The Coral Language Specification

Kateřina Markéta Lisová

July 22, 2014

Contents

1 Lexical Syntax 3

1.1 Identifiers 4

1.2 Keywords 5

1.3 Newline Characters 5

1.4 Operators 6

1.5 Literals 7

1.5.1 Integer Literals 7

1.5.2 Floating Point Literals 9

1.5.3 Imaginary Number Literals 10

1.5.4 Units of Measure 10

1.5.5 Character Literals 10

1.5.6 Boolean Literals 10

1.5.7 String Literals 11

1.5.8 Symbol Literals 11

1.5.9 Type Parameters 11

1.5.10 Regular Expression Literals 12

1.5.11 Collection Literals 12

1.6 Whitespace & Comments 13

2 Identifiers, Names & Scopes 15

3 Types 17

3.1 About Coral's Type System 17

3.2 Paths 19

3.3 Value Types 19

3.3.1 Value & Singleton Type 20
3.3.2 Type Projection 20
3.3.3 Type Designators 20
3.3.4 Parameterized Types 21
3.3.5 Tuple Types 22
3.3.6 Annotated Types 22
3.3.7 Compound Types 22
3.3.8 Function Types 23
3.3.9 Existential Types 24
3.3.10 Nullable Types 24
3.3.11 Dependent Types 25
3.4 Non-Value Types 25
3.4.1 Method Types 25
3.4.2 Polymorphic Method Types 26
3.4.3 Type Constructors 27
3.5 Relations Between Types 27
3.5.1 Type Equivalence 27
3.5.2 Conformance 28
3.5.3 Weak Conformance 29

4 Basic Declarations & Definitions 31
4.1 Value Declarations & Definitions 31
4.2 Variable Declarations & Definitions 34
4.3 Property Declarations & Definitions 35
4.4 Type Declarations & Aliases 36
4.5 Type Parameters 37
4.6 Variance of Type Parameters 37
4.7 Function Declarations & Definitions 38
4.7.1 By-Name Parameters 40

4.7.2 Explicit Parameters 41
4.7.3 Input & Output Parameters 41
4.7.4 Positional Parameters 41
4.7.5 Optional Parameters 42
4.7.6 Repeated Parameters 43
4.7.7 Named Parameters 44
4.7.8 Captured Block Parameter 45
4.7.9 Parameter Kind Combinations 46
4.7.10 Method Types Inference 46
4.8 Overloaded Declarations & Definitions 48
4.9 Use Clauses 49

5 Classes & Objects 51
5.1 Templates 52
5.1.1 Constructor Invocations & Primary Constructor 52
5.1.2 Metaclasses & Eigenclasses 52
5.1.3 Class Linearization 55
5.1.4 Inheritance Trees & Include Classes 55
5.1.5 Class Members 57
5.1.6 Overriding 58
5.1.7 Inheritance Closure 58
5.1.8 Early Definitions 59
5.2 Modifiers 59
5.3 Class Definitions 65
5.3.1 Constructor & Destructor Definitions 68
5.3.2 Clone Constructor Definitions 70
5.3.3 Case Classes 71
5.3.4 Traits 71
5.3.5 Refinements 72

5.3.6 Protocols 72
5.3.7 Interfaces 73
5.4 Object Definitions 73
5.5 Module Definitions 74
5.6 Unions 74
5.7 Enums 75
5.8 Dependent Type Declarations 75
5.8.1 Indexed Types 75
5.8.2 Constrained Types 79
5.8.3 Range Types 80
5.9 Units of Measure 81
5.10 Record Types 82
5.11 Nullability 84

6 Expressions 87
6.1 Expression Typing 88
6.2 Literals 88
6.3 The Nil Value 88
6.4 Designators 88
6.5 Self, This & Super 88
6.6 Use Expressions 88
6.7 Function Applications 88
6.7.1 Named and Optional Arguments 88
6.7.2 By-Name Arguments 88
6.7.3 Input & Output Arguments 88
6.7.4 Function Compositions & Pipelines 88
6.8 Method Values 88
6.9 Type Applications 88
6.10 Tuples 88

6.11 Instance Creation Expressions 88
6.12 Blocks 88
  6.12.1 Local Variable Closure 88
6.13 Prefix & Infix Operations 88
  6.13.1 Prefix Operations 88
  6.13.2 Infix Operations 88
  6.13.3 Assignment Operators 88
6.14 Typed Expressions 88
6.15 Annotated Expressions 88
6.16 Assignments 88
6.17 Conditional Expressions 88
6.18 Loop Expressions 88
  6.18.1 Classic For Expressions 88
  6.18.2 Iterable For Expressions 88
  6.18.3 Basic Loop Expressions 88
  6.18.4 While & Until Loop Expressions 88
  6.18.5 Conditions in Loop Expressions 88
6.19 Collection Comprehensions 88
6.20 Return Expressions 88
  6.20.1 Implicit Return Expressions 88
  6.20.2 Explicit Return Expressions 88
  6.20.3 Structured Return Expressions 88
6.21 Raise Expressions 88
6.22 Rescue & Ensure Expressions 88
6.23 Throw & Catch Expressions 88
6.24 Anonymous Functions 88
6.25 Conversions 88
  6.25.1 Explicit Conversions 88
  6.25.2 Implicit Conversions 88
6.26 Workflows 88

7 Implicit Parameters & Views 89

7.1 The Implicit Modifier 89

7.2 Implicit Parameters 89

7.3 Views 89

8 Pattern Matching 91

8.1 Patterns 92

8.1.1 Variable Patterns 92

8.1.2 Typed Patterns 92

8.1.3 Pattern Binders 92

8.1.4 Literal Patterns 92

8.1.5 Constant Patterns 92

8.1.6 Constructor Patterns 92

8.1.7 Tuple Patterns 92

8.1.8 Extractor Patterns 92

8.1.9 Pattern Sequences 92

8.1.10 Pattern Alternatives 92

8.1.11 Regular Expression Patterns 92

8.2 Type Patterns 92

8.3 Pattern Matching Expressions 92

8.4 Pattern Matching Anonymous Functions 92

9 Top-Level Definitions 93

9.1 Compilation Units 93

9.2 Modules 93

9.3 Module References 93

9.4 Top-Level Classes 93

9.5 Programs 93

10 Annotations 95

11 Naming Guidelines 97

12 The Coral Standard Library 99
12.1 Root Classes 99
12.1.1 The Object Class 99
12.1.2 The Nothing Class 99
12.2 Value Classes 99
12.3 Standard Reference Classes 99

A Coral Syntax Summary 101

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 trait 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#, Clojure and ATS. 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 possible 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.

ATS inspired Coral with dependent types (§3.3.11 & §5.8).

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.

Grammar of lexical tokens is given in the following sections. These tokens are then used as terminal symbols of the semantical grammar.

To construct tokens, characters are distinguished according to the following classes (Unicode general category given in parentheses):

- Whitespace characters. \u0020 | \u0009 | \u000D | \u000A (space, tab character, carriage return, line feed)
- Letters, which include lower case letters (Ll), upper case letters (Lu), title-case letters (Lt), other letters (Lo), letter numerals (Nl), modifier letters (Lm) and the two characters \u0024 ‘`$`’ and \u005F ‘`_`’, which both count as upper case letters.
- Digits ‘`0`’ | … | ‘`9`’.
- Parentheses. ‘`(`’ | ‘`)`’ | ‘`[`’ | ‘`]`’ | ‘`{`’ | ‘`}`’
- Delimiter characters. ‘`'`’ | ‘`"`’ | ‘`.`’ | ‘`;`’ | ’`,`’
- Operator characters. These consist of all printable ASCII characters \u0020-\u007F that are in none of the sets above, mathematical symbols (Sm) and other symbols (So).

1.1 Identifiers

Syntax:

```
simple_id     ::= (lower | '_') [id_rest]
variable_id   ::= simple_id | '_' | '`' simple_id '`'
ivar_id       ::= '@' simple_id
cvar_id       ::= '@@' simple_id
op_id         ::= opchar {opchar}
function_id   ::= simple_id [id_rest_fun]
                | '`' simple_id [id_rest_fun] '`'
constant_id   ::= upper [id_rest_con]
                | '`' upper [id_rest_con] '`'
id_rest       ::= {letter | digit | '_'}
id_rest_con   ::= id_rest [id_rest_mid]
id_rest_fun   ::= id_rest [id_rest_mid] [['_'] op_id]
id_rest_mid   ::= id_rest {('/' | '+' | '-') id_rest}
importable_id ::= simple_id
                | function_id
                | constant_id
                | '`' op_id '`'
```

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). Additionally, *instance variable identifiers* are just prepended with a “`@`” sign and *class instance variable identifiers* are just prefixed with “`@@`”.

Second, *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 “`=`”. Furthermore, function identifiers ending with “`=`” are never used at call site with this last character, but without it and as a target of an assignment expression (they are naming simple setters).

And third, *constant identifiers*, which are just like function identifiers, but starting with an upper-case letter, never just an underscore and never ending with “`?`”, “`!`” or “`=`”.

An identifier may also be formed by an identifier between back-quotes (“`` ` ``”), to resolve possible name clashes with Coral keywords. Instance variable names (`ivar_id`) and class instance variable names (`cvar_id`) never clash with a keyword name, since these are distinguished by the preceding “`@`” and “`@@`” respectively.

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.

```
abstract    alias       annotation  as          begin
bitfield    break       case        cast        catch
class       clone       cloned      constant    constructor
declare     def         destructor  do          else
elsif       end         ensure      enum        extends
final       for         for-some    function    get
goto        if          indexed-with            implements
implicit    in          include     interface   is
lazy        let         loop        match       memoize
message     method      module      native      next
nil         no          object      of          opaque
operator    out         override    prepend     private
property    protected   protocol    public      raise
range       record      redo        refine      rescue
retry       return      requires    sealed      self
set         singleton-type          skip        step
super       template    test        then        this
throw       trait       transparent type        undef
unless      until       union       unit-of-measure
use         val         var         yes         weak
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`**, **`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 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      |        &        ^        ~        ..
...      ,        ->       <-       ~>       <~       =>       ::       :        <:
:>       <<|      |>>      <|       |>       (        )        [        ]        {
}        .        [<       >]       >:       .?       .!       @{
```

---

[1] Read-Eval-Print Loop

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 does'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

Syntax:

```
float_literal ::= ['+' | '-'] non_zero_digit
                  {['_'] digit} '.' digit {['_'] digit}
                  [exponent_part] [float_type]
                | ['+' | '-'] digit {['_'] digit} exponent_part
                  [float_type]
                | ['+' | '-'] digit {['_'] digit} [exponent_part]
                  float_type
                | ['+' | '-'] '0x' hex_digit
                  {['_'] hex_digit} '.' hex_digit
                  {['_'] hex_digit} [float_type]
                | ['+' | '-'] '0b' bin_digit
                  {['_'] bin_digit} '.' bin_digit
                  {['_'] bin_digit} [float_type]
exponent_part ::= 'e' ['+' | '-'] digit {['_'] digit}
float_type    ::= 'f' | 'd' | 'q' | 'df'
```

Floating point literals are of type `Number` as well as integral literals, and have fewer container types. Compiler infers the precision automatically, unless the `float_type` part is present. Floating point literals that have `float_type` of “`dp`” are decimal fixed point literals. Also, floating point literals that are impossible to represent in binary form accurately are implicitly fixed point literals. From user’s perspective, this is only an implementation detail.

1. `Float_32` (IEEE 754 32-bit precision), alias `Float`.
2. `Float_64` (IEEE 754 64-bit precision), alias `Double`.
3. `Float_128` (IEEE 754 128-bit precision).
4. `Decimal` ($-\infty$ to ∞).
5. `Decimal_Unsigned` (0 to ∞).

Letters in the exponent type and float type literals have to be lower-case in Coral sources, but functions that parse floating point numbers do support them being upper-case for compatibility.

1.5.3 Imaginary Number Literals

Syntax:

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

1.5.4 Units of Measure

Coral has an addition to number handling, called *units of measure* (§5.9). Number instances can be annotated with a unit of measure to ensure correctness of arithmetic operations.

Syntax:

```
annotated_number ::= number_literal '[<' uom_expr '>]'
uom_expr         ::= Unit_Conv {',' Unit_Conv}
```

1.5.5 Character Literals

Syntax:

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

1.5.6 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`**: **`no`** equals to **`nil`**, while not actually being **`nil`**. Identity equality is indeed different. The implication is that both **`nil`** and **`no`** are false conditions in **`if`**-expressions.

1.5.7 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_expr
interpolated_expr           ::= '#{' expr '}'
```

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 (\"). Interpolated expression can be preceded only by an even number of escape characters (backslashes, \), so that the # does't get escaped. This is a special *requirement* for any Coral compiler.

1.5.8 Symbol Literals

Syntax:

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

Symbol literals are members of the type `Symbol`. They differ from String Literals in the way runtime handles them: while there may be multiple instances of the same string, there is always up to one instance of the same symbol. Unlike in Ruby, they do get released from memory when no code references to them anymore, so their object id (sometimes) varies with time. Coral does not require their ids to be constant in time.

1.5.9 Type Parameters

Syntax:

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

Type parameters are not members of any type, rather they stand-in for a real type, like a variable which only holds types.

1.5.10 Regular Expression Literals

Syntax:

```
regexp_literal ::= '%/' regexp_content_int '/' [regexp_flags]
    | '%r/' regexp_content_int '/' [regexp_flags]
    | '%r#' regexp_content '#' [regexp_flags]
    | '%r~' regexp_content_int '~' [regexp_flags]
regexp_content_int ::= regexp_element_int {regexp_element_int}
regexp_element_int ::= string_element | int_string_element
regexp_content ::= string_element {string_element}
```

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

1.5.11 Collection Literals

Collection literals are paired syntax tokens and as such, they are a kind of parentheses in Coral sources.

Syntax:

```
collection_literal ::= tuple_literal
    | list_literal
    | dictionary_literal
    | bag_literal
tuple_literal ::= '(' exprs ')'
list_literal ::= '%' collection_flags '[' exprs ']'
dictionary_literal ::= '%' collection_flags '{' dict_exprs '}'
bag_literal ::= '%' collection_flags '(' exprs ')'
exprs ::= expr {',' expr}
dict_exprs ::= dict_expr {',' dict_expr}
dict_expr ::= expr '=>' expr
    | simple_id ':' expr
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` with alias of `Array`. Dictionary literals are members of the `Dictionary` type with alias of `Map`, usually `Hash_Dictionary` with alias of `Hash_Map`. Bag literals are members of the `Bag` type, usually `Hash_Bag` or `Hash_Set`. Collection flags may change the actual class of the literal, along with some other properties, described in the following text.

List literal collection flags:

1. Flag `i` = **i**mmutable, makes the list frozen.

2. Flag `l` = **l**inked, makes the list a member of `Linked_List`.
3. Flag `w` = **w**ords, the following expressions are treated as words, converted to strings for each word separated by whitespace.

Dictionary literals collection flags:

1. Flag `i` = **i**mmutable, makes the dictionary frozen.
2. Flag `l` = **l**inked, makes the dictionary a member of `Linked_Hash_Dictionary` (also has alias `Linked_Hash_Map`).
3. Flag `m` = **m**ulti-map, the dictionary items are then either the items themselves, if there is only one for a particular key, or a set of items, if there is more than one item for a particular key. The dictionary is then a member of `Multi_Hash_Dictionary` (alias `Multi_Hash_Map`) or `Linked_Multi_Hash_Dictionary` (alias `Linked_Multi_Hash_Map`).

