.46 task-id!

2.47 node!

2.48 context!

2.49 error!

To produce a simple user error:

```
make error! "some text"
```

To produce a pre-defined error, see system/errors. Example:

```
make error! [script invalid-arg "argument"]
```

2.50 port!

2.51 handle!

2.52 struct!

Some routines in external libraries take C structures as arguments. In World such structures can be defined with the struct! datatype.

Specification

```
struct-name: make struct! [[
    "struct description"
    C-datatype argument1 "argument1 description"
    C-datatype argument2 "argument2 description"
    ...
] [
    Block of initial values
]]
```

Instead of the block of initial values, **none** can be specified, which will set the memory area to zeros.

The C datatype and argument can be given in reverse order. Example:

```
float1: make struct! [[
    float f
] none]

float2: make struct! [[
    f float
] none]
```

The following fields are optional:

- Struct description (string!)
- Argument description (string!)

The C datatype can be the same as the argument type for routines: uint8, sint8, uint16, sint16, uint32, sint32, uint64, sint64, float, double, uchar, schar, ushort, sshort, uint, sint, ulong, slong and pointer.

To set a pointer to the NULL value, **none** can be specified in the block of initial values.

2.53 library!

Libraries are loaded with

```
lib: load/library %lib-file
```

Libraries don't need to be freed. When there are no more references to the library, it's being freed.

Mac OS X examples

```
w> libc: load/library %/usr/lib/libc.dylib
w> puts: make routine! [libc "puts" [[string!] pointer] sint
    integer!]
w> puts "Hello, World!"
Hello, World!
== 10

w> tanh: make routine! [libc "tanh" [[real!] double] double
    real!]
w> tanh 1.5
== 0.905148253644866
```

Windows examples

```
w> msvcrt: load/library %msvcrt.dll
w> puts: make routine! [msvcrt "puts" [[string!] pointer]
    sint integer!]
w> puts "Hello, World!"
Hello, World!
== 0

w> tanh: make routine! [msvcrt "tanh" [[real!] double] double
     real!]
w> tanh 1.5
== 0.905148253644866
```

2.54 comment!

Comments are recognized by the following four syntaxes:

```
; single-line comment to the end of the line

\word-comment

\"string comment"

\{long string comment
across one or more lines}
```

Comments are ignored by the lexical analysis, so they are stripped from the code. There is one way to produce comment values to be used in e.g. dialects:

```
w> block: make block! "some values ; and a comment"
== [some values ; and a comment]
w> length? block
== 3
w> type? block/3
== comment!
```

Any of the four syntaxes can be utilized to produce comment values:

```
w> make block! {\word \"string" \{long string} ; single-line}
== [\word \"string" \{long string} ; single-line]
```

2.55 KWATZ!

3 Expressions

3.1 Arithmetic operators

Addition (+)

Subtraction (-)

Multiplication (*)

Division (/)

Modulo (//)

The modulo operator, `//`, is defined as:

$$b \mathbin{//} m = b - \left(m \times floor\left(\frac{b}{m}\right)\right)$$

, where the $floor(x)$ function gives the largest integer not greater than x. $floor\,(b/m)$ is also known as *floor division*.

The result of the modulo operation is called the *remainder*.

3.2 Unary minus

3.3 Relational operators

Equal (=)

Strict equal (==)

Same (=?)

Not equal (<>)

Greater (>)

Greater or equal (>=)

Lesser (<)

Lesser or equal (<=)

3.4 Logical operators

and

or

xor

3.5 Math operators

Power (**)

4 Values

4.1 false

```
false: make logic! 0
```

4.2 none

```
none: make none! 0
```

4.3 true

```
true: make logic! 1
```

5 Natives

Natives are built-in functions, where the source isn't available as World source. Words representing natives may be given other values (be redefined). Natives are referred to as functions, as they work exactly like functions.

5.1 Arithmetic

add

subtract

multiply

divide

mod

5.2 Unary minus

Unary minus is a dash, `-`, following immediately after one of these:

- unset!. This also include beginning of a script and beginning of input from the prompt.
- set-word!
- native!
- operator!
- A function!, that takes at least one argument.
- The beginning of a parenthesis.
- The beginning of a block being reduced or evaluated.

Unary minus behaves like **negate**.
Examples of the above seven situations:

```
- tau
a: - 42
print - e
a * - b
my-func - a b c
(- x + y)
do [- x + y]
```

The following all give the same result:

```
e ** - x ** 2
e ** negate x ** 2
e ** -(x ** 2)
e ** (-(x ** 2))
```

5.3 Math

abs

complement

cos

ln

negate

power

rotate

shift

sin

tan

5.4 Context

bind

get

set

value?

5.5 Control

all

any

break

do

either

exit

halt

if

quit

reduce

return

try

while

5.6 Datatype

as

make

to

type?

5.7 Help

comment

trace

5.8 Logic

not

5.9 Port, File and I/O

close

load

open

prin

print

query

read

receive

send

wait

write

5.10 Series

append

back

back'

clear

copy

find

find'

head

head'

index?

insert

length?

newline?

next

next'

pick

poke

remove

select

set-newline

skip

skip'

tail

tail'

5.11 Strings

mold

5.12 System

call

compile

compiled?

disasm

free

now

recycle

retain

stats

tasks

6 Cortex extension

6.1 Values

e

The mathematical constant e, also known as *Euler's number*

e 2.718281828459045

off

```
off: make logic! 0
```

on

```
on: make logic! 1
```

pi

The mathematical constant π

pi 3.141592653589793

tau

The mathematical constant τ equals 2π

tau 6.283185307179586

6.2 Comparison

same?

equal?

strict-equal?

not-equal?

greater?

lesser?

greater-or-equal?

lesser-or-equal?

6.3 Context

context

node

6.4 Control

actor

compose

does

for

forall

foreach

func

has

loop

native

native-op

operator

q

```
q: make function! reduce [pick :quit 1 pick :quit 2]
```

repeat

switch

until

vector

6.5 Datatype

any-block?

any-function?

any-object?

any-paren?

any-path?

any-string?

any-type?

any-word?

as-pair

binary?

bitset?

block?

callback?

char?

comment?

complex?

context?

datatype?

date?

email?

error?

file?

function?

get-path?

get-word?

image?

integer?

issue?

KWATZ?

library?

list?

lit-path?

lit-string?

lit-word?

logic?

map?

node?

none?

number?

operator?

pair?

paren?

path?

percent?

port?

range?

real?

refinement?

routine?

scalar?

series?

set-path?

set-word?

string?

tag?

task?

task-id?

time?

tuple?

typeset?

unset?

url?

vector?

word?

6.6 Help

?

*See **help**.*

help

license

probe

source

6.7 Logic

and'

or'

xor'

6.8 Math

arccos

arcsin

arctan

arg

cosh

deg

exp

log

max

min

random

sinh

sqrt

tanh

to-deg

zero?

6.9 Port, File and I/O

input

save

to-world-file

6.10 Series

after

before

change

empty?

first

from

head?

join

more?

of

parse

reverse

second

sort

tail?

third

6.11 Sets

bitset

6.12 Strings

debase

dehex

detab

enbase

form

lowercase

replace

trim

uppercase

6.13 System

free-all

include

retain-all

.

The World's smallest Hello world program. Example:

```
.
```

The function will print the text "Hello, World!".