World/Cortex Alpha Release

by John Niclasen
Started: 13-Jul-2011

December 9, 2011

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!
    unset!
    none!
    logic!
    scalar!
        number!
            integer!
            real!
            complex!
            percent!
        char!
        range!
        tuple!
        time!
    date!
    series!
        any-string!
            string!
            binary!
            file!
            url!
            tag!
            issue!
        any-block!
            block!
            any-paren!
                paren!
            any-path!
                path!
                set-path!
                get-path!
                lit-path!
    datatype!
    typeset!
    bitset!
    any-word!
        word!
        set-word!
        get-word!
        lit-word!
        refinement!
    any-function!
        operator!
        function!
        routine!
    any-object!
        context!
        error!
```

```
    port!
 handle!
 library!
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.

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:

Spec	Real	Hex
Lowest value	-1.797'693'134'862'315'7e308	`#{ffef ffff ffff ffff}`
Smallest negative	-5e-324	`#{8000 0000 0000 0001}`
Smallest positive	5e-324	`#{0000 0000 0000 0001}`
Highest value	1.797'693'134'862'315'7e308	`#{7fef ffff ffff ffff}`

```
w> 0.
== 0.0
w> .1
== 0.1
w> pi
== 3.141592653589793
w> e: 2.718'281'828'459'045
== 2.718281828459045
w> sqrt 2
== 1.414213562373095
w> 2 ** 53
== 9.007199254740992e+15
w> -1.12321e-320
== -1.123e-320
```

2.7 complex!

2.8 percent!

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:

Characters can also be specified using hex form:

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

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

2.10 range!

2.11 tuple!

2.12 time!

2.13 date!

2.14 string!

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

2.15 binary!

Example:

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

2.36 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

When argument type is pointer, World type can be string!, binary! or handle!. 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.*

2.37 context!

2.38 error!

2.39 port!

2.40 handle!

2.41 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
```

2.42 KWATZ!

3 Expressions

3.1 Arithmetic operators

Addition (+)

Subtraction (-)

Multiplication (*)

Division (/)

Modulo (//)

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

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

, 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:

```
- pi
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

cos

ln

power

sin

tan

5.4 Context

get

set

value?

5.5 Control

all

any

do

either

exit

if

quit

reduce

return

while

5.6 Datatype

as

make

to

type?

5.7 Help

comment

trace

5.8 Port, File and I/O

close

load

open

prin

print

read

wait

write

5.9 Series

back

copy

find

index?

insert

length?

newline?

next

pick

poke

remove

select

set-newline

skip

5.10 Strings

mold

5.11 System

call

compile

compiled?

disasm

now

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

6.2 Comparison

same?

equal?

strict-equal?

not-equal?

greater?

lesser?

greater-or-equal?

lesser-or-equal?

6.3 Context

context

6.4 Control

does

for

foreach

func

has

loop

native

native-op

operator

q

```
q:    :quit
```

---

repeat

switch

until

6.5 Datatype

any-block?

any-function?

any-paren?

any-path?

any-string?

any-type?

any-word?

binary?

block?

char?

complex?

context?

datatype?

date?

file?

function?

get-path?

get-word?

KWATZ?

integer?

library?

lit-path?

lit-word?

logic?

none?

number?

operator?

paren?

path?

percent?

real?

refinement?

routine?

scalar?

series?

set-path?

set-word?

string?

time?

typeset?

word?

6.6 Help

?

See ***help****.*

help

license

probe

source

6.7 Logic

and'

or'

xor'

6.8 Math

arccos

arcsin

arctan

arg

cosh

exp

log

max

min

negate

not

random

sinh

sqrt

tanh

zero?

6.9 Port, File and IO

save

to-world-file

6.10 Series

after

before

empty?

first

from

head

head?

join

parse

reverse

second

sort

tail

tail?

third

6.11 Strings

lowercase

trim

uppercase

6.12 System

.

The World's smallest Hello world program. Example:

```
.
```

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