Bag literal collection flags:

1. Flag `i` = **i**mmutable, makes the bag frozen.
2. Flag `s` = **s**et, the collection is a set instead of a bag (a specific bag, such that for each item, its tally is always 0 or 1, thus each item is in the collection up to once).
3. Flag `l` = **l**inked, makes the collection linked, so either a member of `Linked_Hash_Bag` in case of a regular bag, or `Linked_Hash_Set` in case of a 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 be nested.

Documentation comments are multi-line comments that start with `/*!`.

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. Explicit **`use`** clauses (imports) have the next highest precedence.[1]
3. Inherited definitions and declarations have the next highest precedence.
4. Definitions and declarations made available by module clause 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 (if not resolved dynamically), 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. This restriction is checked in limited scope during compilation[2] and fully in runtime.

---

[1] Explicit imports have such high precedence in order to allow binding of different names than those that would be otherwise inherited.

If at any point of the program execution a binding would change (e.g., by introducing a new type in a superclass that is closer in the inheritance tree to the actual class than the previous binding), and such a change would be incompatible with the previous binding, then a warning[3] will be issued by the runtime. Also, if a new binding would be ambiguous[4], then it is an error.

As shadowing is only a partial order, in a situation like

```
var x := 1
use p::x
x
```

neither binding of x shadows the other. Consequently, the reference to x on the third line above is ambiguous and the compiler will happily refuse to proceed.

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

1. defines an entity with name x in the same 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 or fails to get resolved dynamically.

If x is bound by explicit **`use`** import clause, then the simple name x is consided 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.

---

[2] This is due to the hybrid typing system in Coral, to make use of all the available information as soon as possible.

[3] TBD – shouldn't that be an error?

[4] Coral runtime actually checks for bindings until the binding-candidate would not be able to shadow the already found binding-candidates and caches the result.

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* (e.g. referenced with a type designator, referencing a class or maybe a trait), 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.

3.1 About Coral's Type System

There are two main streams of typing systems out there – statically typed and dynamically typed. Static typing in a language usually means that the language is compiled into an executable with a definite set of types and every operation is type checked. Dynamic typing means that these checks are deferred until needed, in runtime.

Let's talk about Java. Java uses static typing – but, in a very limited and unfriendly way, you may use class loaders and a lot of type casts to dynamically load a new class. And then possibly endure a lot of pain using it.

Let's talk about Ruby. Ruby uses dynamic typing – but, using types blindly can possibly lead to some confusion. Ruby is amazing though, because you can write programs with it really fast and enjoy the process at the same time. But when it comes to type safety, you need to be careful.

And now, move on to Coral. Coral uses hybrid typing. In its core, it uses dynamic typing all the way. But, it allows to opt-in for some limited static typing[1]. Unlike in Ruby, you can overload methods (not just override!). You can constrain variables, constants, properties, arguments and return types to particular types. But you don't have to. Types in Coral were heavily inspired by Scala's type system, but modified for this dynamic environment that Coral provides. Unlike in Ruby, you can have pure interfaces (called protocols[2]), or interfaces with default method implementations (similar to Java 8). Unlike in Java, you can have traits, union types and much more. Unlike in Java, you may easily modify classes, even from other modules (*pimp my library!*). You may even easily add more classes if needed, and possibly shadow existing ones. In face of static typing in Coral, *no type* specified is saying that the value is of any type.

While Coral is so dynamic, it also needs to maintain stability and performance. Therefore, it "caches" its bindings and tracks versions of each type[3]. If a *cached binding* would change, it is ok – as long as the new binding would conform to the old one. Practically, the code that executes first initiates the binding – first to come, first to bind. Bindings are also cached, so that the Coral interpreter does not need to traverse types all the time – it only does so if the needed binding does not exist (initial state), or if the cached version does not match the actual version of the bound type. This mechanism is also used for caching methods, not only types. Moreover, this mechanism ensures that type projections (§3.3.2) are valid at any time of execution, even if their binding changes.

Types in Coral are represented by objects that are members of the `Class` type.

---

[1] This feature is expected to be gradually improved and un-limited.

[2] Interfaces in Coral are used to extract the *public interface* of classes in modules, so that only a small amount of code may be distributed along with the module to allow binding to it.

[3] Versions are simply integers that are incremented with each significant change to the type and distributed among its subtypes.

3.2 Paths

Syntax:

```
Path             ::= Stable_Id
                   | 'this'
                   | [constant_id '#'] 'self'
Stable_Id        ::= constant_id
                   | ['::'] Path '::' constant_id
                   | [constant_id '#'] 'super' [Class_Qualifier]
                     '::' constant_id
Class_Qualifier  ::= '[' constant_id ']'
```

Paths are not types themselves, but they can be a part of named types and in that function form a role in Coral's type system.

A path is one of the following: [4]

- The empty path ϵ (which can not be written explicitly in user programs).
- **`this`**, which references the directly enclosing class.
- C**`#self`**, where C references a class or a trait. The path **`self`** is taken as a shorthand for C**`#self`**, where C is the name of the class directly enclosing the reference.
- p`::`x, where p is a path and x is a member of p. Additionally, p allows modules to appear instead of references to classes or traits, but no module reference can follow a class or a trait reference: `{module_ref '::'} {(class_ref|trait_ref) '::'}` ....
- C**`#super::`**x or C**`#super`**`[`M`]::`x, where C references a class or a trait and x references a member of the superclass or designated parent class M of C. The prefix **`super`** is taken as a shorthand for C**`#super`**, where C is the name of the class directly enclosing the reference, and **`super`**`[`M`]` as a shorthand for C**`#super`**`[`M`]`, where C is yet again the name of the class directly enclosing the reference.

3.3 Value Types

Every value in Coral has a type which is of one of the following forms.

---

[4] This section might need a review of what a path is, since we claim that the referenced entity is a member, yet the syntax only mentions `constant_id`.

3.3.1 Value & Singleton Type

Syntax:

```
Simple_Type ::= Path '#' 'type'
Simple_Type ::= Path '#' 'singleton-type'
```

A singleton type is of the form p#**`singleton-type`** and a special type that denotes the set of values consisting of **`nil`** and the value denoted by p. A value type, on the other hand, is a special type that denotes the set of values consisting of **`nil`** and every value that conforms to the type of value denoted by p.[5]

3.3.2 Type Projection

Syntax:

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

A type projection T#x references type member named x of type T. [6]

3.3.3 Type Designators

Syntax:

```
Simple_Type ::= Stable_Id
```

A type designator refers to a named value type. It can be simple or qualified. All such type designators are shorthands for type projections.

Specifically, the unqualified type name t where t is bound in some class, object or module C is taken as a shorthand for C#**`self`**#**`type`**#t. If t is not bound in a class, object or module, then t is taken as a shorthand for ϵ#**`type`**#t.

A qualified type designator has the form p`::`t, where p is a path (§3.2) and t is a type name. Such a type designator is equivalent to the type projection p#**`type`**#t.

---

[5] This is useful when using a value as prototype of new values.

[6] Type projection operator # is a language construct and can't be overridden by user programs. There is a similarity between this construct and the `::` scope operator. The difference is, type projection operator is expected to be rarely needed, but it does provide a type projection and can refer in a stable way to a type of anything. Scope operator, on the other hand, does not care about types, it merely resolves a member of a particular expression at runtime.

3.3.4 Parameterized Types

Syntax:

```
Simple_Type ::= Simple_Type Type_Args
Type_Args   ::= ':[' Types ']'
Types       ::= Type {',' Type}
```

A parameterized type[7] $T : [T_1, \ldots, T_n]$ consists of a type designator T and type parameters $T_1, \ldots, T_n$, where $n \geq 1$. T must refer to a type constructor which takes exactly n type parameters $a_1, \ldots, a_n$.

Say the type parameters have lower bounds $L_1, \ldots, L_n$ and upper bounds $U_1, \ldots, U_n$. The parameterized type is well-formed if each actual type parameter *conforms to its bounds*, so that $L_i <: \sigma a_i <: U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$. Also, U_i must never be a subtype of L_i, since no other type ever would be able to fulfil the bounds (U_i and L_i may be the exact same type though, but in that case the type parameter would be invariant and the whole point of having a parameterized type would be useless).

Example 3.3.1 Given the generic type definitions:

```
class Tree_Map:[$A <: Comparable:[$A], $B] ... end
class List:[$A] ... end
class I; implements Comparable:[I]; ... end

class F:[$M:[_], $X] ... end
class S:[$K <: String] ... end
class G:[$M:[$Z <: $I], $I] ... end
```

the following parameterized types are well-formed:

```
Tree_Map:[I, String]
List:[I]
List:[List:[Boolean]]

F:[List, Number]
G:[S, String]
```

Example 3.3.2 Given the type definitions of the previous example, the following types are malformed:

---

[7] The ‘`:[`’ and ‘`]`’ token pairs were selected after many considerations. The original idea was to use the same pair as Java or C#: ‘<’ and ‘>’, but this has the drawback of injecting an exception into the parser. Then Scala’s ‘`[`’ and ‘`]`’ were considered, but they kind of collide with an operator that uses the same pair and therefore expressions could become ambiguous. Finally an idea emerged to use ‘`:[`’ and ‘`]`’, since the ‘`:`’ in it hints that it is related to the type system, and ‘`[`’ and ‘`]`’ lack the drawbacks of ‘<’ and ‘>’ (collision with comparison operators).

```
Tree_Map:[I]                // wrong number of parameters
Tree_Map:[List:[I], Number] // type parameter List not within bound

F:[Number, Boolean]         // Number is not a type constructor
F:[Tree_Map, Number]        // Tree_Map takes two parameters,
                           //   F expects a type constructor taking one

G:[S, Number]               // type S constrains its parameter to
                           //   conform to String,
                           // G expects type constructor with a parameter
                           //   that conforms to Number
```

3.3.5 Tuple Types

Syntax:

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

A tuple type $(T_1, \ldots, T_n)$ is an alias for the class `Tuple_`n`:[`$T_1, \ldots, T_n$`]`, where $n \geq 2$.

Tuple classes are available as patterns for pattern matching. The properties can be accessed as methods `[1]`, …, `[`n`]` (using an “offset” that is outside of the tuple’s size results in a method-not-found error, not offset-out-of-bounds – tuple classes do not implement the operator `[i]` for arbitrary i).

Tuple classes are generated lazily by the runtime as needed, so that the language does not constrain users to tuples of only limited sizes, but allows any size.

An effort will be made to introduce a simple enough syntax for variable parameterized types, if possible, until then, `Tuple_`i are the only such types.

3.3.6 Annotated Types

Syntax:

```
Annot_Type ::= {Annotation} Simple_Type
```

An annotated type $a_1 \ldots a_n T$ attaches annotations $a_1, \ldots, a_n$ to the type T.

3.3.7 Compound Types

Syntax:

```
Compound_Type ::= Annot_Type {'with' Annot_Type} [Refinement]
                | Refinement
Refinement    ::= 'refine' '{' Refine_Stat {semi Refine_Stat} '}'
```

A compound type T_1 **`with`** $\ldots$ **`with`** T_n $\{R\}$ represents values with members as given in the component types $T_1, \ldots, T_n$ and the refinement $\{R\}$. A refinement $\{R\}$ contains declarations and definitions (§5.3.5).

If no refinement is given, the type is implicitly equivalent to the same type having an empty refinement.

A compound type may also consist of just a refinement $\{R\}$ with no preceding component types – such a type has an implicit component type `Object` and describes the member values as "any value, as long as it has what the refinement requires", thus it works like an anonymous protocol.

If a compound type does not contain a concrete class type, then `Object` is implied in case the type is used as a concrete class[8].

3.3.8 Function Types

Syntax:

```
Type           ::= [Function_Args {'->' Function_Args}]
                 '->' Return_Type
Function_Args ::= Type
                | '(' [Type {',' Type}] ')'
Return_Type   ::= Type | '()'
```

The type `(`$T_1, \ldots, T_n$`) ->` R represents the set of function values that take arguments of types $T_1, \ldots, T_n$ and yield results of type R. In the case of exactly one argument, type T `->` R is a shorthand for `(`T`) ->` R. Empty arguments list is indeed also possible as `->` R, equivalent to `() ->` R.

Function types associate to the right, e.g. `(`S`) -> (`T`) ->` R is the same as `(`S`) -> ((`T`) ->` R`)`.

Function types are shorthands for class types that conform to the `Function_`i protocol – i.e. having an `apply` function or simply *being* a function. The n-ary function type `(`$T_1, \ldots, T_n$`) ->` R is a shorthand for the protocol `Function_`n`:[`$T_1, \ldots, T_n, R$`]`. Such protocols are defined in the Coral library for any $n \geq 0$:

```
protocol Function_n:[-$T_1, ..., -$T_n, +$R]
  message apply (x_1 : $T_1, ..., x_n : $T_n): $R
```

---

[8] Meaning that the compound type is used as an ad-hoc (possibly anonymous) class, e.g. to create new instances of it.

```
  ...
end protocol
```

Function types are covariant in their result type and contravariant in their argument types (§4.6).

3.3.9 Existential Types

Syntax:

```
Type                ::= Compound_Type Existential_Clauses
Existential_Clauses ::= {'for-some' '{' Existential_Dcl
                        {semi Existential_Dcl} '}'}
Existential_Dcl     ::= 'type' Type_Dcl
```

An existential type has the form T **`for-some`** $\{Q\}$, where Q is a sequence of type declarations. Let $t_1\texttt{:}\,[tps_1]\ \texttt{>:}\ L_1\ \texttt{<:}\ U_1\ , \ldots,\ t_n\texttt{:}\,[tps_n]\ \texttt{>:}\ L_n\ \texttt{<:}\ U_n$ be the types declared in Q.

A *type instance* of T **`for-some`** $\{Q\}$ is a type σT, where σ is a substitution over $t_1, \ldots, t_n$, such that for each i, $L_i\ \texttt{<:}\ t_i\ \texttt{<:}\ U_i$. The set of values denoted by the existential type T **`for-some`** $\{Q\}$ is the union of the set of values of all its type instances.

3.3.10 Nullable Types

Syntax:

```
Nullable_Type ::= Type [Nullable_Mod]
Nullable_Mod  ::= '?' | '!'
```

A nullable type has the form T`?` or T`!`, where “`?`” denotes explicitly a nullable type, and “`!`” denotes explicitly not-nullable type. Although **`nil`** as the singleton member of the `Nothing` type is a subtype of every type, Coral types are implicitly not-nullable, meaning it’s not possible to pass **`nil`** where an instance of T is expected, unless T is of course `Nothing`. Nullability (§5.11) is one of the intrinsic properties of every class type.

Explicitly nullable types are handled by an intrinsic anonymous subtype of T, which is explicitly nullable, overriding the preference of T. Explicitly not-nullable types are handled by an intrinsic anonymous subtype of T, which is explicitly not-nullable, overriding the preference of T. Explicit nullability of already nullable types is redundant, as is explicit non-nullability of already not-nullable types. Explicit nullability of the `Option` type is also redundant and is in fact ignored.

Nullable types in this form can appear as types of variables, parameters and return types.

3.3.11 Dependent Types

Syntax:

```
Dep_Type ::= Simple_Type Dep_Args
Dep_Args ::= '@[' Dep_Arg {',' Dep_Arg} ']'
Dep_Arg  ::= variable_id | Dep_Sort_Val
```

Dependent types in Coral are implemented by simple indexed types. Dependent types are explained in greater detail in §5.8.

Example 3.3.3 The following are examples of dependent types representing the number 42 and all strings of length 42:

```
Integer@[42]
String@[42]
```

3.4 Non-Value Types

The types explained in the following paragraphs do not appear explicitly in programs, they are internal and do not represent any type of value directly.

