AFR-0

The Amlantis System Specification

Version 0.10

Markéta Lisová

November 24, 2019

Special Thanks To

Marie Strnadová

Ondřej Profant

Vojtěch Biberle

Contents

Part I

Introduction

A Brief Introduction to the Amlantis System

Amlantis System is a collection of specifications of programming and data definition languages and their related tools, runtimes and libraries.

Those specifications do not always require particular implementations, but attempt to avoid undefined behaviours as much as possible.

Amlantis also provides an open source implementation of those specifications, but it is indeed possible for other people to write their own implementations or forks[1] of this default implementation, probably focusing on optimizing other aspects of the system, maybe exploring new options of future development to be pull-requested into the default implementation.

A Few Notes on the Name

Amlantis' name has quite some history. The project started being named *Coral*, but that collided with another language of a similar name, *CORAL 66*. Then it got renamed to *Gear*, but that again collided with another language of the same name, which seemed inactive at the time, *zippers/gear*. Than an idea was born and Aml was named *Amlantis*, which is whatever you want it to be. It could be a misspelling of *Atlantis*[2], it could be an acronym like *A ML Language*, or maybe even something like *A ML Language And Neat Technology Improvement System*, or maybe *Caml* without the *C*. For the meaning of the cryptic *ML* part, search for the *Standard ML* or *OCaml*.

[1] Forks and pull requests are preferred!

[2] Intentionally – because otherwise, it would be named Atlantis, but there is already a city of that name.

The Amlantis Languages

There is a couple of languages that come with a fully-featured Amlantis' implementation:

- Virtual Machine Runtime languages:
  - Amlantis Core (or `Aml/Core`, or even simpler `Aml`[3]), the core programming language of the system. It is the most versatile language in this list, also allowing programming of DSLs etc. (§II)
  - Amlantis SE (or `Aml/SE`), also a functional-style focused language, but more *homoiconic*, using *Symbolic Expressions* and defining itself in its own data-structures (which are shared with the other `Aml` languages). (§IV)
- System Runtime languages:
  - Amlantis System (or `Aml/System`), based on `Aml/Core`. (§III)

Languages are interoperable within the same runtime, and have interoperability tools across runtimes. This is due to different ABIs of each runtime.

[3] Whenever this document mentions `Aml` alone, suffix-less, it mentions this core language, `Aml/Core`.

Part II

The Amlantis Core Programming Language

`Aml/Core`

Chapter 1

Introduction to Amlantis Core

Contents

1.1 About The Amlantis Core Language

`Aml` is a scalable, reasonably verbose, type-safe and type-inferred multi-paradigm language. These paradigms are:

- Object-oriented, both class-based and prototype-based
- Functional.
- Imperative.
- Concurrent.
- Metaprogramming.
- Reflective.

`Aml` is a hybrid language in two significant properties. First, the language itself is inherently dynamic, but focuses on minimizing impact of the dynamic changes. Second, it uses hybrid typing discipline, where most types are statically defined and selected types may opt-in for dynamic typing.

It aims to be a good modern high-level self-optimizing language with a scent of retro that programmers will just love.

1.2 Influences

Ada. Ada influenced the base of `Aml`'s syntax, and also had an impact on floating and fixed point types, which are very distinct in `Aml`.

`Aml` uses heavily keyword program parentheses in Ada fashion. There are **`class ... end class`**, `def ... end def`, **`do ... end`**, **`loop ... end loop`** etc.

Ruby. Ruby was the original inspiration in early stages of `Aml`'s development. It still has some visible marks left throughout `Aml`'s syntax, e.g. the metaclass access, and it itself being influenced by Ada, the syntaxes are also similar. `Aml` also admires Ruby's dynamic features and metaprogramming capabilities and wants to be as little obstacle in creating custom DSLs[1] as it can be.

Scala. Scala influenced the type system of `Aml`. The most visible effect that it has left here is the syntax for type arguments and type parameters, which are enclosed in square brackets, instead of the kind of more usual and troubled angle brackets.

[1] Domain-Specific Languages

OCaml & F#. These two languages (which themselves are related) influenced `Aml` so remarkably, that one may almost consider `Aml` to be a distant relative of the ML family of programming languages, it even has *ML* in its name. Function application syntax, function compositions, functional algebraic datatypes, currying, automatic type inference and much more. OCaml also had an impact on `Aml`'s name: it has *ML* in its name, too (although it is less ML-like than F#, which does not have that in its name).

Clojure. While Clojure as a member of the Lisp language family does not have much of a direct effect on the syntax of `Aml`, it influences `Aml` more conceptually.

ATS. ATS convinced `Aml` to give dependent types a try.

Personal experiences. `Aml` is kind of an opinionated language. It makes some subtle or also significant choices that may seem odd at the first sight, but they make more sense over the time (or are bugs). These include decisions like whitespace being required around most infix operators. However, its intentions are friendly – `Aml` prefers readability even one year from now to having the code written and compiled a little bit faster.

1.3 Notational Conventions

All lexical and grammar syntax definitions use a customized form of EBNF. These forms are reused across definitions of all Aml languages.

- `[element]`
  Option, one or zero occurrences of `element`.
- `{element}`
  Optional repetition, zero, one or more occurrences of `element`.
- `{element}+`
  Repetition, one or more occurrences of `element`.
- `? text ?`
  Special sequence, textual description of an element. The following lines could contain additional data, if so described.
- `element1 | element2`
  Alternation, either `element1` or `element2` applies, but not both.
- `(element)`
  Grouping.

- `element1, element2`
  Concatenation, used only for exceptions to enlist elements rather than sequence them.

- `element1 element2`
  Sequence of elements.

- `'data'`
  Terminal string, literal appearance of the given Unicode characters.

- `| ··· |`
  Signals a range from the previous alternative to the following alternative.

- `element - except`
  Exception, an anonymous element that contains all of `element`, except for elements defined by `except` (which can be a grouping).

- `category ::= element`
  Used for definitions; signals that `category` is made of `element`, as in EBNF. The same `category` could appear on the left side multiple times for $\texttt{element}_1$ … $\texttt{element}_n$, in that case, it is taken that `category ::=` $\texttt{element}_1$ `| ... |` $\texttt{element}_n$.

- `category ::= element + extra`
  Extension, a category in which all occurrences of `element` can also be `extra`, including those occurring in `extra` itself.

- `category(param, param, ... ) ::= element`
  A category parameterized by one or more elements that may appear as a part of its definition.

- `category(element, element, ... )`
  A category element parameterized by another elements.

- `-- comment` or `{- comment -}`
  An additional information about the given syntax element, could contain additional restrictions described textually.

Chapter 2

Program Structure

The Amlantis Programming Language System consists of a couple of command line tools, which are described with greater detail in (§VII):

- `aml`, runs Aml interactively in a read-eval-print loop.
- `amlc`, compiles Aml modules or scripts.
- `amlopt`, compiles native optimized Aml modules or scripts.
- `amlrun`, runs Aml modules or scripts.
- `amldoc`, generates documentation for Aml modules.
- `amldbg`, debugger for Aml modules or scripts.
- `amlp`, inspects compiled Aml modules or scripts.
- `amlf`, Aml module bundler and builder.
- `amlprof`, Aml module or script profiler.

The inputs to the Aml tools consist of:

- Source code files.
  - Files with extension `.aml`, implementation files, conforming to the grammar element `Implementation_File`.
  - Files with extension `.amls`, implementation signature files, conforming to the grammar element `Signature_File`.
  - Files with extension `.amlx`, script files, conforming to the grammar element `Script_File`.

  - Files with name `module.aml`, module definition files, conforming to the grammar element `Implementation_File`.
  - Files with name `signature.aml`, module or submodule signature files, conforming to the grammar element `Signature_File`.

- Compiled files.
  - Files with extension `.amlb`, bytecode function files.
  - Files with extension `.amlpsi`, bytecode PSI files.
  - Files with name `module.dependencies.lock`, locked module dependencies.

- Semi-source files.
  - Files with extension `.amlp`, protocol files, conforming to the grammar element `Signature_File`.
  - Files with name `module.dependencies.aml`, module dependencies, conforming to the grammar element `Script_File`.

- Script fragments, used in interactive environments, conforming to the grammar element `Script_Fragment`, where each script fragment can be separated by "`;;`"[1], to trigger evaluation by the interactive environment.
- Compilation and runtime parameters passed to the command line tools.
- Pragma and annotation applications (§15.2) within source files and their compiled forms.

The `Aml/Language.VM` class shall offer methods for querying the compilation environment – methods `is_compiled?` and `is_interactive?`.

Processing of the source code portions of these inputs consists of the following steps:

1. *Decoding*. Each file and source code fragment is decoded into a stream of Unicode characters. UTF-8 is recommended as the source encoding and the default one, but different encodings may be used, if specified in the command line tool's parameters.
2. *Tokenization*. The stream of Unicode characters is broken into a token stream by lexical analysis, described in (§3).
3. *Lexical filtering*. The token stream is filtered by rules described in (§4). These rules describe how additional (virtual) tokens are inserted or removed from the token stream, and how some existing tokens are split into multiple tokens or replaced by others to generate an augmented token stream.

[1] This simulates "End Of File" in regular script files.

4. *Parsing*. The augmented token stream is parsed according to the grammar rules from this document. The result is an AST.

5. *Importing*. Modules referenced from **`module.`**`dependencies.lock` are linked into the AST.

6. *Checking*. The results of parsing are checked. This includes type checking and variable checking.

7. *Elaboration*. The checked AST is pre-cached. This includes pre-filling of local method caches, and type arguments, all dependent on the initial version of each PSI[2] element that plays a part. Elaborated AST and PSI can be saved in a more permanent form, like bytecode files. Each source file is represented by an implicit function[3], but whether these functions are stored in one bytecode file, or multiple bytecode files, or even one bytecode file per each source file, is of no importance – the PSI has to manage mappings of these source files to their actual compiled location.

8. *Evaluation*. Elaborated ASTs are evaluated against the elaborated PSI, according to the commands and parameters the command line tools received.

[2]Program Structure Information, basically a repository of all types that a program uses.

[3]Therefore, DSLs are really easy – since a file is basically a function, it can be used as such, although indirectly.

Chapter 3

Lexical Syntax & Analysis

Contents

Aml programs are written using the Unicode character set; Unicode supplementary characters are supported as well. Aml 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 and text editors 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 rules defined in the following sections. Various combination of different Unicode character classes can also build up tokens, and later on, parts of different higher syntax forms.

3.1 Identifiers & Keywords

Syntax:

```
digit_char
    ::= '0' ... '9'
letter_char
    ::= ? Unicode char of classes Lu, Ll, Lt, Lo, Lm and Nl ?
connecting_char
    ::= ? Unicode char of class Pc ?
combining_char
    ::= ? Unicode char of classes Mn and Mc ?
formatting_char
    ::= ? Unicode char of class Cf ?
lower
    ::= ? Unicode char of class Ll ?
      | '_'
upper
    ::= (letter_char - lower)
      | '§'
var_id
    ::= lower id_rest
plain_id
    ::= upper id_rest
      | var_id
id  ::= (plain_id | '```' {any_id}+ '```')
      | '`' plain_id '`'
      | '`' op_id '`'
id_char
    ::= letter_char
      | digit_char
      | connecting_char
      | combining_char
      | formatting_char
      | '_'
id_rest
    ::= {id_char | op_char} [{'''}+]
any_id
    ::= ? any Unicode chars except for two backquotes: '```' ?
```

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. 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 (§8.3.10).

An identifier may also be formed by an identifier between back-quotes (“ ` ”), to resolve possible name clashes with `Aml` keywords, and to allow to identify operators. An identifier may also be enclosed in double back-quotes (“ `` ”), where any Unicode character may appear, except for another back-quote.

`Aml` 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[1] names.

`Aml` specification does not list the language keywords, because every keyword is only reserved in particular set of contexts. Every reserved word occurrence in grammar is also literal – simply enclosing the keyword in one of the backquote forms removes its keyword-bound meaning, e.g. `` `begin` ``.

3.2 Function Identifiers

Syntax:

```
fun_id
    ::= id
      | '(' op_id ')'
      | Active_Pattern_Ids
```

Function identifiers specify how a function can be named. Usually, it will be the first case. The second case is provided for specifying functions that are in fact operators, and is practically equivalent to the form of `op_id` enclosed in backquotes. The third case is for active patterns (§9.1.5.7) in pattern matching (§9).

3.3 Symbolic Keywords

Symbolic or partially symbolic character sequences can be treated as keywords, if a lexical or syntactic grammars specifies that.

3.4 Symbolic Operators

Syntax:

[1] English, not mistakes.

```
op_id
    ::= Symbolic_Op
      | Qualified_Op
      | 'and' ['also']
      | 'or' ['else']
      | 'xor'
      | 'not'
      | id  -- needs to exist in scope, useful for DSLs
nary_op_id
    ::= '?' {op_char}
Symbolic_Op
    ::= {op_char}+
Qualified_Op
    ::= '`' Long_Id '.' {op_char}+ '`'
      | '`' Long_Id '.' id '`'
op_char
    ::= ? Unicode char of classes Sm, So except for 0x0060 ?
      | '!'
      | '#'
      | '%'
      | '$'
      | '&'
      | '*'
      | '+'
      | '-'
      | '.'
      | '/'
      | ':'
      | '<'
      | '='
      | '>'
      | '?'
      | '@'
      | '\'
      | '^'
      | '|'
      | '~'
```

Symbolic operators are sequences of characters as shown above, except where the sequence of characters is a symbolic keyword (§3.3) and is used by the corresponding grammar as such.[2]

[2]In another words, symbolic keywords have a precedence over symbolic operators, if a grammar would be otherwise ambiguous.

3.5 Delimiter Characters

Some characters are not considered as operator characters (§3.4), but are still part of lexical structure of Aml programs. These include:

Syntax:

```
Delimiter ::= ','
            | '`'
            | '''
            | '"'
```

Delimiter characters are used as components of higher syntactic elements.

3.6 Newline Characters

Syntax:

```
nl   ::= -- Line Feed, LF; Multics, Unix, Unix-like...
         ? \u000A ?

         -- Carriage Return, CR; Mac OS 9-, ZX Spectrum...
       | ? \u000D ?

         -- CR+LF; Windows, DOS, OS/2, Symbian OS...
       | ? \u000D \u000A ?

         -- LF+CR; RISC OS...
       | ? \u000A \u000D ?

         -- Unicode conformance
       | ? \u000B ?  -- Vertical Tab, VT
       | ? \u000C ?  -- Form Feed, FF
       | ? \u0085 ?  -- NExt Line, NEL
       | ? \u2028 ?  -- Line Separator, LS
       | ? \u2029 ?  -- Paragraph Separator, PS

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

Aml 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 Aml source file is treated as the special separator token `nl` if the following criteria are satisfied:

1. The token immediately preceding the newline can terminate an expression.
2. The token immediately following the newline can begin a new expression.

In interactive environment, top-level expressions are ended by entering the "`;;`" sequence of tokens (`tend`).

If the token immediately preceding the newline can not terminate an expression and is followed by more than one newline, Aml still sees that as only a one significant newline, to prevent any confusion.

The grammar rules contain productions where the optional newline ("`[nl]`") is present, in which case it obviously does not end any expression.

3.7 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_Expr
    ::= number_literal
      | percent_literal
      | annotated_number
      | character_literal
      | boolean_literal
      | string_literal
      | symbol_literal
      | regexp_literal
```

3.7.1 Integer Literals

Syntax:

```
sign
    ::= '+' | '-'
number_literal
    ::= [sign] integer_literal
      | ...
integer_literal
    ::= (decimal_numeral
```

```
      | hex_numeral
      | octal_numeral
      | binary_numeral
      | sexagesimal_numeral
      | duodecimal_numeral) [int_suffix]
decimal_numeral
    ::= digit {['_'] digit}
hex_numeral
    ::= hex_prefix ['_'] hex_digits
hex_digits
    ::= hex_digit {['_'] hex_digit}
digit
    ::= '0' | ··· | '9'
hex_digit
    ::= digit
      | 'a' | ··· | 'f'
      | 'A' | ··· | 'F'
octal_numeral
    ::= oct_prefix ['_'] oct_digits
oct_digits
    ::= oct_digit {['_'] oct_digit}
oct_digit
    ::= '0' | ··· | '7'
binary_numeral
    ::= bin_prefix ['_'] bin_digits
bin_digits
    ::= bin_digit {['_'] bin_digit}
bin_digit
    ::= '0' | '1'
sexagesimal_numeral
    ::= sxgs_prefix ['_'] sxgs_digits
sxgs_digits
    ::= sxgs_digit {',' sxgs_digit}
sxgs_digit
    ::= ['0' | ··· | '5'] ('0' | ··· | '9')
duodecimal_numeral
    ::= ddec_prefix ['_'] ddec_digits
ddec_digits
    ::= ddec_digit {['_'] ddec_digit}
ddec_digit
    ::= digit
      | 'a' | 'b'
      | 'A' | 'B'
-- in prefixes, the alternative with lowercase letter is preferred:
hex_prefix
    ::= '0x' | '0X'
```

```
bin_prefix
    ::= '0b' | '0B'
oct_prefix
    ::= '0o' | '0O'
sxgs_prefix
    ::= '0s' | '0S'
ddec_prefix
    ::= '0d' | '0D'
int_suffix
    ::= ('G' | 'I' | 'M' | 'N' | 'T' | 'Q' | 'R' | 'Z') {digit}
```

Integers are usually of type `Number`, which is a class cluster of all types that can represent numbers. Unlike Java, Aml supports both signed and unsigned integral types 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`, but this is implementation-defined. Math operations on numbers are handled internally in such a way that the user doesn't need to worry about the actual types of the numbers — when an integer overflow would occur, the result is stored in a larger container type, or maybe even in a smaller container if fit. Expected type is always respected in any case.

Underscores "_" are allowed between digits for readability, but are otherwise ignored. Decimal integer literals can begin with leading zeros "`0`", but those zeros are likewise ignored.

Integral members of the `Number` class cluster include the following container types. It is implementation-defined how numbers are actually stored and represented internally by the VM – the language itself does not care, so long as types are respected.

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 `Cent`
10. `Integer_128_Unsigned` (0 to 2^{128}), alias `Cent_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. The class cluster types may also be viewed as range types (or constrained types), utilizing their range of values as bounds (or constraints).

3.7.2 Floating & Fixed Point Literals

Syntax:

```
number_literal
    ::= ...
      | [sign] fp_literal
fp_literal
    ::= decimal_numeral '.' decimal_numeral [exponent_part_e] [float_type]
      | decimal_numeral exponent_part_e [float_type]
      | decimal_numeral float_type
      | hex_numeral '.' hex_digits [exponent_part_p [float_type]]
      | hex_numeral '.' hex_digits hex_exp float_type
      | hex_numeral exponent_part_p [float_type]
      | hex_numeral hex_exp float_type
      | octal_numeral '.' oct_digits [exponent_part_e] [float_type]
      | octal_numeral exponent_part_e [float_type]
      | octal_numeral float_type
      | binary_numeral '.' bin_digits [exponent_part_e] [float_type]
      | binary_numeral exponent_part_e [float_type]
      | binary_numeral float_type
      | sexagesimal_numeral ';' sxgs_digits [exponent_part_e] [float_type]
      | sexagesimal_numeral exponent_part_e [float_type]
      | sexagesimal_numeral float_type
      | duodecimal_numeral '.' ddec_digits [exponent_part_e] [float_type]
      | duodecimal_numeral exponent_part_e [float_type]
      | duodecimal_numeral float_type
exponent_part_e
```

```
    ::= int_exp [[digit [digit]] sign] decimal_numeral
exponent_part_p
    ::= hex_exp [[digit [digit]] sign] decimal_numeral
int_exp
    ::= 'e'
hex_exp
    ::= 'p'
float_type
    ::= 'h'  -- Half float precision
      | 'f'  -- Single float precision
      | 'd'  -- Double float precision
      | 'q'  -- Quadruple float precision
      | 'm'  -- Decimal unlimited precision
```

Floating point and fixed 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. If `float_type` part is not present, then the expressions' expected type is used, and if expected type is unable to determine the floating/fixed point type, then a fixed point type is used, with precision set precise enough to store all given significant digits without loss.

Literals that have `float_type` of "`m`" are fixed point literals. Also, floating point literals that are impossible to represent in binary form accurately are implicitly inferred to be fixed point literals, unless specifically converted to a floating point type or using a `float_type` of `h`, `f` or `d`. From `Number`'s user perspective, this is only an implementation detail, and the VM is not required to store floating or fixed point values in any specific representation other than that with the required precision and properties.

1. `Float_16` (IEEE 754-2008 16-bit precision), alias `Half_Float`.
2. `Float_32` (IEEE 754-2008 32-bit precision), alias `Float`.
3. `Float_64` (IEEE 754-2008 64-bit precision), alias `Double`.
4. `Float_128` (IEEE 754-2008 128-bit precision), alias `Quadruple`.
5. `Decimal` ($-\infty$ to ∞).
6. `Decimal_Unsigned` (0 to ∞).

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

3.7.3 Not A Number

A member named `Not_a_Number` exists in the class cluster `Number`, with an alias of `NaN`, to denote results of operations on numbers that are not numbers, e.g., result of division by zero. There is no literal for this special value in Aml.

3.7.4 Imaginary Number Literals

Syntax:

```
number_literal
    ::= ...
      | [sign] imaginary_literal
      | [sign] complex_literal
      | [sign] real_number_literal
real_number_literal
    ::= integer_literal
      | fp_literal
imaginary_literal
    ::= real_number_literal 'i'
complex_literal
    ::= [real_number_literal sign] imaginary_literal
      | imaginary_literal sign real_number_literal
```

Imaginary number literals are of type `Number`, and have a basically a single container type: `Number.Complex`. The syntax requirement here is whitespace around the "`+`" and "`-`" signs, separating the real part from the imaginary part. Newlines as whitespace have the same effect as defined for cases where the sign could be considered to be an operator (§3.6).

3.7.5 Percentage Literals

Syntax:

```
percent_literal
    ::= real_number_literal
        [? whitespace except new line ?]
        ('%' | '‰')
```

Aml offers a way to represent numbers as fractions of 100 (in case of "`%`" suffix) or 1000 (in case of "`‰`" suffix). The type of such literal is not `Number` though, but `Percentage` (which is convertible to a number).

3.7.6 Units of Measure

Aml has an addition to number handling, called *units of measure* (§13). Number instances can be annotated with a unit of measure to ensure correctness of arithmetic operations.

Syntax:

```
annotated_number
    ::= number_literal '[<' measure_literal '>]'
```

Example 3.7.1 Some number literals annotated with units of measure:

```
let min_motorway_speed = 90 [<km/hour>] in
-- in the end, same as:
let distance = 90 [<km>] in
let time = 1 [<hour>] in
min_motorway_speed = distance // time
```

3.7.7 Character Literals

Syntax:

```
printable_char
    ::= ? all visible and printable UTF-8 characters ?
character_literal
    ::= '%'' (printable_char | char_escape_seq) '''
```

Character literals are of type `Character`, and are internally similar to strings (§3.7.9) of length of 1 character.

Character escape sequences are used in strings as well, and are defined as follows:

Syntax:

```
char_escape_seq
    ::= '\''  -- Literal single-quote: '
      | '\"'  -- Literal double-quote: "
      | '\?'  -- Literal question mark: ?
      | '\\'  -- Literal backslash: \
      | '\#'  -- Literal hash: #
      | '\$'  -- Literal dollar: $
      | '\0'  -- Binary zero (NUL, U+0000)
      | '\a'  -- BEL (alarm) character (U+0007)
      | '\b'  -- Backspace (U+0008)
      | '\f'  -- Form feed (FF, U+000C)
```

```
      | '\n'  -- End-of-line (U+000A)
      | '\r'  -- Carriage return (U+000D)
      | '\t'  -- Horizontal tab (U+0007)
      | '\v'  -- Vertical tab (U+000B)
      | '\xnn'  -- Byte value in hexadecimal
      | '\unnnn'  -- Unicode character U+nnnn, hex digits
      | '\Unnnnnnnn'  -- Unicode character U+nnnnnnnn, hex digits
      | '\u{n_1...n_m}'  -- Unicode character U+n_1...n_m, hex digits, n < 8
      | '\&' {named_char}+ ';'  -- Named character entity
      | '\{' {named_char | ' '}+ '}'  -- Named Unicode character

named_char
    ::= 'a' | ··· | 'z' | 'A' | ··· | 'Z'
```

The named character reference (`named_char`) is defined by HTML5 (`http://www.w3.org/TR/html5/syntax.html#named-character-references`); valid Unicode character names are specified by Unicode 12.0 (`https://www.unicode.org/versions/Unicode12.0.0/`)[3].

3.7.8 Boolean Literals

Syntax:

```
boolean_literal
    ::= 'yes'
      | 'no'
```

Both literals are members of type `Boolean`.

3.7.9 String Literals

Syntax:

```
string_literal
    ::= int_string_literal
      | raw_string_literal
      | verbatim_string
int_string_literal
    ::= '"' {int_string_element} '"'
      | '%'  [str_flags] '"' {int_string_element} '"'
raw_string_literal
    ::= '%r' [str_flags] '"' {string_element} '"'
verbatim_string
    ::= '"""' {? any Unicode character ?}+ '"""'
```

[3]Will be updated to newer Unicode versions as ICU components are updated.

```
string_element
    ::= printable_char
      | char_escape_seq
int_string_element
    ::= string_element
      | interpolated_expr
interpolated_expr   -- must not be preceded by an odd number of backslashes
    ::= '${' Expr '}'
      | '$' ? no whitespace ? id
      | '#{' Expr '}' [? no whitespace ? int_format_flags]
str_flags
    ::= 'm'
      | 'i'
int_format_flags
    ::= '%' {printable_char}+
      | '%{' int_format_flags '}'
      | '%(' int_format_flags ')'
```

String literals are members of the type `String`. Double quotes in interpolable string literals have to be escaped (`\"`).

String literals appear in multiple forms:

- *Raw strings* are of the form `%r`*f*`"`*s*`"`. There are no interpolated expressions. If a sequence of string elements appears to be an interpolated expression, it is not, and is instead treated as a part of the raw string as it is. Escape sequences are evaluated.
- *Interpolable strings* are of the forms `"`*s*`"` and `%`*f*`"`*s*`"`. Interpolated expressions can appear there, if its introducing character is not escaped by an odd number of backslashes. Interpolated expressions are evaluated, converted to `String_Like` and their result inserted at each place in the string where they appear, safely.
- *Verbatim strings* are of the form `"""`*s*`"""`. Escape sequences and interpolated expressions are not evaluated, also there is thus no way to escape the delimiter (`"""`) inside it. Verbatim strings are immutable, and can be made mutable by use of appropriate methods.
- *Immutable strings* are all strings that do not carry the flag `m` (**m**utable). The flag `i` (**i**mmutable) is redundant in this manner, and is only provided for completeness.
- *Mutable strings* are only those strings that carry the flag `m`. Immutable strings may be converted to mutable strings and vice versa by use of appropriate methods.

In each form, *s* is the string content, and *f* are its flags, as defined.

In each form, literal newlines do not need to be escaped. Newlines may still be inserted into single-line string literals. Having escaped newlines appearing in a multi-line string is frowned upon.

