The Coral Language Specification

Kateřina Markéta Lisová

September 23, 2014

Contents

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). Also, along with other C-like languages, Scala influenced Coral to allow to choose between Ada-style program parentheses (e.g. **`begin`** … **`end`**) and C-style parentheses (i.e. `{` … `}`) in many places throughout the syntax. 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 (§1.5.4 & §5.7).

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.

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:

```
op_id    ::= opchar {opchar}
var_id   ::= lower id_rest
plain_id ::= upper id_rest
           | ['@' | '@@'] var_id
id       ::= plain_id
           | '`' simple_string_literal '`'
id_rest  ::= {letter | digit | opchar}
```

There are more kinds of identifiers. An identifier can start with a letter, which can be followed by an arbitrary sequence of letters, digits, underscores and operator characters. The identifier may be prefixed with one or two *at* "@" signs, creating an instance variable or a class object variable identifier, respectively. These forms are called *plain identifiers*. An identifier may also start with an operator character, followed by arbitrary sequence of operator characters, forming operator identifiers, which can only be used in expressions that directly involve operators (§6.12).

An identifier may also be formed by an identifier between back-quotes (" ` "), to resolve possible name clashes with Coral keywords, and to allow to identify operators. Instance variable names and class instance variable names 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.

The "$" character is reserved.

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	**advice**	**advice-execution**		**after**
alias	**annotation**	**arguments**	**around**	**as**
aspect	**begin**	**before**	**bitfield**	**break**
broken	**case**	**catch**	**class**	**clone**
cloned	**constant**	**constructor**	**declare**	**def**
delta	**destructor**	**digits**	**do**	**else**

```
elsif          end            ensure         enum           execution
exhausted      extends        final          for            for-some
function       get            goto           handler        if
implements     implicit       in             include        interface
invoke         is             label          lazy           let
loop           macro          match          memoize        message
method         module         native         next           nil
no             object         of             opaque         operator
out            override       pointcut       pragma         prepend
private        property       protected      protocol       public
raise          raising        range          record         redo
refine         refinement     rescue         retry          return
returning      requires       reverse        sealed         self
set            skip           step           super          target
then           this           throw          throwing       trait
transparent    type           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 work-around when used with an implicit receiver.

1.3 Newline Characters

Syntax:

```
semi ::= nl {nl} | ';'
```

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

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

Since Coral may be interpreted in a REPL[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.

[1]Read-Eval-Print Loop

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.

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

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* (§5.7). 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, and **`no`** does not match in pattern matching (§8) as **`nil`** and vice versa. 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 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}
regexp_flags       ::= printable_char {printable_char}
```

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

1.5.10 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 ')'
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. Wildcard **`use`** clauses (imports) have the next highest precedence.
4. Inherited definitions and declarations have the next highest precedence.
5. Definitions and declarations made available by module clause have the next highest precedence.
6. Definitions and declarations that are not in the same compilation unit (a different script or a different module) have the next highest precedence.
7. 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.

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

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.

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, it is an error. Also, if a new binding would be ambiguous[3], 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]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

Syntax:

```
Type                ::= Function_Type
                      | Infix_Type [Existential_Clauses]
Function_Type       ::= Function_Args {'->' Function_Args}
                        '->' Type
Function_Args       ::= Infix_Type
                      | '(' [Function_Arg {',' Function_Arg}] ')'
Function_Arg        ::= [Param_Io] ['*' | '**' | '&' | ['^'] id ':']
                        Param_Type
Existential_Clauses ::= {'for-some' '{' Existential_Dcl
                        {semi Existential_Dcl} '}'}
Existential_Dcl     ::= 'type' Type_Dcl
                      | 'val' Val_Dcl
                      | 'var' Var_Dcl
Compound_Type       ::= Annot_Type {['prepend'] 'with' Annot_Type}
                        ['with' ['refinement'] Refine_Stats]
                      | ['refinement'] Refine_Stats
Infix_Type          ::= Compound_Type {op_id [nl] Compound_Type}
Annot_Type          ::= {Annotation} Simple_Type
Simple_Type         ::= Simple_Type [Type_Args]
                      | Simple_Type '#' id
                      | Stable_Id
                      | Path '.' 'type'
                      | '(' Types ')'
Types               ::= Type {',' Type}
```

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 static typing in most cases. But, it allows to opt-in for some dynamic typing. 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[1]), 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

[1]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.

easily modify classes, even from other modules (*pimp my library!* and *open-class principle*). You may even easily add more classes if needed, and possibly shadow existing ones. In face of static typing in Coral, *no type* specified or inferred is saying that the value is of the special type `Any`, which represents 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[2]. 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 with dynamic typing), 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.

Types in Coral are represented by objects that are members of the `Type` type. Instances of value types are represented by objects that are members of the `Class` type.

3.2 Paths

Syntax:

```
path_id          ::= id [Type_Args]
Path             ::= Stable_Id
                   | [path_id '.'] ('this' | 'self')
                   | 'self' '[' 'cloned' ']'
                   | Container_Path
Stable_Id        ::= path_id
                   | Path '.' path_id
                   | [path_id '.'] 'super' [Class_Qualifier]
                     '.' path_id
Class_Qualifier  ::= '[' Container_Path ']'
```

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:

- 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.

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

- **`self`**`[`**`cloned`**`]`, which references the directly enclosing class of a clone (a cloned instance, see §5.3.2).
- *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.

Paths introduce also *path dependent types*, if the referenced member is a type.

3.3 Value Types

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

3.3.1 Singleton Type

Syntax:

```
Simple_Type ::= Path '.' 'type'
```

A singleton type is of the form *p*`.`**`type`**, where *p* is a path pointing to a value. The type denotes the set of values consisting of **`nil`** and the value denoted by *p*, in spite of nullability.

A *stable type* is either a singleton type or a type which is declared to be a subtype of a trait `Singleton_Type`.

3.3.2 Type Projection

Syntax:

```
Simple_Type ::= Simple_Type '#' id
```

A type projection *T*`#`*x* references type member named *x* of type *T*. This is useful i.e. with nested classes that belong to the class instances, not the class object.

Example 3.3.1 A sample code that shows off what type projections are good for:

```
class A {
  class B {}
  def f (b: B): Unit   := Console.print_line "Got my B."
  def g (b: A#B): Unit := Console.print_line "Got a B."
}

val a1 := A.new
val a2 := A.new
a2.f a1.B.new     // type mismatch, found a1.B, required a2.B
a2.g a1.B.new     // prints "Got a B." to stdout
a2.f a2.B.new     // prints "Got my B." to stdout
```

This is due to the fact that the **class** B is defined as a class instance member of **class** A, not as a class object member (either via object definition (§5.4) or some form of meta-class access (§5.1.4)). Therefore, a1.B refers to type member B of the instance a1, but not of a2. Moreover, A.B is not defined here.

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 C is taken as a shorthand for C**.self.type#**t. If t is bound in some object[3] or module C, it is taken as a shorthand for C**.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.

Example 3.3.2 Some type designators and their expansions are listed below, the type designator being on the left and the expansion on the right of "=".

```
t = ϵ.type#t // "global space"
Number = Lang~[coral].type#Number // predefined import of Lang~[coral]

object An_Object {
  type t
  t = An_Object.type#t // bound by object
}
```

[3]Also in class object methods.

```
class A_Class {
  type t
  def a_method := {
    t = A_Class.self.type#t // bound by class
  }
  class << self
    type u
    u = A_Class.type#u // bound by (class) object
  end
  t = A_Class.self.type#t // bound by class
  self.u = A_Class.type#u // bound by (class) object
}
```

3.3.4 Parameterized Types

Syntax:

```
Simple_Type ::= Simple_Type [Type_Args]
Type_Args   ::= '[' Types ']'
Types       ::= Type_Arg {',' Type_Arg}
Type_Arg    ::= Type | '<' uom_expr '>'
```

A parameterized type $T[T_1, \ldots, T_n]$ 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 $\sigma L_i <: T_i <: \sigma U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$[4]. 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 3.3.3 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
```

[4]The substitution works by replacing occurrences of a_i in the argument by T_i, so that, e.g. `$A <: Comparable[$A]` is substituted into C `<: Comparable[`C`]`.

```
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 3.3.4 Given the type definitions of the previous example, the following types are malformed:

```
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 {['prepend'] 'with' Annot_Type}
                    ['with' ['refinement'] Refine_Stats]
                  | ['refinement'] Refine_Stats
Refine_Stats  ::= '{' [Refine_Stat {semi Refine_Stat}] '}'
Refine_Stat   ::= Dcl
                  | 'type' Type_Def
```

A compound type T_1 **with** ... **with** T_n **with refinement** $\{R\}$ 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[5].

The keyword **refinement** instructs that the following tokens will be a part of a refinement, and the construct **refinement** $\{R\}$ is called an *anonymous refinement*, being equivalent to `Object` **with refinement** $\{R\}$, although when the refinement is a part of a compound type with more elements than just the refinement itself, the `Object` type is replaced with the class type appearing in the compound type, if any.

The **refinement** keyword can be omitted from a compound type that consists of more elements than just the refinement. Constructs `Object` **with** $\{R\}$ and **refinement** $\{R\}$ are then equal.

The **refinement** keyword may also be omitted from a compound type that consists of just the refinement, but only in contexts in which a type is expected, i.e.: parameter type

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

declaration, value or variable type declaration, return type declaration, type argument application or type parameter declaration; but never stand-alone. If used as such, the constructs **`refinement`** $\{R\}$ and $\{R\}$ are then equal.

3.3.8 Infix Types

Syntax:

```
Infix_Type ::= Compound_Type {op_id [nl] Compound_Type}
```

An infix type T_1 *op* T_2 consists of an infix operator *op*, which gets applied to two type operands T_1 and T_2. The type is equivalent to the type application *op*`[`T_1`,` T_2`]`. The infix operator *op* may be an arbitrary identifier, and is expected to represent a type constructor.

Infix type may also result from an infix expression (§6.12), if such operator name is not found on the result type of the expression that it is applied to. In any case, precedence and associativity rules of operators apply here as well.

3.3.9 Function Types

Syntax:

```
Type          ::= Function_Args {'->' Function_Args}
                  '->' Type
Function_Args ::= Infix_Type
                | '(' [Function_Arg {',' Function_Arg}] ')'
Function_Arg  ::= [Param_Io] ['*' | '**' | '&' | ['^'] id ':']
                  Param_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.

Function argument types may be annotated with some extra properties. In that case, these map to annotations of their types, defined as follows:

- "**`out`**" maps to `@[out_param]`.
- "**`in`**" maps to no annotation, as it is implied, and if the parameter is **`out`**-only, the function is still applicable.
- "*" maps to `@[repeated_param]`.
- "**" maps to `@[capturing_named_param]`.
- "&" maps to `@[captured_block_param]`.

- "^*id*" maps to `@[named_param :`*id*`]`. For now, the leading "^" may be omitted, as it could be implied, but it's generally better to not omit it.

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
  ...
end protocol
```

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

A function that returns "nothing" may be declared as returning the type `Unit`, which is similar to **`void`** in C-related languages. Such a type is then written as `(`S`) -> Unit`.

Function arguments may be optionally annotated with more requirements:

- Parameter prefixed with `Param_Io` defines whether the parameter is required to be an output parameter.
- Parameter prefixed with "∗" is a requirement of a repeated parameter.
- Parameter prefixed with "^" is a requirement of a parameter named x.
- Parameter prefixed with "∗∗" is a requirement of a captured named parameters.
- Parameter prefixed with "&" is a requirement for the passed block. It does not tell whether the function must capture the passed block, it only restricts the requirements for the particular block, if any. The actual passed block may have more or even less positional or named parameters (extra ones on the block side are given **`nil`**, unless the type is not nullable (§5.9) – that is an error; and extra ones on the type side are simply discared), but the return type of the passed block must conform (§3.7.2).

Note. Although the function type alone allows to attach the extra annotations to types of arguments in a `1:1` manner (and therefore types of parameters), due to how conformance is defined for function types, it is not always desirable to use them. The only argument extra that might be of any use is the captured block argument, so that a requirement of a passed block is marked (not caring about the actual style of the block passing).

Example 3.3.5 The following definition of functions `g` and `h` conform to a definition of a function `f`, and there are many more such functions that conform to `f`. And vice versa, `f` conforms to both `g` and `h`, although the latter two are more specific than the first one (§6.26.3).

```
def f (*x: Integer) end
def g (x: Integer, y: Integer) end
def h (x: Integer, y: Integer, z: Integer) end
```

3.3.10 Existential Types

Syntax:

```
Type                ::= Infix_Type Existential_Clauses
Existential_Clauses ::= {'for-some' '{' Existential_Dcl
                        {semi Existential_Dcl} '}'}
Existential_Dcl     ::= 'type' Type_Dcl
                      | 'val' Val_Dcl
```

An existential type has the form T **`for-some`** `{` Q `}`, where Q is a sequence of type declarations (§4.4). Let $t_1[tps_1]$ `>:` L_1 `<:` U_1 , ..., $t_n[tps_n]$ `>:` L_n `<:` U_n be the types declared in Q (any of the type parameter sections $[tps_i]$ might be missing). The scope of each type t_i includes the type T and the existential clause Q. The type variables t_i (occurring in the sequence Q) are said to be *bound* in the type T **`for-some`** `{` Q `}`. Type variables that occur in a type T, but which are not bound in T, are said to be *free* in T.

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, $\sigma L_i <: \sigma t_i <: \sigma U_i$. 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. In other words, a type instance of an existential type is a type application (§6.7) of the type T **`for-some`** `{` Q `}`, where the applied type arguments conform to the bounds, or are free in T **`for-some`** `{` Q `}`.

A *skolemization* of T **`for-some`** `{` Q `}` is a type instance σT, where σ is the substitution $[t_1 := t'_1, \ldots, t_n := t'_n]$ and each t'_i is a fresh abstract virtual type with lower bound σL_i and upper bound σU_i[6]. Such type instance is inaccessible to user programs, but is essential to type equality and conformance checks, as it describes the set of values denoted by the existential type without an actual existential type, but with a universal type.

Simplification rules. Existential types obey the following equivalences:

[6]This virtual type t'_i denotes the set of all types, for which $\sigma L_i <: \sigma t'_i <: \sigma U_i$.

1. Multiple **for-some** clauses in an existential type can be merged. E.g., T **for-some** { Q } **for-some** { Q' } is equivalent to T **for-some** { Q; Q' }.

2. Unused quantifications can be dropped. E.g., T **for-some** { Q; Q' }, where none of the types defined in Q' are referred to by T or Q, is equivalent to T **for-some** { Q }.

3. An empty quantification can be dropped. E.g., T **for-some** { } is equivalent to T.

4. An existential type T **for-some** { Q }, where Q contains a clause **type** t[*tps*] >: L <: U is equivalent to the type T' **for-some** { Q }, where T' results from T by replacing every covariant occurrence (§4.6) of t in T by U and by replacing every contravariant occurrence of t in T by L.

Existential quantification over values. As a syntactic convenience, the bindings clause in an existential type may also contain value declarations **val** x: T. An existential type T **for-some** { Q; **val** x: S; Q' } is treated as a shorthand for the type T' **for-some** { Q; **type** t <: S **with** `Singleton_Type`; Q' }, where t is a fresh type name and T' results from T by replacing every occurrence of x**.type** with t.

Placeholder syntax for existential types. Coral supports a placeholder syntax for existential types. A *wildcard type* is of the form `_ >:` L `<:` U. Both bound clauses may be omitted. If a lower bound clause `_ >:` L is omitted, `_ >: Nothing` is assumed. If an upper bound clause `_ <:` U is omitted, `_ >: Object` is assumed. A wildcard type is a shorthand for an existentially quantified type variable, where the existential quantification is implicit.

A wildcard type must appear as a type argument of a parameterized type. Let T := p`.`c[*targs*`,` T`,` *targs*$'$] be a parameterized type, where *targs*, *targs*$'$ may be empty and T is a wildcard type `_ >:` L `<:` U. Then T is equivalent to the existential type

p`.`c[*targs*`,` t`,` *targs*$'$] **for-some** { **type** t >: L <: U }

where t is a fresh type variable. Wildcard types may also appear as parts of compound types (§3.3.7), function types (§3.3.9) or tuple types (§3.3.5). Their expansion is then the expansion in the equivalent parameterized type.

Example 3.3.6 Assume the class definitions

```
class Ref[$T] {}
abstract class Outer { type T }
```

Here are some examples of existential types:

```
Ref[$T] for-some { type $T <: Number }
Ref[x.T] for-some { val x: Outer }
Ref[x_type#T] for-some { type x_type <: Outer with Singleton_Type }
```

The last two types in this list are equivalent. An alternative formulation of the first type above using wildcard syntax is:

```
Ref[_ <: Number]
```

which is equivalent to Java's

```
Ref<? super Number>
```

`Ref[_ <: Number]` then represents any type constructed by the type`Ref` parameterized with a type that is `Number` or any type that conforms to `Number`. `Ref[_ >: Number]` would then represents any type constructed by the type `Ref` parameterized with a type that is `Number` or any type that `Number` conforms to.

Example 3.3.7 The type `List[List[_]]` is equivalent to the existential type

```
List[List[$T] for-some { type $T }] .
```

Example 3.3.8 Assume a covariant type

```
class List[+$T] {}
```

The type

```
List[$T] for-some { type $T <: Number }
```

is equivalent (by simplification rule 4 above[7]) to

```
List[Number] for-some { type $T <: Number }
```

which is in turn equivalent (by simplification rules 2 and 3 above[8]) to

```
List[Number] .
```

Since this `List` type is covariant in its type parameter, then e.g. `List[Integer]` is still a subtype of `List[Number]`.

[7] As `$T` appears in covariant position in `List`, its upper bound can replace the type variable in `List`.
[8] The type variable `$T` is unused, and after dropping it, the quantification is empty.

3.3.11 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 (§5.9) 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.

3.3.12 Unions

Syntax:

```
Const_Type_Def ::= 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.

Unions are indeed virtually "tagged" with the actual type that they represent at the runtime moment, although when it comes to overloading resolution, the union type is used, as it is the expected type.

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 types $(p_1 : T_1, \ldots, p_n : T_n)$ for some $n \geq 0$ and R is a (value or method) type. This type represents named or anonymous methods that take arguments of types $T_1, \ldots, T_n$ and return a result of type R. Types of parameters are possibly annotated with conformance restricting annotations (§10).

Method types associate to the right:[9]
$(Ps_1) \mapsto (Ps_2) \mapsto R$ is treated as $(Ps_1) \mapsto ((Ps_2) \mapsto R)$.

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.[10]

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 (§3.3.9).

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 (x: Integer): Unit
def g (x: Integer)(y: Integer): Integer
def h (x: Integer): (y: Integer) -> Integer
```

produce the typings

```
a : () ↦ Integer
b : (Integer) ↦ Boolean
c : (Integer) ↦ (y: String, z: String) ↦ String
d : (@[named_param :x] Integer) ↦ Integer
e : (@[repeated_param :x] Integer) ↦ Integer
f : (Integer) ↦ Unit
g : (Integer) ↦ (Integer) ↦ Integer
g : (Integer) ↦ (Integer) ↦ Integer
```

[9] Like in Haskell or Scala.

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

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 $[tps] \mapsto T$, where $[tps]$ is a type parameter section $[\pm a_1 >: L_1 <: U_1 , \ldots, \pm a_n >: L_n <: U_n]$ for some $n \geq 0$ and T is a (value or method) type. This type represents (only[11]) named methods that take type arguments $S_1, \ldots, S_n$, for which the lower bounds $L_1, \ldots, L_n$ conform (§3.7.2) 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.

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. $[\pm a_1 >: L_1 <: U_1, \ldots, \pm a_n >: L_n <: U_n] \mapsto T$ 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 Base Types & Member Definitions

Types of class members depend on the way the members are referenced. Central here are these notions:

[11] Not anonymous.

1. The notion of the set of base types of a type T.
2. The notion of a type T in some class C seen from some prefix type S.
3. The notion of the set of member bindings of some type T.

These notions are defined mutually recursively as follows.

1. The set of *base types* of a type is a set of class types, given as follows.
   - The base types of a class type C with parents $T_1, \ldots, T_n$ are C itself, as well as the base types of the compound type T_1 **with** ... **with** T_n.
   - The base types of an aliased type are the base types of its alias.
   - The base types of an abstract type[12] are the base types of its upper bound.
   - The base types of a parameterized type $C[T_1, \ldots, T_n]$ are the base types of type C, where every occurence of a type parameter a_i of C has been replaced by the corresponding parameter type T_i.
   - The base types of a compound type T_1 **with** ... **with** T_n **with** { R } are set of base classes of all T_i's.
   - The base types of a type projection $S\#T$ are determined as follows: If T is an alias or an abstract type, the previous clauses apply. Otherwise, T must be a (possibly parameterized) class type, which is defined in some class B. Then the base types of $S\#T$ are the base types of T in B as seen from the prefix type S.
   - The base types of an existential type T **for-some** { Q } are all types S **for-some** { Q }, where S is a base type of T.
2. The notion of a type *T in class C seen from some prefix type S* makes sense only if the prefix type S has a type instance of class C as a base type, say $S'\#C[T_1, \ldots, T_n]$. Then we define it as follows.
   - If S = ϵ**.type**, then T in C seen from S is T itself.
   - Otherwise, if S is an existential type S' **for-some** { Q }, and T in C seen from S' is T', then T in C seen from S is T' **for-some** { Q }.
   - Otherwise, if T is the i^{th} type parameter of some class D, then:
     - If S has a base type $D[U_1, \ldots, U_n]$, for some type parameters $U_1, \ldots, U_n$, then T in C seen from S is U_i.
     - Otherwise, if C is defined in a class C', then T in C seen from S is the same as T in C' seen from S'.
     - Otherwise, if C is not defined in another class, then T in C seen from S is T itself.

[12] E.g. type members.

- Otherwise, if T is the singleton type D`.self.type` for some class D, then:
  - If D is a subclass of C and S has a type instance of class D among its base types, then T in C seen from S is S.
  - Otherwise, if C is defined in a class C', then T in C seen from S is the same as T in C' seen from S'.
  - Otherwise, if C is not defined in another class, then T in C seen from S is T itself.
- If T is some other type, then the described mapping is performed on all its type components.

If T is a possibly parameterized class type, where T's class is defined in some other class D, and S is some prefix type, then we use "T seen from S" as a shorthand for "T in D seen from S".

3. The *member bindings* of a type T are:

   (a) All bindings d, such that there exists a type instance of some class C among the base types of T and there exists a definition or declaration of d' in C, such that d results from d' by replacing every type T' in d' with T' in C seen from T.

   (b) All bindings of the type's refinement (§3.3.7), if it has one.

   The definition of a type projection $S\#t$ is the member binding d_t of the type t in S. In that case, we also say that $S\#t$ *is defined by* d_t.

3.6 Any-Value Type

Syntax:

```
Simple_Type ::= 'Any'
```

This type does not represent a single concrete value type, but any concrete type. It is used in places where dynamic typing is desired.

With respect to overloading resolution (§6.26.1), this type is always the least specific.

3.7 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.

3.7.1 Type Equivalence

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

- If t is defined by a type alias `type t := T`, then t is equivalent to T.
- If a path p has a singleton type q`.type`, then p`.type` ≡ q`.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 & §5.3.5) are equivalent, if they bind the same names and the modifiers, types and bounds of every declared entity are equivalent in both refinements.
- Two method types (§3.4.1) are equivalent, if they are *override-equivalent* (§4.7.12).
- 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 variances, lower and upper bounds of corresponding type parameters.
- Two existential types (§3.3.10) 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.

3.7.2 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.
- A singleton type p`.type` conforms to the type of the path p.
- A type projection T`#`t conforms to U`#`t if T conforms to U.
- A unit of measure type t conforms to another unit of measure type u if and only if $t \equiv u$ or t extends u, where us is an abstract unit of measure type.
- A parameterized type $T[T_1, \ldots, T_n]$ conforms to $T[U_1, \ldots, U_n]$ if the following conditions hold for $i = 1, \ldots, n$.
  - If the i^{th} type parameter of T is declared covariant, then $T_i <: U_i$.
  - If the i^{th} type parameter of T is declared contravariant, then $U_i <: T_i$.
  - If the i^{th} type parameter of T is declared invariant (neither covariant nor contravariant), then $U_i \equiv T_i$.
  - If the i^{th} type parameter of T is declared with a conformance restricting annotation, then that annotation of T_i must conform to the corresponding annotation of U_i.
- A unit of type N conforms to unit of type U if N extends U or if $N \equiv U$.
- A parameterized type $T[<N_1, \ldots, N_n>]$ conforms to $T[<U_1, \ldots, U_n>]$ if the following conditions hold for $i = 1, \ldots, n$.
  - For the i^{th} type parameter of T, $N_i <: U_i$.
- A compound type T_1 `with` … `with` T_n `with { }` R conforms to each of its component types T_i, and to any compound type created as a combination of subsets of the components, excluding empty sets.
- If $T <: U_i$ for $i = 1, \ldots, n$, and every binding d of a type or value x in R exists a member binding of x in T which subsumes d, then T conforms to the compound type U_1 `with` … `with` U_n `with { }` R.
- The existential type T `for-some { }` Q conforms to U, if its skolemization (§3.3.10) conforms to U. This also means that the t'_i type variables have to fall in between U's type parameter bounds.
- The type T conforms to the existential type U `for-some { }` Q if T conforms to at least one of the type instances (§3.3.10) of U `for-some { }` Q.
- If $T_i \equiv T'_i$ for $i = 1, \ldots, n$ and $R <: R'$, then the method type $(p_1 : T_1, \ldots, p_n : T_n) \mapsto R$ conforms to $(p'_1 : T'_1, \ldots, p'_n : T'_n) \mapsto R'$.
- The polymorphic type or type constructor

  $$[\pm a_1 >: L_1 <: U_1, \ldots, \pm a_n >: L_n <: U_n] \mapsto T$$

conforms to the polymorphic type or type constructor

$$[\pm a_1 \texttt{ >: } L'_1 \texttt{ <: } U'_1, \ldots, \pm a_n \texttt{ >: } L'_n \texttt{ <: } U'_n] \;\mapsto\; T'$$

if one has $T <: T'$, and $L_i <: L'_i$ and $U_i <: U'_i$ for $i = 1, \ldots, n$.

- Polymorphic types or type constructors T and T' must also fulfil the following. We characterize T and T' by their type parameter clauses $[a_1, \ldots, a_n]$ and $[a'_1, \ldots, a'_n]$, where an a_i or a'_i may include a variance annotation, annotations, a higher-order type parameter clauses, and bounds. Then, T conforms to T' if any list $[t_1, \ldots, t_n]$ – with declared variances, annotations, bounds and higher-order type parameter clauses – of valid type arguments for T' is also a valid list of type arguments for T and $T[t_1, \ldots, t_n] \texttt{ <: } T'[t_1, \ldots, t_n]$. Note that this entails that:
  - The bounds on a_i must be the same or weaker than the corresponding bounds declared for a'_i.
  - The variance of a_i must match the variance of a'_i, where covariance matches covariance, contravariance matches contravariance and any variance matches invariance.
  - If annotation of a'_i restricts conformance (§10), then the corresponding annotation of a_i must conform to it.
  - Recursively, these restrictions apply to the corresponding higher-order type parameter clauses of a_i and a'_i.
- A function type $(T_1, \ldots, T_n) \texttt{ -> } R$ (name it f) conforms to a function type $(T'_1, \ldots, T'_m) \texttt{ -> } R'$ (name it f'), if types of arguments that are applicable to f' are also applicable to f (§6.6), and if R conforms to R'. This includes reordering of named arguments/parameters and handling of repeated/optional parameters, and also variances – since although f might be applied to whatever f' might be applied to, f' might not be applied to whatever f might be applied to (and vice versa).
- Polymorphic function traits `Function_`n and `Partial_Function` follow the rules for conformance of function types, as defined above. The repeated parameter, optional parameters, named parameters and all capturing parameters are derived from their conformance restricting annotations (§10). Note that optional parameters may not be expressed with function types in another than with an annotation. The rules may be inverted in the means of constructing a virtual methods m and m' that are reconstructed from the type arguments of the function types f and f' respectively, and applying the rules for function types on them.

A declaration or definition in some compound type of class type C *subsumes* another declaration of the same name in some compound type or class type C', if one of the following conditions holds.

- A value declaration or definition that defines a name x with type T subsumes a value or method declaration that defines x with type T', provided $T <: T'$.
- A method declaration or definition that defines a name x with type T subsumes a method declaration that defines x with type T', provided $T <: T'$.
- A type alias **type** $t[T_1, \ldots, T_n]$:= T subsumes a type alias **type** $t[T_1, \ldots, T_n]$:= T' if $T \equiv T'$.
- A type declaration **type** $t[T_1, \ldots, T_n]$ >: L <: U subsumes a type declaration **type** $t[T_1, \ldots, T_n]$ >: L' <: U', if $L' <: L$ and $U <: U'$.
- A type or class definition that binds a type name t subsumes an abstract type declaration **type** $t[T_1, \ldots, T_n]$ >: L <: U, if $L <: t <: U$.