3.4.1 Method Types

A method type is denoted internally as $(Ps) \mapsto R$, where (Ps) is a sequence of parameter names, types and extra properties $(ep_1 : T_1, \ldots, ep_n : T_n)$ for some $n \geq 0$ and R is a (value or method) type. This type represents named or anonymous methods that take arguments named $p_1, \ldots, p_n$ of types $T_1, \ldots, T_n$, have extra properties e and return a result of type R. Names of parameters are either simple identifiers (for positional argument passing) or symbol literals (§1.5.8, for named arguments passing – they make difference between method types with possibly same parameter types, therefore the name is a part of the method type along with the associated parameter type[9]).

Method types associate to the right:[10]
$(Ps_1) \mapsto (Ps_2) \mapsto R$ is treated as $(Ps_1) \mapsto ((Ps_2) \mapsto R)$.

---

[9] This means that, for simplicity, if we have a method with one parameter, which is a named parameter, represented by having its name expressed with a symbol literal, and the parameters have an equivalent type, but different names, the method types are not equivalent.

[10] Like in Haskell or Scala.

A special case are types of methods without any parameters. They are written here as $() \mapsto R$.

Another special case are types of methods without any return type. They are written here as $(Ps) \mapsto ()$. Methods that have this return type do not have an implicit return expressions and an attempt to return a value from it results in a compile-time error.[11]

Method types do not exist as types of values. If a method name is used as a value, its type is implicitly converted to a corresponding function type (§3.3.8).

Extra properties of parameters are as follows: a $*$ for variable arguments, $**$ for any named arguments and & for a captured block argument, or nothing for regular parameters.

Example 3.4.1 The declarations

```
def a: -> Integer // or def a () -> Integer
def b (x : Integer): Boolean
def c (x : Integer): (y : String, z : String) -> String
def d (:x : Integer): Integer
def e (*x : Integer): Integer
def f (Integer): ()
def g (Integer)(Integer): Integer
def h (Integer): (Integer) -> Integer
```

produce the typings

```
a : () ↦ Integer
b : (Integer) ↦ Boolean
c : (Integer) ↦ (String, String) ↦ String
d : (:x Integer) ↦ Integer
e : (*Integer) ↦ Integer
f : (Integer) ↦ ()
g : (Integer) ↦ (Integer) ↦ Integer
g : (Integer) ↦ (Integer) ↦ Integer
```

The difference between the “`g`” and “`h`” functions is that using the chain of return types as in function “`g`”, the function body is automatically curried to return a function that is of type `(Integer) ↦ Integer`. With the function “`h`”, currying has to be implemented manually.

3.4.2 Polymorphic Method Types

A polymorphic method type is the same as a regular method type, but enhanced with a type parameters section. It is denoted internally as $\texttt{:[}tps\texttt{]} \mapsto T$, where $\texttt{:[}tps\texttt{]}$ is a type parameter section $\texttt{:[}a_1 \texttt{ >: } L_1 \texttt{ <: } U_1 \texttt{ , } \ldots \texttt{, } a_n \texttt{ >: } L_n \texttt{ <: } U_n\texttt{]}$ for some $n \geq 0$ and T is a (value or method) type. This type represents (only[12]) named methods that take type arguments $S_1, \ldots, S_n$, for which the lower bounds $L_1, \ldots, L_n$ conform (§3.5.2) to the type arguments and the type arguments conform and the upper bounds $U_1, \ldots, U_n$ and that yield results of type T. No explicit lower bound implies `Nothing` to be the corresponding lower bound, no explicit upper bound implies `Object` to be the corresponding upper bound. As usual, lower bound must conform to the corresponding upper bound.

---

[11] A compile-time error like this may happen during a runtime evaluation as well.

Example 3.4.2 The declarations

```
def empty:[$A]: List:[$A]
def union:[$A <: Comparable:[$A]] (x : Set:[$A],
         xs : Set:[$A]): Set:[$A]
```

produce the typings

```
empty : :[$A >: Nothing <: Object] () ↦ List:[$A]
union : :[$A >: Nothing <: Comparable:[$A]] (Set:[$A],
        Set:[$A]) ↦ Set:[$A]
```

3.4.3 Type Constructors

A type constructor is in turn represented internally much like a polymorphic method type. $\texttt{:[}\pm a_1 \texttt{ >: } L_1 \texttt{ <: } U_1, \ldots, \pm a_n \texttt{ >: } L_n \texttt{ <: } U_n\texttt{]} \mapsto T$ represents a type that is expected by a type constructor parameter. The difference is that the represented internal entity is not a method, but a type, creating higher-kinded types.

3.5 Relations Between Types

We define two relations between types.

Type equivalence	$T \equiv U$	T and U are interchangeable in all contexts.
Conformance	$T <: U$	Type T conforms to type U.

3.5.1 Type Equivalence

Equivalence ($\equiv$) between types is the smallest congruence, such that the following statements are true:

---

[12] Not anonymous.

- If t is defined by a type alias **type** `t` **is** `T`, then t is equivalent to T.
- If a path p has a singleton type q#**singleton-type**, then p#**singleton-type** $\equiv$ q#**singleton-type**.
- Two compound types (§3.3.7) are equivalent, if the sequences of their components are pairwise equivalent, occur in the same order and their refinements are equivalent.
- Two refinements (§3.3.7 & TBD: named refinements) are equivalent, if they bind the same names and the modifiers, types and bounds of every declared entity are equivalent in both refinements. Two equivalent refinements, both or one attached to a compound type, do not imply the compound types to be equivalent. This applies to both anonymous and named refinements.
- Two method types (§3.4.1) are equivalent, if they have equivalent return types, both have the same number of parameters and corresponding parameters have equivalent types and extra properties. Names of parameters matter for method type equivalence only with named parameters.
- Two polymorphic method types (§3.4.2) are equivalent, if they have the same number of type parameters, the return types are equivalent as well as lower and upper bounds of corresponding type parameters.
- Two existential types (§3.3.9) are equivalent, if they have the same number of quantifiers and the quantified types as well as lower and upper bounds of corresponding quantifiers are equivalent.
- Two type constructors (§3.4.3) are equivalent, if they have the same number of type parameters, the return types are equivalent as well as variances, lower and upper bounds of corresponding type parameters.

3.5.2 Conformance

The conformance relation $(<:)$ is the smallest transitive relation that satisfies the following conditions:

- Conformance includes equivalence, therefore if $T \equiv U$, then $T <: U$.
- For every value type T, `Nothing` $<: T <:$ `Object`.
- For every type constructor T with any number of type parameters, `Nothing` $<: T <:$ `Object`.
- A type variable t conforms to its upper bound and its lower bound conforms to t.
- A class type or a parameterized type conforms to any of its base types.

3.5.3 Weak Conformance

For now, *weak conformance* is a relation defined on members of the `Number` type as a relaxation of conformance. The relation is simple: a type t weakly conforms to another type u when u's size contains all values of t (we say that t can be converted to u without precision loss).

Chapter 4

Basic Declarations & Definitions

Syntax:

```
Dcl         ::= [Val_Mod] 'val' Val_Dcl
              | 'var' Var_Dcl
              | 'def' Def_Dcl
              | 'type' Type_Dcl
Pat_Var_Def ::= [Val_Mod] 'val' Pat_Def
              | 'var' Var_Def
              | 'let' ['!'] Let_Def
Def         ::= Pat_Var_Def
              | 'def' Fun_Def
              | 'type' Type_Def
```

A *declaration* introduces names and assigns them types. Using another words, declarations are abstract members, working sort of like header files in C.

A *definition* introduces names that denote terms or types. Definitions are the implementations of declarations.

Both declarations and definitions produce *bindings* that associate type names with type definitions or bounds, and that associate term names with types.

Even more simply put, declarations declare a binding with a type (or type-less), and definition defines the term behind that binding (along with the binding).

4.1 Value Declarations & Definitions

A value declaration `val` x`:` T introduces x as a name of a value of type T. May appear in any block of code and an attempt to use it prior to initialisation with a value is an error. More specifically, a value declaration **`val`** `@`x**`:`** T introduces x as a name of an instance value of type T, and a value declaration **`val`** `@@`x**`:`** T introduces x as a name of a class instance value of type T.

A value definition **`val`** x**`:`** T **`:=`** e defines x as a name of the value that results from evaluation of expression e.

A value in this sense[1] is an immutable variable. A declared value can be assigned just once[2], a defined value is already assigned from its definition.

The value type T may be always omitted, in that case the type is inferred and bound to the name. If a type T is omitted, the type of expression e is assumed. If a type T is given, then e is expected to conform to it (§3.5.2).

Evaluation of the value definition implies evaluation of its right-hand side e, unless it has a modifier **`lazy`** – in that case, evaluation is deferred to the first time the value is accessed.

A *lazy value* is of the form

`lazy val` x**`:`** T **`:=`** e

A lazy value may only be defined, and a value of the same name (binding) may be declared prior to the value definition, but never as a lazy value.

The effect of the value definition is to bind x to the value of e converted to type T.

A *constant value definition* is of the form

`let` x**`:`** T **`:=`** e

where e is an expression that is supposed to be treated as constant in the same block from its occurrence on. Values defined with **`let`** have certain limitations and properties:

1. They can't use patterns as a name.
2. They can't be lazy.
3. They can't be used in a declaration, only in a definition.

---

[1] Everything in Coral is a value – remember, Coral is also a functional language, to some extent.

[2] A similar way that **`final`** variables or members in Java can be assigned just once, but Java furthermore requires that this assignment will happen in every code path, Coral does not impose such requirement.

4. They can be used to redefine a variable (the name is then treated as a new binding in the scope).
5. They can't define (class) instance variables.
6. They can be used in workflows (§6.26).[3]

The type T may be omitted.

Value declarations & definitions with the type T omitted are of the form

```
val x
val @x
val @@x
val x := e
val @x := e
val @@x := e
let x := e
```

A value declaration without any type is basically only declaring the name, so that a binding is introduced and the actual value is for another code to define.[4]

A value definition can alternatively have a pattern (§8.1) as left-hand side (the name). If p is a pattern other than a simple name or a name followed by a colon and a type, then the value definition **val** p `:=` e is expanded as follows:

1. If the pattern p has bound variables $x_1, \ldots, x_n$ for some $n > 1$:

```
val x$ := match e
  when p then (x₁, ..., xₙ)
end match
val x₁ := x$[1]
...
val xₙ := x$[n]
```

2. If p has exactly one unique bound variable x:

```
val x := match e
  when p then x
end match
```

---

[3] A pragma that would turn all values into lazy values might exist, and lazy values should never appear in workflows, so that's why `val` should not be allowed in workflows.

[4] Usually, that another code should be a `constructor` or the class-level block in another file, maybe.

3. If p has no bound variables:

```
match e
  when p then ()
end match
```

Example 4.1.1 The following are examples of value definitions.

```
val pi := 3.14159
val pi : Double := 3.14159
val Some(x) := f()
val x ~> xs := my_list
```

The last two definitions have the following expansions:

```
val x := match f()
  when Some(x) then x
end match

val x$ := match my_list
  when x ~> xs then (x, xs)
end match
val x := x$[1]
val xs := x$[2]
```

The name of any declared or defined value must not end with =.

A value declaration **`val`** $x_1, \ldots, x_n$**`:`** T is a shorthand for the sequence of value declarations **`val`** x_1**`:`** T**`;`** $\ldots$**`;`** **`val`** x_n**`:`** T. A value definition **`val`** $p_1, \ldots, p_n$ `:=` e is a shorthand for the sequence of value definitions **`val`** p_1 `:=` e**`;`** $\ldots$**`;`** **`val`** p_n `:=` e. A value definition **`val`** $p_1, \ldots, p_n : T$ `:=` e is a shorthand for the sequence of value definitions **`val`** $p_1 : T$ `:=` e**`;`** $\ldots$**`;`** **`val`** $p_n : T$ `:=` e.

4.2 Variable Declarations & Definitions

A variable declaration **`var`** x**`:`** T introduces a mutable variable without a defined initial value of type T. More specifically, **`var`** `@`x**`:`** T introduces a mutable instance variable of type T and **`var`** `@@`x**`:`** T introduces a mutable class instance variable of type T.

A variable definition **`val`** x**`:`** T `:=` e defines x as a name of the value that results from evaluation of expression e. The type T can be omitted, in that case the type of expression e is assumed, but not bound to the variable – the variable is only bound to `Object` then. If the type T is given, then e is expected to conform to it (§3.5.2), as well as every future value of the variable.

Variable definitions can alternatively have a pattern (§8.1) as their left-hand side. A variable definition **var** p `:=` e, where p is a pattern other than a simple name followed by a colon and a type, is expanded in the same way (§4.1) as a value definition **val** p `:=` e, except that the free names in p are introduced as mutable variables instead of values.

The name of any declared or defined variable must not end with =.

A variable declaration **var** $x_1, \ldots, x_n$`:` T is a shorthand for the sequence of variable declarations **var** x_1`:` T`;` $\ldots$`;` **var** x_n`:` T. A variable definition **var** $x_1, \ldots, x_n$ `:=` e is a shorthand for the sequence of variable definitions **var** x_1 `:=` e`;` $\ldots$`;` **var** x_n `:=` e. A variable definition **var** $x_1, \ldots, x_n$`:` T `:=` e is a shorthand for the sequence of variable definitions **var** x_1`:` T `:=` e`;` $\ldots$`;` **var** x_n`:` T `:=` e.

4.3 Property Declarations & Definitions

Syntax:

```
Prop_Dcl   ::= 'property' ['(' Prop_Specs ')'] simple_id
               [':' Type]
Prop_Specs ::= Prop_Spec {',' Prop_Spec}
Prop_Spec  ::= ([Access_Modifier] ('get' | 'set')) | 'weak'
Prop_Def   ::= 'property' ['(' Prop_Specs ')'] simple_id
               [':' Type]
               '{' Prop_Impl {semi Prop_Impl} '}'
Prop_Impl  ::= ('get' [Prop_Get_Impl])
               | ('set' [Prop_Set_Impl])
               | ('val' ':=' Expr)
               | ('var' ':=' Expr)
```

A property declaration **property** x`:` T introduces a property without a defined initial value of type T. Property declaration does not specify any actual implementation details of how or where the declared value is stored.

A property definition **property** x`:` T `{`**get** $\ldots$`;` **set** $\ldots$`}` introduces a property with a possibly defined initial value of type T. Property definition may specify implementation details of the behavior and storage of a property, but may as well opt-in for auto-generated implementation, which is:

1. Storage of the property's value is in an instance variable (or a class instance variable in case of class properties) of the same name as is the name of the property: **property** x is stored in an instance variable `@`x.

2. Properties defined with only **`get`** are stored in immutable instance variables (§4.1).
3. Properties defined with **`set`**[5] are stored in mutable instance variables (§4.2).
4. Properties defined with **`weak`** are stored as weak references. A property **`property`** x`:` T is stored in an instance of type `Weak_Reference:[`T`]`.

Declaring a property x of type T is equivalent to declarations of a *getter function* x and a *setter function* x_=, declared as follows:

```
def x (): T; end
def x_= (y: T): (); end
```

Assignment to properties is translated automatically into a setter function call and reading of properties does not need any translation.

4.4 Type Declarations & Aliases

Syntax:

```
Dcl      ::= 'type' Type_Dcl
Type_Dcl ::= constant_id [Type_Param_Clause] ['>:' Type]
             ['<:' Type]
Def      ::= 'type' Type_Def
Type_Def ::= constant_id [Type_Param_Clause] ':=' Type
```

A *type declaration* **`type`** t`:[`tps`] >:` L `<:` U declares t to be an abstract type with lower bound type L and upper bound type U. If the type parameter clause `:[`tps`]` is omitted, t abstracts over first-order type, otherwise t stands for a type constructor that accepts type arguments as described by the type parameter clause.

