

The Coral Language Specification

Kateřina Nikola Lisová

April 29, 2014

Contents

1	Lexical Syntax	3
1.1	Identifiers	4
1.2	Keywords	4
1.3	Newline Characters	5
1.4	Operators	6
1.5	Literals	6
1.5.1	Integer Literals	7
1.5.2	Floating Point Literals	8
1.5.3	Imaginary Number Literals	8
1.5.4	Units of Measure	8
1.5.5	Character Literals	8
1.5.6	Boolean Literals	9
1.5.7	String Literals	9
1.5.8	Symbol Literals	9
1.5.9	Type Parameters	9
1.5.10	Regular Expression Literals	9
1.5.11	Collection Literals	9
1.6	Whitespace & Comments	9
1.7	Preprocessor Macros	9
2	Identifiers, Names & Scopes	11
3	Types	13
3.1	Paths	14
3.2	Value Types	14

3.2.1	Value Type	14
3.2.2	Type Projection	14
3.2.3	Type Designators	14
3.2.4	Parametrized Types	14
3.2.5	Tuple Types	14
3.2.6	Annotated Types	14
3.2.7	Compound Types	14
3.2.8	Function Types	14
3.2.9	Existential Types	14
3.3	Non-Value Types	14
3.3.1	Method Types	14
3.3.2	Polymorphic Method Types	14
3.3.3	Type Constructors	14
3.4	Relations Between Types	14
3.4.1	Type Equivalence	14
3.4.2	Conformance	14
4	Basic Declarations & Definitions	15
4.1	Variable Declarations & Definitions	16
4.2	Property Declarations & Definitions	16
4.3	Instance Variable Definitions	16
4.4	Type Declarations & Aliases	16
4.5	Type Parameters	16
4.6	Variance of Type Parameters	16
4.7	Function Declarations & Definitions	16
4.7.1	Positional Parameters	16
4.7.2	Optional Parameters	16
4.7.3	Repeated Parameters	16
4.7.4	Named Parameters	16
4.7.5	Procedures	16
4.7.6	Method Return Type Inference	16
4.8	Use Clauses	16

5	Classes & Objects	17
5.1	Class Definitions	18
5.1.1	Class Linearization	18
5.1.2	Constructor & Destructor Definitions	18
5.1.3	Class Block	18
5.1.4	Class Members	18
5.1.5	Overriding	18
5.1.6	Inheritance Closure	18
5.1.7	Modifiers	18
5.2	Mixins	18
5.3	Unions	18
5.4	Enums	18
5.5	Compound Types	18
5.6	Range Types	18
5.7	Units of Measure	18
5.8	Record Types	18
5.9	Struct Types	18
5.10	Object Definitions	18
6	Expressions	19
6.1	Expression Typing	20
6.2	Literals	20
6.3	The Nil Value	20
6.4	Designators	20
6.5	Self, This & Super	20
6.6	Function Applications	20
6.6.1	Named and Optional Arguments	20
6.6.2	Input & Output Arguments	20
6.6.3	Function Compositions & Pipelines	20

6.7	Method Values	20
6.8	Type Applications	20
6.9	Tuples	20
6.10	Instance Creation Expressions	20
6.11	Blocks	20
6.12	Prefix & Infix Operations	20
6.12.1	Prefix Operations	20
6.12.2	Infix Operations	20
6.12.3	Assignment Operators	20
6.13	Typed Expressions	20
6.14	Annotated Expressions	20
6.15	Assignments	20
6.16	Conditional Expressions	20
6.17	Loop Expressions	20
6.17.1	Classic For Expressions	20
6.17.2	Iterable For Expressions	20
6.17.3	Basic Loop Expressions	20
6.17.4	While & Until Loop Expressions	20
6.17.5	Conditions in Loop Expressions	20
6.18	Collection Comprehensions	20
6.19	Return Expressions	20
6.19.1	Implicit Return Expressions	20
6.19.2	Explicit Return Expressions	20
6.19.3	Structured Return Expressions	20
6.20	Raise Expressions	20
6.21	Rescue & Ensure Expressions	20
6.22	Throw & Catch Expressions	20
6.23	Anonymous Functions	20
6.24	Conversions	20
6.24.1	Type Casting	20

7	Implicit Parameters & Views	21
8	Pattern Matching	23
8.1	Patterns	23
8.1.1	Variable Patterns	23
8.1.2	Typed Patterns	23
8.1.3	Literal Patterns	23
8.1.4	Constructor Patterns	23
8.1.5	Tuple Patterns	23
8.1.6	Extractor Patterns	23
8.1.7	Pattern Alternatives	23
8.1.8	Regular Expression Patterns	23
8.2	Type Patterns	23
8.3	Pattern Matching Expressions	23
8.4	Pattern Matching Anonymous Functions	23
9	Top-Level Definitions	25
9.1	Compilation Units	25
9.2	Modules	25
9.3	Module Objects	25
9.4	Module References	25
9.5	Top-Level Classes	25
9.6	Programs	25
10	Annotations	27
11	Naming Guidelines	29
12	The Coral Standard Library	31
12.1	Root Classes	31
12.1.1	The Object Class	31

