The Coral Language Specification

Markéta Nikola Lisová

April 21, 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 6

1.5.2 Floating Point Literals 6

1.5.3 Imaginary Number Literals 6

1.5.4 Units of Measure 6

1.5.5 Boolean Literals 6

1.5.6 String Literals 6

1.5.7 Symbol Literals 6

1.5.8 Type Parameters 6

1.5.9 Regular Expression Literals 6

1.5.10 Collection Literals 6

1.6 Whitespace & Comments 6

1.7 Preprocessor Macros 6

2 Identifiers, Names & Scopes **7**

3 Types **9**

3.1 Paths 10

3.2 Value Types 10

3.2.1 Value Type 10

3.2.2 Type Projection 10
3.2.3 Type Designators 10
3.2.4 Parametrized Types 10
3.2.5 Tuple Types 10
3.2.6 Annotated Types 10
3.2.7 Compound Types 10
3.2.8 Function Types 10
3.2.9 Existential Types 10
3.3 Non-Value Types 10
3.3.1 Method Types 10
3.3.2 Polymorphic Method Types 10
3.3.3 Type Constructors 10
3.4 Relations Between Types 10
3.4.1 Type Equivalence 10
3.4.2 Conformance 10

4 Basic Declarations & Definitions 11
4.1 Variable Declarations & Definitions 12
4.2 Property Declarations & Definitions 12
4.3 Instance Variable Definitions 12
4.4 Type Declarations & Aliases 12
4.5 Type Parameters 12
4.6 Variance of Type Parameters 12
4.7 Function Declarations & Definitions 12
4.7.1 Positional Parameters 12
4.7.2 Optional Parameters 12
4.7.3 Repeated Parameters 12
4.7.4 Named Parameters 12
4.7.5 Procedures 12
4.7.6 Method Return Type Inference 12
4.8 Use Clauses 12

5 Classes & Objects **13**

5.1 Class Definitions 14

5.1.1 Class Linearization 14

5.1.2 Constructor & Destructor Definitions 14

5.1.3 Class Block 14

5.1.4 Class Members 14

5.1.5 Overriding 14

5.1.6 Inheritance Closure 14

5.1.7 Modifiers 14

5.2 Mixins 14

5.3 Unions 14

5.4 Enums 14

5.5 Compound Types 14

5.6 Range Types 14

5.7 Units of Measure 14

5.8 Record Types 14

5.9 Struct Types 14

5.10 Object Definitions 14

6 Expressions **15**

6.1 Expression Typing 16

6.2 Literals 16

6.3 The Nil Value 16

6.4 Designators 16

6.5 Self, This & Super 16

6.6 Function Applications 16

6.6.1 Named and Optional Arguments 16

6.6.2 Input & Output Arguments 16

6.6.3 Function Compositions & Pipelines 16

6.7 Method Values 16
6.8 Type Applications 16
6.9 Tuples 16
6.10 Instance Creation Expressions 16
6.11 Blocks 16
6.12 Prefix & Infix Operations 16
6.12.1 Prefix Operations 16
6.12.2 Infix Operations 16
6.12.3 Assignment Operators 16
6.13 Typed Expressions 16
6.14 Annotated Expressions 16
6.15 Assignments 16
6.16 Conditional Expressions 16
6.17 Loop Expressions 16
6.17.1 Classic For Expressions 16
6.17.2 Iterable For Expressions 16
6.17.3 Basic Loop Expressions 16
6.17.4 While & Until Loop Expressions 16
6.17.5 Conditions in Loop Expressions 16
6.18 Collection Comprehensions 16
6.19 Return Expressions 16
6.19.1 Implicit Return Expressions 16
6.19.2 Explicit Return Expressions 16
6.19.3 Structured Return Expressions 16
6.20 Raise Expressions 16
6.21 Rescue & Ensure Expressions 16
6.22 Throw & Catch Expressions 16
6.23 Anonymous Functions 16
6.24 Conversions 16
6.24.1 Type Casting 16

7 Implicit Parameters & Views **17**

8 Pattern Matching **19**

8.1 Patterns 19

8.1.1 Variable Patterns 19

8.1.2 Typed Patterns 19

8.1.3 Literal Patterns 19

8.1.4 Constructor Patterns 19

8.1.5 Tuple Patterns 19

8.1.6 Extractor Patterns 19

8.1.7 Pattern Alternatives 19

8.1.8 Regular Expression Patterns 19

8.2 Type Patterns 19

8.3 Pattern Matching Expressions 19

8.4 Pattern Matching Anonymous Functions 19

9 Top-Level Definitions **21**

9.1 Compilation Units 21

9.2 Modules 21

9.3 Module Objects 21

9.4 Module References 21

9.5 Top-Level Classes 21

9.6 Programs 21

10 Annotations **23**

11 Naming Guidelines **25**

12 The Coral Standard Library **27**

12.1 Root Classes 27

12.1.1 The Object Class 27

12.1.2 The Nothing Class 27
12.2 Value Classes 27
12.3 Standard Reference Classes 27

A Coral Syntax Summary **29**

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 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. 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.

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

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_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.

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**	**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 workaround 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 REPL[1] 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`**.

[1] Read-Eval-Print Loop

1.4 Operators

1.5 Literals

1.5.1 Integer Literals

1.5.2 Floating Point Literals

1.5.3 Imaginary Number Literals

1.5.4 Units of Measure

1.5.5 Boolean Literals

1.5.6 String Literals

1.5.7 Symbol Literals

1.5.8 Type Parameters

1.5.9 Regular Expression Literals

1.5.10 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