A *type alias* **`type`** t `:=` T defines t to be an alias name for the type T. Since—for type safety and consistence reasons—types are constant and can not be replaced by another type when bound to a constant name, type aliases are permanent. A type remembers the first given constant name, no alias can change that. The left hand side of a type alias may have a type parameter clause, e.g. **`type`** t`:[`tps`] :=` T. The scope of a type parameter extends over to the right hand side T and the type parameter clause tps itself.

---

[5] It is also possible to declare/define properties that are **`set`**-only. That makes them *write-only*, as opposed to *read-only* properties with **`get`**-only.

It is an error if a type alias refers recursively to the defined type constructor itself.

Example 4.4.1 The following are legal type declarations and aliases:

```
type Integer_List := List:[Integer]
type T <: Comparable:[T]
type Two:[$A] := Tuple_2:[$A, $A]
type My_Collection:[+$X] <: Iterable:[$X]
```

4.5 Type Parameters

Syntax:

```
Type_Param_Clause  ::= ':[' Variant_Type_Param
                       {',' Variant_Type_Param} ']'
Variant_Type_Param ::= {Annotation} ['+' | '-'] Type_Param
Type_Param         ::= (constant_id | '_') [Type_param_Clause]
                       ['>:' Type] ['<:' Type]
```

Type parameters appear in type definitions, class definitions and function definitions.

The most general form of a first-order type parameter is `@{`a_1`}`$\ldots$`@{`a_n`}` $\pm$ t `>:` L `<:` U. L is a lower bound and U is an upper bound. These bounds constrain possible type arguments for the parameter. It is an error if L does not conform to U.[6] Then, $\pm$ is a *variance* (§4.6), i.e. an optional prefix of either + or -. The type parameter may be preceded by one or more annotation applications (§6.15 & §10).

Example 4.5.1 The following are some well-formed type parameter clauses:

```
:[$S, $T]
:[@{Specialized} $S, $T]
:[$Ex <: Raisable]
:[$A <: Comparable:[$B], $B <: $A]
```

4.6 Variance of Type Parameters

Variance annotations indicate how instances of parameterised types relate with respect to subtyping (§3.5.2). A “+” variance indicates a covariant dependency, a “-” variance indicates a contravariant dependency, and an empty variance indicates an invariant dependency.

---

[6] This is sometimes detectable as soon as during compilation.

A variance annotation constrains the way the annotated type variable may appear in the type or class which binds the type parameter.

4.7 Function Declarations & Definitions

Syntax:

```
Dcl           ::= 'def' Fun_Dcl 'end' ['def']
                | 'message' Fun_Dcl 'end' ['message']
                | 'function' Fun_Dcl 'end' ['function']
Fun_Dcl       ::= Fun_Sig ':' Type
Def           ::= 'def' Fun_Def 'end' ['def']
                | 'def' Fun_Alt_Def
                | 'method' Fun_Def 'end' ['method']
                | 'method' Fun_Alt_Def
                | 'function' Fun_Def 'end' ['function']
                | 'function' Fun_Alt_Def
Fun_Def       ::= Fun_Sig [':' Type] [Fun_Dec] semi Expr
Fun_Alt_Def   ::= Fun_Sig [':' Type] ':=' Expr
Fun_Dec       ::= [semi] 'declare' Expr [semi] 'begin'
Fun_Sig       ::= Function_Path [Fun_Tpc] Param_Clauses
Fun_Tpc       ::= ':[' Type_Param {',' Type_Param} ']'
Function_Path ::= function_id
                | 'self' '.' function_id
                | variable_id '.' function_id
Param_Clauses ::= {Param_Clause} ['(' 'implicit' Params ')']
Param_Clause  ::= '(' [Params] ')'
Params        ::= Param {',' Param}
Param         ::= {Annotation} [Param_Extra] variable_id
                [':' Param_Type] [':=' Expr]
                | Literal
Param_Extra   ::= [Param_Io] [Param_Rw] ['*' | '**' | '&'] [':']
Param_Io      ::= 'in' | 'out' | 'in' '+' 'out'
Param_Rw      ::= 'val' | 'var'
Param_Type    ::= Type | '$' constant_id | '=>' [Type | '_']
```

A function declaration has the form of **`def`** f $psig$`:` T, where f is the function’s name, $psig$ is its parameter signature and T is its return type.

A function definition **`def`** f $psig$`:` T `:=` e also includes a *function body* e, i.e. an expression which defines the functions’s return value. A parameter signature consists of an optional type parameter clause `:[`tps`]`, followed by zero or more value parameter clauses `(`ps_1`)`...`(`ps_n`)`. Such a declaration or definition introduces a value with a (possibly polymorphic) method type, whose parameter types and return types are as given.

Multiple parameter clauses render curried functions.

The type of the function body is expected to conform (§3.5.2) to the function's declared result type, if one is given.

An optional type parameter clause tps introduces one or more type parameters, possibly with bounds. The scope of a type parameter includes the whole signature, including any of the type parameter bounds as well as the function body, if present.

A value parameter clause ps consists of zero or more formal parameter bindings, such as x`:` T or x`:` T `:=` e, which bind value parameters and associate them with their types. Each value parameter declaration may optionally define a default value expression. The value expression is represented internally by an invisible function, which gets called when the function matched the function call and an explicit value for the parameter was not provided.

The order in which different kinds of value parameters may appear is as follows:

1. n mandatory positional parameters (§4.7.4): x`:` T, where $n \geq 0$.
2. n optional positional parameters (§4.7.5): x`:` T `:=` e, where $n \geq 0$.
3. n repeated parameters (§4.7.6): $*x$`:` T, where $0 \leq n \leq 1$.
4. n post mandatory positional parameters (§4.7.4): x`:` T, where $n \geq 0$.
5. n named parameters (§4.7.7): `:`x `:` T or `:`x `:` T `:=` e, where $n \geq 0$.
6. n captured named parameters (§4.7.7): $**x$`:` T, where $0 \leq n \leq 1$.
7. n captured block parameters (§4.7.8): &x`:` T, where $0 \leq n \leq 1$.

For every parameter $p_{i,j}$ with a default value expression, a function named

`default$`n

is generated inside the function, inaccessible for user programs. Here, n denotes the parameter's position in the method declaration. These methods are parameterized by the type parameter clause `:[`tps`]` and all value parameter clauses `(`ps_1`)`...`(`ps_{i-1}`)` preceeding $p_{i,j}$.

The scope of a formal value parameter name x comprises all subsequent parameter clauses, as well as the method return type and the function body, if they are given.

Example 4.7.1 In the method

```
def compare:[$T](a: $T := 0)(b: $T := a) := (a = b)
```

the default expression `0` is type-checked with an undefined expected type. When applying `compare()`[7], the default value `0` is inserted and `T` is instantiated to `Number`. The functions computing the default arguments have the forms[8]:

```
def default$1:[$T]: Number := 0
def default$2:[$T](a: $T): $T := a
```

Parameters may be optionally flagged with **`var`** and **`val`** keywords, modifying their mutability inside the method. Some combinations are disallowed, as explained in the following sections. All parameters are implicitly **`val`**-flagged, unless the parameter kind implies a **`var`** flag, such as an output parameter (§4.7.3).

4.7.1 By-Name Parameters

Syntax:

```
Param_Type ::= '=>' [Type | '_']
```

The type of a value parameter may be prefixed by "=>", e.g. x`: =>` T. This indicates that the corresponding argument is not evaluated at the point of function application, but instead is evaluated at each use within the function. That is, the argument is evaluated using *call-by-name*.

The by-name modifier is disallowed for output parameters (§4.7.3 & §6.7.3) and for implicit parameters (§7.2). The by-name modifier implies **`val`** parameter and is disallowed for **`var`** parameters.

By-name parameters require a specific application (§6.7.2). A by-name parameter bound to a wildcard type "_" matches any type of by-name argument.

By-name parameters with default value expressions evaluate the default value expression each time the parameter is accessed, unlike optional parameters that evaluate the default value expression only once.

By-name parameters imply the **`val`** flag, and disallow the **`var`** flag.

---

[7] Without any explicit arguments.

[8] See, at the moment `default$2` is called, the parameter `a` is already computed and passed as an argument to it.

4.7.2 Explicit Parameters

Syntax:

```
Param ::= Literal
```

The parameter may be specified by its literal value. Such parameters may only appear where positional parameters may appear. The type of the parameter is the type of the literal value. Methods with explicit parameters are preferred during method resolution to methods with the same parameter types (§6.7), but it is an error if more than one method with explicit parameters match the function application.

The recommendation for usage of these parameters are:

- Use explicit parameters with unary methods only.
- If the value is a collection, use an empty collection literal only.

Example 4.7.2 Sample methods that use explicit parameters:

```
def factorial (0) := 1
def factorial (x) := x * factorial(x - 1)
```

Since the parameter has no name to bind to, it is not accessible inside the method body.

4.7.3 Input & Output Parameters

Syntax:

```
Param_Io ::= 'in' | 'out' | 'in' '+' 'out'
```

If no input/output parameter specifier is explicitly available, then the parameter is implicitly an input parameter. Output parameters require a specific application (§6.7.3).

Output parameters imply `var` parameter and is disallowed for `val` parameters. Input parameters that are not output parameters at the same time can be both `var` and `val`.

4.7.4 Positional Parameters

Positional parameters are of the forms:

```
x: T
var x: T
val x: T
in x: T
in var x: T
in val x: T
out x: T
out var x: T
in+out x: T
in+out var x: T
x: => T
val x: => T
in x: => T
in val x: => T
```

Positional parameters may not have any modifiers, except for input/output modifiers (§4.7.3) and by-name (§4.7.1). Positional parameters can't have any default value expressions.

4.7.5 Optional Parameters

Optional parameters are of the forms:

```
x: T := e
var x: T := e
val x: T := e
in x: T := e
in var x: T := e
in val x: T := e
x: => T := e
val x: => T := e
in x: => T := e
in val x: => T := e
```

Optional parameters may not have any modifiers, except for input/output modifiers (§4.7.3)[9] and by-name modifier (§4.7.1). Optional parameters have a *default value expressions* and may appear between positional parameters, being followed by any number of positional parameters (including no more positional parameters at all), or being followed by repeated parameters and then positional parameters (§6.7.1). Optional parameters are disallowed for output parameters and repeated parameters.

---

[9] Optional parameters are always "input", so that declaration is always redundant.

4.7.6 Repeated Parameters

Repeated parameters are of the forms:

```
*x: T
var *x: T
val *x: T
in *x: T
in var *x: T
in val *x: T
```

Between optional parameters and the tailing positional parameters may be a value parameter prefixed by "$*$", e.g. `(..., *x: T)`. The type of such a *repeated* parameter inside the method is then a list type `List:[T]`. Methods with repeated parameters take a variable number of arguments of type T between the optional parameters block and the last positional parameters block, including no arguments at all (an empty list is then its value).

If a repeated parameter is flagged with **`val`**, the parameter itself is immutable, not the elements of the list. Repeated parameters are **`val`**-flagged implicitly, unless explicitly flagged as **`var`**, to protect the captured elements from accidental overwrite.

Example 4.7.3 The following method definition computes the sum of the squares of a variable number of integer arguments.

```
def sum (*args: Integer): Integer
declare
  var result := 0
begin
  for arg in args loop
    result += arg ** 2
  end loop

  result
end
```

The following applications of this method yield `0`, `1`, `14`, in that order.

```
sum
sum 1
sum 1, 2, 3
```

Furthermore, assume the definition:

```
val xs := %[1, 2, 3]
```

The following application of the method `sum` is not resolved:[10]

```
sum xs // Error: method match not found, wrong arguments
```

By contrast, the following application is well-formed and yields again the result `14`:

```
sum *xs
```

4.7.7 Named Parameters

Named parameters are of the forms:

:x **:** T
var :x **:** T
val :x **:** T
in :x **:** T
in var :x **:** T
in val :x **:** T
out :x **:** T
out var :x **:** T
in+out :x **:** T
in+out var :x **:** T
:x **:** T **:=** e
var :x **:** T **:=** e
val :x **:** T **:=** e
in :x **:** T **:=** e
in var :x **:** T **:=** e
in val :x **:** T **:=** e
:x **:** => T
val :x **:** => T
in :x **:** => T
in val :x **:** => T
:x **:** => T **:=** e
val :x **:** => T **:=** e
in :x **:** => T **:=** e
in val :x **:** => T **:=** e

Captured named parameters are of the form:

x **: T
var **x **:** T
val **x **:** T
in **x **:** T

---

[10] Unless there is an overloaded version of the method that accepts a list of integers as its parameter.

in var $**x$ **:** T
in val $**x$ **:** T

Named parameters are a way of allowing users of the method to write down arguments in any order, provided that their name is given at function application (§6.7 & §6.7.1). Named parameters may have a default value expression and may be both input and output. Named parameters are disallowed for repeated parameters. Named parameters inside the method is then accessible the same way as a positional parameter.

Captured named parameters are capturing any other applied named parameters that were not captured by their explicit declaration (§6.7.1). They are declared after the block of named parameters, prefixed by "**", e.g. `(…, **`x`: `T`)`. The type of such a captured named parameter inside the method is then a dictionary type `Dictionary:[Symbol, `T`]`. Methods with captured named parameters take a variable number of named arguments of type T mixed with other named arguments and before the captured block parameter. Captured named parameters are disallowed for repeated parameters, output parameters and by-name parameters.

If a captured named parameter is flagged with **`val`**, the parameter itself is immutable, not the elements of the dictionary. Captured named parameters are **`val`**-flagged implicitly, unless explicitly flagged as **`var`**, to protect the captured elements from accidental overwrite.

4.7.8 Captured Block Parameter

Captured block parameters are of the forms:

&x
&x**:** T
in &x
in var &x
in val &x
in &x**:** T
in var &x**:** T
in val &x**:** T

Captured block parameter is a way to capture an applied block that is otherwise passed in implicitly as a function into **`yield`** expressions . The forms of captured block parameters explicitly denote the case without the block's function type, since block parameters receive any arguments and those missing are implicitly set to **`nil`**. The function type of the block may be used to further constrain the applied block argument, but is not used during method resolution (§6.7).

It is an error if a block parameter type T is provided and it is not a function type (§3.3.8). It is also an error if the applied block argument does not accept the arguments declared by the type T, or if the block would not return a value conforming to the return type required by the type T. The applied block argument may accept more arguments than required by T, however, these will be set implicitly to **`nil`**. Also, the applied block argument may itself require less constrained parameter types, in which case the arguments applied to it must (and will) always conform (§3.5.2) to the block's parameter requirements. Whether the function type T has a return type or not is irrelevant.

If a block parameter type T is given, then the applied block argument must accept parameters, such that the parameter constrains declared by the function type T conform to the parameter constrains declared by the applied block: the parameters of the applied block must be the same or less restrictive than those declared by T – they must be pairwise contravariant or invariant, never covariant (§4.6).

4.7.9 Parameter Kind Combinations

This section is *normative*.

Users should design their functions in a way that makes them having as few parameters as possible.

In this spirit, an ideal count of parameters lies between 0 and 2 (nullary, unary and binary functions). Named parameters should be used with only up to one positional[11] parameter. Functions with more than 4 different parameters should be avoided.