3.7.10 Symbol Literals

Syntax:

```
symbol_literal
    ::= simple_symbol
      | quoted_symbol
simple_symbol
    ::= ':' plain_id
quoted_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. Aml does not require their ids to be constant in time.

3.7.11 Regular Expression Literals

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

Regular expressions are defined in (§21).

3.8 Whitespace & Comments

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

A multi-line comment is a sequence of characters between `{-` and `-}`. Multi-line comments may be nested.

A single-line comment in Aml is either a multi-line comment, written all on one line, or everything that follows after the token “ `--` ” on a single line.

Documentation comments are multi-line comments that start with `{-|` and end with `-}`. A whitespace is required after the opening token in this case.

As a special case (not only) for UNIX-like systems that support the notation, the initial lines of source file that start with the sequence “`#!`” are ignored.

Whitespace serves two purposes: to separate tokens in the source file, and to determine whether an operator is prefix or infix, and is otherwise ignored. The following characters are considered whitespace: space (\u0020, “ ”), horizontal tab (\u0009) and characters of Unicode

class `Zs`. Newlines (§3.6) are also considered as whitespace, after their special purpose is fulfilled.

Whitespace tokens and comments are not discarded during lexical analysis, but instead after lexical filtering, by rules described in (§4).

Example 3.8.1 Standard comments in `Aml`.

```
-- This is a single line comment.

{- This is a
multi-line
comment.
-}

{-| A documentation comment.
Following grammar element is documented by it.
-}

{--}
   "Removing the right bracket above makes this a part of the comment."
 --}
```

Example 3.8.2 "Shebang" comments in `Aml`.

```
#!/usr/bin/env amlrun

{- Program source follows. -}
```

3.9 Conditional Compilation

Grammar:

```
Pragma           ::= If_Directive
                   | Else_Directive
                   | End_If_Directive
If_Directive     ::= 'pragma' ('if' | 'unless') Arguments
Else_Directive   ::= 'pragma' Else
End_If_Directive ::= 'pragma' 'end-if'
```

If an `If_Directive` token is matched during tokenization, text is recursively tokenized until a corresponding `Else_Directive` or `End_If_Directive`. If the compilation environment

defines the associated identifier, the token stream includes the tokens between the `If_Directive` and the corresponding `Else_Directive` or `End_If_Directive`. Otherwise, the tokens are discarded. The converse applies to the tokens between a corresponding `Else_Directive` and `End_If_Directive`. These directives may be nested.

3.10 Hidden Tokens

Some hidden tokens are inserted by lexical filtering (§4), or are used to replace existing tokens.

Chapter 4

Lexical Filtering

Chapter 5

Identifiers, Names & Scopes

Names in Amlantis 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.[1]
2. Explicit **`use`** clauses (imports) have the next highest precedence.[2]
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 `Dynamic`.

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

Every binding has a *scope*[3] in which the bound entity can be referenced using a simple name

[1] Therefore, local variable and definition names are always preferred to any other name. Function parameters are also treated as local in this sense.

[2] Explicit imports have such high precedence in order to allow binding of different names than those that would be otherwise inherited. Also affects methods, since those can be selected from objects.

[3] This includes, but is not limited to nested templates.

(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[4] and fully in runtime.

If a particular file is a part of a module, then its root scope includes bindings from the module file. Moreover, some bindings are implicitly added to files by the language itself (controllable by pragmas/annotations).

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[5], then it is an error.

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

```
let
  val x = 1
in
  use p.x
  in
    x
  end
end
```

neither binding of *x* shadows the other. Consequently, the reference to *x* on the fourth 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.

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

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

If *x* is bound by explicit **use** import clause (§7.13), then the simple name *x* is considered to be equivalent to the fully-qualified name to which *x* is mapped by the import clause. If *x* is bound by a definition or declaration, then *x* refers to the entity introduced by that binding, thus the type of *x* is the type of the referenced entity.

Chapter 6

Types

Contents

6.1 About Amlantis' Type System

Grammar:

```
Type_Expr
    ::= Function_Type [Quantification_Clauses]
      | Infix_Type [Quantification_Clauses]
      | Type_Pack_Expansion
      | Type_Context Type_Expr
      | Atomic_Type_Expr
Quantification_Clauses
    ::= Quantification_Clause {'and' Quantification_Clause}
Quantification_Clause
    ::= Existential_Clauses
      | Universal_Clauses
      | Given_Clauses
      | Type_Constraint_Clause
Given_Clauses
    ::= 'given' Simple_Type
      | 'given' '{' Given_Dcl {semi Given_Dcl} '}'
Tuple_of_Types
    ::= Type_Expr {',' Type_Expr}
Type_Equation
    ::= '=' Type_Expr
Type_Pack_Expansion
    ::= '...' Type_Expr
Atomic_Type_Expr
    ::= Simple_Type
      | '(' Type_Expr ')'
```

Types in `Aml` comprise three interconnected elements of the language:

- Blueprints of values, describing structure of each value that is a member of the type, its properties, list of messages that can be sent to the object.
- Designators of those blueprints, which give semantic meaning to the blueprints. Two types may share the same structure, but if they are given different names, the semantic difference is preserved.
- Types are values themselves, making runtime reflection an innate part of the language.

6.2 Paths

Grammar:

```
Path
    ::= Stable_Id
      | [id '.'] 'self'
      | 'outer' Class_Qualifier
Long_Id
    ::= [Path '.'] id
      | [id '.'] 'super' [Class_Qualifier] '.' id
      | 'outer' Class_Qualifier '.' id
      | 'global' '.' id
Stable_Id
    ::= Long_Id
Class_Qualifier
    ::= '[' Long_Id ']'
```

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

A path is one of the following:

- The empty path ε (which can't be written explicitly in user programs).
- 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.[1]
- `Function`**`.self`**, where `Function` is defined by Aml's `Language` module (not imported to be anything else), is a reference to the directly enclosing function object and is available only within functions (or methods, anonymous functions, and even blocks).
- p**`.`**x, where p is a path and x is a member of p.
- 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.

[1] This is used to refer to a single object.

6.2.1 Type Variables

Grammar:

```
Stable_Id
    ::= ...
      | Type_Var
Type_Var
    ::= ''' plain_id
      | '''' plain_id
```

A type "`'`*x*" stands for a type variable named *x*. A type "`''`*x*", moreover, comes with a type context, `Equality '`*x* `=> '`*x*, which is added to the type where it appears (or is exactly that if the type comprise just the type variable).

Type variables, unlike type parameters, may also be unified with a specific type by the automatic type inference, and may be made explicitly polymorphic.

6.2.2 Self Path

The path *C*`.`**`self`** and its derivatives are not really special within `Aml`, because it is possible to bind the object (that *C*`.`**`self`** would otherwise refer to) to a different name, say *t*. In such a case, *t* can then be used throughout `Aml`'s type system as *C*`.`**`self`** would have been used. The only difference is then that *C*`.`*t* is not allowed to be used to resolve which **`self`** is meant to be used – but that is the whole point of renaming that particular **`self`** to *t*, to not have to explicitly write the whole (possibly long) path.

6.3 Value Types

Every value in Aml has a type which is of one of the following forms. Remember that the grammar elements from the following sections are used in contexts where a type is expected – not any general expression.

6.3.1 Singleton Type

Grammar:

```
Simple_Type
    ::= Singleton_Type
Singleton_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 solely the value denoted by *p*.

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

6.3.2 Cloned Type

Grammar:

```
Simple_Type
    ::= ...
      | Cloned_Type
Cloned_Type
    ::= Path '.' 'type' '.' 'cloned'
```

A cloned type is of the form *p*`.type.cloned`, where *p* is a path pointing to a value. The type denotes the set of values consisting of the value denoted by *p* and every value that is cloned from the value denoted by *p*.

A *stable type* is either a cloned type or a type which is declared to be a subtype of a trait `Cloned_Type`. Singleton in this view appears to be a special case (more concrete to be precise) of a cloned type, excluding the cloned values.

6.3.3 Literal-Based Singleton Type

Grammar:

```
Simple_Type
    ::= ...
      | Literal_Type
Literal_Type
    ::= Literal ['.' 'type']
```

A singleton type based on a literal is of the form *l*`.type`, where *l* is a literal. The type denotes the set of values consisting of every literal value that is equal to *l*.

A *stable type* is either a literal-based singleton type or a type which is declared to be a subtype of a trait `Literal_Singleton_Type`.

In contexts where a type is expected[2], the "`.type`" part of the type can be omitted.

[2]Most notably type arguments or type annotations of variables or function result types.

6.3.4 Type Projection

Grammar:

```
Simple_Type
    ::= ...
      | Projected_Type
Projected_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 6.3.1 A sample code that shows off what type projections are good for:

```
class A = object
  class B = object end
  method f (b: B): Unit   = Console.print_line "Got my B."
  method g (b: A#B): Unit = Console.print_line "Got a B."
end

let
  val a1 = A()
  val a2 = A()
in
  begin
    a2.f a1.B()   -- type mismatch, found a1.B, required a2.B
    a2.g a1.B()   -- prints "Got a B." to stdout
    a2.f a2.B()   -- prints "Got my B." to stdout
  end
```

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 (§11.10) or some form of metaclass access (§11.4.5)). Therefore, `a1.B` refers to type member `B` of the instance `a1`, but not of `a2`. Moreover, `A.B` is not defined here.

6.3.5 Type Designators

Grammar:

```
Simple_Type
    ::= ...
      | Stable_Id
      | Selected_Type
```

```
      | Self_Type_Expr
Selected_Type
    ::= Parameterized_Type '.' Long_Id
Self_Type_Expr
    ::= 'Self'
```

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

Specifically, the unqualified type name t where t is bound in some class, object or structure C is taken as a shorthand for C**.self.type#**t.[3] If t is not bound in a class, object or structure, 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 (§6.2) and t is a type name. Such a type designator is equivalent to the type projection p**.type#**t.

The type designator **self** refers to the singleton type **self.type**, and C**.self** refers to the singleton type C**.self.type**. Aml also defines a designator named `Self`, which refers to the enclosing class type.

Example 6.3.2 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", not real code
-- predefined import of Aml/Language
Number = Aml/Language.type#Number

val An_Object = object
  type t = ... end
  t = An_Object.type#t        -- bound by standalone object
end

class A_Class = object
  type t = ... end
  method a_method = begin
    t = A_Class.self.type#t   -- bound by class
    t = self.type#t           -- bound by class
  end
  class << self
    type u = ... end
    u = A_Class.type#u        -- bound by class object
    t = A_Class.self.type#t   -- bound by class
  end
```

[3] Note that in objects and structures, the designator C**.self** refers to that object or structure, whereas in classes, it refers to instances of that class.

```
    t = A_Class.self.type#t     -- bound by class
    A_Class.t = A_Class.self.type#t  -- -||-
    A_Class.u = A_Class.type#u  -- bound by class object

    u = A_Class.type#u          -- not bound by class, thus bound by class object;
                                -- if class A_Class defined u itself, it would shadow
                                -- the definition in the class object
end
```

6.3.6 Parameterized Types

Grammar:

```
Simple_Type
    ::= ...
      | Parameterized_Type
Parameterized_Type
    ::= Simple_Type [? no whitespace ? Type_Args]
      | Simple_Type '.' Type_Args
      | Postfix_Parameterized_Type
Postfix_Parameterized_Type
    ::= Atomic_Type_Arg Stable_Id
Type_Args
    ::= '[' [Type_Arg {',' Type_Arg}] ']'
Type_Arg
    ::= Type_Arg_Value
      | Measure_Arg_Value
      | Static_Arg_Value
Atomic_Type_Arg
    ::= Type_Arg_Value
      | '(' Measure_Arg_Value ')'
      | '(' Static_Arg_Value ')'
      | '(' [Type_Arg {',' Type_Arg}] ')'
Type_Arg_Value
    ::= ['~' id ':'] Type
      | '_'
Measure_Arg_Value
    ::= ['~' id ':'] '<' measure_literal '>'
Static_Arg_Value
    ::= Constant_Expr
      | 'constant' 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 $[a_1 \mapsto T_1, \ldots, a_n \mapsto T_n]$[4]. Also, U_i must never be a subtype of L_i, since no other type ever would be able to fulfill 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).

Each type argument may be given a name *n*, using the form "`~`*n*`:` *T*". This attaches the annotation `@[labelled :`*n*`]` to the type argument.

A type may be parameterized by at most two type parameter clauses, although the second one is managed by the Aml system and can't be passed type arguments explicitly. Such a second type parameter clause is called *shadow type parameter clause* and can't be written in user programs explicitly, but may examined in runtime using reflection.

Example 6.3.3 Given the generic type definitions:

```
class Tree_Map['a <: Comparable['a], 'b] = object ... end
class 'a list = object ... end
class I = object inherit Comparable[I]; ... end

class F['m[_], 'x] = object ... end
class S['k <: String] = object ... end
class G['m['z <: 'i], 'i] = object ... end

trait Function[-'t, +'r] = object ... end
```

the following parameterized types are well-formed:

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

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

Function[named: List[String], String]
```

Example 6.3.4 Given the type definitions of the previous example, the following types are malformed:

```
Tree_Map[I]               -- wrong number of parameters
```

[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`]`.

```
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 -}
```

Example 6.3.5 Alternative syntax enabled by postfix type declaration:

```
Integer list         -- equivalent to List[Integer]
'i list              -- equivalent to List['i]
String list          -- equivalent to List[String]
Boolean list list    -- equivalent to List[List[Boolean]]
```

6.3.7 Polymorphic Variant Types

Grammar:

```
Simple_Type
    ::= ...
      | Polymorphic_Variant_Type
Polymorphic_Variant_Type
    ::= '[' Tag_Spec {semi Tag_Spec} ']'
      | '[>' [Tag_Spec {semi Tag_Spec}] ']'
      | '[<' Tag_Spec {semi Tag_Spec} [semi '>' {Tag_Name}+] ']'
Tag_Spec
    ::= Tag_Name ['of' Type]
      | Type
Tag_Name
    ::= '`' ? id that does not start with ` ?
```

Polymorphic variant types describe the values a polymorphic variant may take.

The first case ([ts_1; ...; ts_n]) is an *exact polymorphic variant type*: all possible tags are known, with their associated types, and they can all be present in values; its structure is fully known.

The second case ([> ts_1; ...; ts_n]) is an *open polymorphic variant type*, describing a polymorphic variant value: it givs the list of all tags the value could take, with their associate types, and this type is still compatible with a polymorphic variant type containing more tags.

A special case is the unknown type (`[> ]`), which does not define any tag, and therefore is compatible with any polymorphic variant type.

The third case (`[< ts`$_1$`; ...; ts`$_m$`; > `Tag$_n$ ... `Tag$_o$ `]`) is a *closed polymorphic variant type*. It gives information on all the possible tags and their associated types, along with which tags are known to potentially appear in values. The exact variant type from the first case is equivalent to a closed variant type where all possible tags are also potentially present in values.

In all cases, tags may be either specified directly in the `` `Tag `` and `` `Tag `` **`of`** T forms, or indirectly through a type, which must expand to an exact polymorphic variant type, whose tag specifications are inserted in place.

Intersection-like types may be given in closed polymorphic variant types, likely generated by automatic type inference. Such type may be conjunctive, include multiple incompatible class types, and therefore inherently unsatisfiable. Such a constraint signals an error condition.

6.3.8 Tuple Types

Grammar:

```
Simple_Type
    ::= ...
      | '(' Types ')'
Types
    ::= Named_Subtype {',' Named_Subtype}
Infix_Type
    ::= ...
      | Named_Subtype {'*' Named_Subtype}
```

A tuple type (T_1, ..., T_n) is an alias for the class `Tuple[`T_1, ..., T_n`]`, where $n \geq 1$.

A tuple type T_1 `*` ... `*` T_n is an infix type (§6.3.10) representing the exact same tuple type as (T_1, ..., T_n).

Tuple classes are available as patterns for pattern matching. The properties can be accessed as methods `1`, ..., n.

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.

If any of the tuple's type arguments T_i is annotated to be named n (see §6.3.6, then annotation `@[labelled :`n`]` is added to the type argument.

Tuple type with single type parameter is an alias for the type parameter itself. This is true for the form `(`T`)` as well as `Tuple[`T`]`.

6.3.9 Annotated Types

Grammar:

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

An annotated type $a_1 \dots a_n T$ attaches annotations (§15) $a_1, \dots, a_n$ to the type T.

6.3.10 Infix Types

Grammar:

```
Infix_Type
    ::= Type_Expr {[nl] (Infix_Op - 'and also', 'or else') [nl] Type_Expr}
```

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 (§8.3.10), 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.

6.3.11 Unions

Grammar:

```
Infix_Type
    ::= ...
      | Type {'or' 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.

The first syntax shows a named union type, while the second shows an anonymous union type (which may still be given a name later). If any of the types that are a part of a union type is a union type itself, the two types are merged. The syntax used for the anonymous version is correlating with infix type syntax, but this syntax gives it the meaning of a union

type, which is preferred to infix type syntax. In fact, it might be implemented with an infix type that generates union types:

```
type or [A, B] = Union[A, B]
end
```

Union types also employ the following set of rules, given some types `X`, `Y` and `Z`:

- Commutativity: `X or Y` is equivalent to `Y or X`.
- Associativity: `X or (Y or Z)` is equivalent to `(X or Y) or Z`, which is equivalent to `X or Y or Z`.
- Simplification: if `X` conforms to `Y`, then `X or Y` is equivalent to `Y`.
- Conformance: `X` conforms to `X or Y`.
- Supertypes: if both `X` and `Y` conform to `Z`, then `X or Y` also conforms to `Z`.
- `X or Nothing` is equivalent to `X` for any `X`.
- `X or Any` is equivalent to `Any` for any `X`.
- If `X[T]` is covariant in the type parameter `T`, then `X[U] or X[V]` conforms to `X[U or V]` for any types `U` and `V` that satisfy the type constraints of T.
- If `X[T]` is contravariant in the type parameter `T`, then `X[U] or X[V]` conforms to `X[U with V]` for any types `U` and `V` that satisfy the type constraints of T.

6.3.12 Intersection Types

Grammar:

```
Infix_Type
    ::= ...
      | Type_Expr ['with' Type_Expr] {'and' Type_Expr}
```

An intersection type is a type that conforms to all types that it consists of. Any value that is an instance of all the types is also an instance of the intersection type. The second connected type may be optionally separated using "**with**" instead of "**and**".

Intersection types also employ the following set of rules, given some types `X`, `Y` and `Z`:

- Commutativity: `X and Y` is equivalent to `Y and X`.
- Associativity: `X and (Y and Z)` is equivalent to `(X and Y) and Z`, which is equivalent to `X and Y and Z`.

- Simplification: if `X` conforms to `Y`, then `X and Y` is equivalent to `X`.
- Conformance: `X and Y` conforms to `X`.
- Subtypes: if `Z` conforms to both type `X` and type `Y`, then `Z` also conforms to `X and Y`.
- Distributivity over union: `X and (Y or Z)` is equivalent to `(X and Y) or (X and Z)`.
- `X and Nothing` is equivalent to `Nothing` for any `X`.
- `X and Any` is equivalent to `X` for any `X`.
- If `X[T]` is covariant in the type parameter `T`, then `X[U and V]` conforms to `X[U] and X[V]` for any types `U` and `V` that satisfy the type constraints of T.
- If `X[T]` is contravariant in the type parameter `T`, then `X[U or V]` conforms to `X[U] and X[V]` for any types `U` and `V` that satisfy the type constraints of T.

6.3.13 Function Types

Grammar:

```
Type_Expr
    ::= ...
      | Function_Type
Function_Type
    ::= First_Function_Arg_Type {'->' Function_Arg_Type}
        ['->' Signalling_Types] '->' Atomic_Type_Expr
First_Function_Arg_Type
    ::= Function_Arg_Type
      | '(' 'val' Type_Expr ')'
Function_Arg_Type
    ::= [('~' | '~?') id ':' | '*' | '**' | '&'] Atomic_Type_Expr
Signalling_Types
    ::= Signalling_Kind {'and' Signalling_Kind}  -- each kind at most once
Signalling_Kind
    ::= 'signals' (Simple_Type | '(' Type_Expr {',' Type_Expr} ')')
      | 'throws'  (Simple_Type | '(' Type_Expr {',' Type_Expr} ')')
      | 'raises'  (Simple_Type | '(' Type_Expr {',' Type_Expr} ')')
      | 'resignals'
      | 'rethrows'
      | 'reraises'
```

The type `(`T_1`, ..., `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:

- "`*`" maps to `@[variadic]`.
- "`**`" maps to `@[capture_variadic]`.
- "`&`" maps to `@[captured_block]`.
- "`~`*id*" maps to `@[named :`*id*`]`.
- "`~?`*id*" maps to `@[named :`*id*`] @[optional]`.

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` protocol – i.e. having an `apply` function or simply *being* a function. The tuple-destructuring function type `(`T_1`, ..., `T_n`) -> `*R* is a shorthand for the protocol `Function[`T_1 `*` ... `*` T_n`, `*R*`]`. Such protocol is defined in the Aml library:

```
protocol Function[-T, +R]
  message apply (x₁: T): R
  ...
end protocol
```

Function types are covariant in their result type and contravariant in their argument type (§7.5.1).

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`.

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 6.3.6 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 (§8.9.3).

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

6.3.14 Type Class Constraints

Grammar:

```
Type_Context
    ::= Simple_Type '=>'
      | '(' Types ')' '=>'
Given_Dcl
    ::= 'class' Type
```

A function type can additionally express type class constraints, either with the prefix syntax using `Type_Context`, or the postfix syntax using `Given_Clauses`. Each type occurring in either of those is considered to be adding a type class constraint to the function type (§7.9).

Example 6.3.7 A function type that requires that an instance of a type class `Show['a]` is available for the type parameter `'a`.

```
Show['a] => 'a -> 'b for all { type 'a; type 'b }
```

Example 6.3.8 A function type that is equivalent to the previous type, but uses only the postfix syntaxes.

```
'a -> 'b given Show['a] and for all { type 'a; type 'b }
```

Note. A good style is to not combine the two syntaxes in one function type, however, if so used, the constraints are concatenated.

6.3.15 Functional Dependencies

Grammar:

```
Given_Dcl
    ::= ...
      | Simple_Type '~>' Simple_Type
```

A functional dependency expresses dependencies between type parameters of a function type. A functional dependency in form of `'a ~> 'b` expresses the fact that type variable `'a` determines the type variable `'b`. Note that the involved type variables are still unifiable at that point, unless made explicitly polymorphic.

6.3.16 Type Dependencies, Constraints & Equations

Grammar:

```
Given_Dcl
    ::= ...
      | Stable_Id Type_Equation
      | Stable_Id '>:' Type_Expr ['<:' Type_Expr]
      | Stable_Id '<:' Type_Expr
      | Type_Vars ':' '(' Member_Signature {semi Member_Signature} ')'
      | Stable_Id ':' '(' Constructor_Signature ')'
Type_Vars
    ::= Stable_Id
      | '(' Stable_Id {'or' Stable_Id} ')'
Member_Signature
    ::= ['object' | 'class'] 'member' fun_id ':' Type_Expr
      | ['object' | 'class'] 'message' fun_id ':' Type_Expr
      | ['object' | 'class'] 'attribute' fun_id ':' Type_Expr
      | ['object' | 'class'] 'operator' op_id ':' Type_Expr
      | ['object' | 'class'] 'property' id ':' Type_Expr
        ['with' Property_Signature_Extra]
Constructor_Signature
    ::= 'new' ':' Type_Expr  -- just type of parameters, no result type
Property_Signature_Extra
    ::= 'get' ['and' 'set']
      | 'set' ['and' 'get']
```

A type dependency or equation expresses additional relations between different type components (usually type variables used as type parameters) that make up a type that uses the **given** declarations. It can equate two types (e.g. **given** `{'a = 'b}`), or declare that two or three types have a conformance relation (e.g. **given** `{'a >: 'b <: 'c}`).

6.3.17 Existentially Quantified Types

Grammar:

```
Existential_Clauses
    ::= 'for' 'some' '{' Quantification_Dcl {semi Quantification_Dcl} '}'
Quantification_Dcl
    ::= Type_Declaration
      | 'val' id ':' Type_Expr
Type_Arg
    ::= ...
      | Wildcard_Type_Arg
Wildcard_Type_Arg
```

```
::= '?' ['>:' Type_Expr] ['<:' Type_Expr]
```

An existential type has the form T **for some** { Q }, where Q is a sequence of type declarations (§10.1). 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 variable bound in T is also explicitly polymorphic.

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 (§8.3.7) 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 $[t_1 \mapsto t'_1, \ldots, t_n \mapsto t'_n]$ and each t'_i is a fresh abstract virtual type with lower bound σL_i and upper bound σU_i[5]. 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:

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 (§7.5.1) 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.

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

Placeholder syntax for existential types. Aml supports a placeholder syntax for existential types. A *wildcard type* is of the form `?` `>:` L `<:` U and may only appear as a type argument. 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.[6]

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.

Example 6.3.9 Assume the class definitions

```
class Ref['t] = object ... end
abstract class Outer = object
  type T
end
```

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.

[6]Although the syntax for existential types is heavily influenced by Scala, the particular syntax for the placeholder itself was chosen from Java, as the underscore, which is used in Scala, is already used by wildcard type argument for partial type applications.

Example 6.3.10 The type `List[List[_]]` is equivalent to the existential type

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

Example 6.3.11 Assume a covariant type

```
type List[+T] = ... end
```

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]`.

6.3.18 Explicitly Universally Quantified Types

Grammar:

```
Universal_Clauses
    ::= {'for' 'all' '{' Quantification_Dcl {semi Quantification_Dcl} '}'}
```

A type variable bound in *T* is also explicitly polymorphic.

6.3.19 Nullable Types

Grammar:

```
Simple_Type
    ::= ...
      | Simple_Type ? no whitespace ? Nullable_Mod
Nullable_Mod
    ::= '?'
```

[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.

A nullable type has the form T`?`, which is an abbreviation for `Option[`T`]`.

For intersection types (§6.3.12), the nullability modifier has to be applied to the first connected type, i.e. T_1`?` **`with`** T_2 **`and`** $T_3 \ldots T_n$, which is then equivalent the type `Option[`T_1 **`with`** T_2 **`and`** $T_3 \ldots T_n$`]`.

6.3.20 Types Preventing Nil

In Aml, a type class exists that makes every type that implements it conform to a compatible non-nullable type: `Default[`T`]`.

`Default[`T`]` does that by implementing a default value to runtime-replace **`nil`**.

Types `T` that have got the `Default[`T`]` type class implemented have got an implicit conversion from type `Option[`T`]` to the type `T` enabled, rendering `Option[`T`]` compatible with `T`.

6.3.21 Constrained Types

Grammar:

```
Type_Constraint_Clause
    ::= 'where' Constraint_Block
Simple_Type
    ::= ...
      | '@(' Expr ')'
Constraint_Block
    ::= [Param_Clauses '->'] '(' Expr ')'
```

A constrained type constructs a subset of allowed members of its component type. Such a subset is defined as members for which the constraint function returns boolean **`yes`**. Constraints defined within the component type are implicitly available in such function.

Constrained types, in face of overloading, trigger eager argument expression evaluation, so that the constrained type of a parameter can even be checked.[9]

The constraint function is only applied when a value is tested for membership of the constrained type, i.e. at binding time, be it argument type check, result type check or any other. This has implications on type soundness – if the tested value is modified after the binding is done, it may not be a member of the constrained type subset any more, but it is definitely still a member of the component type.[10]

[9] Therefore, constrained types are not really suitable candidates for parameters that are supposed to be lazy-evaluated, unless the early evaluation is intended.

[10] This is because values in `Aml` are not allowed to change their type membership in runtime.

Note. Constrained types are an implementation of *dependent types* in Aml. Dependent types are basically functions from values to types, where the value part is present in the defined constraint and/or the constraint block.

6.3.22 Types with Explicit Eagerness

Grammar:

