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 *7*
1.4 Operators *7*
1.5 Literals *8*
1.5.1 Integer Literals 9
1.5.2 Floating Point Literals 11
1.5.3 Boolean Literals 12
1.5.4 String Literals 12
1.5.5 Symbol Literals 13
1.5.6 Type Parameters 13
1.5.7 Regular Expression Literals 13
1.5.8 Collection Literals 14
1.6 Whitespace & Comments *15*
1.7 Preprocessor Macros *15*

Identifiers, Names & Scopes **16**

Types **17**
3.1 Paths *18*
3.2 Value Types *19*
3.2.1 Value Type 19
3.2.2 Type Projection 19
3.2.3 Type Designators 19
3.2.4 Parameterized Types 19
3.2.5 Tuple Types 19
3.2.6 Annotated Types 20
3.2.7 Compound Types 20
3.2.8 Function Types 20
3.2.9 Existential Types 20
3.3 Non-Value Types *21*
3.3.1 Method Types 21
3.3.2 Polymorphic Method Types 21
3.3.3 Type Constructors 21
3.4 Relations Between Types *22*

Basic Declarations & Definitions **22**
4.1 *22*

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.

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
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
refine
rescue
retry
return
self
skip
struct
super
template
test
then
this
throw
throws
transparent
type
undef
unless
until
union
use
var
void
yes
with
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.

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 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`**.

1.4 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

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.

`:=`	`<<<`	`<=>`	`xor`	`:>`
`+=`	`>>>`	`+`	`|`	`<<|`
`-=`	`;`	`-`	`&`	`|>>`
`*=`	`=`	`*`	`^`	`<|`
`**=`	`!=`	`**`	`~`	`|>`
`/=`	`==`	`/`	`..`	`(`
`%=`	`!==`	`div`	`...`	`)`
`||=`	`===`	`%`	`,`	`[`
`&&=`	`!===`	`mod`	`->`	`]`
`^^=`	`=~`	`||`	`<-`	`{`
`|=`	`!~`	`or`	`~>`	`}`
`&=`	`<>`	`&&`	`<~`	`.`
`^=`	`<`	`and`	`=>`	
`~=`	`>`	`!`	`::`	
`<<`	`<=`	`not`	`:`	
`>>`	`>=`	`^^`	`<:`	

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

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 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 (-2^7 to 2^7-1)
2. Integer_8_Unsigned (0 to 2^8)
3. Integer_16 (-2^{15} to $2^{15}-1$)
4. Integer_16_Unsigned (0 to 2^{16})
5. Integer_32 (-2^{31} to $2^{31}-1$)
6. Integer_32_Unsigned (0 to 2^{32})
7. Integer_64 (-2^{63} to $2^{63}-1$)
8. Integer_64_Unsigned (0 to 2^{64})
9. Integer_128 (-2^{127} to $2^{127}-1$)
10. Integer_128_Unsigned (0 to 2^{128})
11. Decimal ($-\infty$ 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!).

1.5.2 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.

1.5.3 Imaginary Number Literals

Syntax:

```
imaginary_literal ::= real_number_literal 'i'
complex_literal ::= (real_number_literal ('+' | '-') complex_literal)
     | (complex_literal ('+' | '-') real_number_literal)
real_number_literal ::= (integer_literal | float_literal)
number_literal ::= real_number_literal
     | imaginary_literal
     | complex_literal
```

Examples:

```
0i                          5+1i                        0.5f+1.5fi
5i                          1-5.2i                      3.14159i
```

Coral supports imaginary number literals along with real number literals, by simply putting an extra "`i`" after the real number literal.

1.5.4 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.

1.5.5 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 (`\"`).

1.5.6 Symbol Literals

Syntax:

```
symbol_literal ::= simple_symbol | quoted_symbol
simple_symbol ::= ':' simple_id
quoted_symbol ::= simple_quoted_symbol | interpolable_quoted_symbol
simple_quoted_symbol ::= ':'' { string_element } ) '''
interpolable_quoted_symbol ::= ':"' { int_string_element } '"'
```

Symbol literals are members of the type `Symbol`.