4.7.10 Method Types Inference

Parameter Type Inference. Functions that are members of a class C may define parameters without type annotations. The types of such parameters are inferred as follows. Say, a method m in a class C has a parameter p which does not have a type annotation. We first determine methods m' in C that might be overridden (§5.1.6) by m, assuming that appropriate types are assigned to all parameters of m whose types are missing. If there is exactly one such method, the type of the parameter corresponding to p in that method—seen as a member of C—is assigned to p. It is an error if there are several such overridden methods m'. If there is none[12] (m does not override any m' known at compile-time), then the parameters are inferred to be of type `Object`.

---

[11] In this sense, optional and repeated parameters are also positional, because they are applied on a particular numbered position rather than named.

[12] Detected at compile-time. Dynamically added overridden methods are not used with type inference.

Example 4.7.4 Assume the following definitions:

```
protocol I:[$A]
  def f(x: $A)(y: $A): $A
end
class C
  implements I:[Integer]
  def f(x)(y) := x + y
end
```

Here, the parameter and return types of `f` in `C` are inferred from the corresponding types of `f` in `I`. The signature of `f` in `C` is thus inferred to be

```
  def f(x: Integer)(y: Integer): Integer
```

Return Type Inference. A class member definition m that overrides some other function m' in a base class of C may leave out the return type, even if it is recursive. In this case, the return type R' of the overridden function m'—seen as a member of C—is taken as the return type of m for each recursive invocation of m. That way, a type R for the right-hand side of m can be determined, which is then taken as the return type of m. Note that R may be different from R', as long as R conforms to R'. If m does not override any m', then its return type is inferred to be of type `Object`.

Example 4.7.5 Assume the following definitions:

```
protocol I
  def factorial(x: Integer): Integer
end
class C
  implements I
  def factorial(x: Integer) := {
    if x = 0 then 1 else x * factorial(x - 1) end
  }
end
```

Here, it is ok to leave out the return type of `factorial` in `C`, even though the method is recursive.

For any index i let $fsig_i$ be a function signature consisting of a function name, an optional type parameter section, and zero or more parameter sections. Then a function declaration **`def`** $fsig_1, \ldots, fsig_n$**`:`** T is a shorthand for the sequence of function declarations **`def`** $fsig_1$**`:`** T**`;`** $\ldots$**`;`** **`def`** $fsig_n$**`:`** T. A function definition **`def`** $fsig_1, \ldots, fsig_n$ **`:=`** e is a shorthand for the sequence of function definitions **`def`** $fsig_1$ `:=` e**`;`** $\ldots$**`;`** **`def`** $fsig_n$ `:=` e. A function definition **`def`** $fsig_1, \ldots, fsig_n : T$ `=` e is a shorthand for the sequence of function definitions **`def`** $fsig_1 : T$ `:=` e**`;`** $\ldots$**`;`** **`def`** $fsig_n : T$ `:=` e.

4.8 Overloaded Declarations & Definitions

An overloaded definition is a set of $n > 1$ function definitions in the same statement sequence that define the same name, binding it to types $T_1, \ldots, T_n$, respectively. The individual definitions are called *alternatives*. Overloaded definitions may only appear in the expression sequence of a class-level block. Alternatives always need not to specify the type of the defined entity completely.

Overloaded function definitions have strong impact on method resolution. It is an error if a single set of arguments may be applied type-safely to multiple overloaded functions – to resolve this, explicit argument types have to be applied (§6.7).

Overloaded functions generate new functions that internally merge the overloaded functions into one, which then resolves the correct overloaded function based on the applied types.

Example 4.8.1 Assume the following overloaded declarations

```
def double (arg: Number): Number
def double (arg: Integer): Integer
```

Now, the following method application is invalid, because two functions resolve to the same arguments set:

```
// variable-less:
double 42
```

Now, with explicitly applied argument types, the following method applications are correct:

```
// variable-less:
double 42 as Integer

// with a variable:
var number: Integer := 42
double number

var number := 42 // the type is inferred
double number
```

4.9 Use Clauses

Syntax:

```
Use             ::= 'use' Use_Expr
Use_Expr        ::= (Container_Path | Stable_Id) '::' Import_Expr
Import_Expr     ::= Single_Import
                  | '{' Import_Exprs '}'
                  | '_'
Import_Exprs    ::= Single_Import {',' Single_Import} [',' '_']
Single_Import   ::= importable_id ['as' [constant_id | '_']]
Container_Path  ::= Module_Path ['::' Constant_Path]
                  | Constant_Path
Module_Path     ::= Module_Selector {'::' Module_Selector}
Constant_Path   ::= Const_Selector {'::' Const_Selector}
Module_Selector ::= constant_id [Vendor_Arg]
Const_Selector  ::= constant_id
Vendor_Arg      ::= '~[' vendor_domain ']'
vendor_domain   ::= variable_id {'.' variable_id}
```

A use clause has the form **`use`** p`::`I, where p is a path to the containing type of the imported entity (either a module or another class), and I is an import expression. The import expression determines a set of names (or just one name) of *importable members*[13] of p, which are made available without full qualification, e.g. as an unqualified name. A member m of p is *importable,* if it is *visible* from the import scope and not object-private (§5.2). The most general form of an import expression is a list of *import selectors*

{ x_1 **as** y_1 , …, x_n **as** y_n, _ }

for $n \geq 0$, where the final wildcard "_" may be absent. It makes available each importable member p`::`x_i under the unqualified name y_i. I.e. every import selector x_i **as** y_i renames (aliases) p`::`x_i to y_i. If a final wildcard is present, all importable members z of p other than $x_1, \ldots, x_n, y_1, \ldots, y_n$ are also made available under their own unqualified names.

Import selectors work in the same way for type and term members. For instance, a use clause **`use`** p`::`$\{x$ **as** $y\}$ renames the term name p`::`x to the term name y and the type name p`::`x to the type name y. At least one of these two names must reference an importable member of p.

If the target name in an import selector is a wildcard, the import selector hides access to the source member. For instance, the import selector x_i **as** _ "renames" x to the wildcard symbol, which basically means discarding the name, since _ is not a readable name[14], and thereby effectively prevents unqualified access to x. This is useful if there is a final wildcard in the same import selector list, which imports all members not mentioned in previous import selectors, to selectively not import some members.

---

[13] Dynamically created members are not importable, since the compiler has no way to predict their existence.

The scope of a binding introduced by an import-clause starts immediately after the import clause and extends to the end of the enclosing scope and all nested scopes.

Several shorthands exists. An import selector may be just a simple name x, in which case, x is imported without renaming, so the import selector is equivalent to x **`as`** y. Furthermore, it is possible to replace the whole import selector list by a single identifier of wildcard. The use clause **`use`** p`::`x is equivalent to **`use`** p`::{`x`}`, i.e. it makes available without qualification the member x of p. The use clause **`use`** p`::_` is equivalent to **`use`** p`::{_}`, i.e. it makes available without qualification all importable members of p (this is analogous to **`import`** p`.`$*$ in Java or **`import`** p`._` in Scala).

Example 4.9.1 Consider the object definition:

```
object M
  def z := 0
  def one := 1
  def add (x: Integer, y: Integer): Integer := x + y
end
```

Then the block

```
{ use M::{one, z as zero, _}; add (zero, one) }
```

is equivalent to the block

```
{ M.add (M.z, M.one) } .
```

---

[14] Meaning, it is not possible to use "_" as a variable to read from, it never has any value.

Chapter 5

Classes & Objects

Syntax:

```
Const_Def ::= {Annotation} 'class' Class_Def 'end' ['class']
            | {Annotation} 'class' Class_Alt_Def
            | {Annotation} 'object' Obj_Def 'end' ['object']
            | {Annotation} 'object' Obj_Alt_Def
            | {Annotation} 'module' Module_Def 'end' ['module']
            | {Annotation} 'module' Module_Alt_Def
            | {Annotation} 'trait' Trait_Def 'end' ['trait']
            | {Annotation} 'trait' Trait_Alt_Def
            | {Annotation} 'protocol' Pro_Def 'end' ['protocol']
            | {Annotation} 'protocol' Pro_Alt_Def
            | {Annotation} 'interface' Ifc_Def 'end' ['interface']
            | {Annotation} 'type' Const_Type_Def 'end' ['type']
```

Classes (§5.3), objects (§5.4), modules (§5.5 & §9.2), traits (§5.3.4), protocols (§5.3.6), interfaces (§5.3.7) and constant types (unions §5.6, enums §5.7, range types §5.8.3, units of measure §5.9 & record types §5.10 are all defined in terms of *templates* (§5.1). Their definitions are basically a function that handles internally the definition of each type. As such, these class-level blocks can even have local variables that may appear in closures.

5.1 Templates

5.1.1 Constructor Invocations & Primary Constructor

5.1.2 Metaclasses & Eigenclasses

Metaclasses. A *metaclass* is a class whose instances are classes. Just as an ordinary class defines the behavior and properties of its instances, a metaclass defines the behavior of its class. Classes are first-class citizens in Coral.

Everything is an object in Coral. Every object has a class that defines the structure (i.e. the instance variables) and behavior of that object (i.e. the messages the object can receive and the way it responds to them). Together this implies that a class is an object and therefore a class needs to be an instance of a class (called metaclass).

Class methods actually belong to the metaclass, just as instance methods actually belong to the class. All metaclasses are instances of only one class called `Metaclass`, which is a subclass of the class `Class`.

In Coral, every class (except for the root class `Object`) has a superclass. The base superclass of all metaclasses is the class `Class`, which describes the general nature of classes.

The superclass hierarchy for metaclasses parallels that for classes, except for the class `Object`. The following holds for the class `Object`:

```
Object.class = Class
Object.superclass = nil
```

Classes and metaclasses are “born together”. Every `Metaclass` instance has a method `this_class`, which returns the conjoined class.

Eigenclasses. Coral further purifies the concept of metaclasses by introducing *eigenclasses*, borrowed from Ruby, but keeping the `Metaclass` known from Smalltalk-80. Every metaclass is an eigenclass, either to a class, to a terminal object, or to another eigenclass[1].

Eigenclasses are manipulated indirectly through various syntax features of Coral, or directly using the `eigenclass` method. This method can possibly trigger creation of an eigenclass, if the receiver of the `eigenclass` message did not previously have its own (singleton) eigenclass (because it was a terminal

---

[1] Eigenclasses of eigenclasses (“higher-order” eigenclasses) are supposed to be rarely needed, but are there for conceptual integrity, establishing infinite regress.

Table 5.1: Of objects, classes & eigenclasses

Classes	Eigenclasses of classes	Eigenclasses of eigenclasses
Terminal objects	Eigenclasses of terminal objects	

object whose eigenclass was a regular class, or the reciever was an eigenclass itself).

Another way to access an eigenclass is to use the `class << obj; ...; end` construct. The block of code inside runs is evaluated in the scope of the eigenclass of `obj`.

Example 5.1.1 Direct access to the eigenclass of any object, here a class' eigenclass:

```
class A
begin
  class << self
    def a_class_method
      "A.a_class_method"
    end def
  end
end class
```

Class `A` uses the `class << obj; ...; end` construct to get direct access to the eigenclass. The keyword `self` inside the block is bound to the eigenclass object.

Example 5.1.2 Alternative direct access to the eigenclass of any object, here a class' eigenclass:

```
class B
begin
  self.eigenclass do
    def a_class_method
      "B.a_class_method"
    end def
  end
end class
```

Class `B` uses the `eigenclass` method, which—given a block—evaluates the block in the scope of the eigenclass of `self`, which is bound to the class `B`. The keyword `self` inside the block is again bound directly to the eigenclass object.

Example 5.1.3 Indirect access to the eigenclass using a singleton method definition:

```
class C
begin
  def self.a_class_method
    "C.a_class_method"
  end def
end class
```

Class `C` uses singleton method definition to add methods to the eigenclass of the class `C`. The keyword **`self`** is bound to the class object in the class-level block and in the new method as well, but the eigenclass is accessed only indirectly.

Example 5.1.4 Indirect access to the eigenclass using a class object definition:

```
class D
begin
  object D
    def a_class_method
      "D.a_class_method"
    end def
  end object
end class
```

Class `D` uses the recommended approach, utilizing standard ways of adding methods to the eigenclass of the class `D`. Here, the eigenclass instance itself is not accessed directly.

Example 5.1.5 Alternative indirect access to the eigenclass using a class object definition:

```
object E
  def a_class_method
    "E.a_class_method"
  end def
end object
```

Class `E` uses a similar recommended approach, utilizing standard ways of adding methods to the eigenclass of the class `E` and neither declaring nor defining anything for its own instances. Here, the eigenclass instance itself is not accessed directly.

5.1.3 Class Linearization

The classes reachable through transitive closure of the direct inheritance relation from a class C are called the *base classes* of C. Because of traits, the inheritance relationship on base classes forms in general a directed acyclic graph. A linearization of this graph is defined as follows.

Definition 5.1.6 Let base classes of a class C be the list of every superclass of C with every trait that these classes include and/or prepend and every protocol that these classes implement. Let C be a class with base classes C_1 `with` C_2 `with` $\ldots$ `with` C_n. The *linearization* of C, $\mathcal{L}(C)$ is defined as follows:

$$\mathcal{L}(C) \quad = \quad C\ ,\ \mathcal{L}(C_n) \vec{+} \ldots \vec{+} \mathcal{L}(C_1)$$

Here $\vec{+}$ denotes concatenation, where elements of the right operand replace identical elements of the left operand:

$$\begin{array}{lcll} \{a, A\} \vec{+} B & = & a, (A \vec{+} B) & \textbf{if } a \notin B \\ & = & A \vec{+} B & \textbf{if } a \in B \end{array}$$

Example 5.1.7 Consider the following class definitions.[2]

```
class Abstract_Iterator extends Object; ... end
trait Rich_Iterator extends Abstract_Iterator; ... end
class String_Iterator extends Abstract_Iterator; ... end
class Iterator extends String_Iterator with Rich_Iterator; ... end
```

Then the linearization of class `Iterator` is

```
{ Iterator, Rich_Iterator, String_Iterator, Abstract_Iterator,
  Object }
```

Note that the linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur. Also note that whether a trait is included or prepended is irrelevant to linearization, but essential to function applications (§6.7).

5.1.4 Inheritance Trees & Include Classes

Include classes. A mechanism that allows arbitrary including and prepending of trait into classes and inheritance binary trees[3] uses a transparent structure called *include class*. Include classes are always defined indirectly.

---

[2] Here we say “class”, but that term includes now traits as well.

[3] Yes, trees, not chains: prepended traits make the inheritance game stronger by forking the inheritance chain at each class with prepended traits, forming a shape similar to a rake.

Every class has a link to its superclass. In fact, the link is made up of an include class structure, which itself holds an actual link to the superclass. That superclass has its own link to its superclass and this chain goes forever until the `Object` class is encountered, which has no superclass.

Included traits. When a trait M is included into a class, a new include class Im_i is inserted between the target class and the include class Is that holds a link to its superclass (or to a previously included trait Im_{i+1}). This include class Im_i then holds a link to the included trait. Every class that includes the trait M is available via the `included_in` method of M.

If a trait M is already (included in or prepended to)[4] any of the superclasses, then it is not included again. Included traits act like superclasses of the class they are included in, and they *overlay* the superclasses.

Example 5.1.8 A sample trait schema:

```
class C extends D with Some_Trait
end

C → [Some_Trait] → [D]
D → [Object]
```

Include classes are depicted by the brackets, with their link value inside. Note that include classes only know their super-type (depicted by the arrow "→") and their link value (inside brackets).