```
Simple_Type
    ::= ...
      | Eagerness_Type_Prefix ? no whitespace ? Simple_Type
Eagerness_Type_Prefix
    ::= Explicitly_Strict_Type_Prefix
      | Explicitly_Lazy_Type_Prefix
Explicitly_Strict_Type_Prefix
    ::= '!'
Explicitly_Lazy_Type_Prefix
    ::= '~'
```

An *explicitly strict type* `!`*T* is requesting `Aml` to reduce the expression bound to type *T* in a strict way, even when it would resolve to non-strict reduction otherwise.

An *explicitly lazy type* `~`*T* is requesting `Aml` to reduce the expression bound to type *T* in a non-strict way, even when it would resolve to strict reduction otherwise.

6.3.23 Foreign Types

Grammar:

```
Simple_Type
    ::= ...
      | Foreign_Type
Foreign_Type
    ::= 'foreign' lang_name string_literal
lang_name
    ::= id
```

A *foreign type* is a tool that helps Aml programs integrate with other languages written for the same VM, the Aml VM. It is viewed as extending the type `Any`, can be given an alias name and be a part of a compound type with another foreign types or Aml types.

Parameterized foreign types. If the foreign type is parameterized, it can be a part of a type application as well, but its type parameters are not declared neither defined on the Aml side.

Also, the foreign language defines how the parameterization works – e.g., Java does type erasure with its type parameters.

Example 6.3.12 A foreign type aliases for a few foreign types:

```
type Java_Object = foreign Java "java.lang.Object" end type
type Java_int = foreign Java "int" end type
type CS_Object = foreign `C#` "System.Object" end type
type Rb_Proc_Status = foreign Ruby "Process::Status" end type
```

6.4 Base Types & Member Definitions

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

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[11] 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.

[11] E.g. type members.

- 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.
   - 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:

   - 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.
   - All bindings of the type's refinement (§??), 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.

6.5 Any-Value Type

Grammar:

```
Simple_Type ::= 'Any'
-- also:
Simple_Type ::= 'Aml/Language.Any'
```

This type does not represent a single concrete value type, but any concrete type. It is used in places where the actual type does not matter, and could be even from other language running along Aml in the VM.

With respect to overloading resolution (§8.9.1), this type is always the least specific, as there is no other less specific type available.

Structural constraints may be appended to this type as with any other type by using a new compound type (§??), where `Any` is used as a class type (never a trait type).

The `Any` identifier is not reserved for the language as a keyword, instead, it gets usually automatically imported with the `Language` module, and therefore can be shadowed and aliased (e.g. to `Any_Object`).

The `Any` virtual type is the actual root of the type system of Aml, although explicit inheritance is not allowed from it. Other languages that integrate with Aml on the same VM may have `Any` as their root type as well, disobeying the rule that is set only for Aml. This way, Aml preserves its unified type system.

6.6 Dynamic Type

Grammar:

```
Simple_Type ::= 'Dynamic'
-- also:
Simple_Type ::= 'Aml/Language.Dynamic'
```

This type does not represent a single concrete value type[12], but any concrete type, very much like `Any`.

In runtime, this type is always dynamically replaced by the type of the actual referenced value – e.g., if a variable is typed with `Dynamic` type (§6.7.5), assigned a value, and used in a function application, the type of that value is used, not `Dynamic`, unless it would be typed with `Any` again (§8.5.1). Nonetheless, this does not imply that a value of another type can not

[12]This type is similar to what `dynamic` is to C#.

be assigned to that same variable – such variable is still bound to `Dynamic` and accepts `Any` type.

Structural constraints may be appended to this type as with any other type by using a new compound type (§??), where `Dynamic` is used as a trait type.[13]

With respect to overloading resolution (§8.9.3), this type is almost[14] always the most specific, as it is replaced by the actual runtime type of the value it is assigned to.

Typing an expression with this type triggers early evaluation, which is important to know if the expression is an argument expression and its corresponding parameter is lazily evaluated.

If the `Dynamic` type is used as the type annotation of a parameter, it is translated to `Any` (plus additional structural constraints, if any present), and treated as `Dynamic` in the following code.

The `Dynamic` type is disallowed from use within type parameters, result types and conformance check expressions (§8.5.1).

Using `Dynamic` type is one of possible ways to use multi-methods – the type of arguments typed as `Dynamic` are bound at runtime, not during compilation. Moreover, the two approaches can be combined, as not every argument expression needs to be `Dynamic`.

6.7 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.

6.7.1 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`.

[13] Thus, if `Dynamic` is used in a compound type, the compound type can contain a class type that is not `Dynamic`. This is different from `Any`, which can only appear as a class type.

[14] The type annotation would have to 1:1 copy the runtime type to be the same specific, but no explicit type annotation can ever be more specific.

- Two compound types (§??) are equivalent, if the sequences of their components are pairwise equivalent, occur in the same order and their refinements are equivalent.
- Two constrained types (§6.3.21) are equivalent, if their base types are equivalent and their constraint blocks read the same.
- Two refinements (§?? & §11.8) 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 (§??) are equivalent, if they are *override-equivalent* (§7.10.13).
- Two polymorphic method types (§??) are equivalent, if they have the same number of type parameters, the result types are equivalent as well as variances, lower and upper bounds of corresponding type parameters.
- Two existential types (§6.3.17) 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 (§??) are equivalent, if they have the same number of type parameters, the result types are equivalent as well as variances, lower and upper bounds of corresponding type parameters.
- Two unions (§6.3.11) are equivalent, if they have the same number of member types and those types are pairwise equivalent.
- Two intersection types (§6.3.12) are equivalent, if they have the same number of member types and those types are pairwise equivalent.

6.7.2 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, `Undefined` `<:` `Nothing` `<:` T `<:` `Object` `<:` `Any`.
- For every type constructor T with any number of type parameters, `Undefined` `<:` `Nothing` `<:` T `<:` `Object` `<:` `Any`.
- 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 class type or a parameterized type conforms to a union type, iff it conforms to at least one of the union's component 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 constant constructor T of a variant type U (§10.4) conforms to U. If U is a parameterized variant type, T still conforms to U, even though it itself is not parameterized.
- A non-constant constructor T of a variant type U (§10.4) conforms to U. If U is a parameterized variant type, then T conforms to U if T seen as a parameterized type conforms to parameterized U[15].
- 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`** `...` **`and`** T_n conforms to each of its component types T_i.
- Conformance of intersection types is defined in (§6.3.12).
- A constrained type T **`where`** b conforms to T. A type T **`where`** b_1 conforms to T **`where`** b_2, if any value e that conforms T **`where`** b_1 by being a member of T and passing the test presented by b_1, also conforms to T **`where`** b_2 by passing the test presented by b_2.[16]
- 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`** `...` **`and`** U_n.

[15] Quite obviously.

[16] This conformance is only tested based on actual values being tested, not the types themselves.

- The existential type T **for some** { Q } conforms to U, if its skolemization (§6.3.17) 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 (§6.3.17) 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

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

  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 fulfill 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 $T[t_1, \ldots, t_n] <: T'[t_1, \ldots, t_n]$. 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 (§15), 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 (§8.3.4), 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` 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 (§15). 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 union type (§6.3.11) U_1 conforms to U_2, if every member type that is present in U_2 has also an equivalent member type in U_1. U_1 may thus contain more member types than U_2, but must contain all of member types in U_2. See also: (§6.3.11).
- An overloaded type projection conforms to type of every overloaded alternative.

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. Aml 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, Aml is free to pick up any of them.

6.7.3 Least Upper Bound

The term *least upper bound* of a set of types (and the closely related term *weak least upper bound* of a set of types) have two possible representations.

- *Common supertype representation.* This approach selects one compound type that is a common supertype of all types in the set.
- *Union type representation.* This approach simply makes a union type (§6.3.11) out of all types in the set.

Aml implicitly uses the union type representation for least upper bounds. The union type already presents quite a few simplifications of itself. This behaviour can be changed per-scope, per-file or even per-module, using a pragma:

```
pragma least-upper-bound(Union)  -- implicit
pragma least-upper-bound(Common) -- optionally
```

These pragmas are a part of Aml's `Language` module.

6.7.4 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.

6.7.5 Type Consistency

Type consistency is a relation between types, and a part of Aml's *gradual typing*.

Consistency ($\sim$) relation allows an implicit conversion between two types. By writing $T_1 \sim T_2$, we say that T_1 is consistent with T_2 and vice versa. We will denote the `Dynamic` type as **?** (not to be confused with the existential type placeholder).

- For any type T, we have that $T \sim ?$ and also $? \sim T$.
- For every type T, we have $T \sim T$.
- A tuple type (T_1, T_2) is consistent with another tuple type (S_1, S_2) if $T_1 \sim S_1$ and $T_2 \sim S_2$, and so on for tuple types of any other arbitrary size.
- A function type P_1 `->` R_1 is consistent with another function type P_2 `->` R_2 if $P_1 \sim P_2$ and $R_1 \sim R_2$.

6.8 Reified Types

Unlike in Java or Scala, *type erasure* does not exist in Aml. 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 Aml:

- 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 Aml 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.

6.9 Types Representing Emptiness

In Aml, there are a few types and their values that represent emptiness of some sort. The following lists specifies their semantical purposes.

1. The `Nothing` type, which is the bottom type ($\bot$) of Aml's type system.

   It's the only empty type in Aml, i.e. it has no type inhabitants. Aml defines a function that has this type: **`undefined`**, parameterless.

   To signal that a function never returns, the type alias `Diverges` should be used instead, as its name better expresses the intent.

2. The `None` variant of the `Option[`T`]` variant type and represents a missing value.

   The dedicated keyword **`nil`** is the value of that type.

3. The `Unit` type. It is the only `Tuple` type with no type parameters and no data members.

   It's semantical meaning is that no value is even expected. It is used for functions that should never return any meaningful value (but they do return control back to their users), and that are technically procedures, whose purpose lies in side-effects.

The type has just one inhabitant, the "`()`" value, also known as an empty tuple. In a Aml VM, this type is usually an immediate value.

There are no implicit conversions from `Unit` to any other type defined by Aml.

Chapter 7

Common Building Blocks

Contents

Grammar:

```
Reference_Modifier
    ::= 'weak'
      | 'unowned'
Mutability_Modifier
    ::= 'constant'
      | 'mutable'
      | 'immutable'
Storage_Modifier
    ::= Mutability_Modifier [Reference_Modifier]
      | Reference_Modifier
In_Sep
    ::= [nl] 'in' [nl]
      | semi
```

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).

7.1 Value Names

Syntax rules defined in the following sections do not restrict users from choosing whatever value or variable name they want. However, there are a few conventions that are recommended to follow, because otherwise, an inconsistency in naming could arise in face of pattern matching (§9):

- If a value or variable is to contain a type (such as a class), use upper-case first letter in its name (unless it makes more sense to have a lower-case first letter due to the name itself).
- In other cases, where a value or variable is to contain a non-type value, use lower-case first letter in its name, without exceptions.

This is to keep up consistency with pattern matching, where the type will usually be presented with a name that starts with an upper-case letter (but does not need to if necessary), but all bound variables have to start with a lower-case letter, without exceptions. So basically, these conventions follow the restrictions imposed by pattern matching.

7.2 Value Definitions

Grammar:

```
Val_Def
    ::= 'let' ['rec'] [nl] Let_Bindings_Sequence
Let_Bindings_Sequence
    ::= Let_Binding {('and' | semi) Let_Binding}
Let_Binding
    ::= Let_Value_Binding
      | Let_Fun_Binding
Let_Value_Binding
    ::= Pattern [Type_Param_Clause] [':' Type_Expr]
        '=' Expr
Let_Fun_Binding
    ::= ['fun'] fun_id [Type_Param_Clause] [Given_Clauses] Param_Clauses Fun_Type_Expr
        '=' [Function_Declarations] Expr
Fun_Type_Expr
    ::= [(':' | 'returns') Type_Expr] [':>' Type_Expr]
```

A value declaration **`let`** *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.

A value definition **`let`** *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 (§6.7.2).

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

A value definition using the keyword **`let`** is equivalent to value definitions using the keyword **`val`**. Value declarations do not use the keyword **`let`**.

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.[3]

A value definition can alternatively have a pattern (§9.1) as left-hand side (the name). If *p* is a pattern other than a simple name or a name followed by a colon and a type, then the value definition **`val`** *p* `=` *e* is expanded as follows:

[1] Everything in Aml is a value – remember, Aml 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, Aml does not impose such requirement.

[3] 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 $n > 1$:

```
let x' = match e
  when p then (x_1, ..., x_n)
end
let x_1 = x'.1
...
let x_n = x'.n
```

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

```
let x = match e
  when p then x
end
```

3. If p has no bound variables:

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

Example 7.2.1 The following are examples of value definitions.

```
let pi = 3.14159
let pi: Double = 3.14159
let Some (x) = f ()
let Some (x), y = f ()
```

The last three definitions have the following expansions:

```
let x = match f()
  when Some(x) then x
end

let x' = f ()
let x = match x'.1
  when Some (x'') then x''
end
let y = x'.2

let x' = match my_list
  when x :: xs then (x, xs)
end
let x = x'.1
let xs = x'.2
```

The following shorthands are recognized:

A value declaration **`let`** $x_1, \ldots, x_n$`:` 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. Multiple such sequences may appear in a single value definition, then every appearing T defined type of the preceding values without an explicit type.

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.

A value definition **`val`** p_1 ps`:` $T, \ldots, p_n$ ps`:` T `=` e is a shorthand for the sequence of value definitions **`val`** p_1`:` ps `->` $T, \ldots, p_n$`:` ps `->` T `=` e , where ps are parameter sections (§7.10).

A value definition part $*x$`:` T is a shorthand for the type `Sequence[`T`]` of the value name x.

7.3 Property Definitions & Specifications

Grammar:

```
Property_Spec
    ::= 'val' ['(' Prop_Specs ')'] id ':' Type_Expr
Prop_Specs
    ::= Prop_Spec {',' Prop_Spec}
Prop_Spec
    ::= ([Access_Modifier] ('get' | 'set'))
      | 'weak'
      | 'unowned'
      | Mutability_Modifier
Property_Def
    ::= 'let' ['(' Prop_Specs ')'] id [':' Type_Expr] Prop_Impls
Prop_Impls
    ::= [nl] 'with' Prop_Impl {[nl] 'and' Prop_Impl}
      | [nl] 'by' Expr
        [[nl] 'with' Prop_Hook_Impl {[nl] 'and' Prop_Hook_Impl}]
Prop_Impl
    ::= ([Access_Modifier] 'get' Prop_Get_Impl)
      | ([Access_Modifier] 'set' Prop_Set_Impl)
      | ([Access_Modifier] 'val' '=' Expr)
```

A property declaration **`val`** *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 **`let`** *x*`:` *T* **`with get ... and set ...`** introduces a property with a possibly defined initial value of type *T*. Property definition may specify implementation details of the behaviour and storage of a property, but may as well opt-in for auto-generated implementation, by specifying the implementation block as " _ ", which is:

1. Properties defined with **`set`**[4] are stored in mutable instance variables (§??).
2. Properties defined with **`weak`** are stored as weak references. A property **`val`** *x*`:` *T* is stored in an instance of type `Weak_Reference[`*T*`]`.
3. Properties defined with **`unowned`** are stored as unowned references. A property **`val`** *x*`:` *T* is stored in an instance of type `Weak_Reference[`*T*`]` and implicitly unwrapped.
4. The getter and setter, including both implicit and explicit versions, automatically wrap (assignment) or unwrap (evaluation) the corresponding reference type.

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

```
message x (): T
message set_x (y: T): Unit
```

Assignment to properties (`obj.our_property <- new_value`) is translated automatically into a setter function call and reading of properties does not need any translation due to implicit conversions (§8.9).

7.3.1 Property Implementations

```
Prop_Get_Impl
    ::= Specialised_Block_Expr((() | IVar_Param_Clause '->'))
      | '_'
Prop_Set_Impl
    ::= Specialised_Block_Expr((IVar_Param_Clause [Param_Clause] '->'))
      | '_'
IVar_Param_Clause
    ::= Param_Clause
```

[4] 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.

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

If a property setter (**`set`**) does not specify return type, then it is inferred as `Unit`.

Property getter with no parameter does not get access to its instance variable, property getter with one parameter gets access to its instance variable through that single parameter, its type is *`'t`* **`var`**, where *t* is the type of the property.

Property setter with one parameter does not get access to its instance variable, only the new value to be set, property setter with two parameters gets access to its instance variable through the first parameter, its type is *`'t`* **`var`**, where *t* is the type of the property, and the second parameter's type is then the new value to set.

The type of instance variable can be constant, meaning only reading its content is possible through the reference.

Example 7.3.1 The following is a well-formed property declaration and definition:

```
-- declaration:
val (get, set) temporal: Local_Time option

-- definition:
let temporal: Local_Time option
  val = nil
  get { ivar -> !ivar }  -- the default implementation
  set { ivar value -> ivar := value }  -- also a default implementation
```

Properties may be of function types as well as value types, in which case the implementation would return a function value instead of terminal (non-function) value.

7.4 Reference Types

Reference type modifiers are the syntax category `Reference_Modifier`. The implicit modifier would be `strong`, but it does not have a corresponding keyword.[5]

`Aml` provides automatic wrapping and unwrapping of variables that are declared or defined non-`strong` in context where the wrapped type is expected. Strong references are (usually) not wrapped in anything.

Non-`strong` references are intended to break strong reference cycles that would prevent values from being ever released.

[5] This may or may not change in the future.

7.4.1 Mutable & Immutable Storage

Mutability modifiers are the syntax category `Mutability_Modifier`. The implicit modifier would be `unspecified`, but there is not a corresponding keyword.

Mutability modifiers are applied to any values or variables, unlike reference type modifiers.

The **`mutable`** modifier implicitly adds the `@[`**`mutable`**`]` annotation to the type of the value or variable, and likewise, the **`immutable`** modifier implicitly adds the `@[`**`immutable`**`]` annotation to the type of the value or variable; both if not already present in the value's or variable's type. Values returned from function applications are not transitively mutable or immutable based on this modifier, but may be based on the function's declaration.

The **`constant`** modifier does not add any annotation to the value's or variable's type. Marking a value or variable with **`constant`** means that the runtime will not be able modify the object referred to by the value or variable, thus the referred object is seen as if it were **`immutable`**, but other references to the same object (not created from this constant reference) are still able to mutate the object. A *constant value* or a *constant variable* is then a *constant view* of the referred object. Values returned from function applications, where a constant view is the target, are then transitively constant as well. It is not possible to convert a constant view of a value to a mutable reference.

An exception to immutability is presented by the `Reference_Modifier` syntax category and weak, soft and unowned references to objects, where indeed the runtime is allowed to discard the value, and thus the immutability or mutability is transitively applied to the contained type, but not the reference itself.

7.4.2 Strong Reference

This reference does not have a direct representation in Aml, but it can be emulated with the `Strong_Reference[`*T*`]` type.

A *strong reference* is a reference that does keep a strong hold on the value it refers to. As long as there are strong references to a value, it does not get released.

7.4.3 Weak Reference Type

This reference type is represented in Aml by the type `Weak_Reference[`*T*`]`.

A *weak reference* is a reference that does not keep a strong hold on the value it refers to, and thus does not stop automatic reference counting from releasing the referenced value. Such variable may be changed to **`nil`** in runtime at any time without an error, therefore it implies nullable type.

Retrieving the referenced value can end in two different scenarios (and the same applies to all other non-strong reference types): either the value was already destructed (or is being

destructed) and **`nil`** is returned, or the value has been retained and the returned value is a strong reference to it.

Weak reference type is required to be provided by every proper Aml VM implementation.

7.4.4 Unowned Reference Type

This reference type is represented in Aml by the type `Unowned_Reference[`*T*`]`.

Like weak references, an *unowned reference* does not keep a strong hold on the value it refers to, but it assumes that it will always refer to a non-**`nil`** value until itself released, therefore it implies the implicitly unwrapped nullable type. It is an error if the value is accessed in runtime via this reference type and it is already released.

Unowned reference type is required to be provided by every proper Aml VM implementation, and may reuse internal representation of weak references.

7.5 Type Parameters

Grammar:

```
Type_Param_Clause
    ::= '[' [Variant_Type_Param {',' Variant_Type_Param}] ']'
Prefix_Type_Param_Clause
    ::= Variant_Type_Param
      | Atomic_Type_Param_Clause
Atomic_Type_Param_Clause
    ::= '(' Variant_Type_Param {',' Variant_Type_Param} ')'
Variant_Type_Param
    ::= {Annotation} Type_Param_Variance ['...'] Type_Param
      | '<' Type_Var ['<:' Long_Id] '>'
Type_Param_Variance
    ::= ['+' | '-']
Type_Param
    ::= (Type_Var | '_') [Type_Param_Clause] [Type_Abstract]
```

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 (§12).

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* (§7.5.1),

i.e. an optional prefix of either + or -. The type parameter may be preceded by one or more annotation applications (§8.5.4 & §15).

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, and all type parameter clauses that it is nested within. 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.

Type parameters are also named, unless a single type parameter name for some reason denotes several actual type parameters (which can be achieved by using explicit universal or existential quantifications).

Bounds defined for a particular type parameter extend to all of its nested usages, thus it is only necessary and possible to define the bounds once, preferably on the first occurrence of the type parameter.

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

If the names of higher-order type parameters are irrelevant, they may be denoted with a "_", which is nowhere visible and represents a single type parameter that is independent of all other type parameters, including other type parameters that are denoted the same way.

A type parameters clause may contain a variadic type parameter, denoted by the prefix " **...** ", which may appear up to once per one type parameter list, preferably as the last type parameter, and may also appear in the nested type parameter clauses (but always up to once per one list). Such a type parameter clause then captures a tuple of given type arguments (including no type arguments at all – Unit), and in order to expand it, the same prefix operator has to be used. A variadic type parameter may capture arguments of different types (unlike variadic function parameters, which share a common upper bound) – and may be used as a tuple type (not just expanded).

As a consequence of having type parameters visible across nested type parameter clauses, using the same type parameter name gives additional constraint on the parameterized type, requiring the type argument to be physically equivalent among all occurrences of the type parameter.

A type parameter name may optionally be surrounded by 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 7.5.1 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]]  {- requires the polymorphic arguments M and N
                 to have the same type argument, accessible as X -}
[M[_], N[_]]
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
[E, <Length_Unit <: length>]
[+T, -A]
```

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' -}
```

7.5.1 Variance of Type Parameters

Variance annotations indicate how instances of parameterized types relate with respect to subtyping (§6.7.2). 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 (neutral).
- The right-hand side of a type alias is always in invariant position (neutral).
- 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 7.5.2 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 (§11.5) 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 7.5.3 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 7.5.4 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
}
```

Example 7.5.5 Here is a case where a contravariant type parameter is useful.

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

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.

7.6 Explicit Polymorphic Type Annotations

Every time a type parameter *T* appears in the type parameters list, then the type parameter is explicitly polymorphic. This polymorphism can be used in recursive occurrences.

Possible applications of this feature are e.g. polymorphic recursion, and ensuring that some definition is sufficiently polymorphic.

Example 7.6.1 Polymorphic recursion.

```
type T['a] is variant
  Leaf of 'a
  Node of T['a * 'a]
end type
let rec depth ['a] (x: T['a]): 'b = match x
  when Leaf(_) then 1
  when Node(y) then 1 + depth y
end match
```

Note that `'b` is not explicitly polymorphic here, and will be actually unified with `Aml/Language.Number.Integer` (since it is almost equivalent to `Aml/Language.Auto`).

Example 7.6.2 Ensuring that some definition is sufficiently polymorphic.

```
let id ['a] (x: 'a): 'a = x + 1
```

Now type of `x + 1` is required to be polymorphic. The type of the expression `x + 1` is not `Aml/Language.Number.Integer`, because the operator `+` can be implemented for many other different types.

7.7 Locally Abstract Types

Grammar:

```
Param_Clause
    ::= ...
      | Locally_Abstract_Type_Param_Clause
Locally_Abstract_Type_Param_Clause
    ::= '(' 'type' Typedcl {semi Typedcl} ')'
```

A parameter clause `(type` *x*`)` introduces a type constructor named *x*, which is considered abstract in the scope of the function or method it appears in, but then replaced by a fresh type variable of the form `'`*x* outside the function or method, and which is unifiable by (§16).

The implied type variable may be also given some type constraints, like any other type parameters, in the forms of type equation or type abstract constraints.

A parameter clause `(`**`type`** x_1`; ...; `x_n `)` is a syntactic sugar for `(`**`type`** x_1`) ... (`**`type`** x_n`)`.

7.8 Explicitly Polymorphic Locally Abstract Types

Grammar:

```
Variant_Type_Param
    ::= ...
      | {Annotation} 'type' ['+' | '-'] Typedcl
```

Explicitly polymorphic locally abstract types are a combination of (§7.6) and (§7.7).

The locally abstract type syntax form by itself does not make polymorphic the type variable it introduces, but it can be combined with explicit polymorphic type annotations, where needed.

7.9 Type Class Constrained Types

Grammar:

```
Given_Param_Clause
    ::= '(' Given_Clauses ')'
      | '(' 'given' Given_Dcl {semi Given_Dcl} ')'
```

Type class constrained types are used in function definitions and declarations. A *type class constraint* of a form `(`**`given class`** *T*`['`*a*`])` restricts the type *a* to have an instance of the type class *T* (§10.8) in the scope of the function's application. The type *a* is then said to be a *type class constrained type*.

Multiple type class constraints can be specified in a single type class parameter clause.

Then, every application of operations defined by the type class *T* to values of the type class constrained type *a* is applied using the type class instance (be it a function application, a method call or an attribute selection, possibly together with more arguments). In case of a method call, the receiver is unshifted as the first argument.