12.1.2 The Nothing Class	31
12.2 Value Classes	31
12.3 Standard Reference Classes	31
A Coral Syntax Summary	33

Preface

Coral is a Ruby-like programming language which enhances advanced object-oriented programming with elements of functional programming. Everything is an object, in this sense it's a pure object-oriented language. Object blueprints are described by classes. Classes can be composed in multiple ways – classic inheritance and/or mixin composition, along with prototype-oriented inheritance.

Coral is also a functional language in the sense that every function is also an object, and generally, everything is a value. 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 since 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.

Some of the languages that had major influence on the development of Coral, including syntax and behavior patterns, are Ruby, Ada, Scala, Java, C#, F# and Clojure. Coral tries to inherit their good parts and put them together in its own way.

The vast majority of Coral's syntax is inspired by *Ruby*. Coral uses keyword program parentheses in Ruby fashion. There is **class ... end**, **def ... end**, **do ... end**, **loop ... end**. Ruby itself is inspired by other languages, so this relation is transitive and Coral is inspired by those languages as well (for example, Ada).

Coral is inspired by *Ada* in the way that user identifiers are formatted: `Some_Constant_Name` and — unlike in Ada, but quite similar to it — `some_method_name`. Also, some control structures are inspired by Ada, such as loops, named loops, return expressions and record types. Pretty much like in Ada, Coral's control structures can be usually ended the same way: **class ... end class** etc.

Scala influenced the type system in Coral. Syntax for existential types comes almost directly from it. However, Coral is a rather dynamically typed language, so the type checks are made eventually in runtime (but some limited type checks can be made during compile time as well). Moreover, the structure of this mere specification is inspired by Scala's specification.

From *F#*, Coral borrows some functional syntax (like function composition) and F# also inspired the feature of Units of Measure.

Clojure inspired Coral in the way functions can get their names. Coral realizes that turning function names into sentences does not always work, so it is pos-

sible to use dashes, plus signs and slashes inside of function names. Therefore, `call/cc` is a legit function identifier. Indeed, binary operators are required to be properly surrounded by whitespace or other non-identifier characters.

Chapter 1

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.

1.1 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_ext ::= id_rest [id_rest_mid] ['?' | '!' | '=']
id_rest_mid ::= id_rest {'/' | '+' | '-'} id_rest

```

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.

1.2 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	for-some	function
goto	if	implements	in	include
interface	is	let	loop	match
memoize	message	method	mixin	module
native	next	nil	no	of

opaque	operator	out	prepend	property
protocol	raise	range	record	redo
refine	rescue	retry	return	self
skip	struct	super	template	test
then	this	throw	transparent	type
undef	unless	until	union	unit-of-measure
use	val	var	void	yes
when	while	with	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.

1.3 Newline Characters

Syntax:

```
semi ::= 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 REPL¹ 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**.

¹Read-Eval-Print Loop

1.4 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 are supposed to usually comply to user expectations (principle of least surprise), and another desired precedence may be obtained by putting expressions with operators 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 multiple meanings, usually up to two. Some are binary, some are unary, none is ternary.

Binary (infix) operators have to be separated by whitespace or non-letter characters on both sides, unary operators on left side – the right side is what they are bound to.

Unary operators are: +, -, &, not, ! and ~. The first three of these are binary as well. The ; operator is used to separate expressions (see Newline Characters). Parentheses are postcircumfix operators. Coral has no postfix operators.

Coral allows for custom user-defined operators, but those have the lowest precedence and need to be parenthesized in order to express any precedence. Such custom operators can't be made of letter characters.

1.5 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'

```

1.5.1 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 represent numbers. Unlike Java, Coral supports both signed and unsigned integers directly. Usually integer literals that are obviously unsigned integers are automatically represented internally by a class that stores the integer unsigned, like `Integer_64_Unsigned`. Math operations on numbers are handled internally in such a way that the user doesn't need to worry about the actual types of the numbers — when an integer overflow would occur, the result is stored in a larger container type.

Underscores used in integer literals have no special meaning, other than 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` (-2^7 to $2^7 - 1$), alias `Byte`

2. `Integer_8_Unsigned` (0 to 2^8), alias `Byte_Unsigned`
3. `Integer_16` (-2^{15} to $2^{15} - 1$), alias `Short`
4. `Integer_16_Unsigned` (0 to 2^{16}), alias `Short_Unsigned`
5. `Integer_32` (-2^{31} to $2^{31} - 1$)
6. `Integer_32_Unsigned` (0 to 2^{32})
7. `Integer_64` (-2^{63} to $2^{63} - 1$), alias `Long`
8. `Integer_64_Unsigned` (0 to 2^{64}), alias `Long_Unsigned`
9. `Integer_128` (-2^{127} to $2^{127} - 1$), alias `Double_Long`
10. `Integer_128_Unsigned` (0 to 2^{128}), alias `Double_Long_Unsigned`
11. `Decimal` ($-\infty$ to ∞)
12. `Decimal_Unsigned` (0 to ∞)

The special `Decimal` & `Decimal_Unsigned` container types are also for storing arbitrary precision floating point numbers. All the container types are constants defined in the `Number` class and can be imported into scope if needed.

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.

Weak conformance applies to the inner members of `Number` class.

For use with range types, `Number::Integer` and `Number::Integer_Unsigned` exist, to allow constraining of the range types to integral numbers.

1.5.2 Floating Point Literals

1.5.3 Imaginary Number Literals

1.5.4 Units of Measure

1.5.5 Character Literals

Syntax:

```
character_literal ::= '%' (character | unicode_escape) ''
```


1.5.6 Boolean Literals

1.5.7 String Literals

1.5.8 Symbol Literals

1.5.9 Type Parameters

1.5.10 Regular Expression Literals

1.5.11 Collection Literals

1.6 Whitespace & Comments

1.7 Preprocessor Macros

Chapter 2

Identifiers, Names & Scopes

Chapter 3

Types

3.1 Paths

3.2 Value Types

3.2.1 Value Type

3.2.2 Type Projection

3.2.3 Type Designators

3.2.4 Parametrized Types

3.2.5 Tuple Types

3.2.6 Annotated Types

3.2.7 Compound Types

3.2.8 Function Types

3.2.9 Existential Types

3.3 Non-Value Types

3.3.1 Method Types

3.3.2 Polymorphic Method Types

3.3.3 Type Constructors

3.4 Relations Between Types

Chapter 4

Basic Declarations & Definitions

4.1 Variable Declarations & Definitions

4.2 Property Declarations & Definitions

4.3 Instance Variable Definitions

4.4 Type Declarations & Aliases

4.5 Type Parameters

4.6 Variance of Type Parameters

4.7 Function Declarations & Definitions

4.7.1 Positional Parameters

4.7.2 Optional Parameters

4.7.3 Repeated Parameters

4.7.4 Named Parameters

4.7.5 Procedures

4.7.6 Method Return Type Inference

4.8 Use Clauses

Chapter 5

Classes & Objects

5.1 Class Definitions

5.1.1 Class Linearization

5.1.2 Constructor & Destructor Definitions

5.1.3 Class Block

5.1.4 Class Members

5.1.5 Overriding

5.1.6 Inheritance Closure

5.1.7 Modifiers

5.2 Mixins

5.3 Unions

5.4 Enums

5.5 Compound Types

5.6 Range Types

5.7 Units of Measure

5.8 Record Types

Chapter 6

Expressions

6.1 Expression Typing

6.2 Literals

6.3 The Nil Value

6.4 Designators

6.5 Self, This & Super

6.6 Function Applications

6.6.1 Named and Optional Arguments

6.6.2 Input & Output Arguments

6.6.3 Function Compositions & Pipelines

6.7 Method Values

6.8 Type Applications

6.9 Tuples

6.10 Instance Creation Expressions

6.11 Blocks

Chapter 7

Implicit Parameters & Views

Chapter 8

Pattern Matching

8.1 Patterns

8.1.1 Variable Patterns

8.1.2 Typed Patterns

8.1.3 Literal Patterns

8.1.4 Constructor Patterns

8.1.5 Tuple Patterns

8.1.6 Extractor Patterns

8.1.7 Pattern Alternatives

8.1.8 Regular Expression Patterns

8.2 Type Patterns

8.3 Pattern Matching Expressions

8.4 Pattern Matching Anonymous Functions

Chapter 9

Top-Level Definitions

9.1 Compilation Units

9.2 Modules

9.3 Module Objects

9.4 Module References

9.5 Top-Level Classes

9.6 Programs

Chapter 10

Annotations

Chapter 11

Naming Guidelines

Chapter 12

The Coral Standard Library

12.1 Root Classes

12.1.1 The Object Class

12.1.2 The Nothing Class

12.2 Value Classes

12.3 Standard Reference Classes

Chapter A

Coral Syntax Summary