Prepended traits. When a trait M is prepended to a class, a new include class Im_i is inserted between the target class and its last prepended trait, if any. Prepended traits are stored in a secondary inheritance chain just for prepended traits, forming an inheritance tree. Every class that has M prepended is available via the `prepended_in` method of M. The effect of prepending a trait in a class or a trait is named *overlaying*.

If a trait M is already prepended to any of the superclasses, it has to be prepended again, since the already prepended traits of superclasses are in super-position to the class or trait that gets M prepended. Prepended traits of superclasses do not *overlay* child classes. Prepended traits are inserted into the inheritance tree more like subclasses than superclasses.

---

[4] This is important since both included and prepended traits act like superclasses to every subclass.

Nested includes. When a trait M itself includes one or more traits $M_1, \ldots, M_n$, then these included traits are inserted between the included trait M and the superclass, unless they are already respectively included by any of the superclasses. If M with included $M_1, \ldots, M_n$ is prepended to C, then $M_1, \ldots, M_n$ are inserted before the superclass of C, if not already included in any of the superclasses. It is an error if nested includes form a dependency cycle: any *auto-included*[5] trait must not require to include a trait that triggered the auto-include.

Nested prepends. When a trait M itself prepends one or more traits $M_1, \ldots, M_n$, then nothing happens to the class or the trait that M is included in. If M is prepended to C, then every prepended trait in $M_1, \ldots, M_n$ is prepended to C in this way:

1. M is inserted as the head of the prepend chain in an include class.
2. For each $1 \leq i \leq n$, M is switched with previously prepended traits until M_i is closer to the chain head than M, if M_i is contained in the chain, or else M_i is inserted as the head of the prepend chain.
3. If M_i includes any other traits, these are included in the same way as nested included traits.
4. Check for prepend cycles – if any *auto-prepended*[6] trait requires to prepend a trait that triggered the auto-prepend, then it is an error.

Traits & Metaclasses. Since traits may contain “class” methods as well as instance methods, all the operations with include classes are mirrored on the respective metaclasses.[7]

5.1.5 Class Members

A class C defined by class definition can define members in its class-level block, can inherit members from all parent classes[8] and included traits, and can have its members overlayed by all prepended traits.

Member definitions fall basically into two categories: *concrete* & *abstract*. Members of the class C are either *directly defined*, *overlayed* or *inherited*.

---

[5] Unlike in Ruby, Coral does not need concerns to employ automatic injection of dependencies. Every trait in Coral is a concern.

[6] Again, this is unlike in Ruby, which does not automatically prepend traits.

[7] This makes Coral in no need of constructs like `module ClassMethods`, known from Ruby.

[8] Including an interface and any number of protocols.

Concrete members are those that have a definition, abstract members are those that are only declared without any subsequent or preceding definition.[9]

Members are instance variables, class instance variables, methods & messages.

5.1.6 Overriding

A member M of a class C that matches a member M' of a base class of C is said to override that member. In this case the binding of the overriding member M must conform (§3.5.2) to the binding of the overridden member M'. Furthermore, the following restrictions on modifiers apply to M and M':

- If M is labeled **`private:`** `[`C`]` for some enclosing class or module C, then M' must be labeled **`private:`** `[`C'`]`, where C' equals C or C is contained in C'.
- If M is labeled **`protected`**, then M' must also be labeled **`protected`**.
- If M' is labeled **`protected`**, then M' must also be labeled **`protected`** or **`public`**.
- If M' is labeled **`private`**, then M' must be labeled **`private`**, **`protected`** or **`public`**.
- If M' is not an abstract member, then M may be annotated with `@[Override]`.

To generalize the conditions, the modifier of M must be the same or less restrictive than the modifier of M'.

5.1.7 Inheritance Closure

Let C be a class type. The *inheritance closure* of C is the smallest set $\mathcal{S}$ of types such that

- If T is in $\mathcal{S}$, then every type T' which forms syntactically a part of T is also in $\mathcal{S}$.
- If T is a class type in $\mathcal{S}$, then all parents of T are also in $\mathcal{S}$.

It is an error if the inheritance closure of a class type consists of an infinite number of types.

---

[9] There is a special case, when a protocol introduces an abstract member, but the class that implements this protocol already inherited or directly defined the matching member.

5.1.8 Early Definitions

5.2 Modifiers

Syntax:

```
Modifier         ::= Local_Modifier
                   | Access_Modifier
                   | 'override'
Local_Modifier   ::= 'implicit'
                   | 'lazy'
                   | 'final'
                   | 'sealed'
                   | 'abstract'
Access_Modifier  ::= 'public'
                   | ('protected' | 'private') [Access_Qualifier]
Access_Qualifier ::= ':[' (constant_id | 'self') ']'
```

Access modifiers may appear in two forms:

Accessibility flag modifier. Such a modifier appears in a class-level block (§??) alone on a single line. All subsequent members in the same class-level block than have accessibility of this modifier applied to them, if allowed to (does not apply to destructors). Only access modifiers can be used this way.

Directly applied modifier. Such a modifier appears on a line preceding a member to which the modifier is solely applied, or a list of arguments with symbols that the modifier will be applied to. A directly applied modifier expression has a return type of `Symbol`, so that it may be used as a regular function and chained.

Example 5.2.1 An example of an accessibility flag modifier:

```
class C
  public
    def hello; end
  private
    def private_hello; end
end class
```

Example 5.2.2 An example of directly applied modifier:

```
class C
  def hello; end
  def private_hello; end
  def salute; end
  public :hello
  private :private_hello, :salute
  protected def goodbye; end
end class
```

By default[10], the **`public`** access modifier affects every member of the class type, except for instance variables and class instance variables, which are object-private (`private:[self]`).

Modifiers affect the accessibility and usage of the identifiers bound by them. If several modifiers are given, their order does not matter, but the same modifier may not occur more than once and combinations of **`public`**, **`protected`** & **`private`** are not allowed (using them as accessibility flag modifiers overwrites the previous accessibility, not combines them). If a member declaration has a modifier applied to it, then the subsequent member definition has the same modifier already applied to it as well, without the need to explicitly state that. It is an error if the modifier applied to the member definition would contradict the modifier applied to the member declaration.

Accessibility modifiers can not be applied to instance variables and class instance variables (both declarations and definitions). These are by default *instance-private*. This is a sort of relaxation in access restriction, say, every method that is at least *public* and at most *object-private* restricted, and that has the instance as a receiver, can access the instance variable or the class instance variable. Any other method that does not have the particular instance as the receiver, does not have any access to the instance variable or the class instance variable, even if the method is a method of the same class as the particular instance.

The rules governing the validity and meaning of a modifier are as follows:

- The **`private`** access modifier can be used with any declaration or definition in a class. Such members can be accessed only from within the directly enclosing class, the class object (§5.4) and any member of the directly enclosing class, including inner classes. They are not inherited by subclasses and they may not override definitions in parent classes.

  The modifier may be *qualified* with an identifier C (e.g. **`private:`**`[`C`]`) that must denote a class or a module enclosing the declaration or definition. Members labeled with such a modifier are accessible respectively only from code inside the module C or only from code inside the class C and the class object C (§5.4).

---

[10] That is, without any explicit modifier being applied.

A different form of qualification is **`private:[self]`**. A member M marked with this modifier is called *object-private*; it can be accessed only from within the object in which it is defined. That is, a selection p**.**M is only legal if the prefix ends with **`this`** or **`self`** and starts with O for some class O enclosing the reference. . Moreover, the restrictions for unqualified **`private`** apply as well.

Members marked **`private`** without any qualifier are called *class-private*. A member *is private* if it is either class-private or object-private, but not if it is marked **`private:`** [C], where C is an identifier, in the latter case the member is called *qualified private*.

Class-private and object-private members must not be **`abstract`**, since there is no way to provide a concrete implementation for them, as private members are not inherited. Moreover, modifiers **`protected`** & **`public`** can not be applied to them (that would be a contradiction[11]), and the modifier **`override`** can not be applied to them as well[12].

- The **`protected`** access modifier can be used with any declaration or definition in a class. Protected members of a class can be accessed from within:

  - the defining class
  - all classes that have the defining class as a base class
  - all class objects of any of those classes

  A **`protected`** access modifier can be qualified with an identifier C (e.g. **`protected:`** [C]) that must denote a class or module enclosing the definition. Members labeled with such a modifier are *also*[13] accessible respectively from all code inside the module C or from all code inside the class C and its class object C (§5.4).

  A protected identifier x can be used as a member name in a selection r**.**x only if one of the following applies:

  - The access is within the class defining the member, or, if a qualification C is given, inside the module C, the class C or the class object C, or

---

[11] E.g., a member can not be public and private at the same time.

[12] Otherwise, if a private member could override an inherited member, that would mean there is an inherited member that could be overridden, but private members can not override anything: only protected and public members can be overridden. If a member was overriding an inherited member, the parent class would *lose access* to it.

[13] In addition to unqualified **`protected`** access.

  - r ends with one of the keywords **`this`**, **`self`** or **`super`**, or
  - r's type conforms to a type-instance of the class which has the access to x.

  A different form of qualification is **`protected:`**`[`**`self`**`]`. A member M marked with this modifier can be accessed only from within the object in which it is defined, including methods from inherited scope. That is, a selection p**`.`**M is only legal if the prefix ends with **`this`**, **`self`** or **`super`** and starts with O for some class O enclosing the reference. Moreover, the restrictions for unqualified **`protected`** apply.

- The **`override`** modifier applies to class member definitions and declarations. It is never mandatory, unlike in Scala or C# (in further contrast with C#, every method in Coral is virtual, so Coral has no need for a keyword "`virtual`"). On the other hand, when the modifier is used, it is mandatory for the superclass to define or declare at least one matching member (either concrete or abstract).

- The **`override`** modifier has an additional significance when combined with the **`abstract`** local modifier. That modifier combination is only allowed for members of traits.

  We call a member M of a class or trait *incomplete* if it is either abstract (i.e. defined by a declaration), or it is labeled **`abstract`** and **`override`** and every member overridden by M is again incomplete.

  The **`abstract override`** modifier combination does not influence the concept whether a member is concrete or abstract. A member is *abstract* if only a declaration is given for it; it is *concrete* if a full definition is given. This behavior can be turned off only in tests, if needed, and is implicitly turned on.

  The **`abstract override`** modifier combination can be thus used with a full definition in a trait and yet affect the class or trait with which it is used, so that a member access to member **`abstract override`**M, such as **`super.`**M, is legal. But, the **`abstract override`** modifier combination does not need to be applied to a definition, a declaration is good enough for it.

  Additionally, an annotation `@[Override]` exists for class members that triggers only warnings in case the member has no inherited member to override, but does not prevent the class from being created. Thus, the annotation only signals an intention, the keyword modifier signals a requirement.

- The **`abstract`** local modifier is used in class declarations. It is never mandatory for classes with incomplete members or for declarations and definitions. It is implied (and therefore redundant) for traits. Abstract classes can not be instantiated (an exception is raised if tried to do so), unless provided with traits and/or a refinement which override all incomplete members of the class. Only abstract classes and (all) traits can have abstract term members. This behaviour can be turned off only in tests, if needed, and is implicitly turned on.

The **`abstract`** local modifier can be used with conjunction with **`override`** modifier for class member definitions.

Additionally, an annotation `@[Abstract]` exists for classes and class members that triggers only warnings in case of instantiation, but does not prevent the instantiation. Thus, the annotation only signals an intention, the keyword modifier signals a requirement.

- The **`final`** local modifier applies to class members definitions and to class definitions. Every **`final`** class member can not be overridden in subclasses. Every **`final`** class can not be inherited by a class or trait. Members of final classes are implicitly also final. Note that **`final`** may not be applied to incomplete members, and can not be combined in one modifier list with the **`sealed`** local modifier.

  Additionally, an annotation `@[Final]` exists for classes and class members that triggers only warnings in case of inheriting or overriding respectively, but does not prevent the inheritance or overriding respectively. Thus, the annotation only signals an intention, the keyword modifier signals a requirement.

- The **`sealed`** local modifier applies to class definitions. A **`sealed`** class can not be directly inherited, except if the inheriting class or trait is defined in the same source file as the inherited sealed class. However, subclasses of a sealed class have no restriction in inheritance, unless they are final or sealed again.

  Additionally, an annotation `@[Sealed]` exists for classes and class members that triggers only warnings in case of inheriting outside the same source file, but does not prevent the inheritance. Thus, the annotation only signals an intention, the keyword modifier signals a requirement.

- The **`lazy`** local modifier applies to value definitions. A **`lazy`** value is initialized the first time it is accessed (which might eventually never happen). Attempting to access a lazy value during its initialization is a blocking invocation until the value is initialized of failed to initialize. If an exception is thrown during initialization, the value is considered uninitialized and the initialization is restarted on later access, re-evaluating its right hand side.

Example 5.2.3 The following code illustrates the use of qualified and unqualified private:

```
module Outer_Mod::Inner_Mod
  class Outer
    class Inner
      private:[self] def e() end def
      private def f() end def
      private:[Outer] def g() end def
      private:[Inner_Mod] def h() end def
      private:[Outer_Mod] def i() end def
    end class
  end class
end module
```

Here, accesses to the method `e` can appear anywhere within the instance of `Inner`, provided that the instance is also the receiver at the same time. Accesses to the method `f` can appear anywhere within the class `Inner`, including all receivers of the same class. Accesses to the method `g` can appear anywhere within the class `Outer`, but not outside of it. Accesses to the method `h` can appear anywhere within the module `Outer_Mod::Inner_Mod`, but not outside of it, similar to package-private methods in Java. Finally, accesses to the method `h` can appear anywhere within the module `Outer_Mod`, including modules and classes contained in it, but not outside of these.

A rule for access modifiers in scope of overriding: Any overriding member may be defined with the same access modifier, or with a less restrictive access modifier. No overriding member can have more restrictive access modifier, since the parent class would *lose access* to the member, and that is unacceptable.

- Modifier **`public`** is less restrictive than any other access modifier.
- Qualified modifier **`protected`** is less restrictive than an unqualified **`protected`**, only if the class that the modifier is qualified with is among base classes of the original class – the original class must not lose access.
- Qualified modifier **`protected`** is less restrictive than object-protected, only if the class that the modifier is qualified with is among base classes of the original class – the original class must not lose access.
- While **`protected`** is certainly less restrictive than **`private`**, private members are not inherited and thus can not be overridden.
- While qualified **`private`** is certainly less restrictive than unqualified **`private`**, private members are not inherited and thus can not be overridden.

The relaxations of access modifiers for overriding members are then available as follows:

- **protected:**[**self**] → { **protected, protected:**[C]**, public** }
- **protected** → { **protected:**[C]**, public** }
- **protected:**[C] → { **protected:**[D]**, public** }

  This is only for the case where C is accessible from within D.
- **protected:**[C] → { **public** }
- **public** → { **public** }

  This is just for the sake of completeness, since change from public to public is not much of a relaxation.

5.3 Class Definitions

Syntax:

```
Class_Def  ::= constant_id [Type_Param_Clause]
               [Class_Param_Clauses] (Superclass [semi] | 'begin')
               {Class_Expr}
Class_Alt_Def
           ::= constant_id [Type_Param_Clause]
               [Class_Param_Clauses] [Superclass] '{'
               {Class_Expr} '}'
Class_Param_Clauses
           ::= {Annotation} [Access_Modifier] {Class_Param_Clause}
               ['(' 'implicit' Class_Params ')']
Class_Param_Clause
           ::= '(' [Class_Params] ')'
Class_Params
           ::= Class_Param {',' Class_Param}
Class_Param
           ::= {Annotation} [{Modifier} ('val' | 'var')]
               variable_id [':' Param_Type] [':=' Expr]
Superclass ::= 'extends' Compound_Type
Class_Expr ::= Ctor_Expr
             | Dtor_Expr
             | Clone_Expr
             | Includes_Expr
             | Prepend_Expr
             | Implements_Expr
| Expr
| {Annotation} 'class' Class_Def 'end' ['class']
| {Annotation} 'object' Obj_Def 'end' ['object']
| {Annotation} 'trait' Trait_Def 'end' ['trait']
| {Annotation} 'protocol' Pro_Def 'end' ['protocol']
| {Annotation} 'interface' Ifc_Def 'end'
  ['interface']
| {Annotation} 'type' Const_Type_Def 'end' ['type']
```

A class definition defines the type signature, behavior and initial state of a class of objects (the instances of the defined class) and of the class object, which is the class instance itself, with behavior defined in its metaclass (§5.1.2).