1.5.7 Type Parameters

Syntax:

```
type_param ::= '$' (variable_id | constant_id)
```

1.5.8 Regular Expression Literals

Syntax:

```
regexp_literal ::= '%/' regexp_content '/' [ regexp_flags ]
     | '%r/' regexp_content '/' [ regexp_flags ]
     | '%r(' regexp_content ')' [ regexp_flags ]
     | '%r~' regexp_content '~' [ regexp_flags ]
     | '%r<' regexp_content '>' [ regexp_flags ]
     | '%r[' regexp_content ']' [ regexp_flags ]
     | '%r{' regexp_content '}' [ regexp_flags ]
regexp_content ::= regexp_element { regexp_element }
regexp_element ::= string_element | interpolated_expression
regexp_flags ::= printable_char { printable_char }
```

Regular expression literals are members of the type `Regular_Expression` with alias of `Regexp`.

1.5.9 Collection Literals

Syntax:

```
collection_literal ::= tuple_literal
     | list_literal
     | dictionary_literal
     | bag_literal
tuple_literal ::= '(' list_expressions ')'
list_literal ::= '%' collection_flags '[' list_expressions ']'
dictionary_literal ::= '%' collection_flags '{' dictionary_expressions '}'
bag_literal ::= '%' collection_flags '(' list_expressions ')'
list_expressions ::= expression { ',' expression }
dictionary_expressions ::= dict_expression { ',' dict_expression }
dict_expression ::= expression ' => ' expression
     | simple_id ': ' expression
collection_flags ::= printable_char { printable_char }
```

Tuple literals are members of the `Tuple` type family. List literals are members of the `List` type, usually `Array_List`, but the collection flags may change the resulting class. Dictionary literals are members of the `Dictionary` type with alias of `Map`, usually `Hash_Dictionary` with alias of `Hash_Map`, but the collection flags may again change the resulting class. Bag literals are members of the `Bag` type, usually `Hash_Bag`, but the collection flags may yet again change the resulting class. Order of each flag is not significant.

List literal collection flags are the following:

1. Flag `i` = immutable, makes the resulting list frozen.
2. Flag `l` = linked, makes the resulting list a member of `Linked_List` instead.
3. Flag `w` = words, the following expressions are treated as words, converted to strings separated by whitespace.

Dictionary literal collection flags are the following:

1. Flag `i` = immutable, makes the resulting dictionary frozen.
2. Flag `l` = linked, makes the resulting dictionary a member of `Linked_Hash_Dictionary`.

Bag literal collection flags are the following:

1. Flag `i` = immutable, makes the resulting bag frozen.

2. Flag `s` = set, the resulting collection is a set instead of a bag (a special bag that does support multiple item occurences, and tally of each item is always 0 or 1). The resulting literal is a member of `Hash_Set`.
3. Flag `l` = linked, the resulting collection is linked, so either a member of `Linked_Hash_Bag` or `Linked_Hash_Set`.

Linked collections have a predictable iteration order in case of bags and dictionaries, or are simply stored differently in case of lists.

1.6 Whitespace & Comments

Tokens may be separated by whitespace characters and/or comments. Comments come in two forms:

A single line comment is a sequence of characters that starts with `//` and extends to the end of the line.

A multi-line comment is a sequence of characters between `/*` and `*/`. Multi-line comments may not be nested.

Documentation comments are multi-line comments that start with a `/*!`, and multi-line comments that start with a `/**` are also recognized as documentation comments, while the former is preferred.

1.7 Preprocessor Macros

TBD

Chapter 2

Identifiers, Names & Scopes

Names in Coral identify various types, values, methods and constants, which are the *entities*. Names are introduced by local definitions and declarations, inheritance, use clauses or module clauses, which are the *bindings*.

Bindings of different kinds have a different precedence defined on them:

1. Definitions and declarations that are local have the highest precedence.
2. Inherited definitions and declarations have the next highest precedence.
3. Definitions and declarations made available by module clause have the next highest precedence.
4. Explicit **use** imports have the next highest precedence.
5. Definitions and declarations that are not in the same compilation unit (a different script or a different module) have the next highest precedence.
6. Definitions and declarations that are not bound have the lowest precedence. This happens when the binding simply can't be found anywhere, and probably will result in a name error, while being inferred to be of type Object.

There is only one root name space, in which a single fully qualified binding designates always up to one entity.

Every binding has a *scope* in which the bound entity can be referenced using a simple name (unqualified). Scopes are nested, inner scopes inherit the same bindings, unless shadowed. A binding in an inner scope *shadows* bindings of lower precedence in the same scope (and nested scopes) as well as bindings of the same or lower precedence in outer scopes. Shadowing is a partial order, and bindings can become ambiguous – fully qualified names can be used to resolve binding conflicts.

A reference to an unqualified identifier *x* is bound by a unique binding, which

1. defines an entity with name *x* in the same name scope as the identifier *x*, and

2. shadows all other bindings that define entities with name *x* in that name scope.

It is syntactically not an error if no such binding exists, thanks to the dynamic features of the language (unbound references are implicitly bound to the same scope and are resolved by dynamic method callbacks). The same applies to fully qualified bindings that don't resolve into any entity. However, it is an error if a binding is ambiguous.

If *x* is bound by explicit **use** import clause, then the simple name *x* is considered to be equivalent to the fully qualified name to which *x* is mapped by the import clause. If x is bound by a definition or declaration, then *x* refers to the entity introduced by that binding, thus the type of *x* is the type of the referenced entity.

Chapter 3

Types

When we say *type* in the context of Coral, we are talking about a blueprint of an entity, while the type itself is an entity. Every type in Coral is backed by a *class*, which is an instance of the type `Class`.

We distinguish a few different properties of types in Coral. There are first-order types and type constructors, which take type parameters and yield new types. A subset of first-order types called *value types* represents set of first-class values. Value types are either *concrete* or *abstract*.

Concrete value types can be either a *class type* (i.e. referenced with a type designator, referencing a class or a mixin), or a *compound type* representing an intersection of types, possibly with a refinement that further constrains the types of its members. Both class types and compound types may be bound to a constant, but only class types referencing a concrete class can be blueprints of values (objects). Compound types can only constrain bindings to a subset of other types.

Non-value types capture properties of identifiers that are not values. For instance, a type constructor does not directly specify a type of values, but a type constructor, when

applied to the correct type arguments, yields a first-order type, which may be a value type. Non-value types are expressed indirectly in Coral. In example, a method type is described by writing down a method signature, which is not a real type itself, but it creates a corresponding method type.

To sum it up, type system in Coral is a way to describe sets of values (objects, instances of concrete classes – class types). Types can be applied to method arguments, return values, variables, as a way of binding them to a particular set of possible values with predictable properties and behaviors. However, Coral is a dynamically typed language, and therefore types are type-checked in runtime rather than bound forever by compile time – Coral is able to perform only limited type checking while compiling, the worst it can do is to issue a warning when a type mismatch is apparent.

3.1 Paths

Paths are Coral's way of referencing types. A path represents a binding in a particular scope. If a binding can not be found while compiling a program, then the path is bound to the first enclosing type.

Syntax:

```
Path ::= [ '::' ] Const_Path [ '::' Self_Path ]
     | Self_Path
Self_Path ::= ( 'this' | 'self' | 'super' [Class_Qualifier] ) [Const_Path]
Class_Qualifier ::= '<' Const_Path '>'
Const_Path ::= constant_id { '::' constant_id }
```

Note that the `::` token is an operator used while dynamically referencing not only types, but any properties or methods that happen to be found along the way through the path.

The **`this`** keyword refers to the same class that is directly enclosing its occurence. The **`self`** keyword refers to the actual class that the value is a member of. The **`super`** keyword refers to the direct superclass of the value, or the superclass designated by the class qualifier. If there is no constant path before the self path, then the directly enclosing class is implicitly a part of the constant path. A path that is prefixed with an extra `::` operator is a stable root path, and therefore not bound to the scope.

3.2 Value Types

Value types are entities, for which we can tell if a value is of that type or not.

3.2.1 Value Type

To retrieve the actual type of a value, the method `.class` can be used. This method is defined on every object and its return type is covariant – it is always a `Class` (or a subtype, if any exists).

Syntax:

```
Simple_Type ::= Path '.' 'class'
```

3.2.2 Type Projection

Syntax:

```
Simple_Type ::= Simple_Type '##' constant_id
```

A type projection *T##x* references type member named *x* of type *T*.

3.2.3 Type Designators

A type designator refers to a named value type (constant) and can be qualified or unqualified. Every type designator is a shorthand for a type projection.

Every unqualified type designator *t* with a stable binding in scope of a type *C* is a shorthand for the type projection *C.class##t*. Unqualified type designators are bound to the same scope in which they appear: *C.this.class##t*.

3.2.4 Parameterized Types

Syntax:

```
Simple_Type ::= Simple_Type Type_Args
Type_Args ::= '<' Types '>'
```

3.2.5 Tuple Types

Syntax:

```
Simple_Type ::= '(' Types ')'
```

3.2.6 Annotated Types

Syntax:

```
Annotated_Type ::= {Annotation} Simple_Type
```

3.2.7 Compound Types

Compound types can be used in two ways: to further describe values which possess behaviors of multiple types (up to one class, unlimited number of protocols and mixins and an optional refinement constraint), or to create types ad-hoc (if no class is given, `Object` is implied).

Syntax:

```
Compound_Type ::= Annotated_Type {'with' Annotated_Type} [Refinement]
     | Refinement
Refinement ::= 'refine' '{' Refine_Expr {semi Refine_Expr} '}'
```

3.2.8 Function Types

Syntax:

```
Function_Type ::= Args_List {'->' Args_List} '->' Return_Type
Args_List ::= '(' [Arg_Type {',' Arg_Type}] ')'
Return_Type ::= (Type | 'void')
```

Multiple arguments lists declare function types that are (automatically) curried, so this way a value can be declared as a function that returns a function.

3.2.9 Existential Types

Syntax:

```
Type ::= Compound_Type [Existential_Clauses]
     | Function_Type {* for clarity *}
Existential_Clauses ::= 'for-some' '{' Existential_Decl
           {semi Existential_Decl} '}'
```

3.3 Non-Value Types

The types described as non-value types do not represent sets of values, nor do they appear explicitly in programs, only implicitly: you can't write them in your programs, they are the internal types of defined identifiers.

3.3.1 Method Types

This type represents methods and functions that take (named) arguments and return a result of a type.

Examples:

Function declarations:

```
1. def a -> Integer
2. def b (x : Integer) -> Boolean
3. def c (x : Integer) -> (y : String, z : String) -> String
4. def d (:x : Integer) -> Integer
5. def e (*x : Integer) -> Integer
6. {|x : Integer| -> Boolean }
```

… produce the typings:

```
1. :a () Integer
2. :b (Integer) Boolean
3. :c (Integer) (String, String) String
4. :d (:x Integer) Integer
5. :e (*Integer) Integer
6. (Integer) Boolean
```

3.3.2 Polymorphic Method Types

TBD

3.3.3 Type Constructors

TBD

3.4 Relations Between Types

Chapter 4

Basic Declarations & Definitions

4.1 Variable Declarations & Definitions

4.1.1 Property Declarations & Definitions

4.1.2 Instance Variable Definitions

4.1.3 Type Declarations & Aliases

4.1.4 Type Parameters

4.1.5 Variance of Type Parameters

4.1.6 Function Declarations & Definitions