7.10 Function Declarations & Definitions

Grammar:

```
Message_Dcl
    ::= 'message' fun_id [':' Type_Expr]
Operator_Dcl
    ::= 'operator' op_id [':' Type_Expr]
      | Fixity_Binding
Attribute_Dcl
    ::= 'attribute' id [':' Type_Expr]
Method_Def
    ::= 'method' fun_id Method_Type_Clauses
        Single_Method_Identity Param_Clauses Fun_Type_Expr
        '=' Single_Method_Body
      | 'multi' 'method' fun_id Method_Type_Clauses
        Multi_Method_Identity Param_Clauses Fun_Type_Expr
        '=' Single_Method_Body
      | 'multi' 'method' fun_id Method_Type_Clauses
        Single_Method_Identity Param_Clauses Fun_Type_Expr
        '=' Multi_Method_Bodies
      | 'prefer' ['method'] fun_id Expr 'to' Expr
Operator_Def
    ::= 'operator' op_id Method_Type_Clauses
        Single_Method_Identity Param_Clause Param_Clauses Fun_Type_Expr
        '=' Single_Method_Body
      | 'multi' 'operator' op_id Method_Type_Clauses
        Multi_Method_Identity Param_Clause Param_Clauses Fun_Type_Expr
        '=' Single_Method_Body
      | 'multi' 'operator' op_id Method_Type_Clauses
        Single_Method_Identity Param_Clause Param_Clauses Fun_Type_Expr
        '=' Multi_Method_Bodies
      | 'prefer' 'operator' op_id Expr 'to' Expr
      | Fixity_Binding
Attribute_Def
    ::= 'attribute' id Method_Type_Clauses
        Single_Method_Identity Param_Clause Param_Clauses Fun_Type_Expr
        '=' Single_Method_Body
      | 'multi' 'attribute' id Method_Type_Clauses
        Multi_Method_Identity Param_Clause Param_Clauses Fun_Type_Expr
        '=' Single_Method_Body
      | 'multi' 'attribute' id Method_Type_Clauses
        Single_Method_Identity Param_Clause Param_Clauses Fun_Type_Expr
        '=' Multi_Method_Bodies
      | 'prefer' 'attribute' id Expr 'to' Expr
```

```
Method_Type_Clauses
    ::= [Type_Param_Clause] [Given_Clauses]
Single_Method_Identity
    ::= ['(' 'alias' Expr ')']  -- a constant expression
Multi_Method_Identity
    ::= Single_Method_Identity (Multi_Method_Dispatch | Multi_Method_Select)
Multi_Method_Dispatch
    ::= '(' 'dispatch' [Type_Expr] ['use'
        ('type'    -- each pattern must have a distinct type
        | 'match'  -- will use constant patterns
        | Expr)    -- will use a custom hierarchy
        ] ')'
Multi_Method_Select
    ::= '(' 'when' Pattern ')'
      | '(' 'for'  Expr    ')'  -- a constant expression
Single_Method_Body
    ::= [Function_Declarations] Expr
Multi_Method_Bodies
    ::= [nl] Multi_Method_Body {[nl] 'and' Multi_Method_Body}
Multi_Method_Body
    ::= 'method' Multi_Method_Identity
        '=' Single_Method_Body
      | 'prefer' ['method'] Expr 'to' Expr

Param_Clauses
    ::= {Param_Clause} [Variadic_Param] [Block_Capture_Param] [Implicit_Params]
      | {Param_Clause} [Variadic_Param] [Implicit_Params] [Block_Capture_Param]
Param_Clause
    ::= Value_Param_Clause
      | Locally_Abstract_Type_Param_Clause
      | Given_Param_Clause
Value_Param_Clause
    ::= Atomic_Pattern
      | '~' Reflected_Param_Extra id
      | '~(' Reflected_Param_Extra id [':' Type_Expr] ')'
      | '~' Reflected_Param_Extra id ':' Atomic_Pattern
      | '~' Reflected_Param_Extra id ':' '(' Pattern [':' Type_Expr] ')'
      | '~?' Reflected_Param_Extra id
      | '~?(' Reflected_Param_Extra id [':' Param_Type] ['=' Expr] ')'
      | '~?' Reflected_Param_Extra id ':' Atomic_Pattern
      | '~?' Reflected_Param_Extra id ':' '(' Pattern [':' Type_Expr] ['=' Expr] ')'
      | '(' [Tuple_Params] ')'
      | Record_Params
      | Reflected_Param_Clause
Implicit_Params
    ::= 'implicit' ':' Atomic_Pattern
```

```
      | '(' 'implicit' ':' (Atomic_Pattern | Params) ')'
Block_Capture_Param
    ::= '&' id
      | '&(' id [':' Type_Expr] ')'
      | '~&' id ':' Type_Expr ['as' Var_Pattern]
      | '~&' id ':' '(' Type_Expr ['as' Var_Pattern] ['=' Expr] ')'
Variadic_Param
    ::= Positional_Variadic_Param [Capture_Variadic_Param]
      | Capture_Variadic_Param
Positional_Variadic_Param
    ::= '*(' [Mutability_Modifier] var_id [':' Type_Expr] ')'
      | '*'  [Mutability_Modifier] var_id [':' Atomic_Pattern]
Capture_Variadic_Param
    ::= '**(' [Mutability_Modifier] var_id [':' Type_Expr] ')'
      | '**'  [Mutability_Modifier] var_id [':' Atomic_Pattern]

Params
    ::= Positional_Param {',' Positional_Param}
        [',' Rest_Params {',' Positional_Param}]
      | Rest_Params {',' Positional_Param}
Positional_Param
    ::= Pattern
      | Param_Extra id [':' Param_Type]
Rest_Params
    ::= Optional_Param {',' Optional_Param} [',' Rest_Extract]
      | Rest_Extract
Optional_Param
    ::= Optional_Extract
      | '?' Param_Extra id [':' Param_Type]
      | '?(' Param_Extra id [':' Param_Type] ['=' Expr] ')'
Record_Params
    ::= '{' Record_Param {semi Record_Param}
            [semi Record_Rest_Params | semi '_' [':' Type_Expr]] '}'
      | '{' Record_Rest_Params '}'
      | '{' '_' [':' Type_Expr] '}'
Record_Param
    ::= Stable_Id '=' Positional_Param
      | Positional_Param
Record_Rest_Params
    ::= Optional_Record_Param {',' Optional_Record_Param} [',' Capture_Extract]
      | Capture_Extract
Optional_Record_Param
    ::= Stable_Id '=' Optional_Param
      | Optional_Param
Param_Extra
    ::= [Mutability_Modifier]
```

```
Reflected_Param_Extra
    ::= ['reflect'] Param_Extra
Param_Type
    ::= ['->'] Type_Expr

Function_Declarations
    ::= [nl] 'declare' Fun_Dec_Exprs In_Sep
Fun_Dec_Exprs
    ::= Fun_Dec_Expr {semi Fun_Dec_Expr}
Fun_Dec_Expr
    ::= {Annotation} Dcl
      | ('transparent' | 'opaque')
      | 'pure'
      | 'native' [Expr]
      | 'implicit' id
      | Signalling_Types
      | {Annotation} 'in'  '{' Expr '}'
      | {Annotation} 'out' '{' [id '->'] Expr '}'
      | Expr
      | ()
```

A function declaration has the form of `def` *f psig*`:` *T*, where *f* is the function's path, *psig* is its parameter signature and *T* is its result type.

A function definition **`method`** *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 $(ps_1)\ldots(ps_n)$. Such a declaration or definition introduces a value with a (possibly polymorphic) method type, whose parameter types and result types are as given.

The type of the function body is expected to conform (§6.7.2) 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.

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

7.10.1 Function Path

A *function path* is a reference to the function being declared or defined. It can appear in these forms:

1. Function identifier (`fun_id`), which references a name in the directly enclosing scope.[6]
2. Self function identifier (**`self`**`.fun_id`), which references a name in the metaclass of whatever **`self`** references in the scope.[7]
3. Free function identifier (`Long_Id.fun_id`), which references a name in the metaclass of any object referenced by the `Long_Id` part.

7.10.2 Automatically Curried Function Definitions

In Aml, every function may be seen as a function from one parameter to a result value. Thus, functions that define multiple parameters can be seen as functions that return another function of the second parameter and so on. However, Aml also allows overloading of functions, and therefore the actual number of declared parameters is important for function applications (§8.3.4).

7.10.3 Function Declarations with Function Types

As a special useful case, function declarations that have their `Param_Clauses` empty, but not the last type part, and that last type part is of a `Function_Type` kind, then the declaration is not for a parameter-less function returning a function as defined by the function type, but instead, the function type is the type of the declared function. This is especially useful, because one does not have to repeat all pattern matching (i.e. parameter destructuring) names in the declaration, and instead leave that to the definition.

To actually declare a parameter-less function that returns a function, just specify `()` as parameters in `Param_Clauses`.

Note. This does not apply to function definitions, as these obviously need to know the parameter names.

7.10.4 Function Parameters

A value parameter clause *ps* consists of zero or more formal parameter bindings, which bind value parameters and associate them with their types.

Value parameters are of these forms:

[6] Such scope is usually in a (class) template body, or nested in another function's body.

[7] E.g., in a class template definition, it references a class object function name.

1. Positional parameters (§7.10.6).
2. Labelled parameters (§7.10.7).
3. Optional parameters (§7.10.8).
4. Variadic parameters (§7.10.9).
5. Block capture parameter (§7.10.10).
6. Implicit parameter (§7.10.12).

Value parameter kinds may be freely reordered to the extent allowed by the `Param_Clauses` grammar element: positional, labelled and optional parameters may appear in any order, followed by variadic, block capture and implicit parameters, in that order.

Locally abstract types (the `Locally_Abstract_Type_Param_Clause` grammar element, §7.7) are not real parameters and do not play such role in function applications (§8.3.4), but instead, they influence the typing of the other parameters.

7.10.5 Parameter Evaluation Strategies

Note: This section applies (from the other side of the wall) to function applications (§8.3.4) as well, but it's pointless to have it duplicated over there.

Aml utilizes five argument evaluation strategies. Every strategy defers evaluation of argument values until the function application is resolved, using only arguments' expected types – see (§8.9.3).

Evaluations strategies are only applied to arguments as whole, not to any possible sub-expressions or nested data.

Call-by-value, **strict**. Also known as *call-by-object*, *call-by-object-sharing* or *call-by-sharing*, is a strategy such that arguments are evaluated prior function invocation and are not re-evaluated. This strategy may be inferred by `Aml`, or enforced by using explicitly strict type.

Call-by-reference, **strict**. Also known as *pass-by-reference*, the function can modify the original argument variable. Those are only indirectly supported in `Aml`, by usage of wrapper objects. Those are described in (§7.10.5.1).

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 (§7.10.5.2).

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. Those are described in (§7.10.5.3).

7.10.5.1 By-Reference Parameters

It is possible to get a parameter by reference simply by using the constructor pattern "**`ref`** *x*". The corresponding argument has to be a reference cell.

7.10.5.2 By-Name Parameters

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 implicit parameters (§12.2).

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.

7.10.5.3 By-Need Parameters

A parameter definition may be denoted by a parameter-less function type, or an explicitly lazy parameter type. This indicates that the corresponding argument is explicitly not evaluated before function application, but instead is evaluated the first time used within the function.

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.

7.10.6 Positional Parameters

Positional parameters are parameters that were given no explicit label, and thus it is not possible to pass arguments to them by label.

Positional parameters are of the following forms:

1. `Atomic_Pattern`, the most basic form, which uses atomic patterns to extract and bind data from its corresponding argument.

2. (*params*), a form that uses an augmented form of tuple extractions, and can indeed extract also from a tuple of size 1, which corresponds to a single argument.

The second form is augmented by means of the `Param_Extra` and `Param_Type` grammar elements, which give the tuple extraction the possibility to override some properties of the matched tuple's elements. Those augmentations are also present in the other parameter kinds, but will be described here. This utilizes the fact that arguments passed directly or as tuple elements in a function application (§8.3.4) have also deferred evaluation (unless they are already of evaluated forms).

`Param_Extra` can specify:

1. Whether arguments are lazily or eagerly evaluated, by default eagerly.
2. What is the mutability modifier of the top-level variable bound by the parameter, if there is any such variable.
3. Whether the bound variable is itself mutable (**`var`**) or immutable (**`val`**), or whether its type is implicitly wrapped in a `Reference_Cell` (then **`val`** is implied for the variable that holds the reference cell).

`Param_Type` can specify whether the parameter is evaluated in a by-name strategy, or another strategy (by-value, by-reference, by-need).

Note. No two parameters (of any kind, not just positional) can bind their input argument or a value nested in the argument to the same variable name.

7.10.7 Labelled Parameters

Labelled parameters are of the following forms:

1. ~*param-extra id*, which matches arguments of variable type and binds them to variable named *id*.
2. ~(*param-extra id*: *T*), which matches arguments conforming to the type *T* and binds them to variable named *id*.
3. ~*param-extra id*: *p*, which matches arguments that are matched by the pattern *p* and binds them to variable named *id*, but only if *p* does not already bind the argument to a variable.
4. ~*param-extra id*: (*p*: *T*), which matches arguments of variable type or the type *T*, if that is present, and are matched by the pattern *p*, and binds them to variable named *id*, but only if *p* does not already bind the argument to a variable.

It is possible for a single method to have multiple parameters bearing the same label – in which case, it is important that every such parameter binds the argument to a distinct variable.

7.10.8 Optional Parameters

Optional parameters are of the following forms:

1. `~?`*param-extra* *id*, which matches arguments of variable type and binds them to variable named *id*, typed `Option[`*T*`]`, where *T* is the argument's type.
2. `~?(`*param-extra* *id*`:` *T* `=` *e*`)`, which matches arguments of variable type or the type *T*, if that is present, and binds them to variable named *id*, typed `Option[`*T*`]`, where *T* is the parameter's type, if *e* is not present, and typed just *T* otherwise.
3. `~?`*param-extra* *id*`:` *p*, which matches arguments that are matched by the pattern *p* and binds them to variable named *id*, but only if *p* does not already bind the argument to a variable.
4. `~?`*param-extra* *id*`: (`*p*`:` *T* `=` *e*`)`, which matches arguments of variable type or the type *T*, if that is present, and are matched by the pattern *p*, and binds them to variable named *id* (but only if *p* does not already bind the argument to a variable), and typed `Option[`*T*`]`, where *T* is the parameter's type, if *e* is not present, and typed just *T* otherwise.

Optional parameters in overloading. A method that has equivalent positional, labelled, variadic and block capture parameters to another (overloaded) method can not be distinct just by having different optional parameters (or none at all).

7.10.9 Variadic Parameters

Variadic parameters are in fact two possible parameters, that do not have directly mapped arguments, and can appear independently. Each combination of their presence yields different values being bound to variables they define.

The two kinds are:

1. Positional variadic parameter (starting with single asterisk "`*`").
2. Capture variadic parameter (starting with two asterisks "`**`").

The combinations are:

- Only positional variadic parameter is present. All extra arguments are in order of their appearance bound to a `Sequence[`*T*`]` into a variable defined by the parameter. *T* is a variable type, if not defined by the parameter. Labelled arguments are not applicable.
- Only capture variadic parameter is present. All extra arguments are in order of their appearance bound to a `Dictionary[Symbol,`*T*`]`, where *T* is a variable type, if not defined by the parameter. Positional arguments are mapped to a symbol containing the integer number of their order of appearance, interleaved with labelled arguments.
- Both positional and capture variadic parameters are present. Then the positional parameter only binds positional arguments and the capture parameter only binds labelled arguments.

7.10.10 Block Capture Parameter

A block capture parameter binds a block argument to a variable. The block argument may optionally be passed as a regular labelled argument, in which case it cuts that argument off the capture variadic parameter's arguments.

7.10.11 Reflected Parameter

Grammar:

```
Value_Param_Clause
    ::= ...
      | Reflected_Param_Clause
Reflected_Param_Clause
    ::= '(' 'reflect' Params ')'
```

7.10.12 Implicit Parameter

Implicit parameter is described in (§12.2). An implicit parameter does not yield a distinct overloaded method, very much like optional parameters. An implicit parameter appears annotated with `@[implicit]` in the corresponding function type.

7.10.13 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 number of parameters with equivalent types and kinds (positional, labelled, optional etc.), and equivalent result 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 parameter lists[8] and the same parameters within them with equivalent types, and a result type that conforms to result 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.

7.11 Local Fixity Definitions

Grammar:

```
Let_Binding
    ::= ...
      | Fixity_Binding
Fixity_Binding
    ::= Fixity_Id Operator_Ids
Fixity_Id
    ::= 'nonfix'
      | 'infix'  [integer_literal]
      | 'infixl' [integer_literal]
      | 'infixr' [integer_literal]
Operator_Ids
    ::= op_id {',' op_id}
```

Fixity definition can locally change the fixity of given operators, or locally make operators out of the given identifiers, or remove operator status of given identifiers.

The fixities are defined as follows:

- **`infix`** is non-associative.
- **`infixl`** is left-associative.
- **`infixr`** is right-associative.
- **`nonfix`** removes operator status.

Fixities can also be defined in place of operator declarations or definitions, as defaults for the given operator identifiers.

[8] The **`implicit`** modifier does not make a parameter list different in this matter.

7.12 Overloaded Declarations & Definitions

If two member or entity definitions bind to the same name, but do not override each other at the same time, the member or entity is said to be *overloaded*, each member or entity is said to be an *alternative*, and overloading resolution (§8.9.1) needs to be applied to select a unique alternative.

Overloaded members do not need to appear in the same scope, an overloading member may appear e.g. in a subclass and never in the parent class. Overloaded entities however need to appear in the same scope (e.g. two local function definitions – because the names are shadowed in enclosing scopes).

7.13 Use Clauses

Grammar:

```
Use_Clause
    ::= Import_Clause
      | Open_Clause
Import_Clause
    ::= 'use' ['lazy'] (Type_Expr | Stable_Id) ['.' Import_Expr]
Open_Clause
    ::= 'open' (Type_Expr | Stable_Id) ['.' Import_Expr]
      | 'open!' (Type_Expr | Stable_Id) ['.' Import_Expr]
Import_Expr
    ::= Single_Import
      | '{' Import_Exprs '}'
      | '_'
Import_Exprs
    ::= Single_Import {',' Single_Import} [',' '_']
Single_Import
    ::= id ['as' (id | '_')]

Lambda_Declarations
    ::= [Closure_Usage] Function_Declarations
      | Closure_Usage In_Sep
Closure_Usage
    ::= 'use' Capture_List
Capture_List
    ::= Capture_Items {'and' Capture_Items}
Capture_Items
    ::= (id {',' id} | Let_Value_Binding) 'as' Reference_Modifier
```

A use clause has the form **use** *p*.*I*, where *p* is a path to the containing type of the imported entity, and *I* is an import expression. The import expression determines a set of names (or just one name) of *importable members*[9] 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 (§11.5). The most general form of an import expression is a list of *import selectors*

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

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

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

If the target name in an import selector is a wildcard, the import selector hides access to the source member. For instance, the import selector x_i **as** _ "renames" x to the wildcard symbol, which basically means discarding the name, since _ is not a readable name[10], 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).

The use clause **open** *p* is similar to **use** *p*.{_}, just the precedence of bindings introduced by it is the same that of explicit **use** clauses. It is an error if any of those bindings shadow names that are already present in the scope – for which the alternative form **open!** *p* shadows those names without any error.

The use clause **open** *p*.*I* then expands into the form without import expression for every imported name from *p*, including those imported by wildcard import. The **open!** form of it

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

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

can again shadow names that are already present in the scope, but it must not shadow any name it itself imports into the scope.

In case the imported member is a function, its name does not shadow any already imported functions, instead, it adds itself to the scope as an overloaded variant of that name. It is still possible to shadow the same name with a following import though, if that name would not be a function.

Example 7.13.1 Consider the object definition:

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

Then the block

```
{ use M.{one, z as zero, _} in add (zero, one) }
```

is equivalent to the block

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

A dynamic use clause has the form **`use lazy`** *p*`.`*I*, where *p* is a path to the containing type of the imported entity, and *I* is an import expression. The difference from regular use clauses is that a dynamic use clause can import anything, including dynamically created members. In case of multiple dynamic imports providing the same name, the last one to be provided is preferred, and has to be type-compatible with any possibly previously provided name, i.e., it has to override the previously provided name as if it were a regular member. Dynamic use clause can also import a name that does not exist yet in compile time (by not using wildcard import).

Note that when importing names via use clauses (or dynamic use clauses), the prefix *p* of it is always a selection, but never an application. If a name in selection denotes several possible members, there is no way to use overloading resolution on it, other than that provided by type application.

7.14 Local Bindings

Grammar:

```
Let_Binding
```

```
    ::= ...
      | Local_Bindings
Local_Bindings
    ::= Local_Structure_Binding
      | Open_Clause
      | 'local' Let_Bindings_Sequence 'in' Let_Bindings_Sequence ['end']
Local_Structure_Binding
    ::= 'structure' id {'(' [id ':' Struct_Type] ')'}
        [':' Struct_Type] '=' Struct_Expr
```

Chapter 8

Expressions

Contents

Grammar:

```
Expr
    ::= Simple_Expr
      | Application_Expr
      | Object_Creation_Expr
      | Standalone_Object_Expr
      | Type_Application_Expr
      | '(' Expr ')'
      | 'begin' Expr 'end'
      | Infix_Expr
      | Conditional_Expr
      | Parallel_Eval_Expr
      | Match_Expr
      | Quasiquote_Expr
      | Quote_Expr
      | Metaclass_Access
      | Message_Data_Expr
      | Workflow_Expr
      | Rescue_Expr
      | Catch_Expr
      | Type_Ref_Expr
      | Loop_Expr
      | Try_Expr
      | Signal_Expr
      | Raise_Expr
      | Throw_Expr
      | Return_Expr
      | Rebind_Expr
      | Update_Expr
      | Yield_Expr
      | Annotated_Expr
      | Cast_Expr
      | Use_Expr
      | Delayed_Expr
      | Var_Expr
      | Ref_Expr
      | Jump_Expr
      | Function_Expr
      | Binding_Expr
      | Tuple_Exprs
      | Method_Expr
      | Structure_Ref_Expr
      | Member_Constraint_Invocation_Expr
      | Expr semi Expr
      | Scope_Stat
```

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

8.1 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 Aml is in fact yet another kind of an expression, and those expressions themselves always have a type (usually `Unit`) and a value.

8.2 Data Expressions

8.2.1 Simple Constant Expressions

Grammar:

```
Simple_Expr
    ::= Constant_Expr
Constant_Expr
    ::= Literal_Expr
```

Typing of literals is as described in (§3.7); their evaluation is immediate.

Aml guarantees that methods of numeric literals (the class `Number`) always terminate, are idempotent, and do not have any observable side effects.

8.2.2 The Nil and Undefined Values

Grammar:

```
Constant_Expr
    ::= ...
      | 'nil'
      | 'undefined'
```

The **`nil`** value is of type `Option[`*T*`]`, the `None` variant, and is thus compatible with every type that is nullable (§6.3.19).

The **`nil`** represents a "no value", and is itself represented by an object (with a possible immediate representation).

A reference to any member of the **`nil`** object causes `Member_Not_Found` to be raised, unless the member in fact exists. The **`nil`** object is also frozen by default.

The **`undefined`** expression is of type `Nothing`, and is therefore compatible with every type whatsoever.

The **`undefined`** represents "nothing, not even no value", and is used in computations that never finish (e.g. always end by raising an error, or transferring control somewhere else).

See (§6.9) for more details on "nothing" values.

8.2.3 List Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | List_Expr
Words
    ::= Word {{? Unicode whitespace ?}+ Word}
Word
    ::= {id_char}+
List_Expr
    ::= '[' [Expr {semi Expr}] ']'
      | '%[' [Expr {semi Expr}] ']'
      | '%w[' [Words] ']'
```

Expressions of the forms `[`e_1`; ...; `e_n`]`, `%[`e_1`; ...; `e_n`]` and `%w[`w_1 ... w_n`]` are list expressions. Specifically,

- `[`e_1`; ...; `e_n`]` is the most versatile list expression. The elementary type of it is `'a list`, but can be inferred to any other sequence type, including indexed and linear sequences.
- `%[`e_1`; ...; `e_n`]`, on the other hand, hints that the type is `Linear_Sequence['a]`.
- `%w[`w_1 ... w_n`]` is a list of words expression, which hints that its type is `Linear_Sequence[String]`. Each `Word` in the expression is represented by one `String` element of the list.

8.2.4 Array Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Array_Expr
      | Vector_Expr
Array_Expr
    ::= '[|' [Expr {semi Expr}] '|]'
      | '%[|' [Expr {semi Expr}] '|]'
      | '%w[|' Words '|]'
Vector_Expr
    ::= '#[' [Expr {semi Expr}] ']'
```

Expressions of the forms `[|` e_1`;` `...;` e_n`|]`, `%[|` e_1`;` `...;` e_n`|]` and `%w[|` w_1 `...` w_n`|]` are array expressions. Specifically,

- `[|` e_1`;` `...;` e_n`|]` hints that the expression's type is `Indexed_Sequence['a]`, and unless constrained to another type, is implemented by `Array['a]`.
- `%[|` e_1`;` `...;` e_n`|]` is equivalent to the previous form in `Aml/Core`, but may have semantics different in `Aml/System`.
- `%w[|` w_1 `...` w_n`|]` is an array of words expression, which hints that its type is `Indexed_Sequence[String]`.

An expression of the form `#[`e_1`;` `...;` e_n`]` is a vector expression, typed `Vector['a]`.

8.2.5 Dictionary Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Dictionary_Expr
Dictionary_Expr
    ::= '%{' [Dict_Mapping1 {semi Dict_Mapping1}] '}'
      | '%{' [Dict_Mapping2 {semi Dict_Mapping2}] '}'
      | '#{' [Dict_Mapping1 {semi Dict_Mapping1}] '}'
      | '#{' [Dict_Mapping2 {semi Dict_Mapping2}] '}'
Dict_Mapping1
    ::= Expr '=' Expr
Dict_Mapping2
    ::= plain_id ':' Expr
      | '"' {int_string_element} '":' Expr
```

An expression of the form `%{`e_1`; ...; `e_n`}` is a dictionary expression.

8.2.6 Multimap Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Multimap_Expr
Multimap_Expr
    ::= '%{|' [Dict_Mapping1 {semi Dict_Mapping1}] '|}'
      | '%{|' [Dict_Mapping2 {semi Dict_Mapping2}] '|}'
      | '#{|' [Dict_Mapping1 {semi Dict_Mapping1}] '|}'
      | '#{|' [Dict_Mapping2 {semi Dict_Mapping2}] '|}'
```

An expression of the form `%{|`e_1`; ...; `e_n`|}` is a multimap expression.

8.2.7 Bag Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Set_Expr
      | Bag_Expr
Set_Expr
    ::= '%(' [Expr {semi Expr}] ')'
      | '#(' [Expr {semi Expr}] ')'
Bag_Expr
    ::= '%(|' [Expr {semi Expr}] '|)'
      | '#(|' [Expr {semi Expr}] '|)'
```

An expression of the form `%(`e_1`; ...; `e_n`)` is a set expression.

An expression of the form `%(|`e_1`; ...; `e_n`|)` is a bag expression.

8.2.8 Polymorphic Variant Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Tag_Name Expr
```

The expression `` `id e `` evaluates to the polymorphic variant value whose tag is *id*, and whose argument is the value of evaluated expression *e*.

8.2.9 Record Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Record_Expr
Record_Expr
    ::= '{' [nl] Record_Fields_Init [nl] '}'
Record_Fields_Init
    ::= Record_Row
      | Expr 'with' Record_Fields ['without' Record_Labels]
      | Expr 'without' Record_Labels}
Record_Fields
    ::= Record_Field [semi Record_Field]
Record_Field
    ::= Record_Label [':' Type_Expr] ['=' Expr]
      | var_id
      | Record_Include_Fields  -- can appear up to 1 time in Record_Fields
Record_Include_Fields
    ::= '_' '=' Expr
      | 'include' Expr
Record_Label
    ::= Stable_Id
      | decimal_numeral
Record_Labels
    ::= Record_Label [semi Record_Labels]
```

An expression of the form

$$\{\ i_1\ =\ e_1;\ \ldots;\ i_n\ =\ e_n\ \}$$

is a record creation expression.

8.2.9.1 Anonymous Record Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Anonymous_Record_Expr
```

```
Anonymous_Record_Expr
    ::= '{|' [nl] Record_Fields_Init [nl] '|}'
```

8.2.10 Tuple Expressions

Grammar:

```
Expr
    ::= ...
      | Tuple_Exprs
Tuple_Exprs
    ::= Expr {',' Expr}+
Constant_Expr
    ::= ...
      | '()'
```

A tuple expression $e_1, \ldots, e_n$ is an alias for the class instance creation `Tuple(`$e_1, \ldots, e_n$`)`, 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.

Tuple expressions have the runtime type of `Aml/Language.Tuple[`$T_1, \ldots, T_n$`]`, where each T_i is the known type of each e_i, including the annotations that give label to sub-expressions. Tuple expressions prepended by **`mutable`** are of the type `Aml/Language.Mutable_Tuple[`$T_1, \ldots, T_n$`]`.

Tuple expressions with an extended grammar rules are also used within function applications (§8.3.4), and the rules considering element labelling and positions are the same as for the simple data tuple expressions.

8.2.10.1 Tuple Clone Expressions

An expression of the form

`{` e **`with`** i_1 `=` e_1`;` `...`; i_n `=` e_n `}`

where $i_1, \ldots, i_n$ are decimal non-negative numbers, is a tuple clone expression.