The most general form of a class definition is

```
cas
class c :[tps] @[dps] [<uomps>]
  as m(ps1)...(psn)
  extends t
begin
  exprs...
end class
```

for $n \geq 0$.

Here,

> cas is a possibly empty sequence of annotations (§10) that are applied to the whole class.
>
> c is the name of the class to be defined.
>
> tps is a non-empty list of type parameters of the class being defined. The scope of a type parameter is the whole class definition including the type parameter section itself. It is an error to define two type parameters with the same name. The type parameter section `:`[tps] may be omitted. A class with a type parameter section is called *polymorphic*, a class without a type parameter section is called otherwise *monomorphic*. Type arguments are reified in Coral, i.e. type arguments are preserved in runtime[14], generating a new concrete subtype of the original generic class. This ensures type safety in a dynamic environment of Coral in runtime.
>
> dps is a non-empty list of indexing parameters for dependent types (§5.8). The indexing parameter section `@`{tps} may be omitted. A class with an indexing parameter section is called *indexed*. Indexing arguments are reified in Coral, i.e. indexing arguments are preserved in runtime.

---

[14] Unlike in Java or Scala, which both perform type erasure.

$uomps$ is a non-empty list of unit of measure parameters for aggregated types (§5.9). The unit of measure parameter section `[<`$uomps$`>]` may be omitted. A class with a unit of measure parameter section is called *aggregated with units of measure*. Units of measure arguments are reified in Coral, i.e. units of measure arguments are preserved in runtime.

as is a possibly empty sequence of annotations (§10). If any annotations are given at this point, they apply to the primary constructor of the class.

Expressions in the class-level block may define new members or overwrite members in the parent classes or included traits. The whole class-level block serves as a constructor for the defined class, so the expressions may contain regular function applications, even some that may dynamically define new members (e.g. in a DSL fashion).

For superclasses, the following statements hold:

- The superclass is implicitly `Object`, if no explicit superclass is given.
- Only `Object` can state '`()`' as its superclass, since it has no superclass.
- Explicitly given superclass T, if T is a single class, is that single class.
- Explicitly given superclass T, if T is a compound type with a class member (§3.3.7), is the class member of T and all other members of the compound type are mixed in (§5.3.4).
- Explicitly given superclass T, if T is a compound type without a class member (§3.3.7), is implicitly `Object` and all other members of the compound type are mixed in (§5.3.4) if no member of the compound type has a class as its superclass, or that superclass of one of the members. If there are more members with a class-superclass, there must exist an ordering of these class-superclasses $C_1, \ldots, C_n$, such that for each $i, 1 \leq i \leq n-1$: C_i `<:` C_{i+1}. It is an error if no such ordering exists. If such ordering exists, the superclass is implied to be C_1.
- Superclass is never a function type (§3.3.8).
- Superclass is never a special type, i.e. unions (§5.6), enums (§5.7), range types (§5.8.3), units of measure (§5.9) & record types (§5.10)..
- Superclass is always a class, never a trait (§5.3.4), a protocol (§5.3.6) or an interface (§5.3.7).
- Every single class has at most one direct superclass. Only one class has no superclass: `Object`, no other class can be without a superclass.

It is an error if any of the mixed in traits from a compound type has a class-superclass and yet the defined class does not conform to that superclass.

Example 5.3.1 Consider the following class definitions:

```
class Base extends Object; ... end
trait Trait extends Base; ... end
object O extends Trait; ... end
```

In this case, the definition of `O` is implied to be:

```
object O extends Base with Trait {}
```

5.3.1 Constructor & Destructor Definitions

Syntax:

```
Ctor_Def     ::= 'constructor' Ctor_Fun_Def 'end' ['constructor']
               | 'constructor' Ctor_Alt_Def
Ctor_Fun_Def ::= [Fun_Tpc] [Param_Clauses] [Fun_Dec] semi Expr
Ctor_Alt_Def ::= [Fun_Tpc] [Param_Clauses] [Fun_Dec] ':=' Expr
Dtor_Def     ::= 'destructor' Dtor_Fun_Def 'end' ['destructor']
Dtor_Fun_Def ::= [Fun_Dec] semi Expr
```

Default constructor. A default constructor of a class is the explicit parameterless constructor. This differs from Java or C#.

Implicit constructor. An implicit constructor is an automatically generated bridge constructor to the parameterless default constructor. This is what Java and C# call “default constructor”. An implicit constructor “does nothing” but invokes the super-constructor and initializes all members specific to the constructed object to their default values (either implicit one, which is **`nil`**, or explicit ones used in their definitions).

Convenience constructor. A convenience constructor is any other constructor than the parameterless default constructor.

Accessibility of constructors. Constructors may have modified accessibility, so that only certain functions can invoke them indirectly. The accessibility is then transitioned from the calling context.

Example 5.3.2 An example of a convenience constructor of class C.

```
class C
  constructor (param)
    // super is invoked implicitly here
    val @resource := param
  end constructor
end class
```

Example 5.3.3 An example of a pair of constructors of class C.

```
class C
  constructor := self(42)
  constructor (param)
    // super is invoked implicitly here
    val @resource := param
  end constructor
end class
```

Explicit destructor. An explicit destructor does not have any accessibility. The super-destructor is invoked implicitly at the end of its execution, unless explicitly invoked earlier. Destructors are parameterless and have a further requirement that they can not increment the reference count of the object being destructed – doing so could result in zombie objects.

Implicit destructor. An implicit destructor is an automatically generated bridge destructor to the parameterless super-destructor. An implicit destructor "does nothing" but release all members specific to the destructed object and invoke super-destructor afterwards. The destructor of `Object` releases every remaining member of the destructed object. A class can only have a single destructor, either an explicit or an implicit one.

Accessibility of destructors. Destructors, unlike constructors, can not have any accessibility modifiers. They ignore the current accessibility flag of their class-block and trigger a warning if a modifier is used directly with the destructor. Destructors may be invoked independently on the context in which the object is destructed.

Example 5.3.4 An example of an explicit destructor of class C.

```
class C
  destructor
    @resource.close unless @resource.closed?
    // super is invoked implicitly here
    end destructor
  end class
```

5.3.2 Clone Constructor Definitions

Syntax:

```
CCtor_Def     ::= 'clone' CCtor_Fun_Def 'end' ['clone']
                | 'clone' CCtor_Alt_Def
CCtor_Fun_Def ::= [Fun_Tpc] [Param_Clauses] [Fun_Dec] semi Expr
CCtor_Alt_Def ::= [Fun_Tpc] [Param_Clauses] [Fun_Dec] ':=' Expr
```

Clone constructors are pretty much like constructor, except for they are not invoked indirectly by `allocate` on `Class`, but by **`clone`** on the cloned instance. Regular constructors are not invoked on the cloned objects, since they were already invoked on the original object.

Clone constructor implicitly returns the new cloned object, unless returning explicitly a different object. The original object is available with the **`self`** and **`this`** keywords, the new cloned object is available as the **`cloned`** keyword. The **`cloned`** keyword is only recognized as a keyword in a body of the clone constructor.

Clone constructors pass on the eigenclass (if any) of the original object to the cloned object, thus elevating it to an almost regular class – a prototype class, a class that resides not in a constant, but in a class instance, in an object (but that original object may be still assigned to a constant anyway).

A different clone constructor of the same class may be invoked by using the **`self`** keyword as a function name. If a clone constructor invokes a different clone constructor of the same class this way, the super-clone-constructor is not implicitly invoked (since it is invoked in the other clone constructors).

Default clone constructor. A default clone constructor of a class is the explicit parameterless clone constructor.

Implicit clone constructor. An implicit clone constructor is an automatically generated bridge clone constructor to the parameterless default clone constructor. An implicit clone constructor "does nothing" but invokes the super-clone-constructor and makes a shallow copy of every member specific to the cloned object.

Convenience clone constructor. A convenience clone constructor is any other clone constructor than the parameterless default clone constructor. The **`clone`**

method of objects can accept any number of arguments that are then passed into the clone constructor and the clone constructor is resolved based on these passed arguments.

Accessibility of clone constructors. Clone constructors may have modified accessibility, so that only certain functions can invoke them indirectly. The accessibility is then transitioned from the calling context.

Example 5.3.5 An example of a default clone constructor of class C, performing a deep copy.

```
class C
  clone
    // super is invoked implicitly here
    cloned.resource := self.resource.clone
  end constructor
end class
```

5.3.3 Case Classes

5.3.4 Traits

Syntax:

```
Trait_Def  ::= constant_id [Type_Param_Clause] [Superclass] [semi]
               {Trait_Expr}
Trait_Expr ::= Includes_Expr
             | Prepend_Expr
             | Implements_Expr
             | Expr
             | Requires_Expr
             | {Annotation} 'class' Class_Def 'end' ['class']
             | {Annotation} 'object' Obj_Def 'end' ['object']
             | {Annotation} 'trait' Trait_Def 'end' ['trait']
             | {Annotation} 'protocol' Pro_Def 'end' ['protocol']
             | {Annotation} 'interface' Ifc_Def 'end'
               ['interface']
             | {Annotation} 'type' Const_Type_Def 'end' ['type']
```

A trait is a class that is meant to be injected into some other class as a mixin (including another traits). Unlike normal classes, traits can not be instantiated alone.

Assume a trait D defines some aspect of an instance x of type C (i.e. D is a base class of C). Then the *actual supertype* of D in x is the compound type consisting of all the base classes in $\mathcal{L}(C)$ that succeed D. The actual super type gives the context for resolving a **`super`** reference in a trait (§6.5). Note that the actual supertype depends on the type to which the trait is added in a trait composition; it is not statically known at the time the trait is defined (the trait must exist before being added anywhere).

If D is not a trait, then its actual supertype is simply its least proper supertype (which is statically known).

Example 5.3.6 The following trait defines the property of being comparable to objects of some type. It contains an abstract operator < and default implementations of the other comparison operators <=, > and >=. Operators are methods, too. The trait also requires the self-type to be `$T`.

```
trait Comparable:[$T <: Comparable:[$T]]
  requires $T
  operator < (that: $T): Boolean end
  operator <=(that: $T): Boolean := self < that || self = that
  operator > (that: $T): Boolean := that < self
  operator >=(that: $T): Boolean := that <= self
end trait
```

5.3.5 Refinements

Syntax of refinements is given in section about compound types (§3.3.7).

Refinements in Coral are a special kind of traits. There are two cases in which refinements may appear – as nameless extensions to other types, or as a named trait that has the ability to locally override or extend another type. The actual meaning depends on the usage of a refinement (see also §6.6).

5.3.6 Protocols

Syntax:

```
Pro_Def         ::= constant_id [Type_Param_Clause]
                    [Superclass] [semi] {Pro_Expr}
Pro_Expr        ::= Dcl
Implements_Expr ::= 'implements' Pro_Arg {',' Pro_Arg}
Pro_Arg         ::= Simple_Type [Type_Args]
```

Protocols are classes that are abstract and can contain only abstract member declarations. Protocols express the contracts that other classes have to implement, and are added to classes with the keyword “**`implements`**”.

5.3.7 Interfaces

Syntax:

```
Ifc_Def  ::= '[' Ifc_Kind ']' constant_id [Type_Param_Clause]
             [Superclass] [semi] {Ifc_Expr}
Ifc_Kind ::= 'class' | 'trait'
Ifc_Expr ::= Dcl
```

Interfaces are filtered versions of classes with only declarations. Interfaces can be generated from classes or traits by simple transformations and manually edited as needed. Their only purpose is to be used in *module interfaces*, so that implementation is not distributed along, but only declarations in interfaces and protocols.

5.4 Object Definitions

Syntax:

```
Obj_Def  ::= constant_id [Superclass] [semi]
             {Obj_Expr}
Obj_Expr ::= Clone_Expr
           | Includes_Expr
           | Prepend_Expr
           | Implements_Expr
           | Expr
           | {Annotation} 'class' Class_Def 'end' ['class']
           | {Annotation} 'object' Obj_Def 'end' ['object']
           | {Annotation} 'trait' Trait_Def 'end' ['trait']
           | {Annotation} 'protocol' Pro_Def 'end' ['protocol']
           | {Annotation} 'interface' Ifc_Def 'end'
             ['interface']
           | {Annotation} 'type' Const_Type_Def 'end' ['type']
```

Object definitions define singleton instances. If no superclass is given, `Object` is implied, unless the object definition has the same name as an existing or enclosing class – then `Class:[`C`]` is implied, only abstract compound types without class may appear as superclass, and it is an error if a concrete class appears as a superclass (even if that would be `Class:[`C`]`, as it is prohibited to inherit from this class in user programs). If the class definition is not connected to a class, then rules from compound types apply (§3.3.7).

Modifiers (§5.2) are available in the same way as in class definitions.

5.5 Module Definitions

Syntax:

```
Module_Def  ::= constant_id [Vendor_Arg] {'::' constant_id}
                [semi] {Module_Expr}
Module_Expr ::= Expr
              | Implements_Expr
              | Const_Def
```

Module definitions are objects that have one main purpose: to join related code and separate it from the outside. Coral's approach to modules solves these issues:

- *Namespaces*. A class with a name C may appear in a module M or a module N, or any other module, and yet be a different object. Modules may be nested.
- *Vendor packages*. Even modules of the same name may co-exists, provided that they have a different vendor, which is just an identifier that looks like a reverse domain name (similar to Java or Scala packages).
- *Dependencies*. Module may define a tree of dependencies, including module vendor resolution, if a module of the same name is provided by different vendors.

Modifiers (§5.2) are available in the same way as in class definitions. This time, members may be classes and other types as well, beside functions.

5.6 Unions

Syntax:

```
Const_Type_Def ::= constant_id 'is' 'union' 'of'
                   '(' Type {semi Type} ')'
```

Union types represent multiple types, possibly unrelated. Union types are abstract by nature and can not be instantiated, only the types that they contain may, if these are instantiable. For type safety, bindings of union types should be matched for the actual type prior to usage.

5.7 Enums

Syntax:

```
Const_Type_Def ::= constant_id [Superclass] 'is' ['bitfield']
                   'enum' '(' Enum_Field {semi Enum_Field} ')'
Enum_Field     ::= constant_id [':=' scalar_literal]
```

Enums (short for Enumerations) are types that contain constants. Bitfield enums may be combined to still produce a single enum value. Every enum constant is a singleton instance of the enum class.

5.8 Dependent Type Declarations

Dependent types consist of three kinds of types in Coral. First, there are *indexed types* (§5.8.1), that are at the core of dependent types. Not every type in Coral is indexed. Secondly, there are two types that are similar and sometimes interchangeable: *constrained types* (§5.8.2) and *range types* (§5.8.3), each making use of indexed types in a specific way. Arguments applied to these types are then described in §3.3.11.

