The Coral Language Specification

Version 0.1

Kateřina Nikola Lisová

April 11, 2014

Prague, Czech Republic

Contents

Lexical Syntax 5

1.1 Identifiers 5

1.2 Keywords 6

1.3 Newline Characters 8

1.4 Operators 8

1.5 Literals 8

1.5.1 Integer Literals 8

1.5.2 Floating Point Literals 8

1.5.3 Boolean Literals 8

1.5.4 String Literals 8

1.5.5 Symbol Literals 8

1.5.6 Regular Expression Literals 8

1.5.7 Collection Literals 8

1.6 Whitespace & Comments 8

1.7 Preprocessor Macros 8

Identifiers, Names & Scopes 8

Types 8

3.1 Paths 9

3.2 Value Types 9

3.2.1 Type Projection 9

3.2.2 Type Designators 9

3.2.3 Parameterized Types 9

3.2.4 Tuple Types 9

3.2.5 Annotated Types 9

3.2.6 Compound Types 9

3.2.7 Function Types 9

3.3 Non-Value Types 9

3.3.1 Method Types 9

3.3.2 Polymorphic Method Types 9

3.3.3 Type Constructors 9

3.4 Relations Between Types 9

Basic Declarations & Definitions 9

4.1 9

Preface

Coral is a Ruby-like programming language which enhances advanced object-oriented programming with elements of functional programming. Every value is an object, in this sense it is a pure object-oriented language. Object blueprints are described by classes. Classes can be composed in multiple ways – classic inheritance, mixin composition, union and fusion types.

Coral is also a functional language in the sense that every function is also an object. Therefore, function definitions can be nested and higher-order functions are supported out-of-the-box. Coral also has a limited support for pattern matching, which can emulate the algebraic types used in other functional languages.

Coral has been developed from 2012 in a home environment out of pure enthusiasm for programming and out of a desire for a truly versatile language. This document is a work in progress and will stay that way forever. It acts as a reference for the language definition and some core library classes.

# 

Lexical Syntax

Coral programs are written using the Unicode character set; Unicode supplementary characters are supported as well. Coral programs are preferably encoded with the UTF-8 character encoding. While every Unicode character is supported, usage of Unicode escapes is encouraged, since fonts that IDEs might use may not support the full Unicode character set.

## Identifiers

**Syntax:**

simple\_id ::= lower [ id\_rest ]

variable\_id ::= simple\_id | '\_'

constant\_id ::= upper [ id\_rest ]

function\_id ::= simple\_id [ id\_rest\_ext ]

id\_rest ::= { letter | digit | '\_' }

id\_rest\_mid ::= id\_rest [ ( '/' | '+' | '-' ) id\_rest ]

id\_rest\_ext ::= id\_rest [ id\_rest\_mid ] [ '?' | '!' ]

There are three kinds of identifiers.

First, *variable identifiers*, which are simply a lower-case letter followed by arbitrary sequence of letters (any-case), digits and underscores, or just one underscore (which has special meaning).

Second, *constant identifiers*, which are just like variable identifiers, but starting with an upper-case letter and never just an underscore.

And third, *function identifiers*, which are the most complicated ones. They can start as a variable identifier, then optionally followed by one of “/”, “+” and “-”, and then optionally ended with “?” or “!”.

Coral programs are parsed greedily, so that a longest match rule applies. Letters from the syntax may be any Unicode letters, but English alphabet letters are recommended, along with English names.

## Keywords

A set of identifiers is reserved for language features instead of for user identifiers. However, unlike in most other languages, keywords are not being recognized inside of paths, except for a few specific cases.

The following names are the reserved words.

alias

annotation

as

begin

bitfield

break

case

cast

catch

class

clone

constant

constructor

declare

def

destructor

do

else

elsif

end

ensure

enum

for

function

fusion

goto

if

implements

in

include

interface

is

let

loop

match

memoize

message

method

mixin

module

native

next

nil

no

of

opaque

operator

out

property

protocol

raise

range

record

redo

rescue

retry

return

self

skip

struct

super

template

test

then

this

throw

throws

transparent

type

undef

unless

until

union

use

var

void

yes

when

while

yield

Not every reserved word is a keyword in every context, this behavior will be further explained. For example, the bitfield reserved word is only recognized as a keyword inside an enumeration definition context, in a specific place. Every reserved word may be used as a function identifier, with a little work-around when used with an implicit receiver.