8.2.11 Object Creation Expressions

Grammar:

```
Expr
    ::= ...
```

```
      | Object_Creation_Expr
Object_Creation_Expr
    ::= 'new' Type_Expr
      | 'new' ? no whitespace ? Type_Args Type_Expr
      | 'new' '.' Type_Args Type_Expr
```

8.2.12 Object Clone Expressions

Grammar:

```
Simple_Expr
    ::= ...
      | Object_Clone_Expr
Object_Clone_Expr
    ::= '{<' Object_Ivars_Init '>}'
Object_Ivars_Init
    ::= Object_Ivar_Init {semi Object_Ivar_Init}
Object_Ivar_Init
    ::= id ['=' Expr]
```

An expression of the form

```
{< i_1 = e_1; ...; i_n = e_n >}
```

is an object clone expression.

8.2.13 Variable & Ref Expressions

Grammar:

```
Var_Expr
    ::= 'var' Expr
Ref_Expr
    ::= 'ref' Expr
```

Variable expressions of the form **ref** *e* are a syntax sugar for:

```
Aml/Language.Variable_Cell.new (e)
```

Ref expressions of the form **ref** *e* are a syntax sugar for:

```
Aml/Language.Reference_Cell.new (e)
```

Variable expressions can be explicitly converted to ref expressions and vice versa, but never implicitly.

The cell instances are not automatically unwrapped. The contained value can be retrieved from a reference cell using the prefix operator "`!`"[1], or by sending the instance a "`value`" message. The contained value can be overwritten by using the infix operator "`:=`".

Reference cells are lightweight instances that can be used to simulate output parameters at low costs, in both CPU cycles and the language design.

Variable expressions are not allowed as return values of functions and can not be captured in closures, unless converted to reference cells.

Example 8.2.1 An example of how to manipulate reference cells.

```
let
  -- initialize a reference cell
  val a = ref 42
  val b = Aml/Language.Reference_Cell (42)

  -- retrieve contained value from reference cell
  val c = !a
in
  -- change the value the reference cell contains
  a := 64
end
```

Variable cells are manipulated with the same operators.

Example 8.2.2 An example of how pattern matching differs for variable expressions and ref expressions.

```
val Reference_Cell = object
  method unapply ['a] (cell: 'a ref) : 'a ref option = Some cell
end

val Variable_Cell = object
  method unapply ['a] (value: 'a) : 'a option = Some $ var value
end
```

Reference cells are used to "break boundaries" of function applications, to share a mutable value across function applications, whereas variable cells are meant to define locally or object-locally mutable values.

[1] This operator is not used in the meaning of **`not`**, because accidentally overlooking it does not invert the value that it's applied to.

8.2.14 Arguments Expressions

Grammar:

```
Args_Expr
    ::= 'arguments' [nl] '{' Arguments [Block_Argument] '}'
      | 'arguments' [nl] '{' Block_Argument '}'
```

Arguments expressions of the form **`arguments`** $a_1 \ldots a_n$ generate a special value, which is equivalent to what is passed as arguments in function applications. A value x generated with this expression can be used in a function application using "$**x$". The type of such expression is a tuple with each element representing one argument, where the argument itself may again be a tuple (or any other value).

8.2.15 Message Data Expression

Grammar:

```
Message_Data_Expr
    ::= 'message' '{' Expr '}' 'with' Args_Expr
      | 'message' symbol_literal 'with' Args_Expr
      | Args_Expr
```

A value x generated with this expression can be used to send an object a message using e.g. "`obj.'send` x". In the first form, the expression is typed with expected type of `Symbol`.

8.2.16 Workflows

Grammar:

```
Workflow_Expr          ::= Expr '{' Workflow_or_Range_Expr '}'
Workflow_or_Range_Expr ::= Workflow | Short_Workflow | Range_Expr

Workflow
    ::= 'let!' Let_Binding semi Workflow
      | ['implicit'] Val_Def In_Sep Workflow ['end']
      | ['implicit'] Op_Def In_Sep Workflow ['end']
      | ['implicit'] Multi_Op_Def In_Sep Workflow ['end']
      | 'let' Struct_Defs In_Sep Workflow ['end']

        -- because Do_Binding can't be the last element of a workflow:
      | 'do' Expr In_Sep Workflow ['done']
      | 'do!' Expr In_Sep Workflow ['done']
```

```
      | 'use!' [Storage_Modifier] Let_Binding In_Sep Workflow ['end']
      | 'use' [Storage_Modifier] Let_Binding In_Sep Workflow ['end']
      | 'yield!' Expr
      | Yield_Expr
      | 'return!' Expr
      | Return_Expr

      | 'if' Wf_Condition ('then' | semi) Workflow
        [[semi] Else Workflow] 'end'
      | 'unless' Wf_Condition ('then' | semi) Workflow
        [[semi] Else Workflow] 'end'

        -- Match_Expr where When_Expr is extended with Workflow
      | Wf_Match_Expr
        {- Catch_Expr or Rescue_Expr, where initial Expr,
        Handle_Block, Catch_Block and Rescue_Block are extended with Workflow -}
      | Wf_Catch_Expr
      | Wf_Rescue_Expr

      | 'for' Val_Dcls 'in' ['reverse'] Expr
        ['step' Expr] Wf_For_Loop
      | ('while' | 'until') Condition Wf_For_Loop
      | 'for' id '=' Expr (['up' | 'down'] 'to') Expr Wf_For_Loop
      | Workflow ('while' | 'until') Condition

      | Workflow semi Workflow
      | Annot_Workflow
      | Expr

Annot_Workflow
    ::= {Annotation}+ Workflow
Short_Workflow
    ::= 'for' Generator_Expr
      | [Label_Dcl] 'for' Val_Dcls 'in' Expr
        ['step' Expr] Wf_For_Loop

Wf_For_Loop
    ::= 'loop' Wf_Loop_Block 'end'
Wf_Loop_Block
    ::= Workflow + Loop_Ctrl_Expr
Wf_Condition
    ::= Workflow - Conditional_Expr
Wf_Match_Expr
    ::= Match_Expr   -- see Match_Expr for extensions
Wf_Catch_Expr
    ::= Catch_Expr   -- see Catch_Expr for extensions
```

```
Wf_Rescue_Expr
    ::= Rescue_Expr  -- see Rescue_Expr for extensions
```

8.2.17 Collection Comprehensions

Grammar:

```
List_Literal
    ::= '%' List_Flags '['  Collection_Gen  ']'
Array_Literal
    ::= '%' Array_Flags '[|' Collection_Gen '|]'
Dictionary_Literal
    ::= '%' Dict_Flags '{'  Collection_Gen  '}'
      | '%' Dict_Flags '{{'  Collection_Gen  '}}'
Bag_Literal
    ::= '%' Bag_Flags '('  Collection_Gen  ')'
Collection_Gen
    ::= (Workflow_Expr | Expr) ['in' Range_Expr]
      | Short_Workflow
      | Range_Expr
```

Collection comprehensions extend the syntax of collection "literals"[2], so that collections may be defined not by 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, or, if the actual dictionary type defines its own entry type, values that are members of the entry type.

If the `Workflow_Expr` inside the collection literal uses `Seq_Literal` or similar on-demand computations, the collection literal value is also a cache for its results.

If the expression that generates values would create an infinite collection, the **`in`** `Range_Expr` specification can be used to constrain the generated values to the given range.

8.2.18 Sequence Comprehensions

Grammar:

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

```
Collection_Literal
    ::= Seq_Literal
Seq_Head
    ::= 'sequence'
Seq_Literal
    ::= Seq_Head '{' Collection_Gen '}'
```

Apart from collection comprehensions (§8.2.17), Aml offers also sequence comprehensions. The crucial difference is in evaluation: collection comprehension trigger eager evaluation, unless indeed the used source collection does not employ lazy evaluation of some kind by itself. Sequence comprehension, on the other hand, is super-lazy.

A sequence generated by the form **`sequence`** { *e* } works like a forgetting enumerator – it computes values using the expression *e* only when they are required. This is most useful for very large sequences, and/or when the computation is expensive in some way.

The expression *e* may be of a few different kinds:

- A `Range_Expr`, in which case the sequence is based on a given range, optionally including a **`delta`**, which then defines the increment between each value (and which can indeed be `1` or more, not limited to fractions of `1`). Such a sequence is *linear*, and its size is determinable without computing all the values. At most one bound can be infinity, in which case the size of the sequence is infinite.
- A `Method_Expr`, in which case a function that accepts a key (or an index, if you wish) and computes the corresponding value. Such a sequence is *indexed*, but its size is undefined.
- A workflow using **`yield`** and **`yield!`** to feed the sequence with a next value. Such a sequence is again *linear*, and its size is defined when all values are computed.

In every case, a range may be optionally given to constrain the generated sequence.

The Aml Standard Runtime Library offers more functions to manipulate and create sequences.

8.2.19 Generator Expressions

Grammar:

```
Loop_Expr
    ::= ...
      'for' (Generator_Expr | Generator_Iter)
Generator_Iter
    ::= '(' Enumerators ')' {nl} (Expr | For_Loop)
Yield
```

```
    ::= 'yield' | 'yield!'
Generator_Expr
    ::= '{' Enumerators '}' {nl} Yield Expr
Enumerators
    ::= Generator {semi Enumerator}
Enumerator
    ::= Generator [semi Atomic_Pattern '=' Expr]
      | If_Guard
Generator
    ::= [Label_Dcl] Atomic_Pattern 'in' Expr [If_Guard]
If_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* **in** *e* produces bindings from an expression *e*, which are matched in some way against pattern *p* (§9). 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* **in** *e*, where *p* is not irrefutable (§9.2) for the type of *e*, is replaced by

p **in** *e*.with_filter { **when** *p* **then yes otherwise no** }

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

- A comprehensioin

  for {*p* **in** *e*} **yield** *e′*

  is translated to

  e.map { **when** *p* **then** *e′* }

- A comprehension

  for {<<*l*>> *p* **in** *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 comprehensioin

  ```
  for {p in e} yield! e'
  ```

  is translated to

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

- A comprehension

  ```
  for {<<l>> p in e} yield! e'
  ```

  where l is a label name, is translated to

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

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

- A comprehension

  ```
  for (p in e) e'
  ```

  is translated to

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

- A comprehension

  ```
  for (<<l>> p in 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 in e; p' in 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′ in e′ … } yield e″ }
```

If there was **yield!** instead of **yield**, then **yield!** is also in the translated code.

- A comprehension

```
for {<<l>> p in e; p′ in 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′ in e′ … } yield e″ },
  label: l
)
```

where *l′* is a symbol literal for the label *l*. If there was **yield!** instead of **yield**, then **yield!** is also in the translated code.

- A comprehension

```
for (p in e; p′ in e′ …) e″
```

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

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

- A comprehension

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

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

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

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

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

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

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

  p **in** e.with_filter(**fun** ($x_1, \ldots, x_n$) -> { **not** g })

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

- A generator p **in** 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') **in for** {p **as** x **in** e} **yield** { **let** p' **as** x' = e' **in** (x, x') }

Generators in generator expression can optionally have a label l assigned, so that expressions like **break** l could work.[3] 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 8.2.3 The following code produces all pairs of numbers between 1 and $n - 1$, whose sums are prime numbers.

```
for { i in 1 .. n
      j in 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 8.2.4 Generator expressions can be used to express vector and matrix algorithms concisely.

```
method transpose['a] (xss: List[List['a]]): List[List['a]] =
  for {i in 0 .. xss(0).length} yield {
    for (xs in xss) yield xs(i)
  }
```

[3] 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.

The comprehension is translated to:

```
method transpose['a](xss: List[List['a]]): List[List['a]] =
  (0 .. xss(0).length)
    .map {
      when i
        xss.map { when xs then xs(i) }
    }
```

8.2.20 Anonymous Functions

Grammar:

```
Function_Expr
    ::= 'fun' {Anonymous_Params_Clause}+ [If_Guard] Fun_Type_Expr
        '->' [Lambda_Declarations] Expr
      | 'function' [Lambda_Declarations 'match'] When_Clauses 'end'
Block_Expr
    ::= Match_Block_Expr
Match_Block_Expr
    ::= '{' When_Clauses '}'
      | 'do' When_Clauses 'done'
Anonymous_Params_Clause
    ::= Param_Clause
      | Variadic_Param
      | Implicit_Params
      | '(' 'rec' [id] ')'
```

The anonymous function `fun` x_1`:` T_1 `...` 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.

If the expected type of an anonymous function is of the form `Function[`S_1`, ..., `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 (§9.4), where the parameters are inferred from the expected type.

The anonymous function is evaluated as the following expression:

```
(Function[S1, ..., Sn, T] with {
  method apply (x1: S1) ... (xn: Sn): T = b
}).new
```

Example 8.2.5 Examples of anonymous functions:

```
-- identity function
fun x -> x

-- curried function composition
fun f -> g -> x -> f(g(x))

-- or a bit shorter
fun f g x -> f(g x)

-- a summation function
fun (x: Integer) (y: Integer) -> x + y

{- 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
```

Among the parameters of an anonymous function, a special form `(rec n)` or just `(rec)` can appear, giving the anonymous function itself a name for its recursive use within itself, either *n* or **rec**, respectively.

```
fun (rec f) n ->
  if n <= 1 then 1
  otherwise n * f $ n - 1
  end

fun (rec) n ?(acc = 1) ->
  if n = 1 then acc
  otherwise rec (n - 1) (acc * n)
  end
```

8.2.20.1 Placeholder Syntax for Anonymous Functions

Grammar:

```
Simple_Expr
    ::= ...
      | Param_Placeholder_Expr
Param_Placeholder_Expr
```

```
::= '$' ? no whitespace ? {digit_char}+
```

An expression (`Expr`) may contain embedded dollar symbols "`$`" followed by a number at places where identifiers are legal. Such an expression represents an anonymous function, where each numbered percent symbol denotes the corresponding positional parameter, and the zero-numbered symbol denotes the anonymous function itself.[4]

Define an *anonymous section* to be an expression of the form `$n as T`, where T is a type, or else of the form `$n`.

An expression e of syntactic category `Expr` *binds* an anonymous section u, if the following conditions hold:

1. e properly contains u
2. there is no other expression of syntactic category `Expr` which is properly contained in e and which itself properly contains u

If an expression e binds anonymous sections $u_1, \ldots, u_n$, in order specified by the numbering (and with blanks in between filled by a fresh u_i), it is equivalent to the anonymous `(`u'_1`) -> ... -> (`u'_n`) -> `e', where each u'_i results from u_i by replacing the percent symbol with a fresh identifier and e' results from e by replacing each u_i with u'_i. If u_i was a part of a typed expression, the corresponding parameter is typed the same and the u'_i in e does not need to be typed anymore.

Example 8.2.6 The anonymous functions in the left column use placeholder syntax. Each of these is equivalent to the anonymous function to its right.

```
$1                          x -> x
$1 + 1                      x -> x + 1
$1 * $2                     x -> y -> x * y
($1 as Integer) * 2         (x: Integer) -> x * 2
if $1 then x else y end     (z: Boolean) -> if z then x else y end
$1.map f                    x -> x.map f
$1.map ($1 + 1)             x -> x.map (y -> y + 1)
```

8.3 Application Expressions

8.3.1 Designator Expressions

Grammar:

[4] This is different from Scala where the parameters are referenced with an underscore and not numbered – they are successive.

```
Simple_Expr
    ::= ...
      | Designator_Expr
Designator_Expr
    ::= Path
      | Expr '.' Selection
      | Type_Member_Selection
Selection
    ::= Simple_Selection
      | Tuple_Selection
      | Optional_Selection
      | Forced_Selection
Simple_Selection
    ::= id
Tuple_Selection
    ::= decimal_numeral
Optional_Selection
    ::= '?' (Simple_Selection | Tuple_Selection)
Forced_Selection
    ::= '!' (Simple_Selection | Tuple_Selection)
Type_Member_Selection
    ::= Expr ('#' | '.#') (Simple_Selection | Tuple_Selection)
      | '#' ? no whitespace ? (Simple_Selection | Tuple_Selection)
```

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 (§5). 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 (§6.2) 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 **in** $y.x$ }, for some fresh name y.

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

```
begin
  if let? y = e  -- optional binding
  then y.x
  else nil
  end
end
```

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

references as well.

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.

A selection e`.`x, where x is formed by decimal digits, is useful for selecting members whose name is a decimal number. Tuple types have such members, and other types may define such members by enclosing their names in doubled backquotes, e.g. **`method`** `` ``1`` () = 1``. The expression e must not be a number literal.

The distinctive form in which instance and class instance value or variable selections appears practically means that objects have separate namespaces[5] for their instance variables (state) and all other members (methods, nested classes etc.).

8.3.2 Self, Super & Outer

Grammar:

```
Simple_Expr
    ::= ...
      | [id '.'] 'self' ['.' Selection]
      | [id '.'] 'super' [Class_Qualifier] ['.' Selection]
      | 'outer' Class_Qualifier ['.' Selection]
```

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 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

[5] Therefore names of instance values or variables never clash with names of other members of the instance.

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. 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 expression **`outer`**`[`T`]` refers to the current instance in the context of enclosing (or even directly enclosing) type T. It is an error if T is not an enclosing type. The type of the expression is the same as **`outer`**`[`T`]`**`.type`**.

Class qualifier may be a simple name referring to name of an enclosing type or supertype (depending on the preceding keyword: **`outer`** or **`super`**), or—if there is no type of such name or the qualifier is not a simple name—a stable id referring to the enclosing type or supertype. The latter case is useful in cases when there are multiple enclosing types or supertypes of the same simple name. Class qualifiers are evaluated at compile time.

8.3.3 Use Expressions

Grammar:

```
Expr
    ::= ...
      | Use_Expr
      | Local_Open_Expr
      | Import_Clause semi Expr
Local_Bindings
    ::= ...
      | Use_Core_Expr
Use_Expr
    ::= 'use' Use_Core_Expr (Do_Block | In_Sep Expr ['end'])
Use_Core_Expr
    ::= Use_Expr_As
      | Use_Aspect
      | Use_Refinement
      | Use_Droppable
Use_Expr_As
    ::= Expr ('as' | 'as!' | 'as?') [id ':'] Type
Use_Aspect
    ::= 'aspect' Stable_Id
Use_Refinement
    ::= 'refinement' Stable_Id
```

```
Use_Droppable
    ::= [Storage_Modifier] Let_Value_Binding
Local_Open_Expr
    ::= Long_Id '.' '{' Expr '}'
```

Use expressions of the form **`use`** *e* **`as`** *T* are similar to typed expressions (§8.5.1). 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 `Do_Block` 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 (§8.5.1) apply in these expressions as well, including the differences between **`as`** and **`as!`**.

Use expressions of the form **`use`** *e* **`as`** *v*`:` *T* rebinds the evaluated expression *e* to a value named *v*, typed with type *T*.

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, up to the scope end. If the expression is used as a template statement, then the aspect is enabled for the whole template anywhere, unless it has the block part.

Use expressions of the form **`use refinement`** *T* enable the specified refinement, either in the scope defined by the given block, or if no block is given, then from that point onward, up to the scope end. If the expression is used as a template statement, then the refinement is enabled for the whole template anywhere, unless it has the block part.

Use expressions of the form **`use`** *v* `=` *a* **`in`** *e* are droppable expressions – on every value v_i bound in variable sub-patterns of *v*[6] has v_i`.drop()` sent to them right after evaluation of the expression *e* ends (or `drop(`v_i`)`, in case the type of v_i does not implement `Droppable` and such function exists in the scope). The exact behaviour of that application is obviously defined by the implementation of such method.[7]

8.3.4 Function Applications, Message Sending

Grammar:

```
Expr
    ::= ...
      | Application_Expr
Application_Expr
    ::= Regular_Precedence_Application_Expr
      | High_Precedence_Application_Expr
```

[6] Or, if *v* is a variable pattern, than the value bound from such pattern.

[7] Usually, such value should be considered unusable.

```
      | Super_Application_Expr
Regular_Precedence_Application_Expr
    ::= Expr Arguments [Block_Argument]
Super_Application_Expr
    ::= 'super' [Arguments] [Block_Argument]
High_Precedence_Application_Expr
    ::= Expr '.' '(' [Arguments] ')' [Block_Argument]
      | Expr ? no whitespace ? '(' [Argument] ')'
Arguments
    ::= {Argument}+
Argument
    ::= Positional_Argument
      | Labelled_Argument
Positional_Argument
    ::= ['?' | '*'] Expr
      | (Tuple_Argument - Slice_Expr)
      | '(' Tuple_Argument ')'
Labelled_Argument
    ::= ('~' | '~?') id ':' Expr
      |  '~'         id ':' Slice_Expr
      |  '~'         id ':' Tuple_Argument
      |  '~'         id ':' '(' Tuple_Argument ')'
      | ('~' | '~?') id
      | '**' Expr
Tuple_Argument
    ::= Positional_Tuple_Argument {',' Positional_Tuple_Argument}
Positional_Tuple_Argument
    ::= ['?' | '...'] Expr
      | Slice_Expr
Block_Argument
    ::= Block_Expr
      | Method_Expr
```

A function application $f(e_1 \ldots e_m)\ b$ applies the function f to the argument expressions $e_1 \ldots e_m$ and passes the block expression b (§8.3.8) into it. If f is of a value type, the application is taken to be equivalent to f`.apply(`$e_1 \ldots e_m$`)`, i.e. the application of an `apply` method defined by f.

8.3.4.1 Compared to Other Languages

`Aml` has a sort of unusual definition of function applications, for an object-oriented language. The definition is unlike that of C or similar languages, which has arguments following the function always enclosed in parentheses. The parentheses are actually only necessary to resolve precedences. It is more similar to function applications of the ML family of

languages, most similar to that of OCaml. The most prominent difference from OCaml is that OCaml does not have member overloading, while `Aml` does have that. If `Aml` did not have that overloading, all functions could be seen as functions of one or no parameters (procedures), possibly returning another function curried with the given argument. But `Aml` does have overloading, and therefore the number of declared parameters of each overloaded variant actually matters a lot.

8.3.4.2 Total Function Application

`Aml` follows roughly OCaml's definition of function applications. A total application for `Aml` is such an application that fills all required parameters with arguments. Required parameters are those that are not optional, variadic, block capture or implicit: positional and labelled. A total application may be evaluated, while non-total applications are implicitly curried using the remaining unfilled parameters, in their order of appearance.

8.3.4.3 Arguments to Parameters Mapping

A single function application may contain one or more arguments. The mapping of arguments to parameters follows these rules, given that parameters are ordered as specified by their function definition:

- Argument without a label (positional) is mapped to the next unfilled parameter. Such mapped parameter may be non-labelled as well as labelled, without total application making any difference.[8] If there is are no more unfilled parameters to map the argument to, the argument belongs to a consecutive function application, if any.
- Argument with a label is mapped to the next unfilled parameter bearing the same label. If there are multiple such parameters, the first is the one mapped. Thus, the order of parameters matters. If there is no such parameter, the argument belongs to a consecutive function application, if any. The mapped-to parameter may be also optional, that does not make a difference – it is only important when considering whether a function application is or is not total.
- Arguments do not need to appear in the same order as their mapped parameters, which is what parameter/argument labels are for.
- Arguments that have no label can not commute among themselves, their order matters.

[8] This is different from OCaml.

- Arguments that bear the same label and belong to the same function application (meaning, there are parameters these can be mapped to), they can not commute among themselves, because their order matters. Although they can commute with other arguments within the boundaries of the same function application.
- Arguments are typed with the type of the parameter they are mapped to as their expected type.
- Arguments that can not be mapped to parameters by the previous rules can be mapped to variadic parameter (or variadic parameters, if both are specified), defined by the function (see §7.10.9 for details).
- If a function application is total, the function defines an implicit parameter and there is an unmapped argument available (possibly a tuple argument), that argument is mapped to the implicit parameter.
- A function application that is not total, is implicitly curried on the remaining unfilled parameters. The resulting function is a function of all the remaining parameters in order specified by the function definition, preserving also their labels.[9]
- Arguments are not evaluated, until that is specified by the evaluation strategy of the mapped parameter, at the point when the function is actually applied (which is after any overloading resolution happens). Some arguments might be already evaluated though, e.g. local variables. Early argument evaluations are possible, if needed in overloading resolution.[10] In fact, arguments are passed as if they were wrapped in nested parameterless functions (including their closure over any local variables and the **`self`** references).
- Positional arguments that are prefixed with a " `?` " are not wrapped in `Option[`*T*`]` type, and are expected to already be of type `Option[`*T*`]`. This is useful especially inside tuple arguments, when an optional extraction does not specify a default value expression.
- Labelled arguments that are prefixed with a " `~?` " are also not wrapped in `Option[`*T*`]` type, and are expected to already be of type `Option[`*T*`]` as well.
- An argument may also comprise multiple sub-arguments, inside a tuple argument. In that case, positional sub-arguments must always come before named sub-arguments, as required by tuple expressions.
  - Additionally, the rules for arguments prefixed with a " `?` " apply.
  - Mapping of sub-arguments in a tuple argument is defined by tuple pattern matching (§9.1.20).

[9] The part about preserved labels should be obvious.

[10] E.g., when the parameter's type is a constrained type, or type of the argument is `Dynamic`, but the parameter requires a specific type.

  - No two sub-arguments inside a tuple argument can have the same label (does not apply to unlabelled sub-arguments).

- Arguments prefixed with a " `*` " are early evaluated, expected to result in a sequence, values are extracted from such a sequence and inserted in place of the argument, without labels, possibly spanning multiple function applications (unless used within a tuple argument).
- Arguments prefixed with a " `**` " are early evaluated, expected to result in a dictionary from `Symbol` to any type, values are extracted from such a dictionary and inserted in place of the argument, with labels defined by each dictionary key, possibly spanning multiple function applications (unless used within a tuple argument).
- Block argument is mapped to the block capture parameter, if any, otherwise made available only to **`yield`**.
- Tuple argument's sub-arguments do not yield implicit currying. Explicit currying is possible though, see (§8.3.4.6).
- A labelled argument of the forms *~id* and *~?id* are referring to a local variable (usually parameter) and have the same label as is the name of the variable.
- Labelled arguments that do not specify either of the prefixes " ~ " and " ~? ", are implicitly prefixed with " ~ ". This is different from tuple expressions (§8.2.10), where the explicitly written tilde prefix " ~ " actually yields a slightly different data type.

8.3.4.4 Tail-call Optimizations

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).

8.3.4.5 Memoization

How to memoize a function's result is described in (§8.6.4.1).

A memoized function's body is not evaluated, if it was once called with the same arguments (based on structural equality, not physical identity[11]), 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 (§7.10 & §8.8) 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 (§7.10) may opt-in to implicit memoization[12], as well as functions declared as **`transparent`** (§8.8). Functions declared as **`opaque`** (§8.8) should not be memoized.

Parameters of memoization[13] may be controlled, even on per-function basis, with use of specialized annotations and pragmas.

8.3.4.6 Method Values & Explicit Partial Applications

Grammar:

```
Method_Expr
    ::= -- explicitly partially applied function:
        '&' Expr ['.' '(' {Partial_Argument} ')'] [Partial_Block_Argument]
      | '&(' Expr {Partial_Argument} ')' [Partial_Block_Argument]

        -- explicitly partially applied operator:
      | '&(~' ? whitespace ? op_id ')' [Partial_Argument]
      | '&(' op_id ')' [Partial_Argument [Partial_Argument]]
      | '&(' nary_op_id ')' {Partial_Argument}

        -- implicitly partially applied operator:
      | '(' (op_id | nary_op_id) ')'
      | '(~' ? whitespace ? op_id ')'

        -- left section:
      | '(' Expr ? whitespace ? op_id ')'
```

[11] Therefore, it is actually a good idea to memoize function's results based on arguments that have simple or physical structural equality.