The (<:) relation forms partial order between types, i.e. it is transitive, antisymmetric and reflexive. The terms *least upper bound* and *greatest lower bound* of a set of types are understood to be relative to that order.

Note. The least upper bound or the greatest lower bound of a set of types does not always exist. Coral is free to reject a term which has a type specified as a least upper bound or a greatest lower bound, and that bound would be more complex than a preset limit, e.g. this could happen with infinite bounds sequence.

The least upper bound or the greatest lower bound might also not be unique. If there are several such bounds, Coral is free to pick up any of them.

3.7.3 Weak Conformance

For now, *weak conformance* is a relation defined on members of the `Number` type as a relaxation of conformance, written as $S <:_w T$. 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).

A *weak least upper bound* is a least upper bound with respect to weak conformance.

3.8 Reified Types

Unlike in Java or Scala, *type erasure* does not exist in Coral. Instead, type arguments are *reified* – meaning that they persist in runtime. This is achieved by generating a lightweight subtype of parameterized types, containing basically just a reference to the parameterized type and a tuple of type arguments. This also implies that each new combination of type arguments to the exact same parameterized type creates a new lightweight subtype.

Reified types have some major effects on programs in Coral:

- Type arguments are accessible in runtime. The actual type argument can be inspected via reflection.
- Type arguments do not go away after compilation. This means, for example, that mutable collections should have invariant type parameters, since a hypothetical `List[+T]` can have type instance `List[String]` assigned to a variable bound to be a `List[Object]`, but instances of other subclasses than those that conform to `String` will not be able to be added to the collection. This is in fact true even if Coral did have type erasure – the difference is, with reified types, the addition of a new incompatible value will fail immediately, unlike with type erasure, where retrieving the added value would fail later.

Chapter 4

Basic Declarations & Definitions

Syntax:

```
Dcl         ::= 'val' Val_Dcl
              | 'var' Var_Dcl
              | 'def' Def_Dcl
              | 'message' Fun_Dcl 'end' ['message']
              | 'function' Fun_Dcl 'end' ['function']
              | 'type' Type_Dcl
Pat_Var_Def ::= 'val' Pat_Def
              | 'var' Var_Def
              | 'let' ['!'] id ':=' Expr
Def         ::= Pat_Var_Def
              | 'def' Fun_Def
              | 'method' Fun_Def 'end' ['method']
              | 'method' Fun_Alt_Def
              | 'function' Fun_Def 'end' ['function']
              | 'function' Fun_Alt_Def
              | 'type' Type_Def
              | Tmpl_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

Syntax:

```
Dcl         ::= 'val' Val_Dcl
Val_Dcl     ::= var_ids ':' Type
Pat_Var_Def ::= 'val' Pat_Def
              | 'let' ['!'] id ':=' Expr
Pat_Def     ::= Pattern2 {',' Pattern2} [':' Type] ':=' Expr
              | [[var_ids ',']
                '*' id] var_ids
                ':=' [[Expr {',' Expr} ','] '*' Expr]
                Expr {',' Expr}
var_ids     ::= id {',' id}
```

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 (§3.7.2).

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.

[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.

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 instead of a name.
2. They can't be lazy.
3. They can't be used in a declaration, only in a definition.
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 (§6.27).[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:

[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.

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

```
val x$ := match e
  when p then (x_1, ..., x_n)
end match
val x_1 := x$[1]
...
val x_n := x$[n]
```

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

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

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 Some(x), y := f()
val x ~> xs := my_list
```

The last three definitions have the following expansions:

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

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

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 "_=".

The following shorthands are recognized:

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

4.2 Variable Declarations & Definitions

Syntax:

```
Dcl         ::= 'var' Var_Dcl
Pat_Var_Def ::= 'var' Var_Def
Var_Dcl     ::= var_ids ':' Type
Var_Def     ::= Pat_Def
              | var_ids ':' Type ':=' '_'
```

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 **var** 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 (§3.7.2), 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 definition **var** x**:** T **:=** `_` introduces a mutable variable with type T and a default initial value. The default value depends on the type T as follows:

`0` if T is `Integer` or one of its subrange types,
`0L` if T is `Long`,
`0.0f` if T is `Float`,
`0.0d` if T is `Double`,
`no` if T is `Boolean`,
`()` if T is `Unit`,
`nil` for all other types T.

It is an error if the type T is not nullable and is expected to have a default value at the same time.

The following shorthands are recognized:

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.

4.3.1 Property Implementations

```
Prop_Get_Impl ::= '(' ')' '->' '{' Block '}'
                | '{' Block '}'
Prop_Set_Impl ::= '(' id ')' '->' '{' Block '}'
                | '{' '|' id '|' semi Block '}'
```

Property implementations are restricted to parameterless block expressions for property getters and to block expressions with one parameter for property setters. If the property is specified with a **get** or **set**, but without a property getter or setter defined, then a default implementation is provided for the missing definitions, based on the property specifications.

4.4 Type Declarations & Aliases

Syntax:

```
Dcl      ::= 'type' Type_Dcl 'end' ['type']
Type_Dcl ::= id [Type_Param_Clause]
             ['>:' Type] ['<:' Type]
Def      ::= 'type' Type_Def
Type_Def ::= id [Type_Param_Clause]
             (':=' | 'is') ['alias'] Type
```

[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.

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 a first-order type, otherwise *t* stands for a type constructor that accepts type arguments as described by the type parameter clause.

If a type declaration appears as a member declaration of a type, implementations of the type may implement *t* with any type *T*, for which $L <: T <: U$. It is an error if *L* does not conform to *U*. Either or both bounds may be omitted. If the lower bound *L* is omitted, the bottom type `Nothing` is implied. If the upper bound *U* is omitted, the top type `Object` is implied.

A type constructor declaration declaration imposes additional restriction on the concrete types for which *t* may stand. Besides the bounds *L* and *U*, the type parameter clause, indexing parameter clause and units of measure parameter clause may impose higher-order bounds and variances, as governed by the conformance of type constructors (§3.7.2).

The scope of a type parameter extends over the bounds `>:` *L* `<:` *U* and the type parameter clause *tps* itself. A higher-order type parametre clause (of an abstract type constructor *tc*) has the same kind of scope, restricted to the declaration of the type parameter *tc*.

To illustrate nested scoping, these declarations are all equivalent:

```
type t[m[x] <: Bound[x], Bound[x]] end

type t[m[x] <: Bound[x], Bound[y]] end

type t[m[x] <: Bound[x], Bound[_]] end,
```

as the scope of, e.g., the type parameter of *m* is limited to the declaration of *m*. In all of them, *t* is an abstract type member that abstracts over two type constructors: *m* stands for a type constructor that takes one type parameter and that must be a subtype of `Bound`, *t*'s second type constructor parameter. However, the first example should be avoided, as the last `x` is unrelated to the first two occurrences, but may confuse the reader.

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.

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] end
type Two[A] := Tuple_2[A, A]
type My_Collection[+X] <: Iterable[X] end
```

The following are illegal:

```
type Abs := Comparable[Abs] end // recursive type alias

type S <: T end  // S, T are bounded by themselves
type T <: S end

type T >: Comparable[T.That] end // can't select from T
                                 // T is a type, not a value
type My_Collection <: Iterable end
  // type constructor members must explicitly state
  // their type parameters
```

4.5 Type Parameters

Syntax:

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

Type parameters appear in type definitions, class definitions and function definitions. In this section we consider only type parameter definitions with lower bounds `>:` L and upper bounds `<:` U, whereas a discussion of context bounds `:` T and view bounds `<%` T is deferred to chapter about implicit parameters and views (§7).

The most general form of a first-order type parameter is $a_1 \ldots a_n \pm t$[*tps*] `>:` 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. 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 (§6.14 & §10).

The names of all type parameters must be pairwise different in their enclosing type parameter clause. The scope of a type parameter includes in each case the whole type

parameter clause. Therefore it is possible that a type parameter appears as part of its own bounds or the bounds of other type parameters in the same clause. However, a type parameter may not be bounded directly or indirectly by itself.

A type parameter may also contain a nested type parameter. This is for cases when the expected type argument is a type constructor.

The above scoping restrictions are generalized to the case of nested type parameter clauses, which declare higher-order type parameters. Higher-order type parameters (the type parameters of a type parameter *t*) are only visible in their immediately surrounding parameter clause (possibly including clauses at a deeper nesting level) and in the bounds of *t*. Therefore, their names must only be pairwise different from the names of other visible type parameters. Since the names of higher-order type parameters are thus often irrelevant, they may be denoted with a "_", which is nowhere visible.

A type parameter name may optionally be surrounded with angle brackets, `<`*t* `<:` *u*`>`. This makes the parameter name *t* stand in for a unit of measure parameter, which may have an upper bound *u*, representing the abstract unit of measure. As units of measure do not have any deeper hierarchy structure, variance annotations are not applicable to them.

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

```
[S, T]
[@[specialized] S, T]
[Ex <: Raiseable]
[A <: Comparable[B], B <: A]
[A, B >: A, C >: A <: B]
[M[X], N[X]]
[M[_], N[_]] // equivalent to previous clause
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
[E, <Length_Unit <: length>]
```

The following type parameter clauses are illegal:

```
[A >: A]                 // illegal, 'A' has itself as bound
[A <: B, B <: C, C <: A] // illegal, 'A' has itself as bound
[A, B, C >: A <: B]      // illegal lower bound 'A' of 'C'
                         // does not conform to upper bound 'B'
```

4.6 Variance of Type Parameters

Variance annotations indicate how instances of parameterized types relate with respect to subtyping (§3.7.2). A "+" variance indicates a covariant dependency, a "–" variance

indicates a contravariant dependency, and an empty variance indicates an invariant dependency.

A variance annotation constrains the way the annotated type variable may appear in the type or class which binds the type parameter. In a type definition **type** T[*tps*] `:=` S, or a type declaration **type** T[*tps*] `>:` L `<:` U, type parameters labeled "+" must only appear in covariant position (positive), whereas type parameters labeled "–" must only appear in contravariant position (negative), and type parameters without any variance annotation can appear in any variance position. Analogously, for a class definition **class** C[*tps*](*ps*) **extends** T { **requires** x`:` S ... }, type parameters labeled "+" must only appear in covariant position in the self type S and the parent template T, whereas type parameters labeled "–" must only appear in contravariant position in the self type S and the parent template T.

The variance position of a type parameter in a type or template is defined as follows. Let the opposite of covariance be contravariance (thus positive positions are flipped to negative positions and vice versa), and the opposite of invariance be itself. The top-level of the type or template is always in covariant position (positive). The variance position changes at the following constructs.

- The variance position of a method parameter is the opposite of the variance position of the enclosing parameter clause.
- The variance position of a type parameter is the opposite of the variance position of the enclosing type parameter clause.
- The variance position of the lower bound of a type declaration or a type parameter is the opposite of the variance position of the type declaration or the type parameter.
- The type of a mutable (instance) variable is always in invariant position (negative).
- The right-hand side of a type alias is always in invariant position (negative).
- The prefix S of a type projection $S\#T$ is always in invariant position (neutral).
- For a type argument T of a type $S[\ldots T\ldots]$: If the corresponding type parameter is covariant, then the variance position stays unchanged. If the corresponding type parameter is invariant (no variance annotation), then T is in invariant position (neutral). If the corresponding type parameter is contravariant, the variance position of T is the opposite of the variance position of the enclosing type $S[\ldots T\ldots]$.

Example 4.6.1 In the following example, variance of positions is annotated with $^+$ (for positive) or $^-$ (for negative):

```
abstract class Cat[-T⁺, +U⁺] {
  def meow [W⁻] (volume: T⁻, listener: Cat[U⁺, T⁻]⁻):
    Cat[Cat[U⁺, T⁻]⁻, U⁺]⁺
}
```

The positions of the type parameter, `W`, and the two value parameters, `volume` and `listener`, are all negative (flipped on type parameters and method parameters). Looking at the result type of `meow`, the position of the first `Cat[U, T]` argument is negative, because `Cat`'s first type parameter, `T`, is annotated with a "–". The type `U` inside this argument is again in a positive position (two flips), whereas the type `T` inside that argument is still in negative position.

References to the type parameters in object-private or object-protected values, types, variables, or methods (§5.2) of the class are not checked for their variance position. In these members the type parameter may appear anywhere without restricting its legal variance annotations.

Example 4.6.2 The following variance annotation is legal.

```
abstract class P [+A, +B] {
  def first: A end
  def second: B end
}
```

With this variance annotation, type instances of P subtype covariantly with respect to their arguments. For instance,

```
P[Error, String] <: P[Throwable, Object]  .
```

If the members of P are mutable variables, the same variance annotation becomes illegal.

```
abstract class P [+A, +B](x: A, y: B) {
  var @first: A := x   // error: illegal variance:
  var @second: B := y  // 'A', 'B' occur in invariant position
}
```

If the mutable variables are object-private, the class definition becomes legal again:

```
abstract class R [+A, +B](x: A, y: B) {
  private[self] var @first: A := x   // ok
  private[self] var @second: B := y  // ok
}
```

Example 4.6.3 The following variance annotation is illegal, since a appears in contravariant position in the parameter of `append`:

```
abstract class Sequence [+A] {
  def append (x: Sequence[A]): Sequence[A] end
    // error: illegal variance, 'A' occurs in contravariant position
}
```

The problem can be avoided by generalizing the type of append by means of lower bound:

```
abstract class Sequence [+A] {
  def append [B >: A] (x: Sequence[B]): Sequence[B] end
    // error: illegal variance, 'A' occurs in contravariant position
}
```

Example 4.6.4 Here is a case where a contravariant type parameter is useful.

```
abstract class Output_Channel [-A] {
  def write (x: A): Unit
}
```

With that annotation, we have that `Output_Channel[Object]` conforms to `Output_Channel[String]`. That is, a channel on which one can write any object can substitute for a channel on which one can write only strings.

4.7 Function Declarations & Definitions

Syntax:

```
Dcl           ::= 'def' Fun_Dcl 'end' ['def']
                | 'message' Fun_Dcl 'end' ['message']
                | 'function' Fun_Dcl 'end' ['function']
                | 'operator' Op_Dcl 'end' ['operator']
Fun_Dcl       ::= Fun_Sig ':' Type
Op_Dcl        ::= Op_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
                | 'operator' Op_Def 'end' ['operator']
                | 'operator' Op_Alt_Def 'end' ['operator']
Fun_Def       ::= Fun_Sig [':' Return_Type] [Fun_Dec] [semi Fun_Stats]
Fun_Alt_Def   ::= Fun_Sig [':' Return_Type] ':=' Expr
Op_Def        ::= Op_Sig [':' Return_Type] [Fun_Dec] [semi Fun_Stats]
```

```
Op_Alt_Def    ::= Op_Sig [':' Return_Type] ':=' Expr
Fun_Dec       ::= [semi] 'declare' Fun_Dec_Exprs [semi] 'begin'
Fun_Dec_Exprs ::= Fun_Dec_Expr {semi Fun_Dec_Expr}
Fun_Sig       ::= Function_Path [Fun_Tpc] Param_Clauses
Op_Sig        ::= Op_Path [Fun_Tpc] Param_Clauses
Fun_Tpc       ::= '[' Type_Param {',' Type_Param} ']'
Function_Path ::= id
                | 'self' '.' id
                | id '.' id
Op_Path       ::= op_id
                | 'self' '.' op_id
                | id '.' op_id
Param_Clauses ::= {Param_Clause} ['(' 'implicit' Params ')']
Param_Clause  ::= '(' [Params] ')'
Params        ::= Param {',' Param}
Param         ::= {Annotation} [Param_Extra] id
                  [':' Param_Type] [':=' Expr]
                  | Literal
Param_Extra   ::= ['lazy'] [Param_Io] [Param_Rw]
                    ['*' | '&' | '^']
Param_Io      ::= 'in' ['out'] | 'out'
Param_Rw      ::= 'val' | 'var'
Param_Type    ::= ['=>' | '=>>'] (Type | tp_id | '_')
```

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 (§3.7.2) to the function's declared result type, if one is given.

If the function's result type is given as one of "`()`" or `Unit`, the function's implicit return value is stripped and it is an error if a return statement occurs in the function body with a value to be returned, unless the return value is specified again as "`()`".

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.

An operator declaration/definition is also a function declaration/definition, only it does not require backticks around the operator's name.

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

1. n mandatory positional parameters (§4.7.7): x`:` T, where $n \geq 0$.
2. n optional positional parameters (§4.7.8): x`:` T `:=` e, where $n \geq 0$.
3. n repeated parameters (§4.7.9): $*x$`:` T, where $0 \leq n \leq 1$.
4. n post mandatory positional parameters (§4.7.7): x`:` T, where $n \geq 0$.
5. n named parameters (§4.7.10): `^`x`:` T `:=` e, where $n \geq 0$.
6. n capturing named parameter (§4.7.10): $**x$`:` T, where $0 \leq n \leq 1$.
7. n captured block parameters (§4.7.11): `&`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()`[6], the default value `0` is inserted and `T` is instantiated to `Number`. The functions computing the default arguments have the forms[7]:

[6]Without any explicit arguments.

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

```
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 (§4.7.6).

4.7.1 Parameter Evaluation Strategies

Note: This section applies (from the other side of the wall) to function applications (§6.6) as well, but it's pointless to have it duplicated over there.

Coral utilizes five parameter (resp. argument) evaluation strategies. Every strategy defers evaluation of argument values until the function application is resolved, using only arguments' expected types – see (§6.26.3).

Call-by-value, strict. Also known as *call-by-object*, *call-by-object-sharing* or *call-by-sharing*, is the default evaluation strategy, applied to all parameters, unless otherwise specified. Such arguments are evaluated prior function invocation and are not re-evaluated. This includes explicit parameter values (§4.7.5).

Call-by-reference, strict. Also known as *pass-by-reference*, the function can modify the original argument variable. Those are described in (§4.7.6) as *output parameters*.

Call-by-name, non-strict. The argument is not evaluated until accessed, and is re-evaluated each time it is accessed, providing another tool to create DSLs. Those are described in (§4.7.2).

Call-by-need, non-strict. Also known as *lazy evaluation*, this is a memoized version of *call-by-name* strategy. The difference is, *call-by-need* parameters are not re-evaluated, once they are evaluated.

Call-by-future, non-deterministic. This is a rather experimental feature of Coral, depending on whether the Coral VM is capable of parallelization and whether there are defined standard tools for "system" parallelization. These are described in (§4.7.4).

4.7.2 By-Name Parameters

Syntax:

```
Param_Extra ::= ['lazy'] ['in'] ['val']
                ['*' | '^']
Param_Type  ::= '=>' (Type | tp_id | '_')
```

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.

The by-name modifier is disallowed for output parameters (§4.7.6 & §6.6.7) and for implicit parameters (§7.2). The by-name modifier implies **val** parameter and is disallowed for **var** parameters.

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.

4.7.3 By-Need Parameters

Syntax:

```
Param_Extra ::= 'lazy' ['in'] [Param_Rw]
                ['*' | '^']
```

The parameter definition may be preceded by **lazy** keyword. This indicates that the corresponding argument is not evaluated before function application, but instead is evaluated the first time used within the function.

The by-need modifier is disallowed for output parameters (§4.7.6 & §6.6.7) and for implicit parameters (§7.2). The by-need modifier implies **val** parameter and is allowed for **var** parameters.

By-need parameters with default value expressions evaluate the default value expression the first time the parameter is accessed, like optional parameters that evaluate the default value expression only once.

4.7.4 By-Future Parameters

Syntax:

```
Param_Extra ::= ['lazy'] ['in'] ['val']
                ['*' | '^']
Param_Type  ::= '=>>' (Type | tp_id | '_')
```

By-future uses a concurrent evaluation strategy: the value of a future expression is computed concurrently with the flow of the rest of the program (on a new thread/-worker). When the value of the future is needed, the invoking thread blocks until the future finishes computing, if it has not already completed by then.

This strategy is non-deterministic, as the evaluation can occur at any time between when the future is created (when the function is applied) and when the value of the future is used.

The by-future modifier is disallowed for output parameters (§4.7.6 & §6.6.7) and for implicit parameters (§7.2). The by-future modifier implies **`val`** parameter and is disallowed for **`var`** parameters.

By-future parameters with default value expressions evaluate the default value also as a future value, if the argument is not explicitly given.

By-future parameter prefix is omitted from examples of valid parameter definitions, but it's allowed occurrence is the same as of by-name parameters (§4.7.2).

4.7.5 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 inferred type of the literal value. Methods with explicit parameters are preferred during method resolution to methods with the same parameter types (§6.6), 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.

4.7.6 Input & Output Parameters

Syntax:

```
Param_Io ::= 'in' ['out'] | 'out'
```

If no input/output parameter specifier is explicitly available, then the parameter is implicitly an input parameter. Output parameters require a specific application (§6.6.7).

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`**.

4.7.7 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 (§4.7.6) and by-name (§4.7.2). Positional parameters can't have any default value expressions.

4.7.8 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 (§4.7.6)[8] and by-name modifier (§4.7.2). 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 (§6.6.5). Optional parameters are disallowed for output parameters and repeated parameters.

Optional parameters add the annotation (§10) `@[optional_param]` to the corresponding parameter type of the function trait.

4.7.9 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 `Sequence[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
```

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

```
    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:[9]

```
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
```

4.7.10 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
```

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

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

Capturing named parameter are of the form:

$**x$**:** T
var $**x$**:** T
val $**x$**:** T
in $**x$**:** T
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 (§6.6 & §6.6.5). 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.

Capturing named parameter are capturing any other applied named parameters that were not captured by their explicit declaration (§6.6.5). 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 capturing named parameter take a variable number of named arguments of type T mixed with other named arguments and before the captured block parameter. capturing named parameter 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. capturing named parameter are **val**-flagged implicitly, unless explicitly flagged as **var**, to protect the captured elements from accidental overwrite.

4.7.11 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 (§6.6). The captured block parameter may be used also to capture function arguments, e.g. anonymous functions (§6.23), then the type is used during method resolution.

It is an error if a block parameter type T is provided and it is not a function type (§3.3.9). 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 (§3.7.2) 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).

4.7.12 Method Signature

Two methods M and N have the same signature, if they have the same name, the same type parameters (if any), the same parameters with equivalent types, and equivalent return type.

The signature of a method m_1 is a *subsignature* of the signature of a method m_2 if either:

- m_2 has the same signature as m_1, or
- the signature of m_1 has the same name, the same type parameters (if any), the same parameters with equivalent types, and a return type that conforms to return type of m_2.

A method signature m_1 is *override-matching* m_2, if m_1 is a subsignature of m_2. Two method signatures m_1 and m_2 are *override-equivalent*, iff m_1 is the same as m_2.

4.8 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 (§5.1.8) 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[10] (m does not override any m' known at compile-time), then the parameters are inferred to be of type `Object`.

Example 4.8.1 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 4.8.2 Assume the following definitions:

```
protocol I
  def factorial(x: Integer): Integer
end
class C
  implements I
```

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

```
  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.

4.9 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 (§6.6).

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 4.9.1 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
```

4.10 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' [id | '_']]
Container_Path  ::= Module_Path ['.' Constant_Path]
                  | [Root] Constant_Path
Module_Path     ::= [Root] Module_Selector {'.' Module_Selector}
Constant_Path   ::= Const_Selector {'.' Const_Selector}
Module_Selector ::= id [Vendor_Arg]
Const_Selector  ::= id
Root            ::= 'Root' '~'
Vendor_Arg      ::= '~[' vendor_domain ']'
vendor_domain   ::= vendor_char {vendor_char}
vendor_char     ::= lower | '.' | '-' | '_'
```

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*[11] 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*

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

```
{ x1 as y1 ,..., xn as yn, _ }
```

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[12], 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.

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 4.10.1 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) } .
```

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

Chapter 5

Classes & Objects

Syntax:

```
Tmpl_Def ::= ['case'] 'class' Class_Def
           | ['case'] 'object' Object_Def
           | 'trait' Trait_Def
           | 'protocol' Pro_Def
           | ['case'] 'interface' Ifc_Def
           | 'refinement' Refinement_Def
           | 'aspect' Aspect_Def
           | 'type' Const_Type_Def 'end' ['type']
```

Classes (§5.3) & objects (§5.4) are both defined in terms of *templates*.

5.1 Templates

Syntax:

```
Class_Parents  ::= Constr {['prepend'] 'with' Annot_Type}
Trait_Parents  ::= Annot_Type {['prepend'] 'with' Annot_Type}
Template_Body  ::= [Self_Type] Template_Stat {semi Template_Stat}
Self_Type      ::= 'requires' 'self' ':' Type semi
                 | 'requires' [id ':'] Type semi
                 | 'use' ['self'] 'as' id [':' Type] semi
```

A template defines the type signature, behaviour and initial state of a trait, class of objects or of a single object. Templates for part of instance creation expressions (constructors, see §5.1.3 & §5.3.1), class definitions and object definitions. A template *sc* **with** mt_1 **with** ... **with** mt_n { *stats* } consists of constructor invocation *sc*, which defines the template's *superclass*, trait references $mt_1, \ldots, mt_n$ ($n \geq 0$), which

statically define the template's included traits[1], and a statement sequence *stats*, which contains initialization code and additional member definitions & declarations for the template. Unlike in Scala, all trait references in class/trait parents need not to be exhaustive, as more prepended/included traits may be defined as a part of the template body. Trait references declared using **`prepend with`** are prepended to the template body instead of included (§5.1.6).

Each trait reference mt_i that is not prepended must denote a trait (§5.3.4). By contrast, the superclass constructor *sc* normally refers to a class which is not a trait. It is possible to write a list of parents that starts with a trait reference, e.g. mt_1 **`with`** ... **`with`** mt_n . In that case, the list of parents is implicitly extended to include the supertype of mt_1 as first parent type. This new supertype must have at least one constructor that does not take parameters and is accessible to the subclass (§5.2).

The list of parents of a template must be well-formed, i.e. the class denoted by the superclass constructor *sc* must be a subclass (or the superclass itself) of the superclasses of all the traits $mt_1, \ldots, mt_n$.

The *least proper supertype* of a template is the class type or compound type (§3.3.7) consisting of all its parent class types.

The statement sequence *stats* contain member definitions that define new members or overwrite members in the parent classes. It is called also the *class-level block*, as it does not need to contain only member definitions for the template, but also arbitrary other expressions that construct the class object and that are executed while the class is being loaded, in the context of the class. If the template forms part of an abstract class or trait definition, the statement part *stats* may also contain declarations of abstract members. If the template forms part of a concrete class definition, *stats* may still contain declarations of abstract type members, but not of abstract term members. Unlike in Scala, the expressions in *stats* are not forming the primary constructor of the class, but a multi-constructor[2] of the class itself.

The sequence of template statements may be prefixed with a formal parameter definition prefixed with **`requires`** or **`use`**, i.e. **`use self as`** x, **`use self as`** x**`:`** T, **`requires`** T or **`requires`** x**`:`** T. If a formal parameter x is given, it can be used as an alias for the reference **`self`** throughout the body of the template, including any nested types. If the formal parameter x comes with a type T, this definition affects the *self type* S of the underlying class or objects as follows: Let C be the type of the class or trait or object defining the template. If a type T is given for the formal self parameter, S is the greatest lower bound of T and C. If no type T is given, S is simply C. Inside the template, the type of **`self`** is assumed to be S.

The self type of a class or object must conform to the self types of all classes which are inherited by the template t.

[1] Including protocols, which are also traits.

[2] The classes are open in Coral, a single class may have its statements spread across multiple source files (§5.1.1).

A second form of self type definition reads just **`requires self:`** *S* . It prescribes the type *S* for **`self`** without introducing an alias name for it.

Example 5.1.1 Consider the following class definitions:

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

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

```
object O extends Base with Mixin; ... end
```

5.1.1 Open Templates

Unlike in Java, Scala or any other language that does not feature open classes, and similar to Ruby's open classes or C#'s partial classes, Coral has a feature of open templates. This means that a single template (be it a class or a trait) can have its definition spread across several source files.

An open template has a *leading template*, which is the template that specifies the base class – a property that can't be changed once set, and any number of *auxiliary templates*, which have only the limitation of not being able to define the base class (and a primary constructor, including a parent constructor invocation). However, it is possible for the auxiliary templates to define additional traits applied or prepended to the template.

Auxiliary templates do not need to be eager loaded. There are basically three major ways to make use of an auxiliary template:

- Eager loaded auxiliary template, using the construct **`include`** `'path/to/file'`. This way the auxiliary template is loaded along with the leading template.
- Autoload (§5.1.2) of the auxiliary template.
- On-demand load of the auxiliary template via independent `VM.load 'path/to/file'`, after the leading template is defined.

To declare a template as an open template, the following annotations are available. All templates are implicitly open, so the annotations are only useful to prevent implicit inheriting from `Object`, if the intended leading template inherits from a different base class.