## Newline Characters

Syntax:

separator ::= nl { nl } | ';'

Coral is a line-oriented language, in which statements are expressions and may be terminated by newlines, as well as by semi-colon operator. A newline in a Coral source file is treated as the special separator token nl if the following criterion is satisfied:

1. The token immediately preceding the newline can terminate an expression.

Since Coral may be interpreted in a REPL fashion, there are no other suitable criteria. Such a token that can terminate an expression is, for instance, not a binary operator or a message sending operator, which both require further tokens to create an expression. Keywords that expect any following tokens also can not terminate expressions. Coral interpreters and compilers do not look-ahead beyond newlines.

If the token immediately preceding the newline can not terminate an expression and is followed by more than one newline, Coral still sees that as only a one significant newline, to prevent any confusion.

Keywords that can terminate an expression are: break, end, opaque, native, next, nil, no, redo, retry, return, self, skip, super, this, transparent, void, yes, yield.

## Operators

A set of identifiers is reserved for language features, some of which may be overridden by user space implementations. Operators have language-defined precedence rules that usually comply to user expectations, and may be easily restructured by putting expressions inside of parenthesis pairs.

The following character sequences are the operators recognized by Coral.

:=

+=

-=

\*=

\*\*=

/=

%=

||=

&&=

^^=

|=

&=

^=

~=

<<

>>

<<<

>>>

;

=

!=

==

!==

===

!===

=~

!~

<>

<

>

<=

>=

<=>

+

-

\*

\*\*

/

div

%

mod

||

or

&&

and

!

not

^^

xor

|

&

^

~

..

...

,

->

<-

=>

::

:

<<|

|>>

<|

|>

(

)

<

>

{

}

.

## 

Some of these operators have several different meanings, usually up to two. Some are binary, some are unary, none is ternary.

## Literals

There are literals for numbers (including integer, floating point and complex), characters, booleans, strings, symbols, regular expressions and collections (including tuples, lists, dictionaries and bags).

**Syntax:**

literal ::= integer\_literal

| floating\_point\_literal

| complex\_literal

| character\_literal

| string\_literal

| symbol\_literal

| regular\_expression\_literal

| collection\_literal

| '**nil**'

### Integer Literals

**Syntax:**

integer\_literal ::= [ '+' | '-' ] ( decimal\_numeral

| hexadecimal\_numeral

| octal\_numeral

| binary\_numeral )

decimal\_numeral ::= '0' | non\_zero\_digit { [ '\_' ] digit }

hexadecimal\_numeral ::= '0x' hex\_digit { [ '\_' ] hex\_digit }

digit ::= '0' | non\_zero\_digit

non\_zero\_digit ::= '1' | … | '9'

hex\_digit ::= '1' | … | '9' | 'a' | … | 'f'

octal\_numeral ::= '0' oct\_digit { [ '\_' ] oct\_digit }

oct\_digit ::= '0' | … | '7'

binary\_numeral ::= '0b' bin\_digit { [ '\_' ] bin\_digit }

bin\_digit ::= '0' | '1'

Integers are usually of type Number, which is a class cluster of all classes that can hold numbers. Unlike Java, Coral supports both signed and unsigned integers. Usually integer literals that are obviously unsigned integers result in being represented internally by a class that stores the integer unsigned, something like Integer\_64\_Unsigned. Math operations on numbers are handled internally in such a way that the user does not need to care about the actual types of the numbers – when an integer overflow would occur, the result is stored in a larger container type, and when the largest container type would be overflown, a decimal type is used automatically.

Underscores used in integer literals have no special meaning but to improve readability of larger literals, i.e., to separate thousands.

Integral members of the Number class cluster include the following container types.