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

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

```
        -- right section:
      | '(' op_id ? whitespace ? Expr ')'

Partial_Argument
    ::= Argument
      | Positional_Unapplied_Argument
      | Labelled_Unapplied_Argument
Positional_Unapplied_Argument
    ::= '_' ['as' Type_Expr]
      | '*_' ['as' Type_Expr]
      | Partial_Tuple_Argument
      | '(' Partial_Tuple_Argument ')'
Labelled_Unapplied_Argument
    ::= ('~' | '~?') id ':' '_' ['as' Type_Expr]
      |  '~'         id ':' Partial_Slice_Expr
      |  '~'         id ':' Partial_Tuple_Argument
      |  '~'         id ':' '(' Partial_Tuple_Argument ')'
Partial_Tuple_Argument
    ::= Partial_Tuple_Argument_Part
        {',' Partial_Tuple_Argument_Part}
Partial_Tuple_Argument_Part
    ::= Positional_Tuple_Argument
      | '_' ['as' Type_Expr]
      | '..._' ['as' Type_Expr]
      | Partial_Slice_Expr
Partial_Slice_Expr
    ::= Slice_Expr
      | '_' ['as' Type_Expr] ('..' | '..<') ['_' ['as' Type_Expr]]
      | ('..' | '..<') '_' ['as' Type_Expr]
Partial_Block_Argument
    ::= Block_Argument
      | '&_' ['as' Type_Expr]
```

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 (§8.9.4). 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 8.3.1 The method values in the left column are each equivalent to the anonymous functions (§8.2.20) on their right.

```
&(Math.sin)             fun x -> { Math.sin x }
&(Array.range)          fun x1 x2 -> { Array.range x1 x2 }
( * 42)                 fun x -> { x * 42 }
```

Note that if *e* resolves to a parameterless method of type `() -> T` or if *e* has a method type `() ↦ T`, it is evaluated to type *T* (§8.9.2) – and the method value syntax provides a way to prevent this.

If *e* resolves to an overloaded member, then the type of the expression is a type projection to that member, constrained to the matching alternatives.

If *e* contains sub-expressions of the forms "`_`" or "`_ as T`", called *argument placeholders*, then the method value contains only those overloaded alternatives that are applicable to the given sub-expressions, including the argument placeholders. Such an expression is then partially applied and resolves to an anonymous function, where the argument placeholders become parameters of the anonymous function.

8.3.4.7 Member Constraint Invocation Expressions

Grammar:

```
Expr
    ::= ...
      | Member_Constraint_Invocation_Expr
Member_Constraint_Invocation_Expr
    ::= '(' Type_Var ':' '(' Member_Signature ')' Arguments ')'
```

8.3.5 Argument-Dependent Lookup in Scope

Argument-dependent lookup in scope is a mechanism that extends scope for unqualified name lookups (see §5) with all members of objects that belong to the argument-dependent scope, with priority lower than does not shadow already bound names in the scope (same as scope extension from type classes).

The *argument-dependent scope* of a type *T* consists of the type *T* itself and all types, classes and class objects that are associated with the type *T*. Here, we say that a class or a type *C* is *associated* with a type *T*, if it is a base class of some part of *T*, or it is one of the classes or structures that *T* is nested in. The *parts* or a type *T* are defined the same as for implicit scope (§12.2). The argument-dependent scope is indeed possibly (and likely) broader than the implicit scope for the same type *T*. Overloading resolution may need to be used to determine a unique bound entity for the given name from the extended scope.

8.3.6 Attribute Selection Expressions

Grammar:

```
Expr
```

```
    ::= ...
      | Attribute_Expr
Attribute_Expr
    ::= Expr '.'' Simple_Selection
```

Every function that has at least one parameter can be used as an attribute.

An attribute selection $x.'a$ is then viewed as function application $a(x)$. An attribute selection $x.'a(args)$ is then viewed as function application $a(x)(args)$ etc. (§8.3.4).

Note. Attribute selections are similar to function composition operators, but allow for a bit more concise code[14], resembling designator selections (§8.3.1):

```
let
  val y = x.'a
  val z = x |> a
in
  y = z
```

The main difference between attribute selections and function compositions shows up when more arguments are needed or multiple functions are composed and attributes chained, compare:

```
let
  val y = x.'a(b)     -- a(x)(b)
  val z = x |> a(b)   -- a(b)(x)
in
  y /= z


let
  val y = x.'a(b).'c(d)       -- c(a(x)(b))(d)
  val z = x |> a(b) |> c(d)   -- c(d)(a(b)(x))
in
  y /= z
```

8.3.7 Type Applications

Grammar:

```
Type_Application_Expr
    ::= Expr ? no whitespace ? Type_Args
```

[14]Done by not requiring whitespace around the symbolic keyword, unlike whitespace being required around most infix operators.

```
      | Expr '.' Type_Args
      | '(' Postfix_Parameterized_Type ')'
```

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 $[a_1 \mapsto T_1, \ldots, a_n \mapsto T_n]$. 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 automatic type inference (§16) can infer best type arguments for a polymorphic function from the types of the actual function arguments and the expected result type. If any of the type arguments is specified as an underscore "_", it is locally inferred, therefore allowing for partially explicit type application, where the provided type arguments are taken as constant types.

8.3.8 Blocks

Grammar:

```
Block_Expr
    ::= ...
      | Low_Precedence_Block_Expr
      | High_Precedence_Block_Expr
Low_Precedence_Block_Expr
    ::= 'do' [Block_Head] Block_Code_Expr 'done'
High_Precedence_Block_Expr
    ::= '{' [Block_Head] Block_Code_Expr '}'
      | '{' '->' Block_Code_Expr '}'  -- to resolve ambiguity with records
Specialised_Block_Expr(Params)
    ::= 'do' Params Block_Code_Expr 'done'
      | '{' Params Block_Code_Expr '}'
        -- if Params are empty:
      | '{' '->' Block_Code_Expr '}'
Block_Head
    ::= [Block_Self_Type] Block_Params [':' Type_Expr [semi]] '->'
Block_Self_Type
    ::= ['as' id ':' Type_Expr In_Sep]
      | ['as' Type_Expr In_Sep]
Block_Params
    ::= {Param_Clause} [Variadic_Param] [Implicit_Params]
Block_Code_Expr
    ::= [Lambda_Declarations] Expr
Do_Block
```

```
    ::= '{' Expr '}'
      | 'do' Expr 'done'
Do_Binding
    ::= 'do' Expr ['done']
```

A block expression `{` s_1`;` `...;` 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.

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.

8.3.8.1 Block Expression as Argument

A block expression may be used as the very last argument in a function application (§8.3.4), 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 (§8.6.4.1) 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 (§8.2.20), where the return expression is interconnected with the anonymous function itself.

8.3.8.2 Variable Closure

Aml 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.

Variables are made available via inner hidden instance variables of the implicit function. This includes the **`self`** object of the outer scope, if any methods are to be executed from within the block. Those variables are referenced with a strong reference, and therefore may cause in some situations retain cycles – this can be solved by using *capture lists*, which can override this default behaviour and store the reference as either **`weak`** or **`unowned`** reference, to break the retain cycle.[15]

If a block expression is used as an argument in a function application (§8.3.4), 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 (§8.3.2). Write access to variables is provided in a manner equivalent to **`out`** arguments.

Variable closure is applied to anonymous functions (§8.2.20) 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.

8.3.9 Yield Expressions

Grammar:

```
Yield_Expr
    ::= [id '.'] 'yield' [Arguments]
```

A yield expression **`yield`** $a_1, \ldots, a_n$ is an universal **`yield`** operation:

1. A way to invoke the block argument (§8.3.4 & §8.3.8.1) and pass it arguments. It's expected type is the result type of the given block argument. The value of the expression is then whatever the block argument returned.
2. When no block argument was given and the invocation happens inside a fiber that is not the thread's main fiber, a way to return from a fiber without destroying it's stack. The value of the expression is then whatever value is passed to the method that resumes the fiber (so that it continues with that value in place of the **`yield`** expression).
3. A part of workflows (§8.2.16).
4. A part of generators (§8.2.19) and collection comprehensions (§8.2.17).
5. In other cases, it is an error to use **`yield`**.

[15] Quite useful with lazy definitions of instance variables, where the expression is wrapped in an implicit block.

A yield expression T.**`yield`** $a_1, \ldots, a_n$ is a specialized case of the previous definitions of **`yield`**, where T may be one of:

- `Block`, for invoking block argument. To test if a block argument is available, use `Yield.block_available?`, defined in Aml's `Language` module.
- `Fiber`, for yielding from a fiber. To test if a yield from a fiber is available, use `Yield.fiber_available?`, defined in Aml's `Language` module.

8.3.10 Prefix & Infix Operations

Grammar:

```
Expr
    ::= ...
      | Infix_Expr
Infix_Expr
    ::= Prefix_Expr ['use' Simple_Type]
      | Expr op_id Expr ['use' Simple_Type]
      | Expr nary_op_id Expr {':' Expr} ['use' Simple_Type]
      | Range_Expr
Application_Expr
    ::= ...
      | Op_Application_Expr
Op_Application_Expr
    ::= -- binary operator application
        '(' op_id ')' Expr Expr ['use' Simple_Type]
      | '(' op_id Expr Expr ['use' Simple_Type] ')'
        -- n-ary operator application
      | '(' nary_op_id ')' {Expr}+ ['use' Simple_Type]
      | '(' nary_op_id {Expr}+ ['use' Simple_Type] ')'
        -- unary operator application
      | '(~' ? whitespace ? op_id ')' Expr ['use' Simple_Type]
      | '(~' ? whitespace ? op_id Expr ['use' Simple_Type] ')'
Prefix_Expr
    ::= [op_id | 'not' ? whitespace ? | '...'] (Expr - Prefix_Expr)
Range_Expr
    ::= Expr ('..' | '..<') Expr [FP_Delta [FP_Digits]]
Slice_Expr
    ::= -- slice of 1 element
        Expr
        -- slice from index to last
      | Expr '..'
        -- slice from first to index
      | ('..' | '..<') Expr
        -- slice from index to index
```

```
| Expr ('..' | '..<') Expr
  -- slice from first to last (identity view)
| '*'
```

Expressions can be constructed from operands and operators.

8.3.10.1 Prefix Operations

A prefix operation *op e* consists of a prefix operator *op*, which may be any operator identifier, but must not be followed by any whitespace, only identifiers or parentheses (except for the special operator **`not`**, which has to be separated by a space from the expression that it prefixes). The expression *op e* is equivalent to the function application `(`*op e*`)`.

There is also a keyword prefix operator: **`not`**, which must be followed by a whitespace and is always prefix.

The precedence of prefix operators is higher than that of infix operators (§8.3.10.3). For example, `-a + b` is read as `-(a) + b`, and `-(a + b)` as just that.

Prefix operations may also appear in the forms (*op*) *e* or (*op e*).

8.3.10.2 Postfix Operations

Apart from standard function applications (§8.3.4) that may be viewed as postfix, Aml does not include support for postfix operations.

8.3.10.3 Infix Operations

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

Infix operators have to be separated from both sides by whitespace from the expressions that they connect. If the following whitespace is a newline character, then precedence rules can not be followed (due to line-by-line parsing and evaluation) and usual behaviour described in (§3.6) is followed. The only infix operator that can be separated either from both sides or from neither side by whitespace is "^" (power), and this exception is applied only to this particular operator, not to operators derived from it.

Infix operations and *n*-ary operations may alternatively appear in the form (*op*) $e_1 \ldots e_n$, which is equivalent to a function application.

After associativity of an infix operation is determined and a form e_1`.\`op\`(`e_2`)` is known:

1. If type of e_1 defines *op*, that definition is used.

2. Otherwise, if the type of e_1 is a class, and the class object defines *op*, that definition is used.

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 alphanumeric characters)
`|`
`&`
`< > ~`
`= !`
`: @`
(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* (§8.3.10.4). 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.

A number of left-associative alphanumeric operators with different precedence exist:

- **`and`**, with precedence of `&`.
- **`and also`**, with precedence of `&`.
- **`or`**, with precedence of `|`.
- **`or else`**, with precedence of `|`.
- **`xor`**, with precedence of `|`.
- `rem`, with precedence of `/`.
- `mod`, with precedence of `/`.
- `div`, with precedence of `/`.
- `quot`, with precedence of `/`.

The *associativity* of an operator is determined by the operator's last character, if not explicitly defined (using one of **`infix`**, **`infixl`** or **`infixr`**). Operator "`=~`" is 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 $(op)\ e_1\ e_2$. If op is right-associative, the same operation is interpreted as `{ let x = `e_1` in (`op`) x `e_2` }`, where x is a fresh name.
- Operators in a group with the same precedence must have the same associativity.

Solution to Ambiguity between Infix Operations and Function Applications. There is a syntactic overlap between syntaxes of function applications and infix operations. It's solution is not based on any extra syntactic elements, but rather on static AST resolution. The overlap occurs when an identifier is used in a place of a function application, where it could as well be an infix operation.

Identifiers are then resolved with the following precedence:

1. Variables and values that are declared in the scope always come first. The variable is a part of a function application.
2. In an ambiguous sequence of expressions $e_1\ e_2\ e_3$, where e_2 is an identifier, if the type of e_1 defines an infix operator of the name e_2, is implicitly convertible[16] to a type that defines it, or if an infix operator exists in the scope where the sequence of expressions appears, it is a part of an infix operation.
3. Otherwise, it is a part of a function application.

Example 8.3.2 Distinction between infix operations and function applications.

[16]Only statically known implicit conversions are available.

```
-- function application
file.open "hello_world.txt"
-- is equivalent to function application
file.open ("hello_world.txt")

-- whereas expression
a shift-left b
-- is an infix expression equivalent to
shift-left a b
-- and not to the less obvious
a (shift-left) (b)
-- but only if `shift-left` is an infix operator in the scope
```

Infix operations can be grouped together. If there is an even number of expressions separated by whitespace (i.e., no right-hand side of an infix operation), the last one is an argument in "poetry"-style function application, but such code is considered suspicious and a warning is issued during compilation.

Tuple Arguments. If the left hand side of an infix operator is of a tuple type, then an infix expression $(e_{01}, \ldots, e_{0n})$ op $(e_{11}, \ldots, e_{1n})$ is always seen as $op(e_{01}, \ldots, e_{0n})(e_{11}, \ldots, e_{1n})$. Two tuple arguments are used to distinguish which arguments come from the left side and which from the right side.

Standard Operators. This is a short extract of the full set of standard operators. Not all classes implement these operators, e.g. `Object` implements none, not even "`=`".

Language-reserved operators:

```
a == b       -- value physical identity
a /== b      -- not physically identical
not (a == b) -- quite the same as `not physically identical'
```

Standard arithmetic operators:

```
a + b     -- addition
a +. b    -- real addition
a +: b    -- integral addition
a - b     -- subtraction
a -. b    -- real subtraction
a -: b    -- integral subtraction
a * b     -- multiplication
a *. b    -- real multiplication
a *: b    -- integral multiplication
a / b     -- any number division
```

```
a /. b    -- real division
a // b    -- rational division
a //. b   -- rational division with real numerator
a div b   -- integral division
a mod b   -- modulo
a quot b  -- quotient
a rem b   -- remainder
a ^ b     -- power
```

Standard comparison operators:

```
a = b    -- equality
a /= b   -- non-equality; unordered, less than or greater than
a =~ b   -- custom pattern matching, a matches b
a > b    -- greater than
a >= b   -- greater than or equal
a < b    -- less than
a <= b   -- less than or equal
a <> b   -- less than or greater than
a <>= b  -- less than, greater than or equal; ordered
a /<>= b -- unordered
a /<> b  -- unordered or equal
a /<= b  -- unordered or greater than
a /< b   -- unordered, equal or greater than
a />= b  -- unordered or less than
a /> b   -- unordered, less than or equal
a <=> b  -- compare
```

Extended comparison operators (arguments are a tuple on the right hand side of the operator):

```
a = (...)   -- `a` equal to all arguments
a /= (...)  {- `a` not equal to all arguments,
               but arguments may be equal to each other -}
a > (...)   -- `a` greater than all arguments
a >= (...)  -- `a` greater than or equal all arguments
a < (...)   -- `a` less than all arguments
a <= (...)  -- `a` less than or equal all arguments
-- etc.
```

Standard bitwise operators:

```
a << b   -- arithmetic (& logical) shift left
a <<< b  -- shift left, preserving the least significant bit
a >> b   -- arithmetic shift right
a >>> b  -- logical shift right
```

```
a <<@ b  -- rotate left
a >>@ b  -- rotate right
a and b  -- bitwise and (when applied to numbers)
a &&& b  -- bitwise and
a or b   -- bitwise or (when applied to numbers)
a ||| b  -- bitwise or
a xor b  -- bitwise xor (when applied to numbers)
a ^^^ b  -- bitwise xor
¬a       -- bitwise not (U+00AC, NOT SIGN)
not a    -- bitwise not (prefix, unary; when applied to numbers)
```

Standard boolean (logical) operators:

```
-- short circuited:
a && b       -- boolean and
a and b      -- boolean and
a || b       -- boolean or
a or b       -- boolean or
-- non short circuited:
a &&& b      -- boolean and
a and also b -- boolean and
a ||| b      -- boolean or
a or else b  -- boolean or
a ^^ b       -- boolean xor
a xor b      -- boolean xor
¬a           -- boolean not (U+00AC, NOT SIGN)
not a        -- boolean not (prefix, unary; when applied to booleans)
```

Range Expressions. A range expression (the `Range_Expr` syntax category) is a special case of an infix expression, which can optionally be followed by delta and digits specifications, so that a range that may result from it the expression can swap a floating point type with a fixed point type, and thus allowing more operations on the range, e.g. iteration (where the delta defines the "step" of each iteration).

If the expression contains both delta and digits specifications, the expected type of both operands is a decimal fixed point type with corresponding specification of delta and digits. The range of the type is implementation-defined.

If the expression contains only a delta specification, the expected type of both operands is an ordinary fixed point type with corresponding specification of delta. The range of the type is implementation-defined.

Slice Expression. A slice expression (the `Slice_Expr` syntax category) is a special case of an infix expression, which can only appear in function applications as an argument.

The expression is a representation of a range that has optional bounds, and if the bound is missing, it is `None`. This has special meaning for functions that accept slices as arguments – usually, missing left bound means "from first", missing right bound means "to last inclusive".

8.3.10.4 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 "`/==`"[17].

Assignment operators are treated specially in that they can be expanded to assignments if no other interpretation is valid, as previously defined. Assignment operators can be defined as members of a 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 occurs if the following conditions are fulfilled:

1. The left hand side l does not have a member named "`+=`", and also can not be converted by an implicit conversion (§8.9) to a value with a member named "`+=`", applicable to a value of type of r.
2. The expression l `:=` `!`l `+` r is type-correct. In particular, this implies that `!`l refers to an object that is convertible to a value with a member named "`+`" (or itself has such a member without conversion, in the ideal case, indeed).
3. The prefix operator "`!`" exists for the type of l.

The re-interpretation is built into the compiled bytecode in such a way that first tries the assignment operator, and then the re-interpretation only in case where the assignment operator approach failed.[18] It is indeed an error if none of the two approaches succeeded.

Assignment operations are right-associative, despite any rules defined for other operators. The value of an assignment expression is the left-hand side argument after the assignment operation was evaluated.

[17] This one effectively disqualifies "l `/=` r" from being treated as "l `:=` l `/` r", very intentionally: to prevent floating/fixed point division from being privileged to integral division, "`div`".

[18] No appropriate implicit conversion was found, or if implicit conversions are disabled for the expression, the value referred to by l does not have the appropriate member.

Example 8.3.3 Right-associativity of assignment operations, using mutable variables as left-hand sides.

```
let
  val a = var something
  val b = var something_else
in
  -- the assignment
  a := b := c
  -- is equivalent to
  a := (b := c)
end
```

8.3.10.5 *N*-ary Infix Expressions

Grammar:

```
nary_op_id  ::= '?' {op_char}
Infix_Expr  ::= Infix_Expr nary_op_id Infix_Expr {':' Infix_Expr}
```

Operators that begin with a question mark "?" (and may as well consist only of that one character) have the special ability to create an *n*-ary infix expression. Those are recognized by the " : " symbol (separated by whitespace from both sides), which is otherwise not allowed as an operator. Here, it serves as a separator of arguments to the *n*-ary infix operation. The expression is viewed as if it were a regular infix expression, where the infix expressions following the *n*-ary operator are all right-hand side arguments to the operation.

The following infix expression and the function application are equivalent.

```
a ? b : c : d
(? a b c d)
```

Example 8.3.4 The most widely used *n*-ary operator is the ternary conditional operator "?", which could be defined as follows:

```
operator ?
    ['a <% Boolean_Like, 'b, 'c <: 'b, 'd <: 'b]
    (tested: !'a) (if_yes: ~'c) (if_no: ~'d): 'b
  if tested as Boolean
    if_yes
  else
    if_no
  end
end
```

Such definition of the conditional operator includes lazy evaluation of the arguments. A shorthand version could use an object definition from which the operator name may be imported into scope:

```
operator ?:
    ['a <% Boolean_Like, 'b <: 'a]
    (tested: !'a) (default: ~'b): 'a
  if tested as Boolean
    tested
  else
    default
  end
end
```

Then, one may use these operators as follows:

```
val a = b ? c : d
val e = b ?: d  -- which is equal to b ? b : d
```

8.3.10.6 Operator Name Resolution

In any operator-related expression (prefix, infix, *n*-ary), the operator name is first searched in the receiver. Then, if not found, the name scope in which the operator expression appears, is searched for an operator of the given name and arity of arguments count incremented with 1 (for the original receiver). After that, implicit conversions are applied to the receiver, to a type that contains operator with the given name.

If the found operator does not accept *n* arguments, but instead has *n* type parameters, then:

- If the operator references a type, the infix expression is equivalent to an infix type (§6.3.10), and has to be a part of a type reference expression (§8.5.2).
- If the operator does not reference any type, then the infix expression is equivalent to a type application (§8.3.7).

8.3.11 Assignment Expressions

Grammar:

```
Expr
    ::= ...
      | Rebind_Expr
      | Update_Expr
```

```
Rebind_Expr
    ::= Designator_Expr '<-' Expr
Update_Expr
   ::= High_Precedence_Application_Expr '<-' Expr
     | '(' Regular_Precedence_Application_Expr ')' '<-' Expr
```

The interpretation of an assignment expression e in the form of x `<-` e' depends on the form of x, as described in the following sections.

8.3.11.1 Rebind Expressions

Rebind expressions are assignment expressions where x is a designator expression.

- If x evaluates to a value for which implementation of the operator `:=` exists in the scope, then the expression e is evaluated as an application of said operator, i.e. `(` `:=` x e'`)`.
- Otherwise, the expression needs to be re-interpreted as a function application.
  - If x is a single identifier, the expression e is re-interpreted as x`.rebind(`e'`)`, if x responds to the message `rebind`. With pragmas, the type of x might choose a different message name, in which case, that one is used.
  - If x is an expression of the form x'`.`n, where x' is an arbitrary expression and n is an identifier (or a decimal number), the expression e is re-interpreted as x'`.set_`n`(`e'`)`, if the type of x' responds to the message `set_`n.
  - If x is an expression of the form x'`.?`n, where x' is an arbitrary expression of the type `'a option` and n is an identifier, the expression e is re-interpreted as

    ```
    match x'
      when Some val x''
        then Some (x''.set_n(e'))
      else None
    end
    ```

    if the type of x' responds to the message `set_`n.
  - If x is an expression of the form x'`.`n`!`, where x' is an arbitrary expression of the type `'a option` and n is an identifier, the expression e is re-interpreted as

    ```
    match x'
      when Some val x''
        then x''.set_n(e')
      else -- raise error
    ```

end

if the type of x' responds to the message `set_`n.

- If x is an expression of the form `#`n x', where x' is an arbitrary expression and n is an identifier (or a decimal number), the expression e is re-interpreted as x'`.set_`n`(`e'`)`, if the type of x' responds to the message `set_`n.
- If x is an expression of the form x_1`#`n x_2, where x_1 and x_2 are arbitrary expressions and n is an identifier (or a decimal number), the expression e is re-interpreted as x_2`.set_`n`(`e'`)`, if the type of x' responds to the message `set_`n.[19]
- Otherwise, it is an error.

8.3.11.2 Update Expressions

Rebind expressions are assignment expressions where x is a function application expression.

- If x is of the form x'`.(`*args*`)`, where x' is an arbitrary expression, the expression e is re-interpreted as x'`.update(`*args* e'`)`, if the type of x' responds to the message `update`. With pragmas, the type of x' might choose a different message name, in which case, that one is used.
- If x is of the form x'`(`*args*`)`, the expression e is re-interpreted as if it was of the form x'`.(`*args*`)`.
- If x is of the form `(`x' *args*`)`, where x' is an arbitrary expression, the expression e is re-interpreted as x'`.update(`*args* e'`)`

8.4 Definition Expressions

Grammar:

```
Expr
    ::= ...
      | Binding_Expr
Binding_Expr
    ::= Val_Def In_Sep Expr ['end']
```

[19] In this case, x_1 is used to further specify typing within the expression e, since it selects member n from the expression x_1 and that choice in turn defines the expected type of x_2.

8.5 Type-Related Expressions

8.5.1 Typed Expressions

Grammar:

```
Cast_Expr
    ::= Flexible_Cast_Expr
      | Static_Cast_Expr
      | Dynamic_Cast_Expr
Flexible_Cast_Expr
    ::= Expr ('as' | 'as!' | 'as?') Type_Expr
Static_Cast_Expr
    ::= -- type ascription and coercion from type to type
        '(' Expr ':' Type_Expr ')'
      | '(' Expr [':' Type_Expr] ':>' Type_Expr ')'
      | Expr ':>' Type_Expr   -- upcast to type
      | 'upcast' Expr         -- upcast to anonymous type variable
Dynamic_Cast_Expr
    ::= -- downcast from type to type
        '(' Expr [':' Type_Expr] ':?>' Type_Expr ')'
      | Expr ':?>' Type_Expr  -- downcast to type
      | 'downcast' Expr       -- downcast to anonymous type variable
Infix_Expr
    ::= Expr ('is' ['not' | 'not!'] | 'is!') Type_Expr
      | Expr (':?' | '/:?') Type_Expr
```

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*. If the type conversion fails, a runtime error is raised. 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 $c(e)$.

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.

Flow Based Typing. If the conformance check expression is performed on a variable, as defined here, it shadows the type of the variable, if used as a condition in a conditional expression (§8.6.1), in the related branch. Therefore, it is also used in resolution of function applications (§8.3.4). Furthermore, if the type of the variable was already known in the outer code, then it can both restrict and extend the expected type *T* with *U* into *T*:

- If the type *T* did not contain *U*, then *T′* is *T* **`and`** *U*.
- If the type *T* did contain *U* and was a union type, then *T′* is only *U*, dropping the other member types.
- In the other branches of a conditional expression, it means that *T* does not conform to *U*, whatever that implies. Also, the same happens to the same branch, if the conformance check is negated (e.g., *e* **`is not`** *U*).

The conformance check expression *e* **`is not`** *T* has type `Boolean` and tests whether *e* does not conform to *T*, basically by asking a question "Can *e* not be of type *T*?", answering either "It can't be" or "It can be".

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, and the conformance check expression *e* **`is not!`** *T* works like *e* **`is not`** *T*, but also 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.

The typed expression *e* **`as?`** *T* has a result of type *T*?, resulting in a **`nil`** value instead of an error if no conversion (including implicit conversions) is available to treat *e* as *T*.

If an expression is typed to a dynamic path of a (syntactic) type, then the value referenced by such path is expected to be a type, and it is an error if it is not.