Dependent types are neither concrete nor abstract. They only add a way of indexing existing types and defining subsets of all instances. As a side-effect of this restriction, variables involved in dependent type declarations are not involved in the rest of the class declarations and definitions, and for that reason don’t need to be distinguished from other variables.

Dependent types (§3.3.11) are allowed in these positions:

1. Function parameters.
2. Function return types.
3. Variable declarations and definitions.
4. Type conversions.

5.8.1 Indexed Types

Syntax:

```
Class_Def          ::= constant_id [Type_Param_Clause]
                       [Dep_Params] [Superclass] [semi]
                       {Indexed_Class_Expr}
```

```
Dep_Params         ::= '@{' Index_Param {',' Index_Param} '}'
Index_Param        ::= variable_id [':' Simple_Type]
Indexed_Class_Expr ::= Class_Expr
                     | Indexed_By_Clause
Indexed_By_Clause  ::= 'indexed-with' Indexing_Expr
Indexing_Expr      ::= '{|' Index_Param {',' Index_Param} '|'
                       [Index_Exprs] '}'
Index_Exprs        ::= Index_Expr {[semi] Index_Expr}
Index_Expr         ::= Index_Var Index_Op Index_Val
                     | Index_Val [Index_Op Index_Var]
                     | '(' Index_Var Index_Op Index_Var ')'
                     | '(' Index_Expr [Index_Con Index_Expr] ')'
Index_Var          ::= variable_id
                     | ivar_id
                     | 'self'
Index_Val          ::= variable_id
                     | ivar_id
                     | literal
                     | constant_id
                     | Index_Fun '(' Index_Var {',' Index_Var} ')'
```

Indexed types declare what their index is, based on combinations of their input (indexing) variables, instance variables and operators and functions working with these. Since the scope of testing these *indexing constraints* is limited to cases listed in §5.8, independent on the intrinsic state of the instances, the instances may appear in states that do not conform to the constraints in between each indexing constraint test, but to pass as the dependent type, they must conform to the indexing constraint at the time the test is invoked, that is:

1. Method resolution time (function parameters).
2. Returning from a function (function return types).
3. Getting assigned to a variable.
4. Getting converted to a type (unless the conversion is implicit to a different type).

Note that implicit conversions (§6.25.2) can't apply to dependent types, since that would be a conversion from the same type to a subset of the same type.

Sorts. This part of Coral is inspired by the ATS language. "Sorts" are a types for which the language knows ordering of their values implicitly:

- `Boolean`.
- `Integer`.
- `Float`.
- `Char`.
- Every **`enum`** type (§5.7), where the ordering is given by the order in which each enumerated value appears.
- Every type constrained from a pre-existing sort (§5.8.2).

See references of these "sort" types for more details on their particular ordering.

"Sorts" are the types allowed as types of *indexing variables* (see `Index_Param` syntax). If no type is specified, `Integer` "sort" is implied.

For the mentioned reasons, `Dep_Sort_Val` (used in §3.3.11) are values that are members of "sorts".

Note that the `Index_Exprs` are optional – meaning that the entire "sort" is used for indexing, not only a subset of it.

Indexing Operators. Indexing operators used to declare indexing constraints are the following for the `Index_Op` syntax:

- ~ (bitwise negation): `(Number)` ↦ `Number`
- + (addition): `(Number, Number)` ↦ `Number`
- - (subtraction): `(Number, Number)` ↦ `Number`
- $*$ (multiplication): `(Number, Number)` ↦ `Number`
- $**$ (exponentiation): `(Number, Number)` ↦ `Number`
- / (division): `(Number, Number)` ↦ `Number`
- `%` (modulo): `(Number, Number)` ↦ `Number`
- > (greater than): `(Number, Number)` ↦ `Boolean`
- >= (greater than or equal to): `(Number, Number)` ↦ `Boolean`
- < (less than): `(Number, Number)` ↦ `Boolean`
- <= (less than or equal to): `(Number, Number)` ↦ `Boolean`

- = (equal to): `(Number, Number)` ↦ `Boolean)`
- <> (not equal to): `(Number, Number)` ↦ `Boolean`
- `!=` (not equal to): `(Number, Number)` ↦ `Boolean`
- `!` (boolean negation): `(Boolean)` ↦ `Boolean`
- `||` (boolean disjunction): `(Boolean, Boolean)` ↦ `Boolean`
- && (boolean conjunction): `(Boolean, Boolean)` ↦ `Boolean`
- ^^ (boolean exclusive disjunction): `(Boolean, Boolean)` ↦ `Boolean`
- `|` (bitwise or): `(Number, Number)` ↦ `Number`
- & (bitwise and): `(Number, Number)` ↦ `Number`
- ^ (bitwise xor): `(Number, Number)` ↦ `Number`

These operators are static[15]. In the list, `Number` can also be replaced by `Char`, but not by `Complex`.

Indexing Functions. Functions that can be used to constrain the indexing are also static and limited to work only with a restricted number of types:

- `size: (List)` ↦ `Number`
- `length: (String)` ↦ `Number`
- `empty?: (List)` ↦ `Boolean`
- `empty?: (String)` ↦ `Boolean`
- `max: (*Number)` ↦ `Number`
- `min: (*Number)` ↦ `Number`
- `avg: (*Number)` ↦ `Number`
- `sum: (*Number)` ↦ `Number`
- `abs: (*Number)` ↦ `Number`
- `sgn: (Number)` ↦ `Number`
- `log: (Number, Number)` ↦ `Number`

---

[15] That is, not overridable by user programs.

- `ln: (Number) ↦ Number`
- `even?: (Number) ↦ Boolean`
- `odd?: (Number) ↦ Boolean`
- `concat: (*String) ↦ String`
- `coalesce: (*Object) ↦ Object`
- `lowercase: (String) ↦ String`
- `uppercase: (String) ↦ String`

Indexing Constraint Concatenation. If there are multiple indexing constraints separated by the `[semi]` syntax, boolean conjunction is implied. Indexing constraints may be joined by a different boolean operations (`Index_Con`):

- `||` (boolean disjunction): `(Boolean, Boolean) ↦ Boolean`
- `&&` (boolean conjunction): `(Boolean, Boolean) ↦ Boolean`
- `^^` (boolean exclusive disjunction): `(Boolean, Boolean) ↦ Boolean`

Example 5.8.1 The following is an example on how a `String` type might index itself:

```
class String @{length: Integer}
  indexed-with {|length| @length = length}
end
```

5.8.2 Constrained Types

Syntax:

```
Type_Def      ::= constant_id [Type_Param_Clause] ':='
                  Type [Dep_Params_Constr]
Dep_Params_C  ::= '@{|' Index_Param {','} Index_Param '|'
                  [Index_Exprs_C] '}'
Index_Exprs_C ::= Index_Expr_C {[semi] Index_Expr_C}
Index_Expr_C  ::= Index_Var_C Index_Op Index_Val_C
                | Index_Val_C [Index_Op Index_Var_C]
                | '(' Index_Var_C Index_Op Index_Var_C ')'
                | '(' Index_Expr_C
                  [Index_Con Index_Expr_C] ')'
Index_Var_C   ::= variable_id
```

```
Index_Val_C   ::= variable_id
                | literal
                | constant_id
                | Index_Fun '(' Index_Var_C
                  {',' Index_Var_C} ')'
```

Constrained types are basically aliases to indexed types with optionally further restricted indexing, which can only make use of the existing indexing constraints of the base type, using the indexing variables. No additional instance variable can be constrained by a constrained type directly.

The type of the indexing variable is implied to be the same as of the indexed type.

Example 5.8.2 Here is an example of how Coral might declare the `Char` type:

```
type Char := String @{|length| length = 1}
```

Notice how the constraint only uses the already existing indexing variable.

Example 5.8.3 Here is an example of a different constrained type:

```
type Even_Positive_Integers ::= Integer @{|i| even?(i); i >= 0}
```

This constrained type declares `Even_Positive_Ints` to be positive integers that are even at the same time.

5.8.3 Range Types

Syntax:

```
Const_Type_Def ::= constant_id 'is' 'range'
                   (Range_Expr
                 | '(' Range_Expr ')' [':' constant_id])
```

Range types are similar to constrained types, but limited in a few ways: they can constrain only indexed types that are indexed with exactly one indexing variable. The range expression is converted into the corresponding indexing constraint.

Example 5.8.4 An example of a constrained type interchangeable with a range type:

```
type Positive_Integers ::= Integer @{|i| i >= 0}
type Positive_Integers is range
  (0 .. +Integer::Infinity) : Integer
end type
type Positive_Integers is range
  0 .. +Integer::Infinity
end type
```

Types that do not require the “`constant_id`” are those that are “sorts” at the same time, so that the indexed type can be inferred.

5.9 Units of Measure

Syntax:

```
Const_Type_Def ::= Unit_Name 'is' ['abstract'] 'unit-of-measure'
                    [semi Unit_Convs {semi Unit_Convs}]
Unit_Name      ::= variable_id | constant_id
                    [':' Superunit_Name]
Superunit_Name ::= variable_id | constant_id
Unit_Convs     ::= Unit_Name ':=' Unit_Conv
Unit_Conv      ::= '(' Unit_Conv ')'
                 | Unit_Elem [Unit_Op Unit_Elem]
                 | Unit_Conv Unit_Op Unit_Conv
Unit_Elem      ::= number_literal | Unit_Name
```

Numbers in Coral can have associated units of measure, which are typically used to indicate length, volume, mass, distance and so on. By using quantities with units, the runtime is allowed to verify that arithmetic relationships have the correct units, which helps prevent programming errors.

Example 5.9.1 The following defines the measure `cm` (centimeter).

```
type cm is unit-of-measure
end type
```

Example 5.9.2 The following defines the measure `ml` (milliliter) as a cubic centimeter (`cm ** 3`).

```
type ml is unit-of-measure
  ml := cm ** 3
end type
```

Every unit of measure is defined in the same scope as any other type would be, but the application of units of measure to numbers or *aggregated unit types* require to import units of measure by name into the scope where a unit of measure from a different unrelated module would be used.

Types Aggregated with Units of Measure. In addition to type parameters and dependency parameters of each type, every type may be parameterized with a units of measure aggregation. It is recommended to avoid mixing these three together.

Syntax:

```
Class_Def  ::= constant_id [Type_Param_Clause]
               [Dep_Params] [UoM_Params] [Superclass] [semi]
               {Indexed_Class_Expr}
UoM_Params ::= '[<' UoM_Param {',' UoM_Param} '>]'
UoM_Param  ::= '$' Unit_Name
```

Names of unit of measure parameters must not clash with names of type parameters, otherwise it is a compile-time error.

Persistence of Units of Measure. There is a huge difference between the way F# handles units of measure and Coral's way. In F#, the unit of measure information is lost after compilation, but persists in Coral in runtime, since verification of units of measure is deferred also to runtime, as it is limited during compilation. This also means that the information may be accessed in runtime, e.g. using it to print the unit information on screen.

5.10 Record Types

Syntax:

```
Const_Type_Def    ::= Record_Name 'is' ['abstract'] 'record'
                      [semi] Record_Components
Record_Name       ::= constant_id [Type_Param_Clause] [Param_Clause]
Record_Components ::= Record_Component {semi Record_Component}
                      [Superclass]
Record_Component  ::= 'val' Val_Dcl
                    | 'var' Var_Dcl
                    | 'case' variable_id semi
                      {'when' constant_id
                      ('then' | nl)
                      Record_Components} 'end' ['case']
```

Record types are simple syntax sugar for classes that represent *data objects*, i.e., objects that don't really care about behaviour, their only purpose is to store data.

Record types can appear in three different forms: *basic records*, *discriminated records* and *variant records*.[16]

Basic Records. Basic records have no discriminating parameters, and can have only type parameters. Their structure is always the same.

Discriminated Records. Discriminated records are similar to variant records, they can have discriminating parameters and a discriminated record may also be a variant record. Discriminating parameters are to be used in conjunction with dependent types (§5.8) of the record's components. Discriminating a record type does the same thing as declaring a dependent type: it creates a subset of instances of the original record type.

Variant Records. Variant records are similar to discriminated records, they can have discriminating parameters and a discriminated record may also be a discriminated record. Variant parameters are enums (§5.7). Variant records render new subtypes of the original record type, similar to a concrete type constructor.

Example 5.10.1 The following example defines a variant record.

```
type Traffic_Light is bitfield enum
  Red
  Yellow
  Green
end type

type Variant_Record (option: Traffic_Light) is record
  val a: A
  var b: B
  case option
  when Red
    val c: C
  when Yellow
    var d: D
  when Green
    val e: E
```

---

[16] Note that the syntax for record types in Coral differs from Ada's `type Record_Name is record... end record;`. Coral ends a type, not a record. This difference appears in more places than just this syntax.

```
    end case
  end type

  let vr := Variant_Record(Traffic_Light::Red).
    new(a: A.new, b: B.new, c: C.new)
```

5.11 Nullability

Every type in Coral conforms to `Object`. In turn, `Nothing` conforms to any type. But, the only instance of `Nothing`, which is a singleton accessed with the keyword **`nil`**, can not be assigned to every typed variable. If a type is declared as nullable, then it can. If a type is declared as not-nullable, which is the implicit preference for every type, then it can not.[17]

The implicit preference may be changed in two levels. These levels are *preferred nullability* and *explicit nullability* (§3.3.10).

Preferred nullability is switched by annotations on each class: `Nullable`. The annotation has one positional parameter of type `Boolean` defining what the nullability will be (**`yes`** for nullable types, **`no`** for not-nullable types), and a named parameter `:override` of type `Boolean`, defining whether subclasses may override this preference, and implicitly set to **`yes`**, meaning that subclasses may override this preference by default.

All types in Coral are not-nullable implicitly to prevent `Method_Not_Found_Errors`, resulting from sending messages to **`nil`**. But it is clear that for some cases, nullability of types may be desired, such as with dictionaries, so that **`nil`** can be returned for keys that have no value in the dictionary. However, it is preferred to use the `Option` type wherever possible to indicate that the value may not be present, and moreover, the `Option` type can be used in pattern matching even for values that are not of the `Option` type, utilizing an extractor pattern (§8.1.8) with `Some` and a constant pattern (§8.1.5) with `None`, instead of constructor patterns (§8.1.6). This is again different from Scala, where `Some` and `None` can't be used to match any value, only `Option` values. This difference in Coral makes nullable types and the `Option` type interoperable in pattern matching.

Example 5.11.1 Here are all four versions of the nullability annotation.

```
  @[Nullable yes, override: yes]
  @[Nullable yes, override: no]
```

---

[17] An original idea was to make every type not-nullable and force users to use the `Option` type for every no-object scenario. But then, what would **`nil`** be good for?

```
@[Nullable no, override: yes]
@[Nullable no, override: no]
```

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 Use Expressions

6.7 Function Applications

6.7.1 Named and Optional Arguments

6.7.2 By-Name Arguments

6.7.3 Input & Output Arguments

6.7.4 Function Compositions & Pipelines

6.8 Method Values

6.9 Type Applications

6.10 Tuples

6.11 Instance Creation Expressions

Chapter 7

Implicit Parameters & Views

7.1 The Implicit Modifier

7.2 Implicit Parameters

7.3 Views

Chapter 8

Pattern Matching

8.1 Patterns

8.1.1 Variable Patterns

8.1.2 Typed Patterns

8.1.3 Pattern Binders

8.1.4 Literal Patterns

8.1.5 Constant Patterns

8.1.6 Constructor Patterns

8.1.7 Tuple Patterns

8.1.8 Extractor Patterns

8.1.9 Pattern Sequences

8.1.10 Pattern Alternatives

8.1.11 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 References

9.4 Top-Level Classes

9.5 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