```
@[Open_Template :leading]
@[Open_Template :auxiliary]
```

5.1.2 Autoloading

Autoloading is a mechanism of putting code from different source files and source directories together in runtime. There are two kinds of autoloading.

Implicit autoloading. This kind of autoloading is basically the expected layout of source files, source directories and terms inside of a directory. Given that a root directory represents a module sources root, each template definition is expected to appear in a file that directly appears in this directory, preferably with a name of the included template. If multiple templates appear in one source file, then the implicit autoloading uses the mechanism of template index to track location of these templates. However, if the name of the template is generated dynamically in runtime (i.e. not via template definition or a constant initialization), then it needs to be loaded using explicit autoloading. Also, if the source file is used as a script, every such term needs to be loaded using explicit autoloading, since the compiler does not know about anything that hasn't been yet loaded or otherwise defined and is not in the same source file.

Explicit autoloading. This kind of autoloading is basically a custom definition of the layout of source files, source directories and terms inside of a directory. Explicit autoloading happens every time a member is accessed in a template and it is not defined. Explicit autoloading is defined in means of using the `autoload` method, provided by the `Class` class, and therefore available inside of any template definition. The method has the following overloaded declarations:

```
type Autoloaded_Term is union of ( Symbol, Regexp )
end type
def autoload (term: Autoloaded_Term,
              path: String): Unit
end def
def autoload (term: Autoloaded_Term := nil,
              callback: (t: Autoloaded_Term) -> Boolean): Unit
end def
```

The first overloaded variant generates an autoloading record that will autoload the source file from the given path. The second overloaded variant expects a callback that will return a boolean indicating whether the term was successfully autoloaded or not, also having the actual autoloaded term as an optional argument – if the argument is omitted, then the callback is registered for every autoloading attempt.

With explicit autoloading, the runtime is allowed to keep track of templates that make use of explicit autoloading to optimize member resolution. If an explicit autoloading is added in runtime, the version number of the template has to be updated to invalidate member caches.

5.1.3 Constructor Invocations

Syntax:

```
Constr ::= Annot_Type {'(' [Exprs] ')'}
         | '(' ')'
```

Constructor invocations define the type, members and initial state of objects created by an instance creation expression, or of parts of an object's definition, which are inherited by a class or object definition. A constructor invocation is a function application $c\,[targs]\,(args_1)\ldots(args_n)$, where c is a path to the superclass or an alias for the superclass, *targs* is a type argument list, $args_1, \ldots, args_n$ are argument lists, and there is a constructor of that class which is applicable to the given arguments.

A type argument list can be only given if the class c takes type parameters. If no explicit arguments are given, an empty list `()` is implicitly supplied, unless an explicit primary constructor definition is given, calling explicitly a super-constructor – in that case, the constructor invocation only defines the superclass, and the invocation itself is deferred to the explicit primary constructor.

The superclass constructor is implicitly invoked before any code that the primary constructor defines, but not before early definitions are evaluated.

5.1.4 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]
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[3].

Table 5.1: Of objects, classes & eigenclasses

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

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 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`.

Metaclass Access. Metaclasses of classes may be accessed using the following language construct.

Syntax:

```
Metaclass_Access ::= 'class' '<<' Metaclass_Obj semi
                     [Exprs] 'end'
Metaclass_Obj    ::= Type | Path | 'self' | id
```

Example 5.1.2 The following code shows how metaclasses are nested in case of `Object` type. Don't try this at home though.

```
class << Object
  self = Metaclass[Object]
  class << self
    self = Metaclass[Metaclass[Object]]
    class << self
```

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

```
      self = Metaclass[Metaclass[Metaclass[Object]]]
    end
  end
end
```

Example 5.1.3 The following code shows what **self** references when inside of a class definition, but outside of any defined methods.

```
class Object extends ()
  self = Class[Object]
  class << self
    self = Metaclass[Object]
  end
end
```

Example 5.1.4 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 5.1.5 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 5.1.6 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 5.1.7 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 5.1.8 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.

5.1.5 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 5.1.9 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** ... **with** C_n. The *linearization* of C, $\mathscr{L}(C)$ is defined as follows:

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

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

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

Example 5.1.10 Consider the following class definitions.[4]

```
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 (§6.6).

5.1.6 Inheritance Trees & Include Classes

Include classes. A mechanism that allows arbitrary including and prepending of trait into classes and inheritance binary trees[5] uses a transparent structure called *include class*. Include classes are always defined indirectly.

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.

[4]Here we say "class", but that term includes now traits as well.

[5]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.

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)[6] 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 5.1.11 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.

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* trait must not require to include a trait that triggered the auto-include. The order in which traits are included in another trait may change when included in a class, i.e. if the class includes two traits A and B that themselves include the same two traits D and E in reverse order (A includes

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

D, then *E*, but *B* includes *E*, then *D*): the order is then defined by the first trait that included the two auto-included traits and subsequently included traits can not change this order in any way.

Example 5.1.12 Take the following trait and class definitions, where *S* is the superclass of the class *C*:

```
trait D end
trait E end

trait A extends D with E; end
trait B extends E with D; end

class C extends S with A with B; end
```

Then traits are auto-included in the following order:

- `C → [S]`
  First, the superclass is added.
- `C → [A] → [S]`
  Then, trait *A* is included, unless already included in *S*.
- `C → [A] → [E] → [S]`
  Including of trait *A* triggers auto-include of traits included in *A*. Start with the first one in chain of *A*: *E*.
- `C → [A] → [E] → [D] → [S]`
  Then, *A* has *D* in its chain.
- `C → [B] → [A] → [E] → [D] → [S]`
  Finally include *B*. *B* triggers auto-include, but both of its included traits are already included in the chain, so nothing more happens.

Note that the order of *E* and *D* is reversed, since later includes move the trait closer to the including class or trait, and therefore **`super`** calls go through traits that were included before. Also, if *B* was included sooner than *A*, then *D* and *E* would appear in reverse order in the chain.

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 trait $M_1, \ldots, M_n$ prepended to *M* is automatically prepended to *C* in this way:

1. Establish a list of traits that triggered auto-prepend, named here tp. This list is in ideal case empty, so it's actually ok for the runtime to wait with its creation until needed and only increase its size in very small steps.

2. Establish a list of traits that are scheduled to be auto-prepended, named here sp. This list is in ideal case empty, again.

3. There is a *prepend chain* in the class that prepends M. If no trait was prepended so far, create it. The chain's head is the element that is the farthest from the class C, the chain's tail is right before the class C. Traits closer to this chain's head are searched for method overlays sooner in runtime than traits closer to the tail (and the class respectively).

4. Insert M at the chain's head, unless M already is in the chain. If it is, then halt.

5. If M has itself prepended traits, insert M into tp (triggered prepend).

6. For traits $M_1, \ldots, M_n$ that are prepended in M, test if each M_i already appears in the chain. If it does, move M up the chain towards the chain's tail, right until M is closer to the chain's tail than M_i. If it does not, add M_i to sp, unless M_i is in tp. If it is, then it is an error[7].

7. If M has included traits, include them in C in the already described way now.

8. If sp is not empty, then for each M_i in sp, remove M_i from sp and recursively apply steps starting with 4 on it. Keep both sp and tp shared for recursive calls. If M_i moves closer to the chain's tail than M or any other trait prepended in prepending of the original M[8], it is an error[9].

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.[10]

5.1.7 Class Members

A class C defined by a template C_1 **with** ... **with** C_n { *stats* } can define members in its statement sequence *stats* and can inherit members from all parent classes. Coral uses overloading of methods, therefore it is possible for a class to define and/or inherit several methods with the same name. To decide whether a defined member of a class C overrides a member of a parent class, or whether the two co-exist as overloaded alternatives in C, Coral uses the following definition of *matching* on members:

[7]Trait cycle dependency detected.

[8]This can be achieved by having a third list of traits that were prepended.

[9]Trait composition design flaw detected.

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

Definition 5.1.13 A member definition M *matches* a member definition M', if M and M' bind the same name, and one of the following conditions holds.

1. Neither M nor M' is a method definition.
2. M is override-matching M' (§4.7.12).

Member definitions fall into two categories: *concrete* and *abstract*. Members of class C are either *directly defined* (i.e. they appear in C's statement sequence *stats*), or they are *inherited*. There are rules that determine the list of members of a class:

Definition 5.1.14 A *concrete member* of a class C is any concrete definition M in some class $C_i \in \mathscr{L}(C)$, except if there is a preceding (or even prepended) class $C_j \in \mathscr{L}(C)$, where $j < i$, which directly defines a concrete member M' matching M, where the M' is then the concrete member.

An *abstract member* of a class C is any abstract definition M in some class $C_i \in \mathscr{L}(C)$, except C already contains a concrete member M' matching M, or if there is a preceding (or even prepended) class $C_j \in \mathscr{L}(C)$, where $j < i$, which directly defines a concrete member M' matching M, where the M' is then the concrete member.

This definition also determines the overriding relationship between matching members of a class C and its parents (§5.1.8). First, a concrete definition always overrides an abstract definition. Second, for definitions M and M', which are both concrete or both abstract, M overrides M' if M appears in a class that precedes (in the linearization of C) the class in which M' is defined.

It is an error if a template directly defines two matching members.

5.1.8 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 (§3.7.2) to the binding of the overridden member M'. Furthermore, the following restrictions on modifiers apply to M and M':

- M' must not be labeled **`final`**.
- M must not be **`private`**.
- 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 not an abstract member, then M should be labeled **`override`** or annotated `@[Override]`. Furthermore, one of the possibilities must hold:
  - either M is defined in a subclass of the class where M' is defined,
  - or both M and M' override a third member M'', which is defined in a base class of both the classes containing M and M'.
- If M' is labeled **`private`**, then M' must be labeled **`private`**, **`protected`** or **`public`**.
- If M' is incomplete (§5.2) in C, then M should be labeled **`abstract override`**.
- If M and M' are both concrete value definitions, then either none of them is marked **`lazy`**, or both must be marked **`lazy`**.

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

An overriding method, unlike in Scala, does not inherit any default arguments from the definition in the superclass, but, as a convenience, if the default argument is specified as '_', then it gets inherited.

5.1.9 Inheritance Closure

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

- If T is in $\mathscr{S}$, then every type T' which forms syntactically a part of T is also in $\mathscr{S}$.
- If T is a class type in $\mathscr{S}$, then all parents of T (§5.1) are also in $\mathscr{S}$.

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

5.1.10 Early Definitions

Syntax:

```
Early_Defs ::= '{' [Early_Def {semi Early_Def}] '}'
               'with'
Early_Def  ::= {Annotation} {Modifier} Pat_Var_Def
```

A template may start with an *early definition* clause, which serves to define certain field values before the supertype constructor is called. In a template

```
{
  val p1: T1 := e1
  ...
  val pn: Tn := en
} with sc with mt1 with ... with mtn
```

The initial pattern definitions of $p_1, \ldots, p_n$ are called *early definitions*. They define fields which form part of the template. Every early definition must define at least one field.

Any reference to **self** in the right-hand side of an early definition refers to the identity of **self** just outside the template, not inside of it. As a consequence, it is impossible for any early definition to refer to the object being constructed by the template, or refer to any of its fields, except for any other preceding early definition in the same section.

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 ::= '[' (id | 'self') ']'
```

Access modifiers may appear in two forms:

Accessibility flag modifier. Such a modifier appears in a template (§5.1) or a module definition (§9.1.1) 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
begin
  public
    def hello; end
  private
    def private_hello; end
end class
```

Example 5.2.2 An example of directly applied modifier:

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

By default[11], 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:

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

- 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).

  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[12]), and the modifier **override** can not be applied to them as well[13].

- 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*[14] 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:

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

[13] 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.

[14] In addition to unqualified **protected** access.

  - 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
  - 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
  begin
    class Inner
    begin
      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:

```
Tmpl_Def            ::= 'class' Class_Def
Class_Def           ::= id [Type_Param_Clause]
                        [Class_Param_Clauses] Class_Tmpl_Env
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')]
                        id [':' Param_Type] [':=' Expr]
Class_Tmpl_Env      ::= 'extends' [Early_Defs] Class_Parents
                        semi [Template_Body] 'end' ['class']
                      | ['extends' [Early_Defs]]
                        '{' [Template_Body] '}'
                      | ['begin' [Template_Body]] 'end' ['class']
```

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 (§5.1.4).

The most general form of a class definition is

class c [tps]
  as $m(ps_1)\ldots(ps_n)$
  extends t

for $n \geq 0$.

Here,

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[15], generating a new concrete subtype of the original generic class. This ensures type safety in a dynamic environment of Coral 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.

m is an access modifier (§5.2), such as **`private`** or **`protected`** (**`public`** is implied otherwise), possibly with a qualification. If such an access modifier is given, it applies to the primary constructor of the class.

$(ps_1) \ldots (ps_n)$ are formal value parameter clauses for the *primary constructor* of the class. The scope of a formal value parameter includes all subsequent parameter sections and the template *t*. However, a formal value parameter may not form part of the types of any of the parent classes or members of the class template *t* – only parameters from *tps* may form part of the types of any of the parent classes or members of the class template *t*. It is illegal to define two formal value parameters with the same name. If no formal parameter sections are given, an empty parameter section `()` is implied.

If a formal parameter declaration *x*`:` *T* is preceded by a **`val`** or **`var`** keyword, an accessor (getter) definition (§4.1) for this parameter is implicitly added to the class. The getter introduces a value member *x* of class *c* that is defined as an alias of the parameter. Moreover, if the introducing keyword is **`var`**, a setter accessor *x*`_=(`*e*`)` is also implicitly added to the class. The formal parameter declaration may contain modifiers, which then carry over to the accessor definition(s). A formal parameter prefixed by **`val`** or **`var`** may not at the same time be a call-by-name parameter (§4.7.2).

If a formal value parameter is a part of the definition of a property (a property getter and/or a property value), then these accessors are not implicitly added, as the arguments may be traced into the properties. The same applies to instance value definitions that have a formal value parameter forming a part of it. In any case, every formal value parameter is implicitly added as an instance value definition of a **`val`** or **`var`** type respectively.

t is a template (§5.1) of the form

sc **`with`** mt_1 **`with`** ... **`with`** mt_n `{` *stats* `}`

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

for ($n \geq 0$), which defines the base classes, behavior and initial state of objects of the class. The extends clause **extends** *sc* **with** mt_1 **with** ... **with** mt_n can be omitted, in which case **extends** `Object` is implied. The only class that defines *sc* as `()` is `Object` (meaning that it has no superclass) and it is an error if any other class attempts to do the same.

This class definition defines a type *c*[*tps*] and a primary constructor, which, when applied to arguments conforming to types *ps*, initializes instances of type *c*[*tps*] by evaluating the primary constructor parts defined in the template *t*.

Example 5.3.1 The following example illustrates **val** and **var** parameters of a class `C`:

```
class C (x: Integer, val y: String, var z: List[String]) end
val c := C.new(1, "abc", List[String].new)
c.z := c.y ~> c.z
```

Example 5.3.2 The following class can be created only from its class object.

```
class Sensitive private () {
  ...
}
object Sensitive {
  def make_sensitive (credentials: Certificate): Sensitive := {
    if credentials.admin?
      Sensitive.new
    else
      raise SecurityViolationException
    end
  }
}
```

5.3.1 Constructor & Destructor Definitions

Primary constructor. The primary constructor is a special function, more special in its syntax than any other function in Coral. Its definition can spread across three syntactically different places inside of a class definition: formal value parameters, explicit field declarations, explicit primary constructor body. Primary constructor without explicitly defined parameters is equivalent to the default constructor.

Primary constructor parameters. These are described as a part of class definitions (§5.3). Technically, a primary constructor is happy being left with nothing but the parameter definitions, since these are automatically mapped to instance values[16] and

[16]Making this behavior overrideable may be discussed, but it might interfere with case classes.

also implicitly adds accessor methods to the class with appropriate visibility, defined explicitly by accessor modifiers (§5.2) or implicitly as **`public`**.

Explicit primary constructor fields. These are instance value definitions inside the class template (§5.1) that use the formal value parameters either with or without any modification. Formal value parameter used without any modification is equivalent to the expression defining the value of an instance value being only the parameter itself. If the parameter is used without modification, it becomes an alias to the instance value implicitly defined by the primary constructor parameter. As the explicit field can be expected to be accessed more often than the original parameter, a Coral VM implementation may opt in to make the implicit instance value being the alias, instead of the explicit field.

Example 5.3.3 An example of an explicit primary constructor field.

```
class C (val y: String) {
  val @x := y                // without modification
  val @z := y + "_modified"  // with modification
}
```

Explicit primary constructor body. If a primary constructor should do something more than simply map parameters to instance values, then its explicit body comes in. It may co-exist with explicit primary constructor fields, which are then executed right after a call to the super-constructor (either in implicit position, or an explicit one). **Syntax:**

```
Def             ::= PCtor_Def
PCtor_Def       ::= 'constructor' PCtor_Fun_Def
PCtor_Fun_Def   ::= '(' '_' ')' (':=' Constr_Expr | Constr_Block)
Constr_Expr     ::= Self_Invocation
                  | Constr_Block2
Self_Invocation ::= 'self' Argument_Exprs {Argument_Exprs}
                  | Sup_Invocation
Sup_Invocation  ::= 'super' [Argument_Exprs {Argument_Exprs}]
Constr_Block    ::= Constr_Block1 | Constr_Block2
Constr_Block1   ::= [semi Constr_Block3] 'end' ['constructor']
Constr_Block2   ::= '{' [Constr_Block3] '}'
Constr_Block3   ::= [[Block_Stat {semi Block_Stat} semi]
                     Self_Invocation semi]
                     Block_Stat {semi Block_Stat}
                  | Self_Invocation
```

Auxiliary constructors. Besides the primary constructor, a class may define additional constructors with different parameter sections, that form together with the primary constructor an overloaded constructor definition. Every auxiliary constructor is

constrained in means that it has to either invoke another constructor defined before itself, or any superclass constructor. Therefore classes in Coral have multiple entry point, but with great power comes great responsibility: user must ensure that every constructor path defines all necessary members. It is an error if a constructor invocation leads to an instance with abstract members. If a constructor does not explicitly invoke another constructor, an invocation of the implicit constructor is inserted implicitly as the first invocation in the constructor.

Syntax:

```
Ctor_Def     ::= 'constructor' Ctor_Fun_Def
Ctor_Fun_Def ::= [Param_Clauses] (':=' Constr_Expr | Constr_Block)

Dtor_Def     ::= 'destructor' Dtor_Fun_Def
Dtor_Fun_Def ::= (':=' Dtor_Expr | Dtor_Block)
Dtor_Expr    ::= Sup_Invocation
               | Dtor_Block2
Dtor_Block   ::= Dtor_Block1 | Dtor_Block2
Dtor_Block1  ::= [semi Dtor_Block3] 'end' ['destructor']
Dtor_Block2  ::= '{' [Dtor_Block3] '}'
Dtor_Block3  ::= [[Block_Stat {semi Block_Stat} semi]
                  Sup_Invocation semi]
                  Block_Stat {semi Block_Stat}
                | Sup_Invocation
```

Default constructor. A default constructor of a class is the explicit parameterless constructor. This differs from Java or C#. The default root constructor of `Object` carries the modifier **`public`** (§5.2), and therefore to explicitly disallow direct object creating, custom constructors have to be explicitly more restrictive than **`public`**.

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 superconstructor 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, including the primary constructor, if it has parameters.

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 5.3.4 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 5.3.5 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 5.3.6 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
CCtor_Fun_Def ::= [Param_Clauses]
                  (':=' Constr_Expr | Constr_Block)
```

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 (which would possibly invoke its regular constructors). The original object is available with the **`self`** and **`this`** keywords, the new cloned object is available with the **`self`**`[`**`cloned`**`]` construct. The **`self`**`[`**`cloned`**`]` construct is only legal in a body of the clone constructor. Clone constructors are invoked in the context of the original object, and **`self`**`[`**`cloned`**`]` is the only allowed prefix to instance values and variables, allowing to define new instance values or variables in the clone.

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. The `Object` class has a default root clone constructor, which carries the modifier **`protected`** (§5.2). Therefore, any custom clone constructor has to be explicitly protected in a less restrictive way, or public, in order to allow direct object cloning from outside of the class that defines it.

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 5.3.7 An example of a default clone constructor of class *C*, performing a deep copy.

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

5.3.3 Case Classes

Syntax:

```
Tmpl_Def ::= 'case' 'class' Class_Def
```

If a class definition is prefixed with **case**, the class is said to be a *case class*.

The formal parameters are handled differently from regular classes. The formal parameters in the first parameter section of a case class are called *elements* and are treated specially:

> First, the value of such a parameter can be extracted as a field of a constructor pattern.
>
> Second, a **val** prefix is implicitly added to such a parameter, unless the parameter already carries a **val** or **var** modifier. Hence, an accessor definition for the parameter is generated (§5.3).

A case class definition of $c[tps](ps_1)\ldots(ps_n)$ with type parameters *tps*, and value parameters *ps* implicitly adds some methods to the corresponding class object, making it suitable as an extractor object (§8.1.8), and are defined as follows:

```
object c {
  def apply[tps](ps_1)...(ps_n): c[tps] :=
```

```
      c[Ts].new(xs1)...(xsn)
  def unapply[tps](x: c[tps]): Option[e] :=
      unless x = nil
        Some(x.xs11, ..., x.xs1k)
      else
        None
      end
}
```

Here, *Ts* stands for the vector of types defined in the type parameter section *tps*, each xs_i denotes the parameter names of the parameter section ps_i, $xs_{11}, \ldots, xs_{1k}$ denote the names of all parameters in the first parameter section xs_1, and e denotes the type that `Option` is parameterized with. If a type parameter section is missing in the class c, it is also missing in the `apply` and `unapply` methods. The definition of `apply` is missing also if the class c is marked **`abstract`**.[17]

If the case class definition contains an empty value parameter list, the `unapply` returns a `Boolean` instead of an `Option` type and is defined as follows:

```
def unapply[tps](x: c[tps]): Boolean := x != nil
```

For each case of a value parameter list of n parameters, the `unapply` returns these types:

- For $n = 0$, `Boolean` is the return type.
- For $n = 1$, `Option[`Tp_1`]` is the return type, where Tp_1 is the type of the only parameter in the first parameter section.
- For $n \geq 1$, `Option[(`$Tp_1, \ldots, Tp_n$`)]` is the return type, where Tp_i is the type of the i^{th} parameter of the first parameter section. Notice that the `Option` is parameterized with a tuple type (§3.3.5). This allows for the `apply` method of `Some` to accept a tuple of arguments while accepting a single argument only.

A method named `copy` is implicitly added to every case class, unless the case class already has a matching one, or the class has a repeated parameter. The method is defined as follows:

```
def copy [tps](ps'1)...(ps'n): c[tps] :=
    c[Ts].new(xs1)...(xsn)
```

Ts stands for the vector of types defined in the type parameter section *tps*. Each xs_i denotes the parameter names of the parameter section ps'_i. Each value parameter of the first parameter list ps'_1 has the form `:`$x_{1,j}$ `:` $T_{1,j}$ `:= self.`$x_{1,j}$, and the other

[17] Because the implied `apply` method creates new objects and abstract classes can't have any instances of themselves.

parameters $ps'_{i,j}$ of the clone constructor are defined as `:`$x_{1,j}$ `:` $T_{1,j}$. Note that these parameters are defined as named parameters (§4.7.10), and that the parameters of the first value parameter section have default values using **`self`**, which is available, since the copied object is already "complete".

A clone constructor is also implicitly added to every case class, but limited to the first value parameter section. The following definition of the implicitly added clone constructor shares the definition of parameters:

```
clone [tps](ps'_1): c[tps]
  self[cloned].@xs_1,1 := xs_1,1
  ...
  self[cloned].@xs_1,n := xs_1,n
end clone
```

Here, $xs_{1,n}$ denotes the (named) parameter names of the first (and only) parameter section.

Every case class implicitly overrides some method definitions of class `Object`, unless a definition of the same method is already given in the case class itself or a concrete definition of the same method is given in some base class (including traits) of the case class, different from `Object`. Namely:

> Method `equals: (Object) ↦ Boolean` (equivalent to operator "=") is structural equality, where two instances are equal if they both belong to the case class in question and they have equal (with respect to `equals`) elements.
>
> Method `hash_code: () ↦ Number.Integer_Unsigned` computes a hash code.
>
> Method `to_string: () ↦ String` returns a string representation which contains the name of the case class and its elements.

5.3.4 Traits

Syntax:

```
Tmpl_Def       ::= 'trait' Trait_Def
Trait_Def      ::= id [Type_Param_Clause] [UoM_Params]
                   Trait_Tmpl_Env
Trait_Tmpl_Env ::= 'extends' [Early_Defs] Trait_Parents
                   semi [Template_Body] 'end' ['trait']
                 | ['extends' [Early_Defs]]
                   '{' [Template_Body] '}'
                 | ['begin' [Template_Body]] 'end' ['trait']
```

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 $\mathscr{L}(C)$ that succeed D. The actual super type gives the context for resolving a **super** reference in a trait (§6.4). 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 5.3.8 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
}
```

5.3.5 Refinements

There are two separate branches of refinements in Coral, both are quite similar, but used for different purposes. Refinements are kind of a trait (§5.3.4). The branches are as follows:

> First, refinements as part of the type system. Those refinements present only declarations and further type restrictions as part of compound types (§3.3.7).
>
> Second, refinements that are basically traits designed to be locally prepended to classes. The second branch is described here.

Syntax:

```
Tmpl_Def            ::= 'refinement' Refinement_Def
Refinement_Def      ::= id 'refine' id
                        Refinement_Tmpl_Env
Refinement_Tmpl_Env ::= '{' [Template_Body] '}'
                      | (semi | 'begin') [Template_Body]
                        'end' ['refinement']
```

Such a refinement does not declare any type parameters, since those are already declared on the type that it refines. Therefore, the type and units of measure parameters are made visible to the refinement in order to allow it to override the class methods type-correctly.

Refinements need to be "activated" in the scope for it to take effect (§6.5).

If a refinement refines a parameterized type, then the refinement only activates for this parameterized type (§3.3.4).

A refinement is not allowed to include or prepend any new traits to the type that it refines. It is an error if it attempts to do so.

A refinement is though allowed to refine classes that are **`final`** or **`sealed`**.

5.3.6 Protocols

Syntax:

```
Tmpl_Def     ::= 'protocol' Pro_Def
Pro_Def      ::= id [Type_Param_Clause]
                 Pro_Tmpl_Env
Pro_Tmpl_Env ::= 'extends' Trait_Parents
                 semi [Template_Body] 'end' ['trait']
               | ['extends'] '{' [Template_Body] '}'
               | ['begin' [Template_Body]] 'end' ['trait']
```

Protocols express the contracts that other classes have to implement, and are added to classes with the keyword "**`implements`**".

Protocols are basically traits (§5.3.4) that are stripped of some features, namely:

- Only declarations are allowed as the template body. If a method definition is needed, use a trait instead.
- Protocols can't declare anything for the class or metaclass of the class that implements them. If this is needed, use a trait instead.
- Protocols don't have early definitions (§5.1.10). If this is needed, again, use a trait instead.

Protocol references are preferred to traits to be used by library designers to declare contracts for the code that uses them.

5.3.7 Interfaces

Syntax:

```
Tmpl_Def      ::= ['case'] 'interface' [Ifc_Qualifier] Ifc_Def
Ifc_Qualifier ::= '[' Ifc_Kind ']'
Ifc_Kind      ::= 'class' | 'trait' | 'object'
Ifc_Def       ::= id [Type_Param_Clause]
                  [Class_Param_Clauses] Ifc_Tmpl_Env
Ifc_Tmpl_Env  ::= 'extends' Class_Parents
                  semi [Template_Body] 'end' ['interface']
                | ['extends'] '{' [Template_Body '}'
                | ['begin' [Template_Body]] 'end' ['interface']
```

Interfaces are filtered versions of classes and traits with only declarations and without early definitions or any definitions at all, except for auxiliary constructor definitions that become empty constructor declarations (without function bodies). 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 are.

5.4 Object Definitions

Syntax:

```
Object_Def ::= id (Class_Tmpl_Env | Trait_Tmpl_Env)
```

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 and the object definition is called a *class object*. If the class definition is not connected to a class, then rules from compound types apply (§3.3.7), the object definition is simply called a *singleton object definition* and is a single object of a new (anonymous) class. This is unlike in Scala, where objects are defined as terms, in a scope separate from types: Coral has one scope for both types and terms.

It's most general form is **`object`** m **`extends`** t. Here, m is the name of the object to be defined (or of the class object, which is the same as the related class), and t is a template (§5.1) of one of the following forms:

Singleton object form. This is a form of objects that are not class objects (not instances of `Class[`C`]`).

sc **`with`** mt_1 **`with`** ... **`with`** mt_n `{` *stats* `}`

Class object form. This is a form of objects that are class objects, therefore instances of `Class[`C`]`.

mt_1 **with** ... **with** mt_n { *stats* }

Every trait applied to a class object does not affect the associated class itself – here, the class object is the object that inherits features of the included/prepended traits, not the class instances; and the trait is included/prepended in the metaclass of the class object. This also has an implication on self types of traits (§5.3.4) that are to be included in a class object of class C: the self type would have to be `Class[`C`]`, not just C.

5.4.1 Case Objects

Syntax:

```
Tmpl_Def ::= 'case' 'object' Object_Def
```

Case objects are pretty much similar to case classes (§5.3.3). Case objects can not be class objects at the same time. Their template (§5.1) is the singleton object form of an object definition (§5.4).