8.5.2 Type Reference Expressions

Grammar:

```
Expr
    ::= ...
      | Type_Ref_Expr
Type_Ref_Expr
    ::= '(' 'type' Type_Expr ')'
      | Simple_Type - Postfix_Parameterized_Type
```

A type reference expression is a way to refer to a type and use it as a first-class value, which it already is.

8.5.3 Structure Reference Expressions

Grammar:

```
Expr
    ::= ...
      | Structure_Ref_Expr
Structure_Ref_Expr
    ::= '(' 'structure' Struct_Expr [':' Pkg_Type] ')'
```

8.5.4 Annotated Expressions

Grammar:

```
Expr
    ::= ...
      | Annotated_Expr
Annotated_Expr
    ::= {Annotation}+ Expr
```

An annotated expression $a_1 \ \ldots \ a_n \ e$ attaches annotations $a_1 \ \ldots \ a_n$ to the expression e (§15).

8.5.5 Runtime Types

Grammar:

```
Type_Expr
    ::= ...
      | Expr 'as' 'type'
      | Expr 'as!' 'type'
      | Expr ':>' 'type'
```

This grammar does not extend the `Expr` class, but instead, allows one to use an expression as a type in the type system. The expression's expected type is `Type`.

8.6 Control Flow Expressions

8.6.1 Conditional Expressions

Grammar:

```
Expr
    ::= ...
      | Conditional_Expr
Conditional_Expr
    ::= Cond_Block_Expr
      | Cond_Mod_Expr
Cond_Block_Expr
    ::= ('if' | 'unless') Condition ('then' | semi) Expr
        [[semi] Else Expr] 'end'
Cond_Mod_Expr
    ::= Expr Cond_Modifier
Cond_Modifier
    ::= Basic_Cond_Modifier [Else (Expr - Conditional_Expr)]
Basic_Cond_Modifier
    ::= ('if' | 'unless') Condition
Else
    ::= 'else' | 'otherwise'
Condition
    ::= Boolean_Condition
      | Optional_Binding_Condition
Boolean_Condition
    ::= (Expr - Conditional_Expr)
Optional_Binding_Condition
    ::= [Mutability_Modifier] 'let?' Let_Binding
        ['and' Optional_Binding_Condition]
```

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`** (§8.5.1), and **`no`** otherwise. 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 (§6.7.4) 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 also short-circuited ("**and**" and "**or**" respectively). To prevent short-circuited behaviour, one has to use a non short circuiting version of the operator ("**and also**" and "**or else**", or "&&" and " || ").[20]

Example 8.6.1 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 alternative conditional expression **unless** e_1 **then** e_2 **else** e_3 is evaluated as if it was **if not** e_1 **then** e_2 **else** e_3.

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.

Note. Aml provides two equivalent alternatives of the usual "**else**" keyword: "**else**" (obviously) and "**otherwise**". This is only provided so that certain conditions may be read better.

Conditional Variable Binding. The conditional variable definition **if** p `=` e_1 **then** e_2 **else** e_3, where p `=` e_1 is a variable binding using **let?**, works the same way as other conditional expressions, with the following difference: if e_1 evaluates to **nil** or even `()` (`Unit` value), then the condition is regarded as **no**. Beware that if e_1 evaluates to **no** (literally), the condition is still met (because **no** is not **nil** neither `()`) and e_2 is evaluated. Note that if type required by p does not allow **nil** or `()` value, then no error happens and e_3 is evaluated instead. The condition in this case says "if the variable definition p `=` e_1 can be realized, then proceed with e_2, otherwise proceed with e_3". In case the **unless**

[20] This is different from SML, where the short circuiting behaviour is the other way around with "**and also**" and "**or else**". The reason is that Aml prefers the "lazy" behaviour of short circuiting these infix expressions, so it also requires less typing.

keyword is used instead of **`if`**, then the variable binding is effective in e_3, but not in e_2. Multiple conditional variable bindings may be combined, and in that case, all of them must successfully bind for the condition to be true.

8.6.2 Loop Expressions

Aml 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 Aml.

8.6.2.1 Loop Control Expressions

Grammar:

```
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' Do_Block
Broken_Expr
    ::= 'broken' Do_Block
```

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` (§8.6.4.2). All annotations (§8.5.4) 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 behaviour 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 (§8.6.5.3), 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.[21]

8.6.2.2 Iterable For Expressions

Grammar:

```
Loop_Expr
    ::= [Label_Dcl] 'for' Val_Dcls 'in' Expr
        ['step' Expr] For_Loop
For_Loop
    ::= 'loop' Loop_Block_Expr 'end'
Loop_Block_Expr
    ::= Expr + Loop_Ctrl_Expr
Val_Dcls
    ::= id {',' id}
      | id ':' Type
      | Pattern
```

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.

[21] 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.

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 (§8.6.2.1).

Iterable expressions can be seen as simple comprehensions over iterating a single iterable value. For more complex iterating expressions, see generators (§8.2.19).

An expression

for e_1 **in** e_2 **loop** e_3 **end**

is translated to the invocation

e_2`.each {` **when** e_1 **then** e_3 `}`

An expression

`<<`*l*`>>` **for** e_1 **in** e_2 **loop** e_3 **end**

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

e_2`.each({` **when** e_1 **then** e_3 `}, label:` l'`)`

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

An expression

for e_1 **in** e_2 **step** i **loop** e_3 **end**

is translated to the invocation

e_2`.each({` **when** e_1 **then** e_3 `}, 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`.

8.6.2.3 While & Until Loop Expressions

Grammar:

```
Loop_Expr
    ::= ...
      | [Label_Dcl] ('while' | 'until') Condition For_Loop
      | Loop_Modifier_Expr
Loop_Modifier_Expr
    ::= Expr Loop_Modifier
Loop_Modifier
    ::= ('while' | 'until') Condition1
```

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 (§8.6.1). 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:

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

The real implementation has to handle loop control expressions (§8.6.2.1) 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 (§8.6.2.1).

A while loop expression with variable definition works like a regular while loop expression, but the condition is treated as in conditional variable definition, and can be an immutable value definition, since it is local to each loop.

8.6.2.4 Pure Loops

Grammar:

```
Loop_Expr
```

```
    ::= [Label_Dcl] 'loop' semi
        Loop_Block_Expr 'end' ['loop']
```

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 (§8.6.2.1). It is basically equivalent to an iterable expression (§8.6.2.2) 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.

8.6.2.5 Counter Loops

Grammar:

```
Loop_Expr
    ::= ...
      | 'for' id '=' Expr (['up' | 'down'] 'to') Expr For_Loop
```

The *counter loop expressions* **for** *i* = e_1 **to** e_2 **loop** e_3 **end** is typed as `Unit`, same as **for** *i* = e_1 **down to** e_2 **loop** e_3 **end** , so there is no point in using their value.

The expression **for** *i* = e_1 **to** e_2 **loop** e_3 **end** first evaluates the expressions e_1 and e_2 (boundaries) as `Integer_Like` values *a* and *b*. Types of both must be equivalent, and the

value i is typed with their type. The loop body e_3 is then evaluated in an environment where i is successively bound to the values $a, a + 1, \ldots, b - 1, b$. The loop body is never evaluated if $a > b$. It is an error if `e_1 as? Integer_Like` or `e_2 as? Integer_Like` results in **`nil`**.

The expression **`for`** i `=` e_1 **`down to`** e_2 **`loop`** e_3 **`end`** evaluates similarly, except that i is successively bound to the values $a, a - 1, \ldots, b + 1, b$, and the loop body is also never evaluated if $a < b$.

The expression **`for`** i `=` e_1 **`up to`** e_2 **`loop`** e_3 **`end`** is equivalent to **`for`** i `=` e_1 **`to`** e_2 **`loop`** e_3 **`end`**.

8.6.3 Pattern Matching, Case Expressions & Switch Expressions

Grammar:

```
Match_Expr
    ::= Match_When_Expr
      | Match_With_Expr
Fallthrough_Expr
    ::= 'next'
Match_When_Expr
    ::= 'match' Expr [nl] When_Clauses 'end'
Match_With_Expr
    ::= 'match' Expr 'with' [nl] With_Clauses 'end'
When_Clauses
    ::= When_Clause {semi When_Clause}
        [semi Else [nl] Expr]
        [semi 'rescue' Rescue_Clauses]
With_Clauses
    ::= ['|'] With_Clause {semi '|' With_Clause}
        [semi Else [nl] Expr]
        [semi 'rescue' Rescue_Clauses]
When_Clause
    ::= 'when' Pattern [If_Guard] ('then' | semi) When_Expr
When_Expr
    ::= Expr + Fallthrough_Expr
With_Clause
    ::= Pattern [If_Guard] '->' When_Expr
```

Pattern matching is described in (§9). Here, the syntax of expressions that make use of pattern matching is given.

8.6.4 Unconditional Expressions

Unconditional expressions change the flow of programs without a condition.

8.6.4.1 Return Expressions

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

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).

Grammar:

```
Return_Expr
    ::= ['memoize'] 'return' [Expr] [Basic_Cond_Modifier]
      | ['memoize'] 'tailcall' Arguments [Block_Argument]
```

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 (§8.6.5.3) 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 (§8.3.4.5). Memoization is not available from within anonymous functions and blocks.

Implementation Note. The **`return`** expression is not using a reserved word, and neither is **`memoize`** or **`tailcall`** – they are to be implemented as macros, implicitly available through the `Aml/Language.Predefined` module.

8.6.4.2 Local Jump Expressions

Grammar:

```
Jump_Expr
    ::= Goto_Expr
      | Label_Dcl Expr
Goto_Expr
    ::= 'goto' label_name [Cond_Modifier]
Label_Dcl
    ::= '<<' label_name '>>'  -- no whitespace in between
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 (§8.6.2.4) and a **`while`** loop, which are not transformed as comprehensions into method calls.

Labels can also be used as functions (macros) to mark anonymous functions for use with **`return`**.

8.6.4.3 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.

Definitions. The following code shows how a function that create unlimited continuations might be defined.

```
protocol Continuation [-A, +B] : object
  inherit Function[A, B]
  ... end
protocol Unlimited_Continuation [-A] : object
  inherit Continuation[A, Nothing]
  ... end
```

```
let call/cc [A, B <: A] &(ctx: Unlimited_Continuation[A] -> B?): A = ...
```

A continuation, while internally holding a copy of the call stack that it was created in, is basically a function from the value that is passed into it to some other type. Unlimited continuations are restricted in the means that the input and output type has to be the same, as an unlimited continuation directly changes its result value based on that without any further modification – the modification is to be actually performed by the code that invokes the unlimited continuation. Delimited continuations do not have this restriction, as the captured continuation is well defined and delimited to a particular scope.

To vindicate the signature of `Unlimited_Continuation[A]`, it extends `Continuation[A, Nothing]` because when invoked, it does not return any value and instead the given argument is what its `call/cc` application returns – and therefore `call/cc` has to return a value of the same type that the unlimited continuation accepts as argument. This is unlike a delimited continuation, where invocation of the continuation does not continue from `reset`.

Example 8.6.2 The following shows how to invoke a continuation.

```
call/cc {cont -> cont () }
call/cc {cont -> cont 1 }
call/cc {cont -> cont 1 2 3 }
```

In this example, each line captures the current continuation, with unlimited scope – the whole call stack is duplicated for that to be possible. Once a continuation is invoked, the code that follows the corresponding `call/cc` is resumed, with the passed arguments being the result value. On the first line, the continuation is immediately invoked with no arguments, therefore the `call/cc` returns the unit value "`()`". On the second line, the continuation is immediately invoked with argument `1`, therefore the `call/cc` returns the value `1`. On the third line, the continuation is immediately invoked with arguments `1, 2, 3`, therefore the `call/cc` returns the value `Sequence(1, 2, 3)`.

Since invoking the continuation changes history and the return value of `call/cc`, the values passed as arguments to its invocation are limited to the expected type of the original application of `call/cc`. It is an error if a continuation is invoked with a value of an incompatible type. A similar restriction applies to delimited continuations as well.

There is no requirement for functions that use unlimited continuations to define their result type with any special annotations regarding continuation passing style – there would not be any result type available, since the whole remaining program is the result.

The initial application of `call/cc` returns whatever the given block returns, and such value has to be compatible with the expected type of the `call/cc` application. If the block itself invokes the continuation, then its return value is discarded and replaced with the arguments of the continuation invocation.

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 (§8.6.4.1).

The difference between unlimited and delimited continuations is in the scope where the call stack is captured. With delimited continuations, that is defined by the scope of `reset` – once `reset` is applied, there is no changing of its value, unlike with unlimited continuations, where the `call/cc` is similar to delimited continuation's `shift`.

If the delimited continuation is stored to be used outside of `reset`'s bounds, then it can possibly return a value, but never modify the value of the original `reset` application.

A `reset` application is typed with its expected type and its result type is derived from the statically known contents of its passed block. If at any point there is a `Any` type occurring, it might propagate down into the result type.

Definitions. The following code shows how functions that create delimited continuations might be defined.

```
class Shift [+A, -B, +C] (val cont: Continuation[A, B] -> C) = object
  method map [A1]
      (f: A -> A1):
      Shift[A1, B, C] =
    Shift.new fun (k: Continuation[A1, B]) -> { cont(fun (x: A) -> k(f(x))) }

  method flat_map [A1, B1, C1 <: B]
      (f: A -> Shift[A1, B1, C1]):
      Shift[A1, B1, C] =
    Shift.new fun (k: Continuation[A1, B1]) -> { f(x).cont(k) }
}
message reset [A, C] (ctx: => @[CPS_Param[A, C]] A): C end
message shift [A, B, C] (cont: Continuation[A, B] -> C):
    @[CPS_Param[B, C] A end

annotation CPS_Param [-B, +C] = object ... end
type CPS_Type ['a] = CPS_Param['a, 'a]
type Suspendable = CPS_Param[Unit, Unit]
```

Here, the `ctx` parameter represents the block that is passed to `reset`. For typing, each `shift` block is virtually converted to a **`for`**-comprehension:

```
let ctx == for {
  x in Shift.new(y)
```

```
} yield (b)
```

where x is a name representing the value that is assigned with the value of the `shift` application, y is the block passed to `shift`, and b is the code continuation that follows the application of `shift` (and which may possibly include more `shift` applications).

Note. Here, the `ctx` parameter is causing the passed block to be used as a positional argument. But for the type system, the annotation `@[CPS_Param[A, C]]` makes the type of the argument convert to a **`for`**-comprehension as specified, similar to what workflows (§8.2.16) do with their passed blocks. Both conversions may happen dynamically at runtime. Therefore, the passed block is eventually a regular positional argument anyway.

Example 8.6.3 An example of a delimited continuation in use.

```
reset do
  shift {(cont: Integer -> Integer) ->
    cont 5
  } + 1
end
```

For the type system, it looks as if it was the following code:

```
let ctx = for {
  x in Shift.new {(cont: Integer -> Integer) ->
    cont 5
  }
} yield (x + 1)
reset(ctx)
```

Note. Aml uses saguaro stack mechanisms instead of code conversions to actually implement both continuations. Due to this, the continuations are technically less limited, but the typing of its expressions may get complicated easily, as the control flow is manipulated on VM level.

8.6.5 Conditions, Throwables, Raiseables & Abandonments

Aml includes an elaborate system for handling various situations that can happen while executing a program: the *condition system*, inspired by the one found in Common Lisp (`http://clhs.lisp.se/Body/09_a.htm`).

A situation is the evaluation of an expression in a specific context, in a specific thread. A condition is an object that represents a specific situation that was detected. Condition objects are instances of the `Condition` trait. A hierarchy of classes of such instances is predefined in Aml, and can be indeed extended with new ones. A condition is not necessarily an error, as can be seen from the variety of use cases of the instances, including, but not limited to custom flow control operators.

The principal traits that comprise the condition system in Aml are:

- `Condition`, the root condition trait.
- `Throwable`, which extends `Condition`.
- `Raiseable`, which extends `Throwable`. Instances of this trait represent various errors that may occur.
- `Abandonment`, which extends `Raiseable`. Instances of this trait represent such conditions in program's execution, that its execution is no longer possible.[22]

An error is a situation in which normal program execution path can no longer continue correctly without some form of intervention, either interactive or under program's control. All errors are conditions, but not all conditions are errors.

Signalling is the process by which a detected condition can alter the flow control of a program. The condition then can be handled. Aml defines **`signal`** as the major low-level function (a keyword in fact) used to signal conditions, and also a few more functions that make use of it: `warn` and `error`.

The process of signalling involves the selection and evaluation of a handler from a list of active handlers in the thread that performs the evaluation. There are multiple ways to define a handler. One such way is to use one of the predefined functions provided by Aml that define a handler as a function of one argument (the condition object). Signalling a condition has no side-effects on the condition object.

An invoked handler can address the situation in one of these ways:

Decline
: It can decline to handle the condition, by simply returning rather than transferring control elsewhere. When this happens, any values returned by the handler are discarded and the next handler in line is invoked instead. If there is no such handler, the **`signal`** expression simply returns (with result type of `Unit`, the "`()`" value). The function that signalled the condition may make use of that.

Handle
: It can handle the condition by performing transfer of control. There are many ways to do that, including raising an exception or invoking a restart.

[22] A similar thing is called fatal error or logic exception in other languages.

Defer
: It can put off the decision, by many different ways, e.g. signalling another condition, resignalling the same condition, or even entering the debugger.

8.6.5.1 Signal Expressions

Grammar:

```
Signal_Expr
    ::= 'signal' Expr
```

A signal expression **`signal`** *e* evaluates the expression *e*. The type of this expression must conform to `Condition`. It is an error if *e* evaluates to **`nil`** or `()`.

Signalling a condition does not unwind the call stack.[23]

8.6.5.2 Try Operator

Grammar:

```
Try_Expr
    ::= Try_Op (Expr - Try_Expr) [Else 'rescue' '[' Type_Expr ']' ['with' Expr]]
      | Try_Op (Expr - Try_Expr) [Else Expr]
Try_Op
    ::= 'try'
      | 'try?'
      | 'try!'
```

The *try operator* is required to be applied on expressions, where the expressions contains a function application, one such that the function to be applied are statically known to possibly raise errors (instances of `Raiseable`), or reraise them.

- The **`try`** *e* form is the basic form that simply allows one to use a raising function.
- The **`try?`** *e* form returns **`nil`** in case of any raised error. This effectively discards any raised error, except for abandonments, as specified.
- The **`try!`** *e* form works like the basic form, but does not expect the applied function to ever raise, and if it does, a runtime error is raised – an abandonment. This has implications on the surrounding context – the application does not need to be surrounded in **`begin-rescue`** block and the function in which the application appears does not need to declare itself as raising the unexpected error.

[23] Yes, you read it correctly. No stack unwinding when signalling a condition. Implementations of Aml may achieve this by registering handlers in thread-local storage.

- The **try** *e* **else rescue**[*E*] form returns a value of type `Tried_Result[`*T*`, `*E*`]`. Rescued errors that are subtypes of *E* are then wrapped in it.
- The **try** *e* **else rescue**[*E*] **with** e_2 form returns result of evaluating e_2 if an error is rescued.
- The **try** *e* **else** e_2 form returns result of evaluating e_2 if any error is rescued.
- The **try?** *e* **else rescue**[*E*] form returns **nil** if the raised error is a subtype of *E*, otherwise, the error is reraised.
- The **try?** *e* **else rescue**[*E*] **with** e_2 form returns result of evaluating e_2 if the raised error is a subtype of *E*, otherwise, **nil** is returned.
- The **try?** *e* **else** e_2 form is forbidden.[24]
- The **try!** *e* **else rescue**[*E*] form returns a value of type `Result[`*T*`, `*E*`]`. Rescued errors that are subtypes of *E* are then wrapped in it. An abandonment is raised in case an error that is not a subtype of *E* is raised by e_1.
- The **try!** *e* **else rescue**[*E*] **with** e_2 form returns result of evaluating e_2 if an error is rescued.
- The **try!** *e* **else** e_2 form returns result of evaluating e_2 if any error is rescued.

The purpose of the try operator is to allow readers of the code to quickly spot the place where things can go south.

The `Result[`*T*`, `*E*`]` may be defined as follows:

```
type Result['Result, 'Error] = variant
  case Success of 'Result
  case Fail of 'Error
end
```

8.6.5.3 Throw, Handle, Catch & Ensure Expressions

Grammar:

```
Catch_Expr
    ::= 'begin' (Expr + ? Workflow only within Wf_Catch_Expr ?)
        ['handle' Handle_Clauses]
        'catch' Catch_Clauses
        ['ensure' [nl] Expr] 'end'
```

[24] It would be no different from **try** *e* **else** e_2, therefore, it is a redundant form.

```
      | 'begin' (Expr + ? Workflow only within Wf_Catch_Expr ?)
        'handle' Handle_Clauses
        ['ensure' [nl] Expr] 'end'
Throw_Expr
    ::= 'throw' Expr
Catch_Clauses
    ::= Catch_When_Clauses
      | Catch_With_Clauses
      | '[' Type_Expr ']' 'with' Expr
Catch_When_Clauses
    ::= [nl] Catch_When_Clause {semi Catch_When_Clause}
        [semi Else Catch_Block]
Catch_With_Clauses
    ::= 'with' ['|'] Catch_With_Clause {semi '|' Catch_With_Clause}
        [semi Else Catch_Block]
Catch_When_Clause
    ::= 'when' Pattern [If_Guard] ('then' | semi) Catch_Block
Catch_With_Clause
    ::= Pattern [If_Guard] '->' Catch_Block
Catch_Block
    ::= Expr + Rethrow_Expr + ? Workflow only within Wf_Catch_Expr ?
Rethrow_Expr
    ::= 'rethrow' [Cond_Modifier1]
Handle_Clauses
    ::= Handle_When_Clauses
      | Handle_With_Clauses
      | '[' Type_Expr ']' 'with' Expr
Handle_When_Clauses
    ::= [nl] Handle_When_Clause {semi Handle_When_Clause}
        [semi Else Handle_Block]
Handle_With_Clauses
    ::= 'with' ['|'] Handle_With_Clause {'|' Handle_With_Clause}
        [semi Else Handle_Block]
Handle_When_Clause
    ::= 'when' Pattern [If_Guard] ('then' | semi) Handle_Block
Handle_With_Clause
    ::= Pattern [If_Guard] '->' Handle_Block
Handle_Block
    ::= Expr + Resignal_Expr + ? Workflow only within Wf_Catch_Expr ?
Resignal_Expr
    ::= 'resignal' [Cond_Modifier1]
```

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 **`begin`** b **`catch`** h, where h is a handler pattern matching anonymous function (§9.4)

```
function
  when p1 then b1
  ...
  when pk then bk
  otherwise bk+1
end   .
```

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 (§6.7.2) of the type of b and the result type of h.

A **`begin-ensure`** expression **`begin`** b **`ensure`** e **`end`** 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`** (§8.6.4.1) – 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`** (§8.6.4.1) – 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 **`begin`** e_0 **`catch`** e_1 **`ensure`** e_2 **`end`** is a shorthand for **`begin`** (**`begin`** e_0 **`catch`** e_1 **`end`**) **`ensure`** e_2 **`end`**.

8.6.5.4 Raise, Handle & Rescue Expressions

Grammar:

```
Raise_Expr
    ::= 'raise' Raiseable_Body
Raiseable_Body
    ::= string_literal
      | Stable_Id [',' string_literal]
      | Expr
Rescue_Expr
    ::= [Label_Dcl] 'begin' (Expr + ? Workflow only within Wf_Rescue_Expr ?)
        ['handle' Handle_Clauses]
        ['catch' Catch_Clauses]
        'rescue' Rescue_Clauses
        ['ensure' semi Expr] 'end'
Rescue_Clauses
    ::= Rescue_When_Clauses
      | Rescue_With_Clauses
      | '[' Type_Expr ']' 'with' Expr
Rescue_When_Clauses
    ::= Rescue_When_Clause {semi Rescue_When_Clause}
Rescue_With_Clauses
    ::= 'with' ['|'] Rescue_With_Clause {'|' Rescue_With_Clause}
Rescue_When_Clause
    ::= 'when' Pattern [If_Guard] ('then' | semi) Rescue_Block
Rescue_With_Clause
    ::= Pattern [If_Guard] '->' Rescue_Block
Rescue_Block
    ::= Expr + Retry_Expr + Reraise_Expr + ? Workflow only within Wf_Rescue_Expr ?
Retry_Expr
    ::= 'retry' [label_name] [Cond_Modifier1]
Reraise_Expr
    ::= 'reraise' [Cond_Modifier1]
```

A raise expression **raise** *e* is similar to **throw** *e* (§8.6.5.3), 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`.

Rescue expression **`rescue`** h is similar to catch expression (§8.6.5.3), 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 behaviour 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.

8.6.6 Parallel Evaluation Expressions

Grammar:

```
Expr
    ::= Parallel_Eval_Expr
Parallel_Eval_Expr
    ::= 'parallel' Expr {Expr}+
      | Expr '`parallel`' Expr
```

Parallel evaluation expression "**`parallel`** e_1 e_2 ... e_n" indicates that evaluation of the sub-expressions $e_1 \ldots e_{n-1}$ could happen in parallel with that of e_n. It itself evaluates to e_n, and thus is semantically equivalent to e_n.

Note. If the runtime decides to evaluate the sub-expressions in parallel, occurrences of those sub-expressions within the last sub-expressions are to use the same mechanism as evaluation of by-need parameters (§7.10.5.3) – i.e. when the expression's value is to be used, evaluation blocks until the parallel evaluation either completes or fails. Passing the expression as an argument to a method that evaluates it by-name or by-need does not wait for the evaluation to complete.

8.7 Quoted Expressions

8.7.1 Quasi-quotation

Grammar:

```
Quasiquote_Expr
    ::= '<@' (Expr + Splice_Expr) '@>'
      | '<@@' (Expr + Splice_Expr) '@@>'
```

Quasi-quote expression “ `<@ e @>` ” is basically a well-formed piece of Aml 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 `@[quasi-quote]`.

Quasi-quotes are useful in combination with macros (§15.3).

8.7.2 Expression Splices

Grammar:

```
Splice_Expr
    ::= '${' Expr '}'
      | '$${' Expr '}'
      | '%' ? Expr not starting with '%' ?
      | '%%' ? Expr not starting with '%' ?
```

Quasi-quote expressions may optionally contain expression splices, so that values or other nodes may be injected into the represented AST. There are the following ways to interpolate the AST:

- Splice expressions “ `${ e }` ” and “ `%`*e* ”, typed as `Expression[`*T*`]`, where *e*'s value is expanded into the AST as is and *T* is the expected type of *e*. If the expression would require parentheses around it in the source, then parentheses have to be around the splice expression.
- The “ `$${ e }` ” and “ `%%`*e* ” splice expressions is typed as `Expression[Any]`.
- The `@[unquote]` annotation, which puts parentheses around the annotated expression.
- The `@[splice]` annotation, where the annotated expression is expanded into the AST as it is. Otherwise equivalent to “ `${ e }` ”.

8.7.3 Quotation

Grammar:

```
Quote_Expr
    ::= '<#' Expr '#>'
```

A quote expression " `<#` *e* `#>` " is similar to a quasi-quote, but without any interpolation, therefore, any apparent interpolation within a quote stays just that and is never expanded (unquoted or spliced). The annotation that marks an expression for quotation is `@[quote]`.

8.8 Statements

Grammar:

```
Template_Stat  ::= ? Expr, which includes the following extra elements for Expr:  ?
                   ['public'] Use_Clause
                 | [Opt_Req] {Modifier} Dcl
                 | Opt_Req [Alias_Id {',' Alias_Id}]
                 | {Modifier} Val_Def
                 | {Modifier} Fun_Def
                 | {Modifier} Op_Def
                 | {Modifier} Method_Def
                 | {Modifier} Att_Def
                 | {Modifier} Multi_Fun_Def
                 | {Modifier} Multi_Op_Def
                 | {Modifier} Multi_Method_Def
                 | {Modifier} Multi_Att_Def
                 | {Modifier - 'implicit'} Macro_Def
                 | {Modifier} Ctor_Def
                 | {Modifier} PCtor_Def
                 | {Modifier} CCtor_Def
                 | {Modifier} UCtor_Def
                 | {Modifier} Dtor_Def
                 | {Modifier - Local_Modifier} Struct_Def
                 | Tmpl_Member [In_Sep Expr ['end']]
                 | Tmpl_Ifc_Impl
                 | Tmpl_Ifc_Dcl
                 | {Modifier} Struct_Spec
                 | {Modifier} Prop_Dcl
                 | {Modifier} Prop_Def
                 | {Modifier} Constraint_Dcl
                 | {Modifier} Constraint_Def
                   -- parent:
```

```
                  | 'inherit' Long_Id [Arguments] ['as' id] [Cond_Modifier]
                    -- mixin:
                  | 'inherit' 'val' (Expr - Binding_Expr) 'as' id [Cond_Modifier]
                  | 'include' Long_Id [Cond_Modifier]
                  | Expr
                  | Alias_Expr
                  | Struct_Expr
                  | Struct_Spec
                  | ()
                  | Access_Modifier [Alias_Id {',' Alias_Id}]
                  | 'invariant' '{' Block '}'
                  | Do_Binding
Scope_Stat     ::= 'scope' '(' Scope_Id ')' (Do_Block | Expr)
                  | 'defer' (Do_Block | Expr)
Scope_Id       ::= [label_name] Scope_Ending
Scope_Ending   ::= 'exit' | 'success' | 'failure' | 'throws' | 'raises'
Tmpl_Member    ::= {Modifier} 'member' Def
Tmpl_Ifc_Impl  ::= 'interface' Annot_Type
                   'with' {Annotation} Tmpl_Member {'and' {Annotation} Tmpl_Member}
                   'end' ['interface']
Tmpl_Ifc_Dcl   ::= 'interface' Annot_Type
                   'end' ['interface']
Opt_Req        ::= 'optional' | 'required'
```

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 `()`, or the empty statement/expression, whose value is again `()`. Within template statements, the `Expr` element includes elements defined for the statement element.

8.8.1 Function-Specific Statements

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 (§11.5).

Evaluation of a statement sequence entails evaluation of the statements in the order they are written. This behaviour can be overridden for statement sequences in workflows (§8.2.16).

Statement can be an import via a use clause (§7.13), 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 (§8.3.4.5), but skips one call stack frame and works better during compilation, unlike memoization, which is a runtime feature. Moreover, all function parameters are then marked as **`constant`**.

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 **`pure`** has no access to the **`self`** object, the `Function`**`.self`** object is marked **`constant`**, and moreover, it has disabled access to all expressions that were not passed to it via arguments. However, **`constant`** is not added to its parameters. A function that is declared with **`pure`** can't be declared with **`opaque`**, as that would be contradictory. A function that is declared with **`transparent`** combines restrictions from both keywords.

A function that is declared with **`native`** in its **`declare`** block, has its body defined outside of Aml source files. Compilation of a source code that contains such functions result in generation of necessary header files, so that the native implementation may interface with a particular Aml VM implementation. Every Aml VM that has the ability to run native functions defines its own extra annotations that may be attached to the function, to influence the header file somehow (implementation-defined).

A function that is declared with **`immutable`** in its **`declare`** block, has the **`self`** value treated as immutable, therefore it presents a guarantee that applying it will never modify the target value. On the other hand, a function that is declared with **`mutable`** in its **`declare`** block, is no different from a function that is declared without **`immutable`** or **`mutable`**, but it presents a requirement that the target object will be mutable, and thus it is an error if it is not, and also it is an error if a function with such modifier is a member of a template that is declared **`immutable`**, because all of its member values are inherently immutable.

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. If a subtype attempts to override an aliased method, then all methods with that alias are overridden as well.

The `Capture_Usage` syntax construct (of the forms **`use`** $id_1, \ldots, id_n$ **`as weak`**, **`use`** $id_1, \ldots, id_n$ **`as unowned`** and **`use`** $id_1, \ldots, id_n$ **`as`** `soft`), or combinations of those, provide a way to define ownership of captured variables in blocks and anonymous functions. Every captured variable is stored as a property within the function object, and therefore it takes an ownership of the pointed object. By using this construct, the default strong reference can be replaced with a `Weak_Reference[T]`, an `Unowned_Reference[T]` or a

`Soft_Reference[T]`, with automatic unwrapping and wrapping of read and written values. It is an error if id_i is not a variable name in an enclosing scope. If a let binding is used in the capture usage syntax construct, all variables bound by it are captured with the given ownership

8.8.2 Template-Specific Statements

When **`optional`** or `requires` appear in a template, the following message declarations are either optional or required. When a message declaration is optional, then its result type is always nullable. No restriction is put on required messages. When those keywords appear alone on a line, then all following message members are affected. When the keywords directly precede a message member declaration, then only that member is affected. If the keyword precedes a list of identifiers or symbols, message members of those names are affected.

When **`public`**, **`protected`** or **`private`** appear in a template, the following member declarations have their accessibility affected. When those keywords appear alone on a line, then all following message members are affected. When the keywords directly precede a message member declaration, then only that member is affected. If the keyword precedes a list of identifiers or symbols, message members of those names are affected.

The **`public use`** combination, besides importing names into scope as per (§7.13), also defines the imported names. Another way to see that is to say that the combination re-exports the imported names.

8.8.3 Contracts

A body of a contract should consist only of assert expressions, in form of **`assert`** *e*, and also be side-effect free. As a contract, an assert expression represents a guarantee that the code in its expression must uphold. The expected type of the asserted expression is `Boolean`, and can employ implicit conversions to get a boolean value, but only those available in the scope. If the asserted expression evaluates to **`no`**, then the contract is broken and `Contract_Broken` is raised.

A function be declared with contract blocks in its definition:

- Input contracts (*preconditions*) in form of **`in`** { *e* }.
- Output contracts (*postconditions*) in forms of either **`out`** { |*p*| *e* } or **`out`** { *e* }. If the variable *p* is specified, its type is inferred from the result type of the function. The variable *p* references the result value, and the contract code can't change its content.

A template may contain an invariant contract in its definition, in form of **`invariant`** { *e* }.

Contracts are by default always evaluated when needed, unless they are marked as only debugging by annotations. The times contract have to be evaluated are after constructors

are finished and around calls to public methods. Contracts are inherited by traits and classes alike, and the implicit ordering is to run inherited contracts before own contracts.

The exact location of evaluation of an inherited contract can be overridden by application of **`super`** (without arguments) in the contract's code. Contract blocks can access all parameters (except the invariant contract) of the function application, but the output contract may[25] see updated values of parameters in mutable variables.

The evaluation order of contracts code is:

1. Preconditions (excluding parameter-less constructors)
2. Invariant (excluding constructor, where the invariant may not yet be met)
3. Constructor chain or function body
4. Invariant
5. Postconditions

Applications of public methods on the object tested for invariant that are nested in any contract's code do not repeat invariant contract evaluations, instead, only the first and last are evaluated. Preconditions and postconditions are evaluated for every such nested application.

Every type can have only up to one own invariant contract. There is no limit on inheritance of invariant contracts – there is always only up to one from each parent type.

8.8.4 Scope Guard Statements

A *scope guard statement* **`scope`** `(`*l x*`)` **`do`** *e* **`done`** executes the immediately following block or expression (*e*) at the end of an enclosing scope, rather than at the point where it appears. Values defined at the point of its appearance are available also in the executed block or expression. A scope guard statements comprise also a scope ending identifier (combination of optional label *l* and required ending type *x*).

- **`scope`** `(`**`exit`**`)` executes no matter how the enclosing scope ended.
- **`scope`** `(`**`success`**`)` executes if the enclosing scope ended normally.
- **`scope`** `(`**`failure`**`)` and **`scope`** `(`**`raises`**`)` executes if the enclosing scope ended due to stack unwinding caused by a raised error.[26]

[25] Well, only if somebody does update those variables.

[26] It is possible that a particularly annotated **`goto`** may be treated as scope failure, then the former one executes.

- **`scope (throws)`** executes if the enclosing scope ended due to stack unwinding caused by a thrown value, including raised errors.

A scope guard statement with a specified label is executed always at end of the closest enclosing scope that has the same label.[27] If no label is specified, then the directly enclosing scope is taken.

Scope guard statements are executed in reverse order of their lexical appearance (like if they were stacked).

Scope guard statements are not allowed to exit by throwing a value, jumping out with a **`goto`**, using loop control expressions (such as `continue` or **`break`**) or return expressions, nor they may be entered with a **`goto`**.

The scope guard statement **`defer do`** *e* **`done`** is equivalent to **`scope (exit)`**.

8.9 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 (§8.9.4) and view applications (§12.3), if necessary.

8.9.1 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 (§8.9.3) is applied to pick a unique member.

Numeric widening. If *e* is of a number type which weakly conforms (§6.7.2) 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.

[27] Useful to execute scope guards conditionally.

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 (§12.3). 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 (§8.9.5).

8.9.2 Method Conversions

The following implicit conversions can be applied to methods which are not applied to any arguments.

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 (§12.2).

Eta expansion. Otherwise, if the expected type *et* is a function type `(Ts') -> T'`, eta-expansion (§8.9.4) is performed on the expression *e*.

Empty application. Otherwise, if *e* is of a method type `() ⟼ T`, it is implicitly applied to the empty argument list, yielding `e()`.

8.9.3 Overloading Resolution

Deprecated. Will need to rewrite this with respect to the updated function application forms, and automatic type inference.

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. Overloading resolution of e is applied after local type inference, although local type inference may play part in overloading resolution of nested argument expressions, which are applied separately.

8.9.3.1 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 are 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 $(p_1 : T_1, \ldots, p_n : T_n) \texttt{ -> } b$, the shape is $(\texttt{Any}, \ldots, \texttt{Any}) \texttt{ -> } shape(b)$, where `Any` occurs n times in the argument type.
- For a named argument $n : e$, the shape is $\texttt{@[named\_arg :}n\texttt{]}\ shape(e)$, which is an annotated type.[28]
- For all other expressions, the shape is `Nothing`.

Let $\mathscr{B}$ be the set of alternatives in $\mathscr{A}$ that are *applicable* (§8.3.4) to expressions $(e_1, \ldots, e_n)$ of types $(shape(e_1), \ldots, shape(e_n))$. 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 (§8.9.1).

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".

[28] This is different from e.g. Scala, since Aml supports capturing named arguments, which make the definition of applicable functions different.

Triggered early evaluation. If any of the corresponding parameter types of any alternative in $\mathscr{C}$ is a constrained type (§6.3.21) or non-trivial pattern[29], an early argument evaluation is triggered, exactly once per each corresponding argument, to detect whether the alternative is applicable to the constrained type or if the pattern matches.

Definition 8.9.1 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.

$$\begin{aligned} weight(A, B) &= \textstyle\sum_{i=1}^{m} pweight(A_i, B_i) \\ pweight(t, u) &= cweight(t, u) + rweight(t) \end{aligned}$$

$$\begin{aligned} cweight(t, u) &= \begin{cases} 1 & \text{if } t <: u \\ 0 & \text{otherwise} \end{cases} \\ rweight(t) &= \begin{cases} 1 & \text{unless } t \text{ is a variadic or a capturing named parameter} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

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.

Generics based overloading resolution. If there are more alternatives in $\mathscr{C}$ that are equally most specific, then let $\mathscr{D}$ be those equally most specific alternatives. One chooses the most specific alternative from $\mathscr{D}$ based on the following redefinition of being "more specific than".

Definition 8.9.2 The *generic relative weight* of an alternative A over an alternative B is a number between 0 and 1, defined as follows:

- 1, if A is not polymorphic and B is polymorphic.
- 0, in any other case.

An alternative A is *more specific than* an alternative B, if the generic relative weight of A over B is greater than the generic relative weight of B over A.

If there are more alternatives in $\mathscr{D}$ that are equally most specific, continue with overloading resolution on $\mathscr{D}$ using preference declarations. If that fails, one chooses one alternative from $\mathscr{D}$ as in overloading resolution without any application (§8.9.3.3), where $\mathscr{A}$ is the same as $\mathscr{D}$ here, and the expected type is the expected type of the function application, unless there are consecutive function applications.

[29]Non-trivial patterns are all patterns that are not variable patterns or typed patterns – but a typed pattern with a constrained type is considered non-trivial as well.

Consecutive applications based overloading resolution. If there are any consecutive function applications (§??) involved, then let $\mathscr{E}$ be those alternatives from $\mathscr{D}$ that have no following parameter lists, and let $\mathscr{F}$ be those alternatives from $\mathscr{D}$, for which one of the following conditions is true:

- There are a following consecutive function applications and the alternative has a more parameter lists. The last parameter list may or may not be marked **`implicit`**, it does not matter.
- There are a following consecutive function applications and the alternative has $a + 1$ more parameter lists, and the last one is marked **`implicit`**. Note that a might be 0, so that methods with an extra implicit parameter list are preferred.
- There are a following consecutive function applications and the alternative has more than a more parameter lists, and the whole expression containing the consecutive function applications is enclosed in a method value, for partial application.

Then, if $\mathscr{F}$ is not empty, apply overloading resolution on $\mathscr{F}$ recursively, omitting the already processed parameter list and argument list in each turn. If overloading resolution on $\mathscr{F}$ selects a unique member, that one is chosen. If no unique alternative from $\mathscr{F}$ could be selected (either because none was applicable, or there were multiple applicable alternatives at the end of recursion), continue with overloading resolution on alternatives from $\mathscr{E}$ using preference declarations. If $\mathscr{E}$ is empty, it is an error.

Declared preference based overloading resolution, using arguments preference. Let $\mathscr{G}$ be those alternatives that are supposed to be resolved based on declared preference (there are two points from which this can happen). One finds all preference declarations using arguments filter (not limited to) for each alternative over other alternatives in $\mathscr{G}$. If there is one alternative in $\mathscr{G}$ that is preferred strictly more times than any other alternative, then that one is chosen. Otherwise, one continues with overloading resolution without any application (§8.9.3.3), where $\mathscr{A}$ is the same as $\mathscr{G}$ (i.e. no elements from $\mathscr{G}$ are left out).

Note. Nested overloading resolutions happen in depth-first order. If an alternative is polymorphic, it needs to be type-reified with local type inference to first determine what the types are. If an argument expression is itself overloaded, overloading resolution needs to be applied on it first and alone, and the expected type of the argument expression defined by the local type inference algorithm. Notice that this is safe in regard to local type inference of the enclosing alternative, since the type of the argument expression is just a part of type bounds for the enclosing alternative. Indeed, ambiguities may need to be resolved explicitly.

Example 8.9.3 Assume the following overloaded function definitions:

```
method f (*x: Integer): Unit end             -- 1.
method f (x: Integer): Unit end              -- 2.
method f (x: Integer, y: Integer): Unit 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.

8.9.3.2 Function in a type application

Assume next that *e* appears as a function in an explicit type application (not inferred), 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. If there are several such alternatives, overloading resolution (different than this case: so either in a function application or not in a function application) is applied to the whole expression *e*[*targs*].

8.9.3.3 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 (§8.9) 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 8.9.4 The *relative weight* of an alternative *A* over an alternative *B* is defined as a number from 0 to 2, defined as the sum of:

- 1 if A <: B, 0 otherwise, and
- 1 if A is not polymorphic and B is polymorphic, 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.

If there are multiple most specific alternatives in $\mathscr{B}$, one chooses an alternative from $\mathscr{B}$, which is strictly more times preferred to the other alternatives in $\mathscr{B}$, based on preference declarations that include a result type filter for that alternative.[30]

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.[31]

Note. An important note is that when an identifier e references several members of a class, it also references only those members that are visible (§11.5) from the scope where e appears.

8.9.4 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-expansion is then:

```
let
  val x_1 = e_1
  ...
  val x_m = e_m
in
  fun (y_1: T_1) ... (y_n: T_n) -> e'.(y_1 ... y_n)
end
```

Example 8.9.5 A few examples of eta-expansion, the original expression and eta-expanded below it:

[30] This is needed to resolve ambiguities in cases such as when the expected type is the least upper bound of the result types of two or more overloaded alternatives.

[31] This can be fixed e.g. by using typed expressions (§8.5.1).

```
-- expression:
&((1 .. 9).fold(z))
-- expands to:
let
  val eta1 = z
  val eta2 = (1 .. 9)
in
  fun x -> eta2.fold(eta1)(x)
end
```

8.9.5 Dynamic Member Selection

`Aml` defines a trait `Dynamic_Member_Selecting` that enables dynamic invocations rewriting.

Chapter 9

Pattern Matching

Contents

9.1 Patterns

Grammar:

```
Pattern
    ::= Atomic_Pattern
      | Non_Atomic_Pattern
Atomic_Pattern
    ::= Atomic_Var_Pattern
      | Bound_Pattern
      | Typed_Pattern
        -- a portion of Type_Pattern
      | ':?' (Simple_Type - Postfix_Parameterized_Type)
      | Type_Instance_Pattern
      | Wildcard_Pattern
      | Constant_Pattern
      | Grouped_Pattern
        -- a portion of Named_Pattern
      | Stable_Id
      | List_Pattern
      | Array_Pattern
      | Vector_Pattern
      | Dict_Pattern
      | Multimap_Pattern
      | Bag_Pattern
      | Record_Pattern
      | Struct_Pattern
      | '(' Pattern ')'
Non_Atomic_Pattern
    ::= Var_Pattern
      | Type_Pattern
      | Type_Bound_Pattern
      | Infix_Pattern
      | Or_Pattern
      | And_Pattern
      | Named_Pattern
      | Tag_Pattern
      | Tuple_Pattern
      | Cons_Pattern
      | Annotated_Pattern
      | Nested_Pattern
Var_Pattern
    ::= Atomic_Var_Pattern
      | ['implicit'] [Storage_Modifier] 'val' var_id
Atomic_Var_Pattern
    ::= var_id
```

```
Bound_Pattern
    ::= Pattern 'as' Var_Pattern
Wildcard_Pattern
    ::= '_'
Constant_Pattern
    ::= Constant
Typed_Pattern
    ::= '(' Type_Bound_Pattern ')'
Type_Pattern
    ::= ['is' | ':?'] Type_Expr
Type_Bound_Pattern
    ::= Pattern ':' Type_Expr
Type_Instance_Pattern
    ::= Singleton_Type
      | Projected_Type
Grouped_Pattern
    ::= '(' Pattern ')'
Infix_Pattern
    ::= Pattern (Infix_Op - ('and also', 'or else')) Pattern
Or_Pattern
    ::= Pattern '|' Pattern
And_Pattern
    ::= Pattern '&' Pattern
Named_Pattern
    ::= Stable_Id {Pattern_Param} [Atomic_Extractions]
Tag_Pattern
    ::= Tag_Name [Pattern]
Tuple_Pattern
    ::= Pattern {',' Pattern}+
      | Tuple_Extractions
List_Pattern
    ::= '[' [Seq_Extractions] ']'
      | '%[' [Seq_Extractions] ']'
Array_Pattern
    ::= '[|' [Seq_Extractions] '|]'
      | '%[|' [Seq_Extractions] '|]'
Vector_Pattern
    ::= '#[' [Seq_Extractions] ']'
Dict_Pattern
    ::= '%{' [Dict_Extractions] '}'
      | '#{' [Dict_Extractions] '}'
Multimap_Pattern
    ::= '%{|' [Dict_Extractions] '|}'
      | '#{|' [Dict_Extractions] '|}'
Set_Pattern
    ::= '%(' [Seq_Extractions] ')'
```

```
      | '#(' [Seq_Extractions] ')'
Bag_Pattern
    ::= '%(|' [Seq_Extractions] '|)'
      | '#(|' [Seq_Extractions] '|)'
Cons_Pattern
    ::= Pattern '::' Pattern
Annotated_Pattern
    ::= {Annotation}+ Pattern
Nested_Pattern
    ::= Pattern 'with' Pattern '=' Expr
Extractions
    ::= Pattern
      | Positional_Extracts
Tuple_Extractions
    ::= Atomic_Tuple_Extractions
      | Non_Atomic_Tuple_Extractions
Atomic_Extractions
    ::= Atomic_Tuple_Extractions
      | Record_Pattern
Atomic_Tuple_Extractions
    ::= '(' Extractions ')'
Non_Atomic_Tuple_Extractions
    ::= Extractions
Positional_Extracts
    ::= Pattern {',' Pattern} [',' Rest_Extracts {',' Pattern}]
      | Rest_Extracts {',' Pattern}
Rest_Extracts
    ::= Optional_Extract {',' Optional_Extract} [',' Rest_Extract]
      | Rest_Extract
Optional_Extract
    ::= '?' Pattern
      | '?' id [':' Type_Expr]
      | '?' '_' [':' Type_Expr]
      | '?(' Pattern '=' Expr ')'
Rest_Extract
    ::= '*' id [':' Type_Expr]
      | '*' '_' [':' Type_Expr]
Seq_Extractions
    ::= Pattern {semi Pattern} [semi Seq_Rest_Extracts] {semi Pattern}
      | Seq_Rest_Extracts {semi Pattern}
Seq_Rest_Extracts
    ::= Optional_Extract {semi Optional_Extract} [semi Rest_Extract]
      | Rest_Extract
Dict_Extractions
    ::= Dict_Extraction {semi Dict_Extraction} [semi Capture_Extract]
      | Capture_Extract
```

```
      | Seq_Extractions
Dict_Extraction
    ::= Dict_Key '=' Pattern
Dict_Key
    ::= '_'
      | Expr
Capture_Extract
    ::= '**' var_id [':' Type_Expr]
      | '**' '_' [':' Type_Expr]
Record_Pattern
    ::= '{' Field_Patterns '}'
Field_Patterns
    ::= Field_Pattern {semi Field_Pattern} [semi Field_Rest_Pattern]
      | [semi] Field_Rest_Pattern
Field_Pattern
    ::= Record_Label '=' Pattern
      | Var_Pattern [':' Type_Expr] ['as' Pattern]
Field_Rest_Pattern
    ::= '_' ['=' Pattern]
Struct_Pattern
    ::= '(' 'structure' id ['as' Pkg_Type] ')'
Pattern_Param
    ::= Expr
```

Table 9.1: The relative precedences and associativity of operators and non-closed pattern constructs, in decreasing precedence order.

Operator or construct	Associativity
Infix pattern (§9.1.22)	operator-based
Named pattern, tag application	right
`::`	right
`=>`	-
`,`	left
`&`	left
`|`	left
`as`	-

9.1.1 Atomic & Non-Atomic Patterns

We call *atomic patterns* those of pattern forms that can appear consecutively on a line and still have their boundaries clear and unambiguous. On the other hand, *non-atomic patterns* could make such consecutive appearances ambiguous. This distinction is only important

in grammar constructs that allow consecutive appearance of patterns, namely function parameters.

9.1.2 Constant Patterns

Grammar:

```
Constant_Pattern
    ::= Constant
```

A constant pattern *L* matches any value that is structurally equal (defined by the `Equatable` trait, "`=`" operator returning "**`yes`**") to the constant value *L*. The type of *L* must conform to the expected type of the matched value.

Example 9.1.1 An example of a constant pattern.

```
let rotate_3 = function
  when 0 then "two"
  when 1 then "zero"
  when 2 then "one"
end function
```

A constant pattern contributes the type of the constant to type of parameter, when used in function parameter list.

9.1.2.1 Regular Expression Patterns

Grammar:

```
Pattern
    ::= Constant
```

A regular expression pattern *p* (*regexp pattern*) is a variant of constant 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.

Regular expression patterns contribute the `String_Like` type to type of parameter, when used in function parameter list.

9.1.3 Variable Patterns

Grammar:

```
Var_Pattern
    ::= ['implicit'] [Storage_Modifier] 'val' 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 variable pattern contributes the implicit type of a variable to type of parameter, when used in function parameter list. It is by default `Auto`.

A variable pattern prefixed with **`implicit`** is made eligible to be passed as implicit argument (§12.2). The value that is eligible is however the current value at the instant of function application with implicit arguments – therefore if the variable is mutable, it might be a different value than the one bound from the pattern match.

A variable pattern contributes the type `Aml/Language.Any` to type of parameter, when used in function parameter list.

9.1.4 Wildcard Patterns

Grammar:

```
Wildcard_Pattern
    ::= '_'
```

The pattern "_" is a wildcard pattern that matches any input and does not bind any value to any variable.

A wildcard pattern contributes the type `Aml/Language.Any` to type of parameter, when used in function parameter list.

Example 9.1.2 In this example, if `x` is 0, the match returns 1, if `x` has any other value, the match returns 0. The second definition is equivalent.

```
let categorize = function
  when 1 then 0
  when 0 then 1
  when _ then 0
end

let categorize_alternative = function
  when 1 then 0
```

```
  when 0 then 1
  otherwise 0
end
```

Wildcard patterns may also be used in extractions to ignore the input at the given site.

9.1.5 Named Patterns

Grammar:

```
Named_Pattern
    ::= Stable_Id {Pattern_Param} [Atomic_Tuple_Extractions]
```

In a named pattern, the `Stable_Id` part must not start with a lowercase letter as defined in `var_id`, otherwise it would be interpreted as a variable pattern.

Due to the syntactic overlap regarding variable patterns, Aml offers a simple solution to interpret an identifier starting with a lowercase letter as a value instead of a variable pattern, and that is to enclose the identifier in backquotes, so that "`id`" is "`` `id` ``". Such an identifier is never considered to be a variable pattern.

If that condition is met, then the `Stable_Id` part may resolve into the following elements:

- An enumeration element (§9.1.5.1).
- A variant type case (§9.1.5.2).
- A generalized algebraic datatype (§9.1.5.3).
- An open variant type case (§9.1.5.4).
- A throwable or raiseable type case (§9.1.5.5).
- A case class or case object (§9.1.5.6).
- An active pattern case name (§9.1.5.7).
- A value implementing `Pattern_Matching` (§9.1.5.8).
- A (likely constant) value implementing `Equatable` (§9.1.5.9).[1]
- A class object (§11.4.5) of a class that defines a primary constructor with one or more parameters (§??).

After the `Stable_Id` is resolved, the named pattern is treated according to the following subsections.

[1] The runtime should issue a warning if the value is not constant, since that condition would make the pattern match possibly behave differently at different execution times.

9.1.5.1 Enumeration Patterns

If `Stable_Id` from (§9.1.5) resolves to an enumeration element, the pattern is an enumeration pattern. Such pattern then matches inputs that are equal to the enumeration element. The form `Stable_Id` is used if the corresponding enumeration element takes no arguments or if arguments are to be ignored, and the form `Stable_Id Tuple_Extractions` is used if it takes arguments, which are then matched according to the `Tuple_Extractions` (§9.1.21).

An enumeration pattern contributes the enumeration type to type of parameter, when used in function parameter list.

Example 9.1.3 An example of an enumeration pattern with extractions:

```
type Data = enum
  case Kind_1 of Number * Number
  case Kind_2 of String * String
end

let some_data = Data.Kind_1 (4, 2)

let result = match some_data
  when Kind_1 (a, b) then a + b
  when Kind_2 (s1, s2) then s1.length + s2.length
end
```

Which gives `result` a value of `6`.

9.1.5.2 Variant Type Case Patterns

If `Stable_Id` from (§9.1.5) resolves to an variant type case, the pattern is a variant type case pattern. Such pattern then matches inputs that are members of the variant type case element. The form `Stable_