1. Integer\_8 (-27 to 27-1)
2. Integer\_8\_Unsigned (0 to 28)
3. Integer\_16 (-215 to 215-1)
4. Integer\_16\_Unsigned (0 to 216)
5. Integer\_32 (-231 to 231-1)
6. Integer\_32\_Unsigned (0 to 232)
7. Integer\_64 (-263 to 263-1)
8. Integer\_64\_Unsigned (0 to 264)
9. Integer\_128 (-2127 to 2127-1)
10. Integer\_128\_Unsigned (0 to 2128)
11. Decimal (-∞ to ∞)
12. Decimal\_Unsigned (0 to ∞)

The special Decimal and Decimal\_Unsigned container types are also for storing arbitrary precision floating point numbers. All the container types are constants defined in the Number class. (Number::*Container\_Type*)

Aliases for these classes exist as follows:

1. Integer\_8 = Byte
2. Integer\_8\_Unsigned = Byte\_Unsigned
3. Integer\_16 = Short
4. Integer\_16\_Unsigned = Short\_Unsigned
5. Integer\_64 = Long
6. Integer\_64\_Unsigned = Long\_Unsigned
7. Integer\_128 = Double\_Long
8. Integer\_128\_Unsigned = Double\_Long\_Unsigned

Moreover, a helper type Number::Unsigned exists, which can be used for type casting in cases where an originally signed number needs to be treated as unsigned.

These inner classes of Number also have one important feature: if the actual number fits into the range of possible values of another inner class, then it will also pass when performing a type check on that other class.

Notice that some host platforms may not provide a native implementation of 128 bit values, thus in such a case, the class actually becomes an alias of Decimal or Decimal\_Unsigned respectively, or just has a different internal implementation. Users should not rely on these internal classes and rather use range types when constraining value ranges. As helper types for the range types, Number::Integer and Number::Integer\_Unsigned exist, to allow easy constraining of the range types to integral numbers while being machine-size-independent.

Also notice that Coral does not have a Char or Character number literal, since that is not considered to be a fixed-size number by Coral, with respect to Unicode standards being of variable byte length (i.e., UTF-8 characters are usually between 1 to 4 bytes long). This neatly solves another Java issue with their Characters being in UTF-16 with higher and lower surrogates being split into two successive Character instances (or primitive values!).

### Floating Point Literals

**Syntax:**

float\_literal ::= digit { [ '\_' ] digit } '.' digit { [ '\_' ] digit }

[ exponent\_part ] [ float\_type ]

| digit { [ '\_' ] digit } exponent\_part [ float\_type ]

| digit { [ '\_' ] digit } [ exponent\_part ] float\_type

exponent\_part ::= 'e' [ '+' | '-' ] digit { [ '\_' ] digit }

float\_type ::= 'f' | 'd'

Floating point literals are of type Number as well as integer literals, and have fewer container types. Compiler infers the precision automatically, unless the float\_type part is present.

1. Float\_32 (IEEE 754 32-bit precision)
2. Float\_64 (IEEE 754 64-bit precision)
3. Decimal (-∞ to ∞)
4. Decimal\_Unsigned (0 to ∞)

The aliases for these classes exist as follows:

1. Float = Float\_32
2. Double = Float\_64

The letters in exponent type and float type literals have to be lower-case in Coral sources, but functions for parsing of floating point numbers indeed support them being upper-case for compatibility reasons.

### Boolean Literals

**Syntax:**

boolean\_literal ::= 'yes' | 'no'

Both literals are members of type Boolean. The no literal has also a special behavior when being compared to nil: it equals to nil, while not actually being nil. Identity equality is indeed different.

### String Literals

**Syntax:**

string\_literal ::= simple\_string\_literal | interpolable\_string\_literal

simple\_string\_literal ::= ''' { string\_element } '''

string\_element ::= printable\_char | char\_escape\_seq

interpolable\_string\_literal ::= '"' { int\_string\_element } '"'

int\_string\_element ::= string\_element | interpolated\_expression

interpolated\_expression ::= '#{' expression '}'

String literals are members of the type String. Single quotes in simple string literals have to be escaped (\') and double quotes in interpolable string literals have to be escaped (\").

### Symbol Literals

### Regular Expression Literals

### Collection Literals

## Whitespace & Comments

## Preprocessor Macros

Identifiers, Names & Scopes

Types

## Paths

## Value Types

### Type Projection

### Type Designators

### Parameterized Types

### Tuple Types

### Annotated Types

### Compound Types

### Function Types

## Non-Value Types

### Method Types

### Polymorphic Method Types

### Type Constructors

## Relations Between Types

Basic Declarations & Definitions

## 