5.5 Aspects

Syntax:

```
Tmpl_Def        ::= 'aspect' Aspect_Def
Aspect_Def      ::= id Aspect_Tmpl_Env
Aspect_Tmpl_Env ::= '{' Aspect_Body '}'
                  | 'begin' Aspect_Body 'end' ['aspect']
Aspect_Body     ::= Template_Stat
                  | Aspect_Stat
Aspect_Stat     ::= Advice_Stat
                  | Pointcut_Stat
```

5.6 Enums

Syntax:

```
Const_Type_Def ::= id ['extends' Class_Parents] 'is' ['bitfield']
                   'enum' '(' Enum_Field {semi Enum_Field} ')'
Enum_Field     ::= 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.

5.7 Units of Measure

Syntax:

```
Const_Type_Def ::= Unit_Name 'is' ['abstract'] 'unit-of-measure'
                     [semi Unit_Convs {semi Unit_Convs}]
Unit_Name      ::= id ['extends' 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 | id |
Unit_Op        ::= '*' | '/' | '**'
```

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 5.7.1 The following defines the measure `cm` (centimeter).

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

Example 5.7.2 The following defines the measure `ml` (milliliter) as a cubic centimeter (`cm ** 3`).

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

Example 5.7.3 The following shows possible usage of abstract units of measure.

```
type distance is abstract unit-of-measure
end type

type m extends distance is unit-of-measure
  m := km / 1000
end type

type km extends distance is unit-of-measure
  km := m * 1000
end type
```

```
type mi extends distance is unit-of-measure
  mi := km * 0.621
  mi := (m * 1000) * 0.621 // this can be inferred!
end type
```

This enables types aggregated with units of measure require a number tagged with any distance unit of measure and still work with correct units.

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 of each type, every type may be parameterized with a units of measure aggregation. These parameters are each put into angle brackets, and the only available bound is an upper bound for abstract unit of measure type.

Syntax:

```
Variant_Type_Param ::= '<' (id | '_') ['<:' id] '>'
```

Names of unit of measure parameters must not clash with names of type parameters or other unit of measure 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.

5.8 Record Types

Syntax:

```
Const_Type_Def    ::= Record_Name 'is' ['abstract'] 'record'
                      [semi] Record_Components
Record_Name       ::= id [Type_Param_Clause] [Param_Clause]
Record_Components ::= Record_Component {semi Record_Component}
                      ['extends' Class_Parents]
Record_Component  ::= 'val' Val_Dcl
                    | 'var' Var_Dcl
                    | 'case' id semi
```

```
{'when' 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*.[18]

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.

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 (§5.6). Variant records render new subtypes of the original record type, similar to a concrete type constructor.

Example 5.8.1 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
  end case
end type
```

[18]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.

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

5.9 Nullability

Every type in Coral conforms to `Object`. In turn, `Nothing` conforms to any type, in spite of nullability of the particular 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.[19]

The implicit preference may be changed in two levels. These levels are *preferred nullability* and *explicit nullability* (§3.3.11).

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.

Example 5.9.1 Here are all four versions of the nullability annotation.

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

[19] 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?

Chapter 6

Expressions

Syntax:

```
Expr        ::= Cond_Expr
              | Loop_Expr
              | Rescue_Expr
              | Raise_Expr
              | Throw_Expr
              | Catch_Expr
              | Return_Expr
              | Assign_Expr
              | Update_Expr
              | Yield_Expr
              | Infix_Expr
              | Simple_Expr
              | Match_Expr
              | Binding
              | Annot_Expr
              | Cast_Expr
              | Use_Expr
              | Jump_Expr
              | Anon_Fun
              | Anon_Class
              | Metaclass_Access
              | Workflow_Expr
              | Quasiquote_Expr
Infix_Expr  ::= Prefix_Expr
              | Infix_Expr [op_id Infix_Expr]
Simple_Expr ::= Block_Expr
              | ['&'] Simple_Expr1
              | '&' '(' Simple_Expr1 ')'
Result_Expr ::= Anon_Params '->' Block
```

```
| ['memoize'] Expr
```

Expressions are composed of various keywords, operators and operands. Expression forms are discussed subsequently.

6.1 Expression Typing

The typing of expressions is often relative to some *expected type* (which might be undefined). When we write "expression e is expected to conform to type T", we mean:

1. The expected type of e is T.
2. The type of expression e must conform to T.

Usually, the type of the expression is defined by the last element of an execution branch, as discussed subsequently with each expression kind.

What we call "statement", in context of Coral is in fact yet another kind of an expression, and those expressions themselves always have a type and a value.

6.2 Literals

Syntax:

```
Simple_Expr1 ::= Literal
```

Typing of literals is as described in (§1.5); their evaluation is immediate, including non-scalar literals (collection literals).

6.2.1 The Nil Value

Syntax:

```
Simple_Expr1 ::= 'nil'
```

The **`nil`** value is of type `Nothing`, and is thus compatible with every type that is nullable (§5.9), either preferably or explicitly.

The **`nil`** represents a "no object", and is itself represented by an object. This object overrides methods in `Object` as follows:

- `equals(`x`)` and `=(`x`)` return **`yes`** if the argument x is also the **`nil`** object.

- `!=(`*x*`)` return **`yes`** if the argument *x* is not the **`nil`** object.
- `as_instance_of[$T]()` returns always **`nil`**.
- `hash_code()` returns `0`.

A reference to any other member of the **`nil`** object causes `Method_Not_Found_Error` or `Member_Not_Found_Error` to be raised, unless the member in fact exists.[1]

6.3 Designators

Syntax:

```
Simple_Expr1 ::= Path
               | '(' Anon_Class ')' '.' Selection
               | Simple_Expr '.' Selection
Selection    ::= ['?'] id
```

A designator refers to a named term. It can be a *simple name* or a *selection*.

A simple name *x* refers to a value as specified in (§2). If *x* is bound by a definition or a declaration in an enclosing class or object *C*, it is taken to be equivalent (at the resolution time) to the selection *C*`.self.`*x*, where *C* is taken to refer to the class or object containing *x*, even if the type name *C* is shadowed at the occurrence of *x*.

If *r* is a stable identifier (§3.2) of type *T*, the selection *r*`.`*x* refers to a member *m* of *r* that is identified in *T* by the name *x*.

For other expressions *e*, *e*`.`*x* is typed as if it was `{ val` *y* `:=` *e*`; ` *y*`.`*x* `}`, for some fresh name *y*.

The selection *e*`.?`*x* is typed as if it was

```
{ val y := e; if y != nil then y.x else nil }
```

for some fresh name *y*; also called *safe navigation* or *safe selection*.

The expected type of a designator's prefix is undefined. The type of a designator is the type *T* of the entity it refers to.

The selection *e*`.`*x* is evaluated by first evaluating the qualifier expression *e*, which yields an object *r*. The selection's result is then the member *m* of *r* that is either defined by *m* or defined by a definition overriding *m*.

[1] It is even possible to use a refinement to actually implement some methods of **`nil`** locally (preferred approach), or globally implement those methods (discouraged, causes warnings).

6.4 Self, This & Super

Syntax:

```
Simple_Expr1 ::= [Path '.'] 'self'
                 ['.' Selection]
               | [Path '.'] 'this'
                 ['.' Selection]
               | [Path '.'] 'super'
                 [Class_Qualifier]
                 ['.' Selection]
               | [Path '.'] 'self' '[' 'cloned' ']'
                 '.' (ivar_id | Selection)
               | [Path '.'] (ivar_id | cvar_id)
```

The expression **self** stands always for the current instance in the context (and in function resolution searches in the actual class of the instance) in the innermost template containing the reference (thus excluding blocks and anonymous functions).

The expression **this** is the same as **self**, except that function resolution searches from the class that this expression appears in, possibly skipping overrides in subtypes of the actual class of **self**. The **this** expression is interchangeable with **self** in the following paragraphs, although use of **self** is preferred.

The expression C**.self** refers to the current instance in the context of the enclosing (or even directly enclosing) type C. It is an error if C is not an enclosing type. The type of the expression is the same as C**.self.type**.

A reference **super.**m refers to a method or type m in the least proper supertype of the innermost template containing the reference. It evaluates to the member m' in the actual supertype of that template, which is equal to m or which overrides m. If m refers to a method, then the method must be either concrete, or the template containing the reference must have a member m', which overrides m and which is labeled **abstract override**.

A reference C**.super.**m refers to a method or type m in the least proper supertype of the innermost class or object definition named C, which encloses the reference. It evaluates to the member m' in the actual supertype of that template, which is equal to m or which overrides m. If m refers to a method, then the method must be either concrete, or the template containing the reference must have a member m', which overrides m and which is labeled **abstract override**.

The **super** prefix may be followed by a qualifier [T], as in C**.super[**T**].**m. In this case, the reference is to the type or method m in the parent class or trait of C, whose simple name is T. The qualifier allows also paths, in case multiple supertypes had the same simple name, working as a suffix search – the name T then refers the parent class or trait of C, whose qualified name ends with T. It evaluates to the member m'

in the actual supertype of that template, which is equal to m or which overrides m. If m refers to a method, then the method must be either concrete, or the template containing the reference must have a member m', which overrides m and which is labeled **`abstract override`**.

6.5 Use Expressions

Syntax:

```
Use_Expr    ::= Use_Expr_As | Use_Aspect
Use_Expr_As ::= 'use' Simple_Expr ('as' | 'as!')
                [id ':'] Type [Block_Expr]
Use_Aspect  ::= 'use' 'aspect' Path [Block_Expr]
```

Use expressions of the form **`use`** e **`as`** a**`:`** T are similar to typed expressions (§6.13). Their intention is to rebind an expression to a specific type (changing its expected type), and then either have this type to be effective in the same scope from that point onward, or, if a `Block_Expr` is syntactically given, only in the scope of that block expression. If a block is given, then the return value of the block is the value of this expression, otherwise, the value retrieved by evaluation of `Simple_Expr` is the value of this expression. Conversions described in typed expressions (§6.13) apply in these expressions as well, including the differences between **`as`** and **`as!`**.

Use expressions of the form **`use aspect`** T enable the specified aspect, either in the scope defined by the given block, or if no block is given, then from that point onward. If the expression is used as a template statement, then the aspect is enabled for the whole template anywhere, if it does not have the block part.

6.6 Function Applications

Syntax:

```
Simple_Expr1   ::= Simple_Expr1 Argument_Exprs
Argument_Exprs ::= Parens_Args {Parens_Args} [Block_Expr]
                 | Poetry_Args [Block_Expr2]
                 | Block_Expr

Parens_Args ::= '(' Args_Expr ')'
Poetry_Args ::= Args_Expr
Args_Expr   ::= [[Arg_Exprs ','] '*' Expr ','] Arg_Exprs [',' '**' Expr]
              | [Arg_Exprs ','] '*' Expr [',' '**' Expr]
              | '**' Expr
Arg_Exprs   ::= Arg_Expr {',' Arg_Expr}
```

```
Arg_Expr    ::= [['out'] [id ':'] | '&'] Expr
```

A function application $f(e_1, \ldots, e_m)\ \ b$ applies the function f to the argument expressions $e_1, \ldots, e_n$ and passes the block expression b (§6.10) into it. If f has a method type $(p_1\texttt{:}\ T_1, \ldots, p_n\texttt{:}\ T_n) \mapsto U$, the type of each argument expression e_i is typed with the corresponding parameter type T_i (§6.6.2) as expected type. Let S_i be type of argument e_i (for $i = 1, \ldots, m$). If f is a polymorphic method, local type inference (§6.26.4) is used to determine type arguments for f. If f is of a value type, the application is taken to be equivalent to $f\texttt{.apply}(e_1, \ldots, e_m)$, i.e. the application of an `apply` method defined by f.

6.6.1 Argument Evaluation Strategies

Coral defers evaluation of arguments up to the point of function application, and happens then as specified in parameter evaluation strategies (§4.7.1). The type that each argument is type-checked against the corresponding parameter type (defined as follows) is the expected type of the argument expression, i.e. not its actual concrete type, which is known only after its evaluation. If the expected type is undefined, then `Object` is assumed. Typed expressions (§6.13) may be used to give the argument expression a concrete expected type. When the argument expression is evaluated, it is evaluated as if it were in the scope of the function application (which it is), so that visibility rules from that scope apply.

6.6.2 Corresponding Parameters

The argument expressions $a_1, \ldots, a_n$ can be split up to 3 virtual sections:

> *Positional parameters*. Let's refer to them as $p_{1,i,j}$. These are defined by any number of mandatory parameters (where $i = 1$), followed by any number of optional parameters (where $i = 2$), followed by at most one repeated parameter (where $i = 3$ and $j = 1$), ended by any number of post mandatory parameters (where $i = 4$).
>
> *Named parameters*. Let's refer to them as $p_{2,i}$, where i is the position of the named parameter among the section of named parameters.
>
> *Block capturing parameter*. Let's refer to it as $p_{3,1}$

To pair argument expressions with corresponding parameters, the following steps are to be taken:

1. Say that a *positional argument* is of the form a_i.

2. Say that a *named argument* is of the form x_i`:` a'_i, where x_i is one of the named parameter names from the named parameters section.

3. Say that n_1 is the count of positional arguments. If the last argument is prefixed with "&":

   - If the captured block parameter is defined and a block b is given, count the last argument also as a positional argument. It is an error if there are named arguments before it.
   - If the captured block parameter is defined and a block b is not given, do not count the last argument as a positional argument.
   - If the captured block parameter is not defined and a block b is given, count the last argument also as a positional argument. It is an error if there are named arguments before it.
   - If the captured block parameter is not defined and a block b is not given, count the last argument also as a positional argument. It is an error if there are named arguments before it.

4. Say that m_1 is the count of mandatory parameters. Pair each a_i for $1 \leq i \leq m_1$ with $p_{1,1,i}$.

5. Say that m_2 is the count of post mandatory parameters. Pair each a_i for $(n_1 - m_2) \leq i \leq m_2$ with $p_{1,4,i}$.

6. Say that m_3 is the count of optional parameters. Pair each a_i for $m_1 < i < m_2$ with $p_{1,2,i}$, if $p_{1,2,i}$ is an optional parameter. If $p_{1,2,i}$ is not an optional parameter, collect the arguments that don't have a corresponding optional parameter into a sequence and pair it as a single argument with $p_{1,3,1}$. This finishes the positional arguments section.

7. Let σi be a substitution from the named argument name's position in the function application's named arguments section to its position in the named parameters section of the function parameters definition. If the named argument has a name that is not defined in the function parameters definition, then the position of the capturing named parameter is returned, if it exists. It is an error if the substitution does not return any position, in the sense that the function is not applicable to the given arguments, defined also as follows.

8. Say that n_2 is the count of named arguments. Pair each $a_{\sigma i}$ for $1 \leq i \leq n_2$ with $p_{2,i}$.

9. Say that n_3 is the count of arguments given after the section of named arguments. If $n_3 = 0$, then pair the given block b with $p_{3,1}$, or pair **`nil`** with $p_{3,1}$, if no block b is given. If $n_3 = 1$, then pair the last unpaired argument with $p_{3,1}$. If $n_3 > 1$, it is an error, in the sense that the function is not applicable to the given arguments, defined also as follows.

The type of each argument expression a_i is typed with the corresponding parameter type T_i as expected type.

6.6.3 Applicable Function

The function f must be applicable to its arguments $a_1, \ldots, a_n$ of types $S_1, \ldots, S_n$.

If f has a method type $(p_1\texttt{:}\ T_1, \ldots, p_n\texttt{:}\ T_n)R$, the function f is applicable if all of the following conditions hold:

- For every named argument $x_i\texttt{:}\ a'_i$, the type S_i is compatible (§6.26) with the parameter type T_j, whose name p_j matches x_i, or if f defines a capturing named parameter and x_i does not match name of any p_j, then the type S_i is compatible with the parameter type T_j, whose name p_j matches the name of the capturing named parameter.
- For every positional argument a_i, the type S_i is compatible (§6.26) with its corresponding T_i.
- The given block or the last argument prefixed with "&" is of a type compatible (§6.26) with the type of the captured block parameter, if such parameter is defined.
- If the expected type of the function application is defined, the result type R is compatible (§6.26) to it.
- For every argument a_i, if the corresponding parameter is defined as **`out`**, the argument is prefixed with **`out`** as well and must be a local variable. If the corresponding parameter is defined as only **`out`** and not **`in`** at the same time, the argument is converted to **`out`** argument and any previous value of the argument variable is released. See (§6.6.7) for details.
- Every formal parameter $p_j\texttt{:}\ T_j$ which is not specified by either a positional or a named argument has a default value (§4.7.8 & §4.7.10).

If f is a polymorphic method, it is applicable if local type inference (§6.26.4) can determine type arguments, so that the instantiated method is applicable. If f is of a value type, it is applicable if it has a method member named `apply`, which is applicable. Note that if explicit type parameters are given to the polymorphic method, type application (§6.7) happens prior to function application.

If a function application appears to be an argument to another function application (let's call it a nested function application), the expected type of the nested function application is used to determine, whether the outer function is applicable, but the nested function application is not evaluated until time specified by argument evaluation strategy (§6.6.1) corresponding to the argument. Local type inference may indeed occur

for the nested function application, if it involves a polymorphic method, but again, only using the available expected types.

If an argument expression is prefixed with "∗" (let's call it a *sequence argument expression*, or *sequence-splat*, or just *splat*), it is expanded into multiple argument expressions, as its expected type is `Sequence[`*S*`]` and *S* is the expected (and usually actual) type of the arguments resulting from the expansion. The expansion uses methods of the `Sequence` type to determine the length of the sequence and its type, which are then inserted instead of the sequence argument (and the contents of the sequence have their evaluation deferred). The sequence should have a reasonable length, and must not be infinite. Such sequence arguments can appear in the function application multiple times (unlike the repeated parameter), but only in the section of positional arguments.

If an argument expression is prefixed with "∗∗" (let's call it a *map argument expression*, or *map-splat*), it is expanded into multiple argument expressions, as its expected type is `Map[Symbol,` *S*`]` and *S* is the expected (and usually actual) type of the arguments resulting from the expansion. The expansion uses methods of the `Map` type to determine the length of the map and its type, which are then inserted instead of the map argument (and the contents of the map have their evaluation deferred). The map should have a reasonable length, and must not be infinite. Such map arguments can appear in the function application multiple times (unlike the repeated parameter), but only in the section of named arguments.

Example 6.6.1 Assume the following function, which computes the sum of variable number of arguments:

```
def sum (*xs: Integer) := (0 /: xs) ((x, y) -> { x + y })
```

Then

```
sum 1, 2, 3, 4
sum (1, 2, 3, 4)
sum *%[1, 2, 3, 4]
sum (*%[1, 2, 3, 4])
sum 1, 2, *%[3, 4]
sum (1, 2, *%[3, 4])
sum 1, *%[2, 3], 4
sum (1, *%[2, 3], 4)
```

all yield 10 as result. On the other hand,

```
sum %[1, 2, 3, 4]
```

would not be applicable. Moreover, (note the extra space before the sequence-splat operator),

```
sum * %[1, 2, 3, 4]
```

would be interpreted as

```
sum.`*`(%[1, 2, 3, 4])
```

which is an infix expression rather than a function application. On the other hand, a space may appear between the function name and the arguments list.

6.6.4 Tail-call optimization

A function application usually allocates a new stack frame on the program's runtime stack for the current thread. However, if at least one of the following conditions holds and function calls itself as its last action, the application is executed using the stack frame of the caller, replacing arguments and rewinding stack pointer to the first instruction, called *tail-call optimization*:

- The function is local and not overloaded.
- The function is **`final`**.
- The function is **`private`** or **`private[self]`**.
- The function is annotated so that tail-call optimization is explicitly allowed.
- A pragma allowing tail-call optimizations is effective in the scope of the tail call.

The optimization will not happen if the application results in a different (possibly overloaded or overridden) variant of the caller function being applied, and a warning is issued if the tail-call optimization was explicitly expected (either via an annotation or a pragma).

6.6.5 Named & Optional Arguments

If an application uses named arguments p_i**`:`** e_i or default arguments, the following conditions must hold:

- No named argument appears left of a positional argument in the argument list.
- No positional argument appears right of a named argument. A bit of an exception is the captured block argument, which appears to be positional, but is treated specially.
- The names p_i of all named arguments are pairwise distinct. There is no way for named arguments to specify positional arguments – this is also ensured by a similar requirement for the parameters lists.

- Every formal parameter p_j`:` T_j, which is not specified by a positional argument, has a default argument.
- Every formal parameter `^`p_j`:` T_j, which is not specified by a named argument, has a default argument.
- If there are more named arguments than named parameters (excluding the capturing named parameter), a capturing named parameter is defined. (If it is not, the function is not applicable.)

No transformation is applied to convert a function application into an application without named or default arguments – the runtime handles the application itself.

6.6.6 By-Name, By-Need & By-Future Arguments

None of these argument types require any syntactically special treatment. The user of a function that uses these types should however consider the implications of their types on their evaluation.

6.6.7 Input & Output Arguments

Output arguments must be prefixed with the **`out`** keyword, so that the runtime can pass in a reference to the variable, and not to the value. The **`in`** modifier is implied and not used explicitly in the application. If the actual parameter is **`out`**-only, the original value of the variable is released upon being written to by the applied function, but not sooner.

A variable that is defined with **`val`** is not useable as an **`out`** argument. A variable that is declared with **`val`** (but not defined) is useable as an **`out`** argument.

When it comes to closures, variables in variable closures are implicitly also **`out`**, if the anonymous function contains an assignment to them.

An assignment to an output parameter that was released is a no-op, and issues a warning. This can happen when the execution of an anonymous function or a block is delayed past the point where the original stack frame of the calling function is already released.

6.6.8 Curried Functions & Partial Applications

A curried function can appear in two distinct forms:

Implicitly curried form, which is defined by using multiple parameters lists.

Explicitly curried form, which is defined by using function types as return types of functions, or simply by returning a function from within a function.

Each form has some implications on function applications.

Let's define *consecutive function applications*. Such function applications are a continuous sequence of function applications, where each following function application is directly applied to the result of the previous function application, without storing the intermediate values anyhow.

Example 6.6.2 The following is an example of consecutive function applications:

```
f(a, b)(c, d)
```

The following are not consecutive function applications:

```
val e := f(a, b)
e(c, d)
```

An implicitly curried function requires a consecutive function application for all of its parameters lists, excluding the implicit parameters list. If the implicitly curried function is intended to be *partially applied* (not providing all the parameters lists with arguments lists), then a method value (§6.23.1) can be used. This also applies to the implicit parameters list – if providing it is to be deferred, a method value that encloses arguments lists up to the implicit parameters list can be used, but then the implicit arguments list has to be provided later in order to evaluate the curried function.

On the other hand, explicitly curried functions do not care about consecutive function applications.

The consecutive function applications meta-construct is also a solution to providing explicitly the implicit parameters list an arguments list. Without it, the function application would handle the implicits from it and the consecutive application would be applied to the result of the whole function. Therefore, if a consecutive function application is present, the evaluation of implicit parameters list is deferred to this consecutive function application, so that arguments for it can be specified. If there is no consecutive function application, then the implicit parameters list is evaluated as usual.

6.6.9 Function Compositions & Pipelines

These expressions are not in fact syntax features, but rather an implementation on functions and their traits.

A *function composition* is a way to compose two functions and return a function. A *function pipeline* is a way to pass a value to a function and return a value, which can be again passed to another function in a pipeline.

Function composition is usually defined by operators such as "`|>>`" for unary functions, "`||>>`" for binary functions and so on, and "`<<|`" for unary functions, "`<<||`" for binary functions, in reverse order.

Function pipeline is usually defined by operators such as "|>" for unary functions, "||>" for binary functions and so on, and "<|" for unary functions, "<||" for binary functions, in reverse order.

These operators for unary functions can be defined as follows, e.g.:

```
trait Function_1 [-T, +R]
begin

  // right-associative
  operator |>> [T1] (g: T1 -> T): T1 -> R :=
    (a: T1) -> { self(g(a)) }

  // left-associative
  operator <<| [T1] (g: T1 -> T): T1 -> R :=
    (a: T1) -> { self(g(a)) }

  // right-associative
  operator |> (a: T): R :=
    self(a)

  // left-associative
  operator <| (a: T): R :=
    self(a)

end trait
```

Function composition and pipelining makes more sense with the use of positional parameters rather than with named parameters, although with some more verbose syntax, it can be achieved as well, e.g. by assuming that the composed functions or pipelines share the same names of their named parameters.

6.6.10 Memoization

How to memoize a function's result is described in (§6.21.1).

A memoized function's body is not evaluated, if it was once called with the same arguments (based on equality, not identity), and if that result value is still memoized. If so, the memoized result value is immediately returned without evaluation of the function's body, which can speed up execution of some functions significantly. Such functions should however be referentially transparent in best-case scenario (§4.7 & §6.25) or at least tolerant to being memoized.

Memoization is better with small parameter numbers, so that searching the result values cache would not actually take longer than evaluation of the function's body. Functions that are defined with the **`function`** keyword (§4.7) may opt-in to implicit mem-

oization[2], as well as functions declared as **`transparent`** (§6.25). Functions declared as **`opaque`** (§6.25) should not be memoized.

Parameters of memoization[3] may be controlled, even on per-function basis, with use of specialized annotations and pragmas.

6.7 Type Applications

Syntax:

```
Simple_Expr1 ::= Simple_Expr Type_Args
```

A type application $e[T_1, \ldots, T_n]$ instantiates a polymorphic value e of type $[a_1 >: L_1 <: U_1 , \ldots, a_n >: L_n <: U_n] \mapsto S$ with argument types $T_1, \ldots, T_n$. Every argument type T_i must obey the corresponding bounds L_i and U_i. That is, for each $i = 1, \ldots, n$, we must have $\sigma L_i <: T_i <: \sigma U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$. The type of the application is σS.

If the function part e is of some value type, the type application is taken to be equivalent to e`.apply[`$T_1, \ldots, T_n$`]`, i.e. the application of an `apply` method defined by e.

Type applications can be omitted if local type inference (§6.26.4) can infer best type arguments for a polymorphic function from the types of the actual function arguments and the expected result type.

6.8 Tuples

Syntax:

```
Simple_Expr ::= '(' [Exprs] ')'
```

A tuple expression $(e_1, \ldots, e_n)$ is an alias for the class instance creation `Tuple_`n`(`$e_1, \ldots, e_n$`)`, where $n \geq 2$. The empty tuple `()` is the unique value of type `Unit`. A tuple with only one value is only the value itself, without being wrapped in a tuple.

6.9 Instance Creation Expressions

Unlike languages like Java, Scala, C# and similar, Coral does not have dedicated language construct for creating new instances of classes. Instead, all such attempts are

[2]E.g., based on the computed complexity of the function. If the function is decided to be simple, then memoization could actually worsen performance.

[3]Parameters include things like: cache policy, ttl, cache size and so on.

made through the `Class#new` method (not to be confused with `Class.new`), which in the end[4] has all arguments for a constructor, which gets invoked by a native implementation.

6.10 Blocks

Syntax:

```
Block_Expr      ::= Block_Expr1 | Block_Expr2
Block_Expr1     ::= '{' [Block_Args semi] Block '}'
Block_Expr2     ::= 'do' [Block_Args semi] Block 'end'
Block_Args      ::= '|' [Params] [Block_Shadowing] '|' [':' Type]
Block_Shadowing ::= ';' [Shad_Val_Dcl {',' Shad_Val_Dcl}]
Shad_Val_Dcl    ::= 'val' Val_Dcl
                  | 'var' Var_Dcl
                  | 'def' Def_Dcl
Block           ::= {Block_Stat semi} [Result_Expr]
```

A block expression `{` s_1`;` $\ldots$`;` s_n`;` e `}` is constructed from a sequence of block statements $s_1, \ldots, s_n$ and a final expression e. The statement sequence may not contain two definitions or declarations that bind the same name in the same namespace, except for local function definitions, which then create overloaded local function definitions (behaving pretty much like regular overloaded functions). The final expression may be omitted, in which case the unit value `()` is assumed.

The expected type of the final expression e is the expected type of the block expression. The expected type of all preceding statements is undefined.

The type of a block `{` s_1`;` $\ldots$`;` s_n`;` e `}` is T **`for-some`** `{` Q `}`, where T is the type of e and Q contains existential clauses (§3.3.10) for every value or type name which is free in T and which is defined locally in any of the statements $s_1, \ldots, s_n$. We say that the existential clause *binds* the occurence of the value or type name. Specifically,

- A locally defined type definition **`type`** t `:=` T is bound by the existential clause **`type`** t `>:` T `<:` T. It is an error if t carries type parameters.
- A locally defined value definition **`val`** x`:` T `:=` e is bound by the existential clause **`val`** x`:` T.
- A locally defined class definition **`class`** c **`extends`** t is bound by the existential clause **`type`** c `<:` T, where T is the least class type of refinement type which is a proper supertype of the type c. It is an error if c carries type parameters.

[4]Because constructor currying can happen, the constructor is invoked by a native implementation from outside of the new method, by the curried function, which is a mechanism inaccessible to users otherwise than via the constructor definition.

- A locally defined object definition **`object`** *x* **`extends`** *t* is bound by the existential clause **`val`** *x*`:` *T*, where *T* is the least class type of refinement type which is a proper supertype of the type *x*`.`**`type`**.

Evaluation of the block entails evaluation of its statement sequence, followed by an evaluation of the final expression *e*, which defines the implicit result of the block.

6.10.1 Block Expression as Argument

A block expression may be used as the very last argument in a function application (§6.6), being equivalent to an anonymous function.

As such, the block parameters section is optional to be defined, and its parameters are, unlike with anonymous functions, tolerant to different shapes of given arguments. If parameters of the block are typed and arguments for the corresponding parameters are given, the types of the arguments must be compatible with the expected parameter types.

If less arguments are provided, the remaining parameters have default values of their types, and it is an error if such parameter is typed with a non-nullable type.

If more arguments are provided, the extra arguments are discarded and released.

An explicit return expression (§6.21.1) within the block is interconnected with the innermost function that defines the block, i.e. evaluating it returns from the function (as well as from the block expression). This does not apply to anonymous functions (§6.23), where the return expression is interconnected with the anonymous function itself.

6.10.2 Variable Closure

Coral uses variable closure when defining a block expression.

If a block expression is used just as a statement, it implicitly inherits access to all variables and methods defined in the scope in which it itself is defined.

If a block expression is used as an argument in a function application (§6.6), it is used as a functor, and is provided with read and write access to variables in the scope that the function application appears in, and with an access to the **`self`** reference, including any nested *C*`.`**`self`** references (§6.4). Write access to variables is provided in a manner equivalent to **`out`** arguments.

Variable closure is applied to anonymous functions (§6.23) as well.

A block expression may opt-in to *shadow variables* that it would otherwise have access to from its scope, specified with the `Block_Shadowing` syntax element. Variables and methods with the same names as those of the shadowing variables will not be a part of the variable closure.

6.11 Yield Expressions

Syntax:

```
Yield_Expr ::= 'yield' (Parens_Args | Poetry_Args | ())
```

A yield expression **yield** $a_1, \ldots, a_n$ is a way to invoke the block argument (§6.6 & §6.10.1) and pass it arguments. It's expected type is the result type of the given block argument.

The **yield** keyword is also used in collection comprehensions (§6.19) and generators (§6.18), with a different meaning.

6.12 Prefix & Infix Operations

Syntax:

```
Infix_Expr  ::= Prefix_Expr
              | Infix_Expr [op_id Infix_expr]
Prefix_Expr ::= [op_id] Simple_Expr
```

Expressions can be constructed from operands and operators.

6.12.1 Prefix Operations

A prefix operation *op e* consists of a prefix operator *op*, which may be any operator identifier, unlike in Scala, but must not be followed by any whitespace, only identifiers or parentheses. The expression *op e* is equivalent to the method application `e.op()`.

6.12.2 Postfix Operations

Apart from standard function applications (§6.6) that may be viewed as postfix, Coral does not include support for postfix operations.

6.12.3 Infix Operations

An infix operator can be an arbitrary operator identifier. Infix operators have static precedence and associativity borrowed from Scala, and defined as follows:

The *precedence* of an infix operator is determined by the operator's first character. Characters are listed below in increasing order of precedence, with characters on the same line having the same precedence.

```
|
^
&
< > ~
= !
:
(all other special characters)
+ -
* / %
```

That is, operators starting with "|" have the lowest precedence, followed by operators starting with "^", etc.

There's one exception to this rule, which concerns *assignment operators* (§6.12.4). The precedence of an assignment operator is the same as the one of simple assignment (=). That is, it is lower than the precedence of any other operator.

The *associativity* of an operator is determined by the operator's last character. Operators ending in a colon "`:`" are right-associative, and operators ending in a greater-than sign ">" are right-associative, if they consist of more than one operator character. All other operators are left-associative.

Precedence and associativity of operators determine the grouping of parts of an expression as follows.

- If there are several infix operations in an expression, then operators with higher precedence bind more closely than operators with lower precedence.
- If there are consecutive infix operations $e_0\ op_1\ e_1\ op_2 \ldots op_n\ e_n$ with operators $op_1 \ldots op_n$ of the same precedence, then all those operators must have the same associativity (i.e. it is an error if they don't). If all operators are left-associative, then the sequence is interpreted as $((e_0\ op_1\ e_1)\ op_2 \ldots)\ op_n\ e_n$. Otherwise, if all operators are right-associative, the sequence is interpreted as $e_0\ op_1\ (e_1\ op_2\ (\ldots op_n\ e_n))$.
- A left-associative binary operation $e_1\ op\ e_2$ is interpreted as e_1`.\`op\`(`e_2`)`. If op is right-associative, the same operation is interpreted as `{ val x :=` e_1`;` e_2`.\`op\`(`x`) }`, where x is a fresh name.
- The right-hand operand of a left-associative operator may consist of several arguments enclosed in parentheses, e.g. $e\ op\ (e_1, \ldots, e_n)$. This expression is then interpreted as e`.\`op\`(`$e_1, \ldots, e_n$`)`.
- The left-hand operand of a right-associative operator may consist of several arguments enclosed in parentheses, e.g. $(e_1, \ldots, e_n)\ op\ e$. This expression is then interpreted as e`.\`op\`(`$e_1, \ldots, e_n$`)`.

6.12.4 Assignment Operations

An assignment operator is an operator symbol that ends in an "equals" character "=", with the exception of operators for which one of the following conditions holds:

1. the operator also starts with an equals character and has more than one character, or
2. the operator is one of "<=", ">=" or "!=".

Assignment operators are treated specially in that they can be expanded to assingments if no other interpretation is valid, as previously defined. Assignment operators can't be defined as members of any type.

Let's consider an assignment operator, such as "+=", in an infix operation l += r, where l & r are expressions. This operation can be re-interpreted as an assignment

l := l + r

except that the operations's left-hand-side l is evaluated only once.

The re-interpretation is correct always, since, unlike in Scala,

1. The left hand side can never have a member named "+=", as specified in this section.
2. The assignment l := l + r is only expected to be type-correct, not enforced. If it is not type correct, a runtime error will be raised.

6.13 Typed Expressions

Syntax:

```
Cast_Expr  ::= Infix_Expr ('as' | 'as!') Type
Infix_Expr ::= Infix_Expr ('is' | 'is!') Type
```

The typed expression e **as** T has type T. The type of expression e is expected to conform to T. The result of the expression is the value of e converted to type T. The conversion can take these forms, preferred in the following order:

1. No conversion, if e conforms to T directly.
2. If an implicit conversion c from expression type E of method type $(E) \mapsto T$ exists in the scope, then the conversion is of the form $c(e)$.

3. Otherwise, the conversion is of the form `e.as_instance_of[T]()`.

The conformance check expression `e is T` has type `Boolean` and tests whether e conforms to T, basically by asking a question "Can e be of type T?", answering either "It can be" or "It can't be". The expression e conforms to type T if at least one of the following conditions hold:

1. Type of e is a subtype of T.
2. An implicit conversion c from expression type E of method type $(E) \mapsto T$ exists in the scope.
3. As a last resort, type of e overrides the method `is_instance_of[T]()` and evaluating it results in **`yes`** value.

Note: the conformance check expression as defined here is not used in resolution of function applications (§6.6).

The typed expression `e as! T` works like `e as T`, but only uses the first form. Similarly, the conformance check expression `e is! T` works like `e is T`, but it uses only the first condition. The bang character "`!`" signalizes that the operation is more dangerous, in means of that its easier for the expression e to not successfully convert to the target type or conform to it.

6.14 Annotated Expressions

Syntax:

```
Annot_Expr ::= Annotation {Annotation} Infix_Expr
```

An annotated expression $a_1 \ldots a_n\ e$ attaches annotations $a_1 \ldots a_n$ to the expression e (§10).

6.15 Assignments

Syntax:

```
Assign_Expr ::= [Simple_Expr '.'] id ':=' Expr
              | Mul_Assign_Expr
Update_Expr ::= Simple_Expr1 Argument_Exprs ':=' Expr
```

The interpretation of an assignment to a simple variable x `:=` e depends on the definitions of x. If x denotes a mutable variable, then the assignment changes the current value of x to the result of evaluating the expression e. The type of e is expected to conform to the type of x.

If x is defined as a property of some template, or the template contains a setter function x`_=` as a member, then the assignment is interpreted as the invocation x`_=(`e`)` of that setter function.

Analogously, an assignment f`.`x `:=` e is interpreted as the invocation f`.`x`_=(`e`)`. If f is evaluated to **`nil`**, then the invocation is forwarded to **`nil`**.[5]

An assignment f`.?`x `:=` e is interpreted as the invocation f`.?`x`_=(`e`)`. If f is evaluated to **`nil`**, then the invocation is evaluated to **`nil`**. See (§6.3) for more on behavior of the "`.?`" navigation.

An assignment f`(`*args*`)` `:=` e with a function application to the left of the "`:=`" operator is interpreted as f`.update(`*args*`)(`e`)`, i.e. the invocation of an `update` function defined by f. If f is evaluated to **`nil`**, then the invocation is forwarded to **`nil`**. The "`.?`" navigation is not available with this expression.

Example 6.15.1 Here are some assignment expressions and their equivalent interpretations.

```
f := e                      f_=(e)
f() := e                    f.update()(e)
f(i) := e                   f.update(i)(e)
f(i, j) := e                f.update(i, j)(e)
x.f := e                    x.f_=(e)
x.f() := e                  x.f.update()(e)
x.f(i) := e                 x.f.update(i)(e)
x.f(i, j) := e              x.f.update(i, j)(e)
f()() := e                  f().update()(e)
f(i)() := e                 f(i).update()(e)
f()(i) := e                 f().update(i)(e)
f(i)(j) := e                f(i).update(j)(e)
f(i, j)(k) := e             f(i, j).update(k)(e)
f(i, j)(k, l) := e          f(i, j).update(k, l)(e)
f(i, j) := (e_1, e_2)       f.update(i, j)((e_1, e_2))
```

6.15.1 Multiple Assignments

Syntax:

[5]This likely results in a runtime error being raised, unless **`nil`** would actually implement method `x_=`.

```
Mul_Assign_Expr ::= Mul_Vars ':=' Mul_Exprs
Mul_Vars        ::= [(id | '_') {',' (id | '_')} ',']
                    ['*'] (id | '_') {',' (id | '_')}
Mul_Exprs       ::= [Expr {',' Expr} ',']
                    ['*'] Expr {',' Expr}
```

Multiple assignment is a way to assign multiple variables at once. On the left-hand side of the assignment are variables separated by commas, where at most one of which may be prefixed with an asterisk "∗". On the right-hand side of the assignment are expressions separated by commas, where one or more expressions may be prefixed with an asterisk "∗" as a sequence-splat operator.

The left-hand side of the multiple assignment must contain only variable names that can be assigned to – so either mutable variables, or declared and not defined variables.

The right-hand side is expanded into a single sequence of expressions in the following way:

1. Say that $e_1, \ldots, e_n$ are the original right-hand side expressions.
2. For each e_i, where $1 \leq i \leq n$, if e_i is prefixed with a sequence-splat operator, replace e_i with a comma-separated expressions $e_{i,j}$, where j is the index of the sub-expression contained in the original e_i, and move to the next expression e_{i+1}.

To match the left-hand side variable names with the expanded right-hand side expressions, match first the variables until the one prefixed with an asterisk, if any, and remove the matched expressions. Count variables that are following the one prefixed with an asterisk as m and match them with the remaining expressions, starting from expression $n - m$ where n is the count of the remaining expressions, or from the first expression, if $m \geq n$. If $m \leq n$, then collect the remaining expressions into a sequence and assign it to the variable prefixed with an asterisk. In any case, if there are less expressions available than variables to assign to, assign the extra variables with **`nil`**. If there are more expressions than variables to assign to and no variable is prefixed with an asterisk, then the extra expressions are discarded and released.

The multiple assignment evaluates all expressions on the right-hand side of the assignment prior to the actual assignment. The expected type of each assigned expression is the expected type of the corresponding variable it assigns to, or if the corresponding variable is prefixed with an asterisk, then the expected type is the expected type of the elements of the sequence declared by that variable.

If the name of the assigned variable is "_", the assignment for that variable is not evaluated and is discarded.

Example 6.15.2 The following examples show how multiple assignment works.

```
// swap two variables
a, b := b, a

// `a` will be `e`
// `b` will be the first element of `f` (if `f` contains anything)
// `d` will be the last element of `f` (if `f` contains anything)
// `c` will be the remaining elements of `f` (if `f` contains anything)
// if `f` is an empty sequence, then `b` and `d` are assigned nil
//   and `c` is an empty sequence
// if `f` has one element, then `d` is assigned nil
//   and `c` is assigned an empty sequence
// if `f` has two elements, then `c` is assigned an empty sequence
a, b, *c, d := e, *f
```

6.16 Conditional Expressions

Syntax:

```
Cond_Expr        ::= Cond_Block_Expr | Cond_Mod_Expr
Cond_Block_Expr  ::= Cond_Block_Expr1 | Cond_Block_Expr2
Cond_Block_Expr1 ::= 'if' Expr ('then' | semi) Cond_Block
                     {[semi] 'elsif' Expr ('then' | semi) Cond_Block}
                     [[semi] 'else' Cond_Block] 'end' ['if']
Cond_Block_Expr2 ::= 'unless' Expr ('then' | semi) Cond_Block
                     {[semi] 'elsif' Expr ('then' | semi) Cond_Block}
                     [[semi] 'else' Cond_Block] 'end' ['unless']
Cond_Mod_Expr    ::= Expr Cond_Modifier
Cond_Modifier    ::= Cond_Modifier1
                     ['else' Infix_Expr]
Cond_Modifier1   ::= ('if' | 'unless') Infix_Expr
Cond_Block       ::= Expr | Block
```

The conditional expression **`if`** e_1 **`then`** e_2 **`else`** e_3 chooses one of the values of e_2 and e_3, depending on the value of e_1. The condition e_1 is expected to conform to type `Boolean`, but can be virtually any type – if it is not a `Boolean`, then it is equal to **`yes`** if it implements the method `to_boolean(): Boolean` and that implementation returns **`yes`**, or can be converted to **`yes`** (§6.13), and **`no`** otherwise. The **`nil`** object converts always to **`no`**. If the e_1 is the single instance "`()`" of type `Unit`, it is an error. The **`then`**-part e_2 and the **`else`**-part e_3 are both expected to conform to the expected type of the conditional expression, but are not required to. The type of the conditional expression is the weak least upper bound (§3.7.3) of the types of e_2 and e_3. A semicolon preceding the **`else`** symbol of a conditional expression is ignored.

The conditional expression is evaluated by evaluating first e_1. If this evaluates to `true`,

the result of evaluating e_2 is returned, otherwise the result of evaluating e_3 is returned.

The evaluation of e_1 utilizes the so-called *short-circuit evaluation*. The expression e_1 is split by binary boolean operators. Then every first argument is evaluated, but the second argument is evaluated only if the evaluation of the first argument does not suffice to determine the value of the expression. When the first argument of "&&" evaluates to **no**, the overall value must be **no** and the result of evaluating the second argument does not change that. When the first argument of "||" evaluates to **yes**, the overall value must be **yes**. These boolean operators are in fact short-circuited source-code-wide, not only as part of conditional expressions. Word equivalents of these operators are not short-circuited ("`and`" and "`or`" respectively).

Example 6.16.1 The following examples show how short-circuit evaluation behaves. Let's mark the short-circuited "&&" as "`sand`" and the short-circuited "||" as "`sor`". On the right side are the equivalent conditional expressions.

x `sand` y	**if** x **then** y **else no**
x `sor` y	**if** x **then yes else** y

A short form of the conditional expression eliminates the **else**-part. The conditional expression **if** e_1 **then** e_2 is evaluated as if it was **if** e_1 **then** e_2 **else** `()`, and is therefore expected to be the weak least upper bound of the type `Unit` and the type of e_2.

The conditional expression

if e_1 **then** e_2 **elsif** e_3 **then** e_4 ... **elsif** e_n **then** e_{n+1} **else** e_{n+2}

is evaluated as if it was

if e_1 **then** e_2 **else if** e_3 **then** e_4 ... **else if** e_n **then** e_{n+1} **else** e_{n+2}

Basically, **elsif** is a simple syntax sugar for the little longer **else if** keyword tokens sequence.

The alternative conditional expression **unless** e_1 **then** e_2 **else** e_3 is evaluated as if it was **if** `!`e_1 **then** e_2 **else** e_3. Unlike in Ruby, the **elsif**-part is allowed to appear with this conditional expression. However, there is no syntax sugar for the **else unless** keyword tokens sequence.

The modifier-fashion conditional expression e_1 **if** e_2 **else** e_3 is interpreted as if it was **if** e_2 **then** e_1 **else** e_3. Similarly with the **unless** version and the short form of the modifier conditional expression (without the explicit **else**-part).

Unlike in some languages, conditional expressions do not require to place parentheses around the conditions – but it is possible to do so, the result is equivalent. That might be useful when the condition is inevitably long and needs to span multiple lines – so

that boolean operators may be situated at the beginning of each new line, instead of being at the end of the previous line.

6.17 Loop Expressions

Coral has an elaborate support for loop expressions. Not all structures known from other languages are supported though, e.g. the **`do`** … **`while`** expression, which is expressed differently in Coral.

6.17.1 Loop Control Expressions

Syntax:

```
Loop_Ctrl_Expr ::= Break_Expr
                 | Skip_Expr
                 | Next_Expr
                 | Redo_Expr
                 | Exhausted_Expr
                 | Broken_Expr
Break_Expr     ::= 'break' [label_name] [Cond_Modifier1]
Skip_Expr      ::= 'skip' [integer_literal] [Cond_Modifier1]
Next_Expr      ::= 'next' [label_name] [Cond_Modifier1]
Redo_Expr      ::= 'redo' [label_name] [Cond_Modifier1]
Exhausted_Expr ::= 'exhausted' Block_Expr
Broken_Expr    ::= 'broken' Block_Expr
```

Loop control expressions are made available inside of loop expressions to allow control of the enclosing loops.

In the following paragraphs, a *loop identified by the label l* is a loop expression preceded in its syntax with the syntax element `Label_Dcl` (§6.21.3). All annotations (§6.14) that precede the label declaration are applied to the following loop expression, never to the label.

The **`break`** *l* expression stops the loop labeled with *l*, and omitting the *l* label stops the directly enclosing loop.

The **`skip`** *i* expression skips *i* loop iterations, or with the *i* omitted, skips 1 loop iteration (the current iteration).

The **`next`** *l* expression skips the current loop iteration and every other enclosing iteration until the loop identified by the given label *l* is found, and continues with its next iteration. If the label *l* is omitted, then its behavior is equal to **`skip`** `1`.

The **`redo`** *l* restarts the loop identified by the label *l* (and stops all loops in between), or if *l* is omitted, restarts the directly enclosing loop.

The **exhausted** *e* expression evaluates the expression *e* only if the directly enclosing loop *was not broken* with the **break** keyword.

The **broken** *e* expression evaluates the expression *e* only if the directly enclosing loop *was broken* with the **break** keyword.

The standard library provides loop-like methods, where these loop control structures are not available as keywords, but as methods instead (either imported or available on some given loop-control object) – they might be implemented e.g. using the **throw** expressions (§6.22), that the enclosing loop-like method catches and resolves as appropriate. Only the **exhausted** and **broken** constructs need to be simulated, possibly by optional parameters or additional parameter sections.[6]

6.17.2 Iterable For Expressions

Syntax:

```
Loop_Expr       ::= [Label_Dcl] 'for' Val_Dcls 'in' ['reverse'] Expr
                    ['step' Expr] For_Loop
For_Loop        ::= 'loop' Loop_Block_Expr 'end' ['loop']
                  | '{' Loop_Block_Expr '}'
Loop_Block_Expr ::= {Block_Stat | Loop_Ctrl_Expr}
Val_Dcls        ::= Val_Dcl
                  | Pattern1
```

The *iterable expression* is typed as `Unit`, so there is no point in using its value.

In an expression **for** e_1 **in** e_2 **loop** e_3 **end**, the type of e_2 is expected to conform to `Iterable_Like[`E`]`. The type of e_1 is expected to conform to the type E. The type of e_3 is evaluated to "`()`" anyway. The scope of variables defined in e_1 extends to the e_3 expression.

In expression **for** e_1 **in reverse** e_2 **loop** e_3 **end**, the type of e_2 is expected to conform to `Reverse_Iterable_Like`.

Iterable expressions make use only of the two mentioned traits and the methods defined by them, and therefore advanced iterating mechanisms, such as parallel computations, are not performed – they are simply too complex to be generalized by a simple language construct.

Iterable expression repeats evaluation of the expression e_3 for every value that comes from the `Iterable_Like`'s `Iterator`, unless the loop controls alter this flow (§6.17.1).

Iterable expressions are simple comprehensions over iterating a single iterable value. For more complex iterating expressions, see generators (§6.18).

An expression

[6] In fact, all loop expressions may be interpreted as syntax sugar to such methods. How exactly – that may get into this specification as soon as it is clearly defined.

```
for e1 in e2 loop e3 end
```

is translated to the invocation

```
e2.each { when e1 then e3 }
```

An expression

```
<<l>> for e1 in e2 loop e3 end
```

where l is a label name, is translated to the invocation

```
e2.each({ when e1 then e3 }, label: l')
```

where l' is a symbol literal for the label l.

An expression

```
for e1 in reverse e2 loop e3 end
```

is translated to the invocation

```
e2.reverse.each { when e1 then e3 }
```

An expression

```
<<l>> for e1 in reverse e2 loop e3 end
```

where l is a label name, is translated to the invocation

```
e2.reverse.each({ when e1 then e3 }, label: l')
```

where l' is a symbol literal for the label l.

An expression

```
for e1 in e2 step i loop e3 end
```

is translated to the invocation

```
e2.each({ when e1 then e3 }, step: i)
```

Analogously, other combinations of **`reverse`**, **`skip`** and labeled loops are translated. If the e_3 expression contains a **`exhausted`** expression, then it's block is passed to the `each` method as an argument named `exhausted`, and analogously, if the e_3 expression contains a **`broken`** expression, then it's block is passed to the `each` method as an argument named `broken`.

6.17.3 While & Until Loop Expressions

Syntax:

```
Loop_Expr     ::= [Label_Dcl] ('while' | 'until') Expr For_Loop
                | Loop_Mod_Expr
Loop_Mod_Expr ::= Expr Loop_Modifier
Loop_Modifier ::= ('while' | 'until') Expr
```

The *while loop expression* **while** e_1 **loop** e_2 **end** is typed as `Unit`, so there is no point in using its value.

In an expression **while** e_1 **loop** e_2 **end**, the expression e_1 is treated the same way as the condition part in conditional expressions (§6.16). The type of e_2 is evaluated to "`()`" anyway.

The while loop expression **while** e_1 **loop** e_2 **end** is alone typed and evaluated as if it was an application of a hypothetical function `while_loop (`e_1`) (`e_2`)`, where the function `while_loop` would be defined as follows, with the e_1 and e_2 would be passed by-name:

```
def while_loop (condition: => Boolean)(body: => Unit): Unit := {
<<repeat>>
  if (condition) { body; goto repeat } else {}
}
```

The real implementation has to handle loop control expressions (§6.17.1) around the evaluation of `body` and also handle a label, if one is given; so it is not this simple.

A while loop expression repeats evaluation of the expression e_2 as long as e_1 evaluates to **yes**, unless the loop controls alter this flow (§6.17.1).

6.17.4 Pure Loops

Syntax:

```
Loop_Expr  ::= [Label_Dcl] 'loop'
               (semi Loop_Block_Expr 'end' ['loop'] |
               '{' Loop_Block_Expr '}')
```

The *pure loop expression* **loop** e **end** is typed as `Unit`, so there is no point in using its value.

A pure loop expression repeats evaluation of the expression e as long as the loop controls don't alter this flow (§6.17.1). It is basically equivalent to an iterable expression (§6.17.2) that iterates over an endless iterator.

This expression may also be used to replace the **do** { e_1 } **while** (`(`e_2`)`) expression, known from other languages, using the following structure:

```
loop
  e1
  break if e2
end loop
```

A pure loop expression is the only expression that is not translated into a method call, but rather into another expression. The following constructs are practically the same:

```
// construct with loop
loop
  ...
end loop

// construct with goto
label loop_begin
  ...
  goto loop_begin
```

However, the loop construct has built-in support for loop control expressions.

6.18 Generator Expressions

Syntax:

```
Loop_Expr      ::= 'for' (Generator_Iter | Generator_Expr)
Generator_Iter ::= '(' Enumerators ')' {nl} (Expr | For_Loop)
Generator_Expr ::= '{' Enumerators '}' {nl} 'yield' Expr
Enumerators    ::= Generator {semi Enumerator}
Enumerator     ::= Generator
                 | Guard
                 | Pattern1 ':=' Expr
Generator      ::= [Label_Dcl] Pattern1 '<-' Expr [Guard]
Guard          ::= Cond_Modifier1
```

A *generator iteration* **for** (*enums*) *e* executes expression *e* for each binding generated by the enumerators *enums* and as an expression, it is typed as `Unit`. A *generator expression* **for** {*enums*} **yield** *e* evaluates expression *e* for each binding generated by the enumerators *enums* and collects the results.

An enumerator sequence always starts with a generator; this can be followed by further generators, value definitions or guards. A *generator* *p* <- *e* produces bindings from an expression *e*, which are matched in some way against pattern *p* (§8). A *value definition* *p* := *e* binds the value name *p* (or several names in a pattern *p*) to the result of evaluating the expression *e*. A *guard* **if** *e* (or **unless** *e*) contains a boolean expression *e*, which restricts enumerated bindings. The precise meaning of generators

and guards is defined by translation to invocations of four methods: `map`, `with_filter`, `flat_map` and `each`. These methods can be implemented in different ways for different carrier types.

The translation scheme is defined as follows. In a first step, every generator p `<-` e, where p is not irrefutable (§8.1.15) for the type of e, is replaced by

p `<-` e`.with_filter { when` p `then yes else no }`

Then, the following rules are applied repeatedly, until all comprehensions are eliminated.

- A comprehensioin

  `for {`p `<-` e`}` `yield` e'

  is translated to

  e`.map { when` p `then` e' `}`

- A comprehension

  `for {<<`l`>>` p `<-` e`}` `yield` e'

  where l is a label name, is translated to

  e`.map({ when` p `then` e' `}, label:` l'`)`

  where l' is a symbol literal for the label l.

- A comprehension

  `for (`p `<-` e`)` e'

  is translated to

  e`.each { when` p `then` e' `}`

- A comprehension

  `for (<<`l`>>` p `<-` e`)` e'

  where l is a label name, is translated to

  e`.each({ when` p `then` e' `}, label:` l`)`

  where l' is a symbol literal for the label l.

- A comprehension

```
for {p <- e; p′ <- e′ …} yield e″
```

where … is a (possibly empty) sequence of generators, value definitions, or guards, is translated to

```
e.flat_map { when p then for {p′ <- e′ … } yield e″ }
```

- A comprehension

```
for {<<l>> p <- e; p′ <- e′ …} yield e″
```

where l is a label name, and where … is a (possibly empty) sequence of generators, value definitions, or guards, is translated to

```
e.flat_map(
  { when p then for {p′ <- e′ … } yield e″ },
  label: l
)
```

where l' is a symbol literal for the label l.

- A comprehension

```
for (p <- e; p′ <- e′ …) e″
```

where … is a (possibly empty) sequence of generators, value definitions, or guards, is translated to

```
e.each { when p then for (p′ <- e′ …) e″ }
```

- A comprehension

```
for (<<l>> p <- e; p′ <- e′ …) e″
```

where … is a (possibly empty) sequence of generators, value definitions, or guards, is translated to

```
e.each(
  { when p then for (p′ <- e′ …) e″ },
  label: l
)
```

- A generator `p <- e` followed by a guard `if g` is translated to a single generator

```
p <- e.with_filter((x1, …, xn) -> { g })
```

where $x_1, \ldots, x_n$ are the free variables of the pattern p.

- A generator p `<-` e followed by a guard **`unless`** g is translated to a single generator

  p `<-` e`.with_filter((`$x_1, \ldots, x_n$`) -> { !(`g`) })`

  where $x_1, \ldots, x_n$ are the free variables of the pattern p.

- A generator p `<-` e followed by a definition p' `:=` e' is translated to the following generator of pairs of values, where x and x' are fresh names:

  `(`p`, `p'`) <-` **`for`** `{`x `@` p `<-` e`}` **`yield`** `{` **`val`** x' `@` p' `:=` e'`; (`x`, `x'`) }`

Generators in generator expression can optionally have a label l assigned, so that expressions like **`break`** l could work.[7] Like with other loop expressions, if an **`exhausted`** or a **`broken`** loop control expression is given in the generator iteration expression e, it is passed to the outermost `each` method as a named argument, possibly along the **`label`** argument.

Example 6.18.1 The following code produces all pairs of numbers between 1 and $n-1$, whose sums are prime numbers.

```
for { i <- 1 .. n
      j <- 1 .. i
      if is_prime? i + j
} yield (i, j)
```

The comprehension is translated to:

```
(1..n).flat_map {
  when i then (1 .. i).
    with_filter { (j) -> { is_prime? i + j } }.
    map { when j then (i, j) }
}
```

Example 6.18.2 Generator expressions can be used to express vector and matrix algorithms concisely.

```
def transpose[$A](xss: List[List[$A]]): List[List[$A]] :=
  for {i <- 0 .. xss(0).length} yield {
    for (xs <- xss) yield xs(i)
  }
```

The comprehension is translated to:

[7] It is up to the concrete method how it handles the invocation, which uses **`throw-catch`** expressions – however, ignoring it may result in an uncaught `Throwable` killing the thread.

```
def transpose[$A](xss: List[List[$A]]): List[List[$A]] :=
  (0 .. xss(0).length).
    map {
      when i then xss.
        map { when xs then xs(i) }
    }
```

6.19 Collection Comprehensions

Syntax:

```
List_Literal       ::= '%' Collection_Flags '[' 'for' Generator_Expr ']'
Dictionary_Literal ::= '%' Collection_Flags '{' 'for' Generator_Expr '}'
Bag_Literal        ::= '%' Collection_Flags '(' 'for' Generator_Expr ')'
```

Collection comprehensions extend the syntax of collection "literals"[8], so that collections may be defined by not their explicit values, but by a function that generates them – and that function is a generator. Only tuple literals don't have collection comprehension, due to their special nature within the language.

Note that the generator expression for dictionary literal comprehension has to generate values of type `(`K`,` E`)`, where K is the type of the keys and E is the type of mapped values.

6.20 Pattern Matching & Case Expressions

Syntax:

```
Match_Expr     ::= Pat_Match_Expr | Case_Expr
Pat_Match_Expr ::= 'match' Simple_Expr1 Match_Body
Match_Body     ::= semi When_Clauses 'end' ['match']
                 | '{' When_Clauses '}'
When_Clauses   ::= When_Clause {['next'] semi When_Clause}
                   [['next'] 'else' Cond_Block]
When_Clause    ::= 'when' Pattern [Guard] ('then' | semi) Cond_Block
Case_Expr      ::= 'case' Simple_Expr1 Case_Body
Case_Body      ::= semi Case_Clauses 'end' ['case']
                 | '{' Case_Clauses '}'
Case_Clauses   ::= Case_Clause {['next'] semi Case_Clause}
                   [['next'] 'else' Cond_Block]
```

[8]Pure literals are terminal symbols in the language, but collection literals are wrappers around virtually any expression.

```
Case_Clause    ::= 'when' Case_Patterns ('then' | semi) Cond_Block
Case_Patterns  ::= Case_Pattern {',' Case_Pattern}
Case_Pattern   ::= Stable_Id
                 | literal [('..' | '...') literal]
                 | id
                 | Infix_Expr
```

Pattern matching is described in (§8). Here, the syntax of expressions that make use of pattern matching is given.

Case expressions

case e { **when** c_1 **then** b_1 ... **when** c_n **then** b_n **else** b_{n+1} }

are simplified pattern matching expressions, though they do not use pattern matching at all, but *case equality* instead, defined with the method "===". Thus, case expressions do not aim at matching the selector expression e, thus decomposing the selector e, but rather tests if it falls into a particular set of values, defined using the case equality method, by:

- a type of values c.`===`(e),
- a range of values (e_1 .. e_2).`===`(e),
- a set of values defined by another value e_1.`===`(e).

Let T be the type of the selector expression e. The parameter p_i of each invocation of the method "===" is typed with T as its expected type and `Boolean` as the return type. It is an error if T does not conform to the actual type of the parameter, as the invocation would not be applicable (§6.6). The method "===" may be overloaded for multiple parameter types, then overloading resolution (§6.26.3) applies as usual.

The method "===" is defined basically as follows:

```
operator === (x: T): Boolean
  ...
end
```

where x is the parameter name and T is the expected type of the parameter, and the type of the selector expression e.

The expected type of every block b_i is the expected type of the whole pattern matching expression. The type of the pattern matching expression is then the weak least upper bound (§3.7.2) of the types of all blocks b_i.

Multiple values can define a case pattern, for convenience. Note that no variables are bound from the case pattern to the corresponding block.

6.21 Unconditional Expressions

Unconditional expressions change the flow of programs without a condition.

6.21.1 Return Expressions

Implicit return expressions. Implicit return expression is always the value of the last expression in a code execution path.

Syntax:

```
Result_Expr ::= Anon_Params '->' Block
              | ['memoize'] Expr
```

Explicit return expressions. Explicit return expressions unconditionally change the flow of programs by making the enclosing function definition return a value early (or return no value).

Syntax:

```
Return_Expr ::= ['memoize'] 'return' [Expr] [Cond_Modifier1]
```

A return expression **`return`** *e* must occur inside the body of some enclosing method, or inside a block nested in the body of the innermost enclosing method. Unlike in Scala, the innermost enclosing method in a source program, f, does not need to have an explicitly declared result type, as the result type can be inferred as the weak least upper bound of all return paths, including the explicit return path. The return expression evaluates the expression *e* and returns its value as the result of f. The evaluation of any statements or expressions following the return expression is omitted.The type of a return expression is `Nothing`.

The expression *e* may be omitted, then the return expression **`return`** is type-checked and evaluated as if it was **`return`** `()`, typed as `Unit`.

Returning from a nested block is implemented by throwing and catching a specialized exception, which may be seen by **`throw-catch`** expressions (§6.22) between the point of return and the enclosing method. If such a block is captured and run later, at the point where the original call stack frame is long gone, the exception might propagate up the call stack that ran the captured block. Returning from anonymous functions does not affect the enclosing method.

Memoized return expressions. A returned expression may optionally be memoized, by using the keyword **`memoize`** right before the returned expression *e* or **`return`** *e*. In that case, arguments and reference to **`self`** are captured and stored along the returned

value, so that further calls to the same method with the same arguments may be sped up significantly (§6.6.10). Memoization is not available from within anonymous functions and blocks.

6.21.2 Structured Return Expressions

Syntax:

```
Return_Expr ::= ['memoize'] 'return' Var_Def 'do'
                Block_Stat {semi Block_Stat}
                'end' ['return']
```

A structured return expression is practically the same as explicit return expression. The variable defined in it has its scope extended to the following block statements, which are evaluated, and then the variable is returned.

6.21.3 Local Jump Expressions

Syntax:

```
Jump_Expr  ::= Goto_Expr | Label_Dcl
Goto_Expr  ::= 'goto' label_name [Cond_Modifier1]
Label_Dcl  ::= 'label' label_name
             | '<<' label_name '>>'
label_name ::= plain_id
```

Local jumps transfer control from the points of **`goto`** statements to the statements following a **`label`**. Such a jump may only occur inside of the same function, i.e. it is not possible to jump from one method to another. Also, the jump can't happen to be from outside of a loop into a loop, but the other way around is possible. The only loop expressions that may be jumped out of are the pure loop (§6.17.4) and a **`while`** loop, which are not transformed as comprehensions into method calls.

6.21.4 Continuations

Unlimited continuations. Unlimited continuations are defined by the whole program, as the unlimited continuation allows almost arbitrary non-local jumps. The unlimited continuation is captured with `call/cc` function.

Delimited continuations. Delimited continuations are defined with `reset` and `shift` functions. Reset and shift expressions are actually not language constructs, but rather regular functions that have a native implementation capable of unconditionally changing the standard control flow of a program. Moreover, the first `shift` expression (which captures the delimited continuation) controls the return value of the `reset` expression, which overrides the implicit return expression (§6.21.1).

6.22 Throw, Catch & Ensure Expressions

Syntax:

```
Catch_Expr  ::= Catch_Expr1 | Catch_Expr2
Catch_Expr1 ::= 'begin' Block
                'catch' semi When_Clauses
                ['ensure' semi Block_Stat {semi Block_Stat}] 'end'
                'end'
Catch_Expr2 ::= '{' Block '}'
                'catch' '{' When_Clauses '}'
                ['ensure' '{' Block_Stat {semi Block_Stat} '}']
Throw_Expr  ::= 'throw' Expr
```

A throw expression **`throw`** e evaluates the expression e. The type of this expression must conform to `Throwable`. It is an error if e evaluates to **`nil`** or `()`. If there is an active **`begin-catch`** expression that handles the thrown value, evaluation is resumed with the handler, otherwise a thread executing the **`throw`** is aborted. The type of a **`throw`** expression is `Nothing`.

A **`begin-catch`** expression is of the form `{` b `}` **`catch`** h, where h is a handler pattern matching anonymous function (§8.4)

`{` **`when`** p_1 **`then`** b_1 … **`when`** p_k **`then`** b_k **`else`** b_{k+1} `}` **`.`**

This expression is evaluated by evaluating the block b – if evaluation of b does not throw any value, the result of b is returned, otherwise the handler h is applied to the thrown value. If the handler h contains a **`when`** clause matching the thrown value, the first such clause is invoked (and may throw another value, or the same value). If the handler contains no such clause, the value is re-thrown.

Let T be the expected type of the **`begin-catch`** expression. The block b is expected to conform to T. The handler h is expected to conform to type `Partial_Function[Throwable,` T`]`. The type of the **`begin-catch`** expression is the weak least upper bound (§3.7.2) of the type of b and the result type of h.

A **`begin-ensure`** expression `{` b `}` **`ensure`** `{` e `}` evaluates the block b. If evaluation of b does not cause any value to be thrown, the block e is evaluated. If any value is thrown during evaluation of e, the evaluation of the whole expression is aborted with the thrown value. If no value is thrown during evaluation of e, the result of b is returned as the result of the whole expression, unless e contains an explicit **`return`** (§6.21.1) – in that case, the value returned from e replaces the value returned from b, even if b returns a value explicitly.

If a value is thrown during evaluation of b, the **`ensure`** block e is also evaluated. If another value is thrown during evaluation of e, evaluation of the whole expression is aborted with the new thrown value and the previous is discarded. If no value is thrown

during evaluation of *e*, the original value thrown from *b* is re-thrown once evaluation of *e* has completed, unless *e* again contains an explicit **`return`** (§6.21.1) – in that case, the value thrown from *b* is discarded, and the value returned from *e* is returned.

The block *b* is expected to conform to the expected type of the whole expression and the **`ensure`** block *e* is expected to conform to type `Unit`.

An expression `{` *b* `}` **`catch`** e_1 **`ensure`** `{` e_2 `}` is a shorthand for `{{` *b* `}` **`catch`** e_1 `}` **`ensure`** `{` e_2 `}`.

6.22.1 Raise Expressions

Syntax:

```
Raise_Expr ::= 'raise' Raiseable
Raiseable  ::= string_literal
             | Path [',' string_literal]
             | Expr
```

A raise expression **`raise`** *e* is similar to **`throw`** *e* (§6.22), it throws a value (raises an error) that is expected to be of type `Raiseable`. It has three variants:

`raise` *s*, where *s* is a string provided to constructor of the type `Runtime_Error`.

`raise` *T*`,` *s*, where *s* is a string provided to constructor of the type *T*.

`raise` *e*, where *e* is an expression, whose type is expected to conform to `Raiseable`, and whose result value will be raised after its evaluation.

`raise`, which raises a value of type `Runtime_Error` without any message. Such errors should not propagate outside of the method that raises them.

`Raiseable` is a subtype of `Throwable`.

6.22.2 Rescue Expressions

Syntax:

```
Rescue_Expr  ::= [Label_Dcl] (Rescue_Expr1 | Rescue_Expr2)
Rescue_Expr1 ::= 'begin' Block
                 'rescue' [Pattern [Guard]] semi Rescue_Block
                 {'rescue' [Pattern [Guard]] semi Rescue_Block }
                 ['ensure' semi Block_Stat {semi Block_Stat}] 'end'
Rescue_Expr2 ::= '{' Block '}'
                 'rescue' [Pattern [Guard]] '{' Rescue_Block '}'
                 {'rescue' [Pattern [Guard]] '{' Rescue_Block '}'}
```

```
                ['ensure' '{' Block_Stat {semi Block_Stat} '}']
Fun_Stats    ::= Block
                {'rescue' [Pattern [Guard]] semi Rescue_Block }
                ['ensure' semi Block_Stat {semi Block_Stat}]
Rescue_Block ::= {Rescue_Stat semi} [Result_Expr]
Rescue_Stat  ::= Block_Stat | Retry_Expr
Retry_Expr   ::= 'retry' [label_name] [Cond_Modifier1]
```

Rescue expression **`rescue`** h is similar to catch expression (§6.22), with two major differences: First, each **`rescue`** is followed by **`when`** clause; second, all **`rescue`** expressions in the same scope form together a handler h, where rules from catch expressions apply, only that h is expected to conform to type `Partial_Function[Raiseable,` T`]`, where T is the expected type of the whole **`begin`**-**`rescue`** expression.

The syntactic overlap with **`ensure`** expression signifies that the same expression with the exact same behavior may be used with **`rescue`** expressions as well.

Optionally, the **`rescue`** may appear before any **`begin`** keyword, being connected to the function body instead as the expression protected agains raiseables (this does not apply to **`catch`**), where the **`begin`** is implied to be at the very start of the function's body.

The **`retry`** l expression is available inside of each raiseable handler block, and evaluating it restarts evaluation of the whole expression since **`begin`** of the labeled rescue expression, or if no label is given, then of the innermost (if nested) rescue expression. Again, it is not available in catch handler block.

The keyword **`try`** is not available in Coral – in Coral, there is no trying, there is doing or not doing.

6.23 Anonymous Functions

Syntax:

```
Anon_Fun        ::= Anon_Params '->' '{' Block '}'
Result_Expr     ::= Anon_Params '->' Block
Anon_Params     ::= Bindings {'->' Bindings}
                  | Param_Clause
                  | '(' ['implicit'] id ')'
                  | '(' [Nameless_Param] ')'
Bindings        ::= '(' Binding {',' Binding} ')'
Binding         ::= (id | '_') [':' Type]
Nameless_Params ::= Nameless_Param {',' Nameless_Param}
Nameless_Param  ::= '_' [':' Type]
```

The anonymous function `(`x_1`:` $T_1, \ldots,$ x_n`:` T_n`) -> {` b `}` maps parameters x_i of types T_i to a result value given by evaluation of block b. The scope of each formal

parameter x_i is e. Formal parameters must have pairwise distinct names.[9]

If the expected type of an anonymous function is of the form `Function_`n`[`$S_1, \ldots, S_n$`, `R`]`, the expected type of b is R and the type T_i of any of the parameters x_i can be omitted, in which case $T_i = S_i$ is assumed. If there is no expected type of the anonymous function, then for each parameter x_i which has no explicit type T_i, T_i is assumed to be `Object`, and the type of the result value is also assumed to be `Object`.

Note that anonymous functions explicitly specify all of their parameters, unlike anonymous pattern matching functions (§8.4), where the parameters are inferred from the expected type.

The anonymous function is evaluated as the following expression:

```
(Function_n[S_1, ..., S_n, T] with {
  def apply (x_1: T_1, ..., x_n: T_n): T := { b }
}).new
```

In the case of a single untyped formal parameter, `(`x`) -> { `b` }` can be abbreviated to `$x$ `-> { `$b$` }`. If an anonymous function `(`$x$`: `$T$`) -> { `$b$` }` with a single typed parameter appears as the result of expression of a block, it can be abbreviated to `x`: `T `-> { `b` }`.

A formal parameter may also be a wildcard represented by an underscore "_". In that case, a fresh name for the parameter is chosen arbitrarily.

A parameter of an anonymous function may optionally be preceded by an **implicit** modifier. In that case the parameter is labeled **implicit** (§7); however the parameter section itself does not count as an implicit parameter section in the sense of (§7.2). Such a parameter **implicit** x_i is then added transparently to the block b as **implicit val** y_i := x_i, where y_i is a fresh name. Also, therefore arguments to anonymous functions always have to be given explicitly.

Example 6.23.1 Examples of anonymous functions:

```
// identity function
x -> { x }

// curried function composition
f -> g -> x -> { f(g(x)) }

// a summation function
(x: Integer, y: Integer) -> { x + y }
```

[9] In future versions of Coral, a syntax where curly brackets are not required to be surrounding an anonymous function's body may be allowed.

```
// a function which takes an empty parameter list,
// increments a non-local variable (via closure)
// and returns the new value
() -> { count += 1; count }

// a function that ignores its argument and returns 5
_ -> { 5 }
```

6.23.1 Method Values

Syntax:

```
Simple_Expr ::= '&' Simple_Expr1
              | '&' '(' Simple_Expr1 ')'
```

The expression `&`e (or alternatively `&(`e`)`) is well-formed if e is of method type or if e is a call-by-name parameter. If e is a method with parameters, `&`e represents e converted to a function type by eta expansion (§6.26.5). If e is a parameterless method or call-by-name parameter of type `=>` T, `&`e represents the function of type `() ->` T, which evaluates e when it is applied to the empty parameter list `()`.

Example 6.23.2 The method values in the left column are each equivalent to the anonymous functions (§6.23) on their right.

```
&(Math.sin)               (x)      -> { Math.sin(x) }
&(Array.range)            (x1, x2) -> { Array.range(x1, x2) }
&(List.map_2)             (x1, x2) -> (x3) -> { List.map_2(x1, x2)(x3) }
&(List.map_2(xs, ys))     (x)      -> { List.map_2(xs, xy)(x) }
&(42.`*`)                 (x)      -> { 42 * x }
```

Note that if e resolves to a parameterless method of type `() ->` T or if e has a method type `()` $\mapsto$ T, it is evaluated to type T (§6.26.2) – and the method value syntax provides a way to prevent this.

6.24 Anonymous Classes

Syntax:

```
Anon_Class      ::= ['class' [Class_Param_Clauses] 'extends']
                    [Early_Defs] Anon_Class_Tmpl
Anon_Class_Tmpl ::= Class_Parents 'with' '{' [Template_Body] '}'
```

Anonymous classs are a mechanism to implement an abstract class or override a concrete class "ad hoc", in place where needed, without needing to create a new constant (although as an expression, the anonymous class definition can indeed be assigned to a constant and gain its name). Anonymous classes can't be type constructors (§3.4.3).

A minimal anonymous class expression is of the form *c* **with** { *t* }, where *c* is the class that the anonymous class inherits from (can be even `Object`), and *t* is the template of the anonymous class. The anonymous class inherits all traits mixed into this parent class, and can itself include or prepend more traits (via the `Class_Parents` syntax element).

Optionally, the anonymous class may define its own primary constructor parameters, in which case the form of the anonymous class is **class** $(ps_1)\ldots(ps_n)$ **extends** *c* **with** { *t* }, where ps_1 to ps_n are the primary constructor parameters. Superclass constructor arguments may be specified in any case.

6.25 Statements

Syntax:

```
Block_Stat    ::= Use
                | {Annotation} ['implicit'] Def
                | {Annotation} {Local_Modifier} Tmpl_Def
                | Expr
                | Alias_Expr
                | ()
Template_Stat ::= Use
                | {Annotation} {Modifier} Def
                | {Annotation} {Modifier} Dcl
                | 'include' Container_Path [Cond_Modifier]
                | 'prepend' Container_Path [Cond_Modifier]
                | Expr
                | Alias_Expr
                | ()
Fun_Stats     ::= [Fun_Stat {semi Fun_Stat}] Return_Expr
                | Block
                  {'rescue' [Pattern [Guard]] semi Block }
                  ['ensure' semi Block_Stat {semi Block_Stat}]
Fun_Stat      ::= Block_Stat
Fun_Dec_Expr  ::= {Annotation} Dcl
                | {Annotation} ['implicit'] Def
                | 'transparent'
                | 'opaque'
                | 'native' [Expr]
```

```
                | ()
Alias_Expr    ::= 'alias' symbol_literal 'is' symbol_literal
```

Statements occur as parts of blocks and templates. Despite their name, they are actually generally expressions as well, except that for some statements, their value is not much of a use, i.e. use clauses, whose value is a **`nil`**, or the empty statement/expression, whose value is again **`nil`**.

Function statements is an umbrella term for a series of statements and expressions, so their effective value is more complex.

An expression that is used as a statement can have an arbitrary value type. An expression statement *e* is evaluated by evaluating *e* and discarding and releasing the result of the evaluation.

Block statements may be definitions, which bind local names in the block. The only modifier allowed in all block-local definitions is **`implicit`**. When prefixing a class or object definition, modifiers **`abstract`**, **`final`** and **`sealed`** are also permitted (§5.2).

Evaluation of a statement sequence entails evaluation of the statements in the order they are written. This behavior can be overridden for statement sequences in workflows (§6.27).

Statement can be an import via a use clause (§4.10), a definition or an expression, or it can be empty. Statements used in the template of a class definition can also be declarations.

A function that is declared with **`transparent`** in its **`declare`** block, is visible as referentially transparent, and therefore the compiler and possibly the runtime as well are given the possibility to replace function applications of this same function with its previously computed result with the same arguments on the same receiver instance. In that sense, it is similar to memoization (§6.6.10), but skips one call stack frame and works better during compilation, unlike memoization, which is a runtime feature. On the other hand, a function that is declared with **`opaque`** in its **`declare`** block, is visible as referentially opaque and those optimizations are disabled for it, so the function is re-evaluated each time it is applied.

A function that is declared with **`native`** in its **`declare`** block, has its body defined outside of Coral source files, possibly in other languages, native to the platform of the Coral VM. For now, some CSL functions are declared this way and a possibility to let users define their own native functions is upcoming in future versions of Coral. Note that the native implementation of the function would have to match the expected platform of the Coral VM, or maybe the expression would map platforms to the native implementations of the function.

An alias to a function name creates a duplicate record in method table of a class or a duplicate variable pointing to the aliased function name. From that scope on, the functions are bound by name, and aliased function names are also inherited. It is an error if a subtype tries to override an aliased function name.

6.26 Implicit Conversions

Implicit conversions can be applied to expressions whose type does not match their expected type, to qualifiers in selections, and to unapplied methods. The available implicit conversions are given in the next two sub-sections.

We say that a type T is *compatible* to a type U if T weakly conforms to T after applying eta-expansion (§6.26.5) and view applications (§7.3), if necessary.

6.26.1 Value Conversions

The following implicit conversions can be applied to an expression e, which is of some value type T and which is type-checked with some expected type et. Some of these implicit conversions may be disabled with pragmas.

Overloading resolution. If an expression denotes several possible members of a class, overloading resolution (§6.26.3) is applied to pick a unique member.

Type instantiation. An expression e of a polymorphic type

$$[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n] \rightarrow T$$

which does not appear as the function part or a type application is converted to a type instance of T by determining with local type inference (§6.26.4) instance types $T_1, \ldots, T_n$ for the type variables $a_1, \ldots, a_n$ and implicitly embedding e in the type application $e[T_1, \ldots, T_n]$ (§6.7).

Numeric widening. If e is of a number type which weakly conforms (§3.7.2) to the expected type, it is widened to the expected type.

Numeric narrowing. If the expected type has smaller range than the number type of e, but the value of e fits into the expected type, it is narrowed to the expected type.

Value discarding. If e is of some value type and the expected type is `Unit`, e is converted to the expected type by embedding it in the block `{ e; () }`.

View application. If none of the previous conversions applies, view applications are not disallowed by pragmas (implicitly they are allowed), and e's type does not conform to the expected type et, an attempt is made to convert e to the expected type with a view application (§7.3). This can happen in compile time only if all necessary type information is available, otherwise, runtime handles it by using specialized instructions (and those instructions are disabled from compilation when view applications are disabled).

Dynamic member selection. If none of the previous conversions applies, and e is a prefix of a selection $e.x$, then if e's type conforms to `Dynamic_Member_Selecting`, the selection is rewritten according to rules for dynamic member selection (§6.26.6). Otherwise, `member_not_found` is invoked on the receiver, whose implementation in `Object` is to raise an error.

6.26.2 Method Conversions

The following implicit conversions can be applied to methods which are not applied to some argument list.

Evaluation. A parameterless method m of type `() -> `T is always converted to type T by evaluating the expression to which m is bound.

Implicit application. If the method takes only implicit parameters, implicit arguments are passed following the rules of (§7.2).

Eta expansion. Otherwise, if the expected type et is a function type `(`Ts'`) -> `T', eta-expansion (§6.26.5) is performed on the expression e.

Empty application. Otherwise, if e is of a method type `() `$\mapsto$` `T, it is implicitly applied to the empty argument list, yielding e`()`.

6.26.3 Overloading Resolution

If an identifier or selection e references several members of a class, the context of the reference is used to identify a unique member, if possible. The way this is done depends on whether or not e is used as a function. Note that even if overloaded resolution picks up a unique member, that member still may not be applied in regard of the actual expected types of the function application. Let $\mathscr{A}$ be the set of members referenced by e.

6.26.3.1 Function in an application

Assume first that e appears as a function in an application, as in $e(e_1, \ldots, e_m)$.

Shape-based overloading resolution. One first determines the set of functions that is potentially applicable based on the *shape* of the arguments.

The shape of an argument expression e, written *shape*`(`e`)`, is a type that is defined as follows:

- For a function expression `(`p_1`:` T_1`, ..., `p_n`:` T_n`) -> `b, the shape is `(Any, ..., Any) ->` *shape*`(`b`)`, where `Any` occurs n times in the argument type.
- For a named argument n`:` e, the shape is `@[named_arg :`n`]` *shape*`(`e`)`, which is an annotated type.[10]
- For all other expressions, the shape is `Nothing`.

Let $\mathscr{B}$ be the set of alternatives in $\mathscr{A}$ that are *applicable* (§6.6) to expressions `(`$e_1, \ldots, e_n$`)` of types `(`*shape*`(`e_1`), ..., `*shape*`(`e_n`))`. If there is precisely one alternative in $\mathscr{B}$, that alternative is chosen. It is an error if that alternative is not applicable to the expected types of the argument expressions – the method is unapplied (§6.26.1).

Argument counts based overloading resolution. Otherwise, let $S_1, \ldots, S_m$ be the vector of types obtained by typing each argument with an undefined expected type (kind of equivalent to typing it with `Any`), keeping the annotations of named arguments attached (from the previous step with the shape of arguments). For every member m in $\mathscr{B}$, one determines whether it is applicable to expressions `(`$e_1, \ldots, e_m$`)` of types $S_1, \ldots, S_m$, which drops requirements set up by *shape*`(`e`)`, namely those for function expressions, and therefore members in $\mathscr{B}$ are more likely to be selected. It is an error if none of the members in $\mathscr{B}$ are applicable – the method is unapplied. If there is one single applicable alternative, that alternative is chosen.

Applicability based overloading resolution. Otherwise, let $\mathscr{C}$ be the set of applicable alternatives in the application to $e_1, \ldots, e_m$. It is again an error if $\mathscr{C}$ is empty. Otherwise, one chooses the *most specific* alternative among the alternatives in $\mathscr{C}$, according to the following definition of being "more specific than".

Definition 6.26.1 The *relative weight* of an alternative A over an alternative B is defined as the sum of relative weights of each argument e_i in the application to $e_1, \ldots, e_m$. In the following equation, A_i is the type of the parameter corresponding to e_i in the alternative A, and B_i is the type of the parameter corresponding to e_i in the alternative B.

$$\begin{aligned} \mathit{weight}(A,B) &= \textstyle\sum_{i=1}^{m} \mathit{pweight}(A_i, B_i) \\ \mathit{pweight}(t,u) &= \mathit{cweight}(t,u) + \mathit{rweight}(t) \end{aligned}$$

$$\begin{aligned} \mathit{cweight}(t,u) &= \begin{cases} 1 & \text{if } t <: u \\ 0 & \text{otherwise} \end{cases} \\ \mathit{rweight}(t) &= \begin{cases} 1 & \text{unless } t \text{ is a repeated or a capturing parameter} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

[10] This is different from e.g. Scala, since Coral supports captured named arguments, which make the definition of applicable functions different.

An alternative A is *more specific than* an alternative B, if the relative weight of A over B is greater than the relative weight of B over A.

If there are more alternatives in $\mathscr{C}$ that are equally most specific, one chooses one of them as in overloading resolution without any application (§6.26.3.3), where $\mathscr{A}$ is the same as $\mathscr{C}$ here, and the expected type is the expected type of the function application.

Note. As overloading on return type is possible, note that overloading based on arguments takes precedence over overloading on return type, even if overloading on return type would make different alternative to be chosen.

Example 6.26.2 Assume the following overloaded function definitions:

```
def f (*x: Integer) end             // 1.
def f (x: Integer) end              // 2.
def f (x: Integer, y: Integer) end  // 3.
```

In the application `f(1)`, there are two applicable alternatives in regard to both shape and argument counts – the first two. Applicability test gives relative weight to (1) over (2) of 1, since it has a repeated parameter, and relative weight to (2) over (1) of 2, therefore the second is chosen.

In the application `f(1, 2)`, there are again two applicable alternatives – the first and the last. Applicability test gives relative weight to (1) over (3) of 2, since it has a repeated parameter matching both arguments, and relative weight to (3) over (1) of 4, therefore the second is chosen.

In the application `f(1, 2, 3)`, there is only one applicable alternative (the first), which can be detected (as soon as) based on the shape of its argument expressions.

6.26.3.2 Function in a type application

Assume next that e appears as a function in a type application, as in e`[`*targs*`]`. Then let $\mathscr{B}$ be the set of all alternatives in $\mathscr{A}$ which take the same number of type parameters as there are type arguments in *targs* are chosen. It is an error if no such alternative exists – the type application is unapplied. If there is one such alternative, that one is chosen.

Otherwise, let $\mathscr{C}$ be the set of those alternatives in $\mathscr{B}$ that are applicable to the type arguments, so that the bounds defined by the alternative's type parameters are satisfied. It is an error if no such alternative exists, and it is also an error if there are several such alternatives, as there is (for now) no way to select a unique member. If there is one such alternative, that one is chosen.

6.26.3.3 Expression not in any application

Assume finally that e does not appear as a function in either an application or a type application, (or that overloading resolution on a function in an application was left with several most specific alternatives). If an expected type is given, let $\mathscr{B}$ be the set of those alternatives in $\mathscr{A}$ which are compatible (§6.26) to it. Otherwise, let $\mathscr{B}$ be the same as $\mathscr{A}$. It is an error if there is no such alternative. If there is one such alternative, that one is chosen.

Otherwise, one chooses the *most specific* alternative among the alternatives in $\mathscr{B}$, according to the following definition of being "more specific than":

Definition 6.26.3 The *relative weight* of an alternative A over an alternative B is defined as a number from 0 to 1, defined as:

- 1 if $A <: B$, 0 otherwise.

An alternative A is *more specific than* an alternative B, if the relative weight of A over B is greater than the relative weight of B over A.

It is an error if there is no alternative in $\mathscr{B}$ which is more specific than all other alternatives in $\mathscr{B}$ – the method is unapplied.[11]

6.26.4 Local Type Inference

Local type inference infers type arguments to be passed to expressions of polymorphic type. Say e is of a type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n] \mapsto T$ and no explicit type arguments are given.

Local type inference converts this expression to a type application $e[T_1, \ldots, T_n]$. The choice of the type arguments $T_1, \ldots, T_n$ depends on the context in which the expression appears and on the expected type *et*.

Local type inference is not always able to infer all type arguments, and sometimes may even infer useless ones – in that cases, explicit type arguments are to be used to solve the problems.

Case 1: Selections. If the expression appears as a prefix of a selection with a name x, then type inference is *deferred* to the whole expression $e.x$. That is, if $e.x$ has type S, it is now treated as having type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n] \mapsto S$, and local type inference is applied in turn to infer type arguments for $a_1, \ldots, a_n$, using the context in which $e.x$ appears.

[11] This can be fixed e.g. by using typed expressions (§6.13).

Case 2: Values. If the expression e appears as a value without being applied to some value arguments in a function application, the type arguments are inferred by solving a constraint system which relates the expression's type T with the expected type *et*. Without loss of generality we can assume that T is a value type; if it is a method type, we apply eta-expansion (§6.26.5) to convert it to a function type (which is a value type and we're home again). Solving means finding a substitution σ of types T_i for the type parameters a_i, such that all of the following conditions hold:

- None of inferred types T_i is a singleton type (§3.3.1).
- All type parameter bounds are satisfied, i.e. $\sigma L_i <: \sigma a_i$ and also $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The expression's type conforms to the expected type, i.e. $\sigma T <: \sigma et$.

It is an error if no such substitution exists. If several substitutions exist, local type inference will choose for each type variable a_i a minimal or a maximal type T_i of the solution space (where a maximal type is closer to `Object` and minimal is closer to `Nothing`). A *maximal* type T_i will be chosen if the type parameter a_i appears contravariantly (§4.6) in the type T of the expression. A *minimal* type T_i will be chosen in all other situations, i.e. if the type variable appears covariantly, non-variantly or not at all[12] in the type T. We call such a substitution σ an *optimal solution* of the given constraint system for the type T.

Case 3: Methods. This case applies if the expression e appears in an application $e(d_1, \ldots, d_m)$. In that case T is a method type $(p_1 \colon R_1, \ldots, p_n \colon R_n) \Rightarrow T'$. Without loss of generality we can assume that the result type T' is a value type; if it is a method type, we apply eta-expansion (§6.26.5) to convert it to a function type (which is a value type and we're home, yet again). Once computes first the types S_j of the argument expressions d_j, using two alternative schemes. Each argument expression d_j is typed first with the expected type R_j, in which the type parameters $a_1, \ldots, a_n$ are taken as type constants. If this fails the argument d_j is typed instead with an expected type R'_j, which results from R_j by replacing every type parameter in $a_1, \ldots, a_n$ with *undefined*.

In a second step, type arguments are inferred by solving a constraint system, which relates the method's type with the expected type *et* and the argument types $S_1, \ldots, S_m$. Solving the constraint system means finding a substitution σ of types T_i for the type parameters a_i, such that all of the following conditions hold:

- None of inferred types T_i is a singleton type (§3.3.1).
- All type parameter bounds are satisfied, i.e. $\sigma L_i <: \sigma a_i$ and also $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.

[12] In fact, this particular case with the type not appearing at all might be changed in future versions of Coral to defer inference for it.

- The method's result type T' conforms to the expected type, i.e. $\sigma T' <: \sigma et$.
- Each argument type weakly conforms (§3.7.2) to the corresponding formal parameter type (§6.6.2), i.e. $\sigma S_j <:_w \sigma R_j$ for $j = 1, \ldots, m$.

If a type argument does not appear in the expected type and neither in the argument types, it stays defined with an inaccessible value *undefined*, and is possibly inferred in a future inference run, in case of curried functions. The type arguments that are inferred to be *undefined* are deferred and preserve their order in a virtual implicit type parameter list of the following parameter lists.[13]

It is an error if no such substitution exists. If several solutions exists, an optimal one for the type T' is chosen.

All or parts of an expected type *et* may be undefined. The rules for conformance (§3.7.2) are extended to this case by adding the rule that for any type T, the following two statements are always true:

- $\mathit{undefined} <: T$
- $T <: \mathit{undefined}$

It is possible that no minimal or maximal solution for a type variable exists, which is an error. Because "<:" is a partial order, it is also possible that a solution set has several optimal solutions for a type. In that case, Coral is free to pick any one of them.

Example 6.26.4 Consider the two methods, where `List` is covariant in its type parameter:

```
def cons [A] (x: A, xs: List[A]): List[A] := x ~> xs
def list_nil [B]: List[B] := List.Nil
```

and the definition

```
val xs := cons(1, list_nil)  .
```

The application of `cons` is typed with an undefined expected type. This application is completed by local type inference to `cons[Integer](1, list_nil)`. Here, one uses the following reasoning to infer the type argument `Integer` for the type parameter `A`:

First, the argument expressions are typed. The first argument `1` has type `Integer`, whereas the second argument `list_nil` is itself polymorphic. One tries to type `list_nil` with an expected type `List[A]`. This leads to the constraint system

[13] This is different from Scala, where Scala will infer all type arguments of the first parameter list, even if it would infer not very helpful types.

```
List[?B] <: List[A]  ,
```

where we have labeled `?B` with a question mark to indicate that it is a variable in the constraint system. Because class `List` is covariant, the optimal solution of this constraint is a minimal type,

```
B := Nothing  .
```

In the second step, one solves the following constraint system for the type parameter `A` of `cons`:

```
Integer <: ?A              // for the parameter x
List[Nothing] <: List[?A]  // for the parameter xs
List[?A] <: undefined       // for the result type
```

The optimal solution of this constraint system is

```
A := Integer  ,
```

so `Integer` is the type inferred for `A`.

The optimal solution is found based on the following reasoning:

1. Solutions for the first constraint are types `Integer` and its supertypes.
2. Solutions for the second constraint are `Nothing` and all supertypes of `Nothing`, basically any type. Together with the first constraint, only those types from the solution of the first constraint are possible.
3. Solutions for the third constraint are all types. Together with the first two constraint, only those types from the solution of the first constraint are possible.
4. Therefore, the minimal type is chosen from the solution space, as `A` appears covariantly (§4.6) in the type T of the expression (which is `List[A]`), which is in turn `Integer`.

Example 6.26.5 Consider now the definition

```
val ys := cons("abc", xs)  ,
```

where `xs` is defined as type `List[Integer]` as before. In this case local type inference proceeds as follows.

First, the argument expressions are typed. The first argument `"abc"` is of type `String`. The second argument `xs` is first tried to be typed with expected type `List[A]`. This leads to the constraint system

```
List[Integer] <: List[?A]
```

In a second step, one solves the following constraint system for the type parameter `A` of `cons`:

```
String <: ?A
List[Integer] <: List[?A]
List[?A] <: undefined
```

The optimal solution of this constraint system is

```
A := Object  ,
```

so `Object` is the type inferred for `A`.

6.26.5 Eta-Expansion

Eta-expansion converts an expression of a method type (not a function application) to an equivalent expression of a function type. It is especially useful to prevent re-evaluation of the expression's subexpressions, if the expression is passed using a by-name strategy. It proceeds in two steps.

First, one identifies the maximal subexpressions of e, let's say these are $e_1, \ldots, e_m$. For each of these, one creates a fresh name x_i. Let e' be the expression resulting from replacing every maximal subexpression e_i in e by the corresponding fresh name x_i. Second, one creates a fresh name y_i for every argument type T_i of the method, for $i = 1, \ldots, n$, using named arguments and parameters as defined by the method. The result of eta-conversion is then:

```
val x_1 := e_1
...
val x_m := e_m
(y_1: T_1, ..., y_n: T_n) -> e'(y_1, ..., y_n)
```

6.26.6 Dynamic Member Selection

Coral defines a marker trait `Dynamic_Member_Selecting` that enables dynamic invocations rewriting without resorting to use error handling mechanisms to implement dynamic dispatch.

Instances x of this trait allow method invocations `x.method(args)` for arbitrary method `method` and argument lists `args`, as well as property accesses `x.property` for arbitrary property names `property`.

If an invocation is not implemented by x (i.e. if type checking fails and no other implicit conversion provides a way to proceed with the invocation), it is virtually rewritten

according to the following rules:

```
foo.method("blah")      foo.apply_dynamic(:method)("blah")
foo.method(x: "blah")   foo.apply_dynamic_named(:method)((:x, "blah"))
foo.method(1, x: 2)     foo.apply_dynamic_named(:method)(
                            (nil, 1), (:x, "blah"))
foo.property            foo.select_dynamic(:property)
foo.property := 10      foo.update_dynamic(:property)(10)
foo.array(10)           foo.apply_dynamic(:array)(10)
foo.array(10) := 11     foo.select_dynamic(:array).update(10)(11)
```

6.27 Workflows

Workflows are basically a syntax sugar for a block expression that is passed as an argument to a workflow builder (an object that conforms to `Workflow_Builder`). In the following translation table, the left column presents the constructs, and the right column its de-sugared form, where *b* is the workflow builder. Note that it has to be statically known at compile time that an object is an instance of `Workflow_Builder` for this language feature to work properly.

```
b do cexpr end              b.delay(() -> { cexpr })
let pat := expr; cexpr      let pat := expr; cexpr
let! pat := expr; cexpr     b.bind(expr, (pat) -> { cexpr })
return expr                 b.return(expr)
return! expr                b.return_from(expr)
yield expr                  b.yield(expr)
yield! expr                 b.yield_from(expr)
for pat in expr             b.for(expr, (pat) -> { cexpr })
  loop cexpr end
while expr loop cexpr end   b.while(
                                () -> { expr },
                                b.delay(() -> { cexpr }))
if expr then cexpr1         if expr then cexpr1 else cexpr2 end
  else cexpr2 end
if expr then cexpr          if expr then cexpr else b.zero
cexpr1; cexpr2              b.combine(cexpr1, b.delay(() -> { cexpr2 }))
begin cexpr1                b.catch(expr, (v) -> { match v { cexpr2 } })
  catch cexpr2 end
begin cexpr1                b.ensure(cexpr1, () -> { cexpr2 })
  ensure cexpr2 end
```

6.28 Syntactic Forms

6.28.1 Quasi-quotation

Syntax:

```
Quasiquote_Expr ::= '`(' Any_Expr ')'
Any_Expr        ::= Expr
                  | Block_Stat {semi Block_Stat}
                  | Template_Stat {semi Template_Stat}
                  | Alias_Expr
                  | Compilation_Unit
                  | Top_Stat_Seq
Expr            ::= '#{' Expr '}'
```

Quasi-quote expression `` `( `` *e* `)` is basically a well-formed piece of Coral source code, wrapped in parentheses preceded by a backtick. The expression represents the compiled expression *e* in means of an AST node. Alternative way to define a quasi-quoted expression is with the annotation `@[quasiquote]`.

Quasi-quote expressions may optionally be interpolated, so that values or other nodes may be injected into the represented AST. There are the following ways to interpolate the AST:

- Interpolating expression `#{` *e* `}`, where *e*'s value is expanded into the AST as is. If the expression would require parentheses around it in the source, then parentheses have to be around the interpolating expression.[14]
- The `@[unquote]` annotation, which puts parentheses around the annotated expression.
- The `@[splice]` annotation, where the annotated expression is expanded into the AST as is.

A quasi-quote without any interpolation is equivalent to a quote (§6.28.2).

Quasi-quotes are useful in combination with macros (§10.3).

6.28.2 Quotation

A quote expression is basically a quasi-quote, but without any interpolation. The annotation that marks an expression for quotation is `@[quote]`.

[14]Note that the same delimiters are used to interpolate string literals.

Chapter 7

Implicit Parameters & Views

7.1 The Implicit Modifier

Syntax:

```
Local_Modifier ::= 'implicit'
Param_Clauses  ::= {Param_Clause} '(' 'implicit' Params ')'
```

Template members and parameters labeled with **implicit** modifier can be passed to implicit parameters (§7.2) and can be used as implicit conversions called views (§7.3).

If the member marked with **implicit** is a class, it makes the primary constructor of the class available for implicit conversions as a function. Such primary constructor may take exactly one non-implicit argument in its first parameter list.

Example 7.1.1 The following code defined an abstract class of monoids and two concrete implementations, `String_Monoid` and `Int_Monoid`. The two implementations are marked implicit and will be used throughout the following discussions.

```
abstract class Monoid [A] extends Semi_Group [A] {
  def unit: A end
  def add (x: A, y: A): A end
}
object Monoids {
  implicit object String_Monoid extends Monoid[String] {
    def unit: String := ""
    def add (x: String, y: String): String := x + y
  }
  implicit object Int_Monoid extends Monoid[Integer] {
    def unit: Integer := 0
    def add (x: Integer, y: Integer): Integer := x + y
```

```
    }
  }
```

7.2 Implicit Parameters

An implicit parameter list (**implicit** $p_1, \ldots, p_n$) of a method marks the parameters $p_1, \ldots, p_n$ as implicit. A method or constructor can have at most one implicit parameter list, and it must be the last parameter list given.

A method with implicit parameters can be applied to arguments just like normal method. In this case the **implicit** label has no effect. However, if such a method misses arguments for its implicit parameters (determined by a missing consecutive function application – §6.6.8), such arguments will be automatically provided, if possible.

The actual arguments that are eligible to be passed to an implicit parameter of type T fall into two categories.

- First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition (§7.1) or an implicit parameter. An eligible identifier may thus be a local name, or a member of an enclosing template, or it may have been made accessible without a prefix through a use clause (§4.10).
- If there are no eligible identifiers under the previous rule, then, second, eligible are also all **implicit** members of some object that belongs to the implicit scope of the implicit parameter's type, T.

The *implicit scope* of a type T consists of the type T itself and all classes and class objects that are associated with the type T. Here, we say a class C is *associated* with a type T, if it is a base class of some part of T. The *parts* of a type T are:

- If T is a compound type T_1 **with** ... **with** T_n, the union of the parts of $T_1, \ldots, T_n$, as well as T itself.
- If T is a parameterized type $S[T_1, \ldots, T_n]$, the union of the parts of S and $T_1, \ldots, T_n$.
- If T is a singleton type p**.type**, the parts of the type of p.
- If T is a type projection $S\#U$, the parts of S as well as T itself.
- If T is a union type **union of** (T_1**;** ...**;** T_n), the union of all types $T_1, \ldots, T_n$.

- In all other cases, just T itself.

If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of overloading resolution without any application (§6.26.3). If the parameter has a default argument and no implicit argument can be found, the default argument is used.

Example 7.2.1 Assuming the classes from Example 7.1.1, here is a method which computes the sum of a list of elements using the monoid's `add` and `unit` operations.

```
def sum [A] (xs: List[A])(implicit m: Monoid[A]): A
  if xs.is_empty?
    m.unit
  else
    m.add xs.head, sum xs.tail
  end
end
```

The monoid in question is marked as an implicit parameter, and can therefore be inferred based on the type of the list. Consider e.g. the call

```
sum %[1, 2, 3]
```

in a context where `String_Monoid` and `Int_Monoid` are visible. We know that the formal type parameter `A` of `sum` needs to be instantiated to `Integer`. The only eligible object which matches the implicit formal parameter type `Monoid[Integer]` is `Int_Monoid`, thus this object will be passed as implicit parameter.

7.3 Views

Implicit parameters and method can also define implicit conversions called *views*. A *view* from type S to type T is defined by an implicit value, which has function type S `->` T, or `(()` `->` S`)` `->` T, or by a method convertible to a value of one of those function types.

Views are applied in the following situations.

1. If an expression e is of a type T, and T does not conform to the expression's expected type *et*. In this case an implicit v is searched, which is applicable to e and whose result type conforms to *et*, and this process makes T compatible with *et*. The search proceeds as in the case of implicit parameters, where the implicit scopes are the scope of T followed by the scope of *et*, searched in this order. If such a view is found, the expression e is converted to v`(`e`)`.

2. In a selection $e.m$ with e of a type T, if the selector m does not denote an accessible member of T, including restrictions imposed by access modifiers, i.e. the member m may actually exist in T. In this case, a view v is searched which is applicable to e and whose result contains an accessible member named m. The search proceeds as in the case of implicit parameters, where the implicit scope is just that of T. If such a view is found, the selection $e.m$ is converted to $v(e).m$.

3. In a selection $e.m(args)$ with e of a type T, if the selector m denotes some members of T, but none of these members is applicable to the arguments *args*. In this case a view v is searched, which is applicable to e and whose result contains a member m, which is applicable to *args*. The search proceeds as in the case of implicit parameters, where the implicit scope is just that of T. If such a view is found, the selection $e.m(args)$ is converted to $v(e).m(args)$.

The implicit view, if it is found, can accept its argument e as a a call-by-value based or call-by-name based parameter, and no precedence is imposed on the views. It is an error if there are multiple equally specific views that differ only in the parameter evaluation strategy.

As for implicit parameters, overloading resolution (§6.26.3) is applied if there are several possible candidates.[1]

7.4 View & Context Bounds

Syntax:

```
Type_Param ::= (id | '_') [Type_Param_Clause]
               ['>:' Type] ['<:' Type]
               {'<%' Type} {':' Type}
             | '<' (id | '_') '>' ['<:' id]
```

A type parameter A of a method or a non-trait class may have one or more view bounds A `<%` T. In this case, the type parameter may be instantiated to any type S, which is convertible by an application of a view to the bound T.

A type parameter A of a method or a non-trait class may have one or more view bounds A `:` T. In this case, the type parameter may be instantiated to any type S for which *evidence* exists at the instantiation point that S satisfies the bound T. Such evidence consists of an implicit value with type T`[`S`]`.

A method or class containing type parameters with view or context bounds is treated as being equivalent to a method with implicit parameters, and if it already contains

[1]The overloading resolution here is for the case of function applications, and the shape takes into account just one argument and corresponding parameter pair.

explicitly some implicit parameters, those are added right after the section of any positional parameters and before the section of any named parameters. Consider the case of a single parameter with a view and/or context bounds, such as:

```
def f [A <% T_1 ... <% T_m : U_1 : U_n] (ps): R := ...
```

Then the method definition above is equivalent to

```
def f [A] (ps)(implicit v_1: A -> T_1, ..., v_m: A -> T_m,
                        w_1: U_1[A], ..., w_n: U_n[A]): R := ...
```

where the v_i and w_j are fresh names for the newly introduced implicit parameters. These parameters are called *evidence parameters*.

If a class or method has several view- or context-bounded type parameters, each such type parameter is expanded into evidence parameters in the order they appear and all the resulting evidence parameters are concatenated in one implicit parameter section. Since traits do not have constructor parameters, such translation is impossible for them.

If the type of a context-bound parameter uses the type parameter in its definition, then the translation is simple – it stays the same and the type argument is still possibly inferred. Such type is then called a *partially applied type*, although all type arguments are necessarily provided in the end. It is an error if the type is a parameterized type and it does not use the type parameter that it is context-bound with. Thus, a context bound `A : T` is in fact a shortcut for `A : T[A]`.

Example 7.4.1 The following example shows a method with a context-bound parameter using a partially applied type, and the resulting translation below it.

```
def f [T : T -> String](t: T) := ...
def f [T] (t: T)(implicit w_1: Function_1[T, String]) := ...
```

Chapter 8

Pattern Matching

8.1 Patterns

Syntax:

```
Pattern        ::= Pattern1 {'|' Pattern1}
Pattern1       ::= ['implicit'] var_id ':' Type_Pat
                 | '_' ':' Type_Pat
                 | Pattern2
Pattern2       ::= ['implicit'] var_id ['@' Simple_Pattern]
                 | Simple_Pattern
Simple_Pattern ::= ['∗'] '_'
                 | ['∗'] var_id
                 | Literal
                 | Stable_Id
                 | Stable_Id '(' [Patterns] ')'
                 | '(' [Patterns] ')'
                 | Pattern '&' Pattern
                 | Pattern '~>' Pattern
                 | Pattern '<~' Pattern
                 | Pattern ('..' | '...') Pattern
Patterns       ::= Pattern {',' Pattern}
```

8.1.1 Variable Patterns

Syntax:

```
Simple_Pattern ::= '_'
                 | var_id
```

A variable pattern x is a simple identifier which starts with a lower case letter. It matches any value and binds the variable name to that value. The type of x is the expected type of the pattern as given from the outside. A special case is the wildcard pattern "_", which is treated as if it was a fresh variable on each occurence, and which does not bind itself to the value, i.e., it is alone equivalent to the **else** clause of `When_Clauses`.

8.1.2 Typed Patterns

Syntax:

```
Simple_Pattern ::= '_' ':' Type_Pat
                 | var_id ':' Type_Pat
Type_Pat       ::= Type
```

A typed pattern x`:` T consists of a pattern variable x and a type pattern T. The type of x is the type T, where each type variable and wildcard is replaced by a fresh, unknown type. This pattern matches any value matched by the type pattern T (§8.2), and it binds the variable name to that value (unless the variable name is "_").

8.1.3 Pattern Binders

Syntax:

```
Pattern2 ::= var_id '@' Simple_Pattern
```

A pattern binder x `@` p consists of a pattern variable x and a pattern p. The type of the variable x is the type T resulting from the pattern p. This pattern matches any value v matched by the pattern p, provided the type of v is also an instance of T, and it binds the variable name to that value.

Example 8.1.1 In the following example, `person` binds to the whole `Person` object.

```
def f (someone: Person) := match someone
  when person @ Person('John Galt', _, _) then ...
end match
```

8.1.4 Literal Patterns

Syntax:

```
Simple_Pattern ::= Literal
```

A literal pattern L matches any value that is equal (in terms of =) to the literal L. The type of L must conform to the expected type of the pattern. Literal kinds that are considered legal with this pattern are: string literals, number literals, boolean literals and the **nil** value.

8.1.5 Stable Identifier Patterns

Syntax:

```
Simple_Pattern ::= Stable_Id
```

A stable identifier pattern is a stable identifier r (§3.2). The type of r must conform to the expected type of the pattern. The pattern matches any value v, such that $r = v$.

To resolve the syntactic overlap with a variable pattern (§8.1.1), a stable identifier pattern may not be a simple name starting with a lower case letter. However, it is possible to enclose such a variable or method name in backquotes, then it is treated as a stable identifier pattern.

Example 8.1.2 Consider the following function definition:

```
def f (x: Integer, y: Integer) := match x
  when y then ...
end match
```

Here, `y` is a variable pattern, which matches any value, namely here it would bind simply to `x`. If we wanted to turn the pattern into a stable identifier pattern, this can be achieved as follows:

```
def f (x: Integer, y: Integer) := match x
  when `y` then ...
end match
```

Now, the pattern matches the `y` parameter of the enclosing function `f`. That is, the match succeeds only if the `x` argument and the `y` argument of `f` are equal.

8.1.6 Constructor Patterns

Syntax:

```
Simple_Pattern ::= Stable_Id '(' [Patterns] ')'
```

A constructor pattern is of the form $c(p_1, \ldots, p_n)$, for $n \geq 0$. It consists of a stable identifier c, followed by element patterns $p_1, \ldots, p_n$. The constructor c is a simple or qualified name which denotes a case class (§5.3.3). If the case class is monomorphic,

then it must conform to the expected type of the pattern, and the formal parameter types of c's primary constructor (§5.3.1) are taken as the expected types of the element patterns $p_1, \ldots, p_n$. If the case class is polymorphic, then its type parameters are instantiated so that the instantiation of c conforms to the expected type of the pattern, unless the type arguments are already given. These types of the formal parameter types of c's primary constructor are then taken as the expected types of the component patterns $p_1, \ldots, p_n$. The pattern matches all objects created from constructor invocations $c(p_1, \ldots, p_n)$, where each element pattern p_i matches the corresponding value v_i. Any extra parameter sections of c's primary constructor do not affect this behavior.

A special case arises when c's formal parameter types contain a repeated parameter. This is further discussed in (§8.1.9).

8.1.7 Tuple Patterns

Syntax:

```
Simple_Pattern ::= '(' [Patterns] ')'
```

A tuple pattern $(p_1, \ldots, p_n)$ is an alias for the constructor pattern `Tuple_`$n(p_1, \ldots, p_n)$, where $n \geq 2$. The empty tuple `()` is the unique value of type `Unit`.

8.1.8 Extractor Patterns

Syntax:

```
Simple_Pattern ::= Stable_Id '(' [Patterns] ')'
```

An extractor pattern $x(p_1, \ldots, p_n)$, where $n \geq 0$, is of the same syntactic form as a constructor pattern. However, instead of a case class, the stable identifier x denotes an object which has a member method named `unapply` or `unapply_sequence` that matches the pattern.

An `unapply` method in an object x *matches* the pattern $x(p_1, \ldots, p_n)$ if it takes exactly one argument and one of the following applies:

> $n = 0$ and `unapply`'s result type is `Boolean`. In this case the extractor pattern matches all values v for which x`.unapply(`v`)` returns **`yes`**.
>
> $n = 1$ and `unapply`'s result type is `Option[`T`]`, for some type T. In this case, the only argument pattern p_1 is typed in turn with expected type T. The extractor pattern matches then all values v for which x`.unapply(`v`)` returns a value of form `Some(`v_1`)`, and p_1 matches v_1.

$n > 1$ and `unapply`'s result type is `Option[(`$T_1, \ldots, T_n$`)]`, for some types $T_1, \ldots, T_n$. In this case, the argument patterns $p_1, \ldots, p_n$ are typed in turn with expected types $T_1, \ldots, T_n$. The extractor pattern matches then all values v for which x`.unapply(`v`)` returns a value of form `Some((`$v_1, \ldots, v_n$`))`, and each pattern p_i matches the corresponding value v_i.

An `unapply_sequence` method in an object x matches the pattern $x(p_1, \ldots, p_n)$, if it takes exactly one argument and its result type is of the form `Option[`S`]`, where S is a subtype of `Sequence[`T`]` for some element type T. This case is further discussed in (§8.1.9).

8.1.9 Pattern Sequences

Syntax:

```
Simple_Pattern ::= Stable_Id '(' [Patterns ',']
                   '*' (var_id | '_')
                   [',' Patterns] ')'
```

A pattern sequence $p_1, \ldots, p_n$ appears in two contexts. First, in a constructor pattern $c(q_1, \ldots, q_a, p_1, \ldots, p_n, r_1, \ldots, r_b)$, where c is a case class, which has $a + 1 + b$ primary constructor parameters, with a repeated parameter (§4.7.9) of type $*S$ in the middle. Second, in an extractor pattern $x(p_1, \ldots, p_n)$, if the extractor object x has an `unapply_sequence` method with a result type conforming to `Sequence[`T`]`, but does not have an `unapply` method that matches $p_1, \ldots, p_n$. The expected type for the pattern sequence is in each case the type S.

8.1.10 Conjunction Patterns

Syntax:

```
Simple_Pattern ::= Pattern '&' Pattern
```

A conjunction pattern ("and" *pattern*) matches only if both patterns match. Moreover, only the first pattern in a sequence of conjunction patterns may bind variable names, but variables from the other patterns have their scope extended to the following patterns. This behavior is unlike in pattern alternatives, which aren't allowed to bind variable names at all, due to the fact that each alternative may match a completely different structure, whereas a conjunction pattern matches on the same structure.

8.1.11 List Patterns

Syntax:

```
Simple_Pattern ::= Pattern '~>' Pattern
                 | Pattern '<~' Pattern
```

A list pattern[1] is a pattern specialized to match elements of list instances. A list pattern of the form p_1 `~>` p_2 matches a list, where p_1 matches the head element of the list and p_2 matches the remainder of the list, ending with the tail element. A list pattern of the form p_1 `<~` p_2 matches a list, where p_1 matches the tail element of the list and p_2 matches the remained of the list, starting with the head element. If the list has one element, then the matched remainder is **`nil`**.

8.1.12 Range Patterns

Syntax:

```
Simple_Pattern ::= Pattern ('..' | '...') Pattern
```

A range pattern is a pattern specialized to match scalar values with ordering.

8.1.13 Pattern Alternatives

Syntax:

```
Pattern ::= Pattern {'|' Pattern}
```

A pattern alternative p_1 `|` ... `|` p_n, where $n \geq 2$, consists of a number of alternative patterns p_i. All alternative patterns are type checked with the expected type of the pattern. They may not bind variable names other than wildcards (which are discarded). The alternative pattern matches a value v if at least one of its alternatives matches v. Consequently, if a first such match is successful, the remaining patterns are not tested.

Non-normatively: Usually, each pattern alternative is a literal pattern, and therefore binding a variable name makes no sense, since the value that the pattern matching expression matches against is the already bound variable. Still, if needed, the pattern alternatives may be used together with a pattern binder (§8.1.3), which can be useful when the structure that is being matched contains union types.

8.1.14 Regular Expression Patterns

Syntax:

```
Simple_Pattern ::= regexp_literal
```

[1] This is in fact a simplified form of an infix pattern, which is a feature that might be enabled in future versions.

A regular expression pattern p (*regexp pattern*) is a variant of literal pattern, designed to match `String_Like` values. Literally, the pattern p matches a value v, if v is of a type that conforms to `String_Like` and its contents match the regular expression. Moreover, sub-patterns bind to variable names of a name of the form `match_`n, where n is either the position of the sub-pattern (unless the sub-pattern is explicitly not captured), or the name of a named sub-pattern.

Regular expression patterns are not designed to match against algebraic data structures.

8.1.15 Irrefutable Patterns

A patter p is *irrefutable* for a type T, if one of the following applies:

1. p is a variable pattern (§8.1.1),
2. p is a typed pattern x`:` T' (§8.1.2), and T `<:` T',
3. p is a constructor pattern $c(p_1 \ldots p_n)$ (§8.1.6), the type T is an instance of a class c, the primary constructor (§5.3) of type T has argument types $T_1, \ldots, T_n$, and each p_i is irrefutable for type T_i.

8.2 Type Patterns

Syntax:

```
Type_Pat ::= Type
```

Type patterns consist of types, type variables and wildcards. A type pattern T is of one of the following forms:

A reference to a class C, p`.`C, p`.type` or T`#`C. This type pattern matches any non-**`nil`** instance of the given class (therefore, it does match the empty tuple `()` with type `Unit`). Note that the prefix of the class, if it is given, is irrelevant for determining class instances, unlike in Scala.

The bottom type `Nothing` (with singleton instance **`nil`**) is the only type pattern that matches **`nil`** (only), but its preferable to match against `Option[`T`]` with implicit conversion of **`nil`** to object `None`.

A singleton type p`.type`. This type pattern matches only the value denoted by the path p (only the single value denoted by the path p, since **`nil`** is not matched).

A parameterized type pattern $T[a_1 \ldots a_n]$ [<$t_1 \ldots t_n$>], where the a_i are type variable patterns or wildcards "_" and u_i are unit of measure kinds. This type pattern matches all values which match T for some arbitrary instantiation of the type variables and wildcards.

A compound type pattern T_1 **with** ... **with** T_n, where each T_i is a type pattern. This type pattern matches all values that are matched by each of the type patterns T_i, and in this sense it is equivalent to the pattern T_1 & ... & T_n.

Types are not subject to any type erasure (§3.8), so it is basically safe to use any other type as type pattern, unlike in Scala.

A *type variable pattern* is a simple identifier which starts with a lower case letter.

8.3 Pattern Matching Expressions

Syntax:

```
Match_Expr     ::= Pat_Match_Expr | Case_Expr
Pat_Match_Expr ::= 'match' Simple_Expr1 Match_Body
Match_Body     ::= semi When_Clauses 'end' ['match']
                 | '{' When_Clauses '}'
When_Clauses   ::= When_Clause {['next'] semi When_Clause}
                   [['next'] 'else' Block]
When_Clause    ::= 'when' Pattern [Guard] ('then' | semi) Block
```

A pattern matching expression

match e { **when** p_1 **then** b_1 ... **when** p_n **then** b_n **else** b_{n+1} }

consists of a selector expression e and a number $n > 0$ of cases. Each case consists of a (possibly guarded) pattern p_i, a block b_i and optionally the default block b_{n+1}, if none of the patterns matched. Each p_i might be complemented by a guard **if** e or **unless** e, where e is a guarding expression, that is typed as `Boolean`. The scope of the pattern variables in p_i comprises the pattern's guard and the corresponding block b_i. If the following when clause **when** p_{i+1} **then** b_{i+1} is preceded by the keyword **next**, then the pattern variables in p_i do not comprise the block b_{i+1} and neither the pattern p_{i+1}.

Let T be the type of the selector expression e. Every pattern $e \in \{p_1 \ldots p_n\}$ is typed with T as its expected type.

The expected type of every block b_i is the expected type of the whole pattern matching expression. The type of the pattern matching expression is then the weak least upper bound (§3.7.3) of the types of all blocks b_i.

When applying a pattern matching expression to a selector value, patterns are tried in given order, until one is found that matches the selector value. Say this **when** clause is **when** p_i **then** b_i. The result of the whole expression is then the result of evaluating b_i, where all pattern variables of p_i are bound to the corresponding parts of the selector value. If no matching pattern is found, a `No_Match_Error` is raised.

The pattern in a **when** clause may also be followed by a guard suffix **if** e with a boolean expression e. The guard expression is evaluated if the preceding pattern in the case matches. If the guard expression evaluates to **yes**, the pattern match succeeds as normal. If the guard expression evaluates to **yes**, the pattern in the case is considered not to match and the search for a matching pattern continues.

The pattern in a case may also be followed by a guard suffix **unless** e with a boolean expression e. The guard expression is evaluated as if it was **if** `!`e.

In the interest of efficiency the evaluation of a pattern matching expression may try patterns in some other order than the textual sequence, even parallelized (indeed, compiler would not decide this on its own – it has to be specified with an annotation or a pragma (§10) applied to the pattern matching expression). This might affect evaluation through side effects in guards. However, it is guaranteed that a guard expression is evaluated only if the pattern it guards matches.

If the selector of a pattern match is an instance of a **sealed** class (§5.2), the compilation of the pattern matching expression can emit warnings, which diagnose that a given set of patterns is not exhaustive, i.e. there is a possibility of a `No_Match_Error` being raised at runtime.

8.4 Pattern Matching Anonymous Functions

Syntax:

```
Block_Expr ::= '{' When_Clauses '}'
```

An anonymous function can be defined by a sequence of cases

`{` **when** p_1 **then** b_1 ... **when** p_k **then** b_k **else** b_{k+1} `}`

which appears as an expression without a prior **match**. The expression is expected to be of a block type, unless it is expected to be a function type, in that case it is converted to `Function_`k`[`$S_1, \ldots, S_k$`,` T`]` automatically for $k \geq 1$, and for $k = 0$ the expression does not match the expected function type in function applications (§6.6).[2]

The expression is taken to be equivalent to the anonymous function:

[2]This is due to Coral not knowing the expected types in advance, but this anonymous function expression is able to match any non-empty arguments list, which is simply passed into the implicit pattern matching expression.

```
(x_1: S_1, ..., x_k: S_k) -> {
  match (x_1, ..., x_k) {
    when p_1 then b_1
    ...
    when p_k then b_k
    else b_{k+1}
  }
}
```

Here, each x_i is a fresh name. As was shown in (§6.23), this anonymous function is in turn equivalent to the following instance creation expression, where T is the weak least upper bound of the types of all b_i.

```
(Function_k[S_1, ..., S_k, T] with {
  def apply (x_1: S_1, ..., x_k: S_k): T := match (x_1, ..., x_k)
    when p_1 then b_1
    ...
    when p_k then b_k
    else b_{k+1}
  end match
}).new
```

Example 8.4.1 Here is a method which uses a fold-left operation `/:` to compute the scalar product of two vectors:

```
def scalar_product (xs: List[Double], ys: List[Double]) :=
  (0.0 /: xs.zip(ys)) {
    when (a, (b, c)) then a + b * c
  }
```

The when clauses in this code are equivalent to the following anonymous function:

```
(x, y) -> {
  match (x, y) {
    when (a, (b, c)) then a + b * c
  }
}
```

Note that the fold-left operation `/:` is an operator ending in a colon "`:`", and therefore right-associative, and therefore the expression is interpreted as specified in (§6.12.3) for right-associative operations: `{ val x$ := 0.0; xs.zip(ys).`/:`(x) }`.

Chapter 9

Top-Level Definitions

9.1 Compilation Units

9.1.1 Modules

Syntax:

```
Compilation_Unit ::= {'module' Module_Path semi [Top_Stat_Seq]}
                     Top_Stat_Seq
Top_Stat_Seq     ::= Top_Stat {semi Top_Stat}
Top_Stat         ::= {Annotation} {Modifier} Tmpl_Def
                   | Use
                   | Packaging
                   | Module_Object
                   | Expr
                   | ()
```

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).

A compilation unit (a single source file) consists of a sequence of packagings, import clauses, and class and object definitions, which may be preceded (and should be preceded) by a module clause.

A compilation unit

module p_1
…
module p_n
stats

starting with one or more module clauses is equivalent to a compilation unit consisting of the packaging

module p_1
  …
  module p_n
    stats
  end module
end module

Implicitly imported into every compilation unit are, in that order:

1. the module `Root~Lang~[coral]`
2. the object `Root~Lang~[coral].Predef`

Members of a later import in that order hide members of an earlier import.

The implicitly added code looks like the following code listing, with all its implications:[1]

```
use Root~Lang~[coral].{_}
use Root~Lang~[coral].Predef.{_}
```

9.1.2 Packagings

Syntax:

```
Packaging  ::= 'module' Module_Path (Packaging1 | Packaging2)
Packaging1 ::= semi Top_Stat_Seq 'end' ['module']
Packaging2 ::= '{' Top_Stat_Seq '}'
```

A module is a special object which defines a set of member classes, objects and another modules. Like open templates (§5.1.1), modules are introduced by multiple definitions across multiple source files.

A packaging **module** p { *stats* } or **module** p~[v] *stats* **end** injects all definitions in *stats*as members into the module whose qualified name is p. Members of a module

[1]The `Root` is actually redundant, as explained in (§9.1.4).

are called *top-level* definitions. If a definition in *stats*is labeled **`private`**, it is visible only for other members in the same module.

Inside the packaging, all members of package p are visible under their simple names. This rule extends to members of the enclosing modules of p that are of the same *vendor*. However, every other module needs to either import the members with a use clause (§4.10), or refer to it via its fully qualified name.

The special `Root` "module" can only be specified as the first element of each packaging name.

Example 9.1.1 Given the packagings

```
module A~[org.net] {
  module B~[org.net] {
    ...
  }
  module B~[org.net.prj] {
    ...
  }
}
module C~[org.net] {}
module D~[org.net.prj] {}
```

all members of the module `B~[org.net]` are visible under their simple names to the modules `B~[org.net]` and `A~[org.net]`, but not the others: module `C` has the same vendor, but is located outside of the packaging of module `B~[org.net]`, and module `D` is completely out of the packaging game.[2] All members of the module `B~[org.net.prj]` are visible under their simple names to the module `B~[org.net.prj]`, but not the other modules. Since the module `B~[org.net.prj]` is nested in the module `A~[org.net]`, all of its members are not visible to members of a potential module `A~[org.net.prj]` , since it is not nested in it.

The fully qualified names of these modules are as follows:

- `A~[org.net]`
- `A~[org.net].B~[org.net]`, same as `A~[org.net].B` (§9.1.4)
- `A~[org.net].B~[org.net.prj]`
- `C~[org.net]`
- `D~[org.net.prj]`

[2]The packaging game is too strong for the module `D`.

Notice how these fully qualified names do not use the `Root~` path prefix, explained in (§9.1.4).

Selections *p*`.`*m* from *p* as well as imports from *p* work as for objects. Moreover, unlike in Scala, modules may be used as values, instances of `Module` class, which shares some behavior with `Class` class. It is illegal to have a module with the same fully qualified name (minus the vendor parts) as a class or a trait.

Top-level definitions outside a packaging are assumed to be injected into the `Object` class directly, and therefore visible to each other without qualification. However, as `Object` is actually a simple name for the fully qualified name `Root~Lang~[coral].Object`, no member is ever defined outside of packaging – it may only seem to be so: the type of **`self`** pseudo-variable in "global" context (outside of any packagings) is `Root~Lang~[coral].Object`—a special instance of `Object` dedicated to handling "global" space—unless the source file is loaded in context of another instance, used with DSLs.

9.1.3 Module Object

Syntax:

```
Module_Object ::= 'module' 'object' Trait_Tmpl_Env
```

A module object **`module object`** *p* **`extends`** *t* can specify some properties of the module object, add new traits to it, and adds the members of the template *t* to the module object *p*. There can be only one module object per module, but the module object definition is still an open template (§5.1.1). The module object has to have the leading template defined in a file named *p*`.coral` in the module's root directory, otherwise the module object has an implied empty template.

The module object should not define a member with the same name as one of the top-level objects or classes defined in module *p*. If there is a name conflict, it is an error.

The module object has also a special role in defining entry points of the module. An entry point is a method of the module object that can be invoked from the outside, to actually run the module as a program (§9.2).

9.1.4 Module References

Syntax:

```
Path ::= Module_Path
```

A reference to a module takes the form of a qualified identifier. Like all other references, module references are relative, i.e., a module reference starting in a name *p* will be

looked up in the closest enclosing scope that defines a member named p, and continue with the next closest enclosing scope, as long as the modules share the exact same vendor.

The special predefined name `Root` (which is vendor-less and separated from the other names with a "~" instead of a "`.`") refers to the outermost "root module", which contains all top-level modules.

A nested module inherits the vendor from its directly enclosing module, unless there is no enclosing module, or the vendor is specified explicitly. If a module inherits the vendor, then it doesn't need to specify it again in its fully qualified identifier, as it is implied that nested modules will have the same vendor.

Example 9.1.2 Consider the following program:

```
module B {
  class C {}
}
module A.B {
  class D {
    val x := Root~B.C.new
  }
}
```

Here, the reference `Root~B.C` refers to class `B` in the top-level module `B`. If the `Root~` prefix had been omitted, the name `B` would instead resolve to the module `A.B`, and, provided that the module does not also contain a class `C`, a runtime error would result (constant not found).

9.2 Programs

A *program* is a module that has 1 or more entry points. An entry point is a method of the module object that can be invoked from the outside, to actually run the module as a program. To mark a method as an explicit entry point, use the `entry` pseudo-modifier[3] before the method definition or declaration. An implicit entry point is a method with a name `main` and a method type `(Sequence[String]) ↦ Unit`. A module does not need to have any entry points at all – that renders it a "library-only" module.

Example 9.2.1 The following example will create a hello world program by defining a module entry point in module `Test`.

Syntax:

[3]Actually, it is implemented as a method of the class `Module`, therefore not a keyword, but IDEs may opt-in to highlight it as such, due to its importance.

```
module Test ~[com.example]
module object {
  entry def main (args: Sequence[String]) := {
    Console.print_line "Hello world!"
  }
}
```

This program can be started by the command

```
% coral Test
```

Chapter 10

Annotations, Pragmas & Macros

Syntax:

```
Annotation     ::= '@[' Simple_Type [NB_Arg_Exprs] ']'
NB_Arg_Exprs   ::= Parens_Args {Parens_Args}
                 | Poetry_Args
Annotation_Def ::= 'annotation' Class_Def
Expr           ::= Pragma
Pragma         ::= 'pragma' Simple_Type [NB_Arg_Exprs]
Def            ::= 'def' 'macro' Fun_Def 'end' ['def']
                 | 'def' 'macro' Fun_Alt_Def
```

Annotations, pragmas & macros are a way to provide metadata to both the compiler and runtime of Coral, possibly affecting the resulting bytecode and abstract syntax trees.

10.1 Annotations

Annotations are classes that must conform to `Annotation`, and which can thus be applied in annotated expressions and annotated definitions or types.

Annotations always appear before element that they annotate, and if multiple annotations are applied, their order is preserved in respect of reflection, although the actual order may or may not matter.

10.2 Pragmas

Pragmas are basically annotations that are applied in its scope from that point on and in any nested scopes, not binding to just a single expression, definition or a type. Some annotations can only be applied as a pragma.

10.3 Macros

Macros are a way to directly manipulate with abstract syntax trees. Unlike in languages such as C, macros in Coral are written using the same language. The only essential restriction here is that while compiling a Coral module or another source file, every macro that is applied in it must be pre-compiled, e.g. available from a separate compilation phase, and applied in the same compilation phase, not runtime. The only way to apply macros in runtime is by ad-hoc compilation.

Macro authors are encouraged to use syntactic forms (§6.28) to manipulate and generate abstract syntax trees.

A macro is defined as a regular function, but it's body is required to pass invocation to a method that implements the macro body, where every parameter type T is replaced with `c.Expr[`T`]`, and where `c` is the first parameter of type `Context`. The macro implementation has exactly two parameter lists:

1. A parameter list with exactly one parameter of type either
   - `Lang~[coral].Reflection.Macros.Whitebox.Context` or
   - `Lang~[coral].Reflection.Macros.Blackbox.Context`.
2. A parameter list with the parameters of the macro definitions, with the described parameter type translation applied.

Example 10.3.1 The following code is an example implementation of a simple `assert` macro.

```
// import a blackbox context
use Lang~[coral].Reflection.Macros.Blackbox.Context

// define the macro
def macro assert (condition: Boolean, message: String_Like): Unit :=
    Asserts.assert_impl

// implement the macro
object Asserts {
  def assert_impl
      (c: Context)
      (cond: c.Expr[Boolean], msg: c.Expr[String_Like]): Unit :=
    `(
      if !#{cond}
        raise #{msg}
      else
        ()
      end
```

```
    )
}
```

Macros can be applied as an intermediate step by IDEs, so that their resulting transformations could be viewed prior to proceeding with compilation.

10.3.1 Whitebox & Blackbox Macros

Blackbox macros are such macros that exactly follow their type signature, including the result type, and therefore can be treated as blackboxes. Their implementations are irrelevant to understanding their behavior. On the otherhand, *whitebox macros* do not necessarily follow their type signature, which they have, but only as an approximation, which may or may not be precise. Therefore, whitebox macros may be used to create e.g. type providers, fundep materialization or extractor macros.

Blackbox macros have the following restrictions applied to them:

1. As an application of a blackbox macro expands into a tree x, the expansion is wrapped in a typed expression `(`x `as` T`)`, where T is the declared result type of the blackbox macro with type arguments and path dependencies applied in consistency with the particular macro application being expanded. This invalidates blackbox macros as a possible implementation of type providers.

2. When an application of a blackbox macro still has undetermined type parameters even after type inference, these type parameters are forcedly inferred, in the same manner as type inference works for normal methods. This invalidates blackbox macros from creating fundep materialization. On the contrary, whitebox macros defer type inference of undetermined type parameters until the macro application is expanded.

3. When an application of a blackbox macro is used as an implicit candidate, no expansion is performed until the particular macro is finally selected as the result of the search for an implicit.

4. It is an error if a blackbox macro is used as an extractor in pattern match.

10.3.2 Macro Annotations

Macro annotations are a combination of macros and annotations – such that a macro annotation is basically an annotation that defines a method `apply_macro` with a single macro definition, where the definition has exactly one parameter list with exactly one parameter of a reflection type that the macro annotation can be applied to. The same annotation may also define the implementation method of the macro, but that is not required. The macro definition has to return a single value, or multiple values via a

tuple. The single value may be `Unit`, therefore effectively discarding the annotated expression or definition.

Macro annotations are the most suitable way to create type providers.

Note. With macro annotations, the order of appearance of the annotations is inherently important, since the reflection type passed to the macro's implementation could be different with each different order of macro annotation applications.

Chapter 11

Design Guidelines

This chapter is *non-normative*, but it should be followed nonetheless.

11.1 Userland Naming Guide

The naming guidelines are particularly inspired by Ada. The purpose of the decisions made here is to improve readability over time required to write programs, as developers tend to spend a lot more time staring into source code and trying to understand it than actually writing it.

The general rule of thumb is to use full words separated by underscores, or if the special character is more suitable for the purpose, with that character.

Example 11.1.1 Examples of the naming recommended for Coral:

```
Some_Class_Name
E-Mail
do_something_with_something
```

Another recommended approach is to use names as long as necessary, but not longer; and if a shortcut for the name exists that is widely recognized, that shortcut may be used as an alias.

The `camelCase` is deprecated, as it worsens readability and in some cases even yields misleading names, such as `withOut` method, which reads as "without", but in fact means "with 'out'", where the name recommended by Coral, `with_out`, makes the intention way more clear. Camel case should be used iff the name that it represents itself is known in that form and placing an underscore in between would actually make it represent something else.

For methods that return boolean values, the rule is to name them with one of the following patterns:

`is_`*something*`?`
`has_`*something*`?`
something`?`

The latter case is especially better for cases where the method accepts parameters, the former two for parameterless methods.

For methods that modify the receiver, raise exceptions or errors, or do other potentially dangerous operations, the rule is to name them with the following pattern:

something`!`

Class names and object names should start with an upper-case letter, unless it makes sense to name it otherwise. Method and variable names should start with a lower-case letter.

Characters such as "+" can be used in names to replace words like "and", although such name inherently suggests possible problems with the named entity.

11.2 Program Parentheses

Coral supports in many syntactical cases two options for parentheses – symbol based (e.g. "`(`" and "`)`") and word based (e.g. **`do`** and **`end`**). The general rule of thumb here is to use symbol-delimited scopes only inside of word-delimited scopes, preferably for shorter definitions and procedures, or outside of any word-delimited scopes. Reversely, word-delimited scopes should not appear inside of symbol-delimited scopes, unless there is no other option.

Chapter 12

The Coral Standard Library

The Coral Standard Library (CSL[1]) consists of the module `Lang~[coral]` with a number of classes and submodules. Some of these classes are described in the following sections.

12.1 Root Classes

The root of the class hierarchy of Coral is formed by the class `Object`. On the other side, the bottom of the class hierarchy is formed by the class `Nothing`. The implicit contract here is that every class conforms to `Object`, and `Nothing` conforms to every class.

All classes and types in Coral are accompanied by the `Any` type (§3.6), for which every class conforms to it, pretty much like `Object`, but using it enables dynamic behavior, as this type is implicitly used for "unknown" types throughout Coral.

User-defined Coral classes that do not explicitly inherit from any class inherit from `Object` implicitly.

A part of the signatures of these root classes is described by the following definitions.

```
module Lang~[coral]

class Object
    extends ()
  method equals (that: Any): Boolean := ...
  alias :'=' is :equals

  method to_string: String := ...
```

[1]Not to be confused with CLS, which is this document.

```
  method hash_code: Integer := ...

  method tap: self.type := { yield self; self }
end

sealed abstract class Nothing extends () {}
```

12.2 Scalar Value Classes

12.2.1 Numeric Scalars

Types `Number.Integer`, `Number.Real` and `Number.Complex` are class clusters for various underlying classes, representing various subrange classes, and each of them inherits from class `Number`.

Partial order and ranking is defined on the numeric types as follows, using definitions of form $(T_1, \ldots, T_n) <:_w T$, where $T_1, \ldots, T_n$ weakly conforms to T.

- `(Integer_8, Integer_8_Unsigned)` $<:_w$ `Integer_16`
- `(Integer_8_Unsigned)` $<:_w$ `Integer_16_Unsigned`
- `(Integer_16, Integer_16_Unsigned)` $<:_w$ `Integer_32`
- `(Integer_16_Unsigned)` $<:_w$ `Integer_32_Unsigned`
- `(Integer_32, Integer_32_Unsigned)` $<:_w$ `Integer_64`
- `(Integer_32_Unsigned)` $<:_w$ `Integer_64_Unsigned`
- `(Integer_64, Integer_64_Unsigned)` $<:_w$ `Integer_128`
- `(Integer_64_Unsigned)` $<:_w$ `Integer_128_Unsigned`
- `(Integer_128, Integer_128_Unsigned)` $<:_w$ `Decimal`
- `(Integer_128_Unsigned)` $<:_w$ `Decimal_Unsigned`
- `(Decimal_Unsigned)` $<:_w$ `Decimal`
- `(Integer_128_Unsigned)` $<:_w$ `Decimal_Unsigned`
- `(Decimal, Decimal_Unsigned)` $<:_w$ `Integer`
- `(Decimal_Unsigned)` $<:_w$ `Integer_Unsigned`
- `(Float_32)` $<:_w$ `Float_64`

- `(Float_64)` $<:_w$ `Float_128`
- `(Float_128)` $<:_w$ `Decimal`
- `(Integer, Integer_Unsigned)` $<:_w$ `Real`
- `(Real)` $<:_w$ `Complex`

`Integer_8`, alias `Byte`, and `Integer_8_Unsigned`, alias `Byte_Unsigned`, are the lowest-ranked types in this order, whereas `Complex` is the highest-ranked. Ranking does not imply conformance (§3.7.2), e.g. `Integer_32` is not a subtype of `Integer_64`. However, the object `Predef` (§12.4) defines views (§7.3) from every numeric value to all higher-ranked numeric value types.

The `Integer` type has also a shorter alias of `Int`.

12.2.2 The `Boolean` Class

The `Boolean` class has only two member values, represented by the keywords **`yes`** and **`no`**.

12.2.3 The `Unit` Class

The `Unit` class has only one value, represented by the construct `()`.

12.3 Standard Reference Classes

This section presents some standard Coral reference classes, which are treated in a special way by the Coral compiler – Coral provides a syntactic sugar for them.

12.3.1 The `String` Classes

Coral's String has a literal defined for it. Strings in Coral are immutable, and a mutable version exists: `Mutable_String`. Both classes inherit from `String_Like`.

Note. Single characters in Coral are abstracted by strings of length exactly 1.

12.3.2 The `Symbol` Class

The Symbol class is intended to represent names throughout programs. As strings may or may not exist in multiple instances representing the exact same sequence of code points, there is always only up to one instance of a symbol that represents the exact

same sequence of code points. Class names, method names and all other names are represented by symbol instances, created on demand if necessary. Symbols are also a good choice for dictionary keys.

12.3.3 The `Tuple` Classes

Scala defines tuple classes `Tuple_`n for $n = 2, \ldots, +\infty$, using autoloading (§5.1.2). These are defined as follows:

```
module Lang~[coral]
case class Tuple_n [+T_1, ..., +T_n] (_1: T_1, ... _n: T_n)
    extends Object
  def apply (1) := _1
  ...
  def apply (n) := _n

  def to_string := "(#{_1}, #{...}, #{_n})"
end
```

The implicitly imported `Predef` object (§12.4) defines the names `Pair` as an alias of `Tuple_2` and `Triple` as an alias of `Tuple_3`.

12.3.4 The `Function` Traits

Coral defines function traits `Function_`n for $n = 2, \ldots, +\infty$, using autoloading (§5.1.2). These are defined as follows:

```
module Lang~[coral]
trait Function_n [-T_1, ..., -T_n, +R]
    extends Object
  message apply (x_1: T_1, ..., x_n: T_n): R
  def to_string := 'Function_n'
end
```

A subtype of `Function_1` represents partial functions, which are undefined on some points in their arguments domain. In addition to the `apply` method of functions, partial functions also have a `defined_at?` message, which tells whether the function is defined at the given argument:

```
module Lang~[coral]
trait Partial_Function [-A, +B]
    extends Function_1[A, B]
  message defined_at? (x: A): Boolean
  def to_string := 'Partial_Function'
end
```

12.4 The Predef Object

The Predef object defines some usual standard functions and type aliases for Coral programs. It is always implicitly imported, so that all its defined members are available without qualification. The Predef object may not be implicitly imported in a scope where **pragma** no-predef is applied.

Some of its members are defined as follows:

```
module Lang~[coral]
object Predef extends Object

  // identity functions
  def identity [A] (x: A): A := x
  def implicitly [T] (implicit e: T) := e

  // tuples
  type Pair [+A, +B] := Tuple_2[A, B]
  type Triple [+A, +B, +C] := Tuple_3[A, B, C]

  // implicit conversions
  implicit def int_8_to_int_16 (i: Integer_8): Integer_16 := ...
  ...

  // math aliases
  type Number := Math.Number
  type Integer := Number.Integer
  type Real := Number.Real
  type Complex := Number.Complex
  type Int := Integer
  ...

  // reading, printing
  def print (x: Any) := VM.Console.print(x)
  def print_line (x: Any) := VM.Console.print_line(x)
  def printf (text: String_Like, *xs: Any) := VM.Console.printf(text, *xs)

  def read_line: String := VM.Console.read_line
  def read_boolean := VM.Console.read_boolean
  def read_byte := VM.Console.read_byte
  ...

  // definitions
  implicit final class Arrow_Assoc [A] (private val myself: A)
      extends Object
    operator => [B] (that: B): Tuple_2[A, B] := Tuple_2(myself, that)
```

```
    end class

  end
```

Chapter A

Coral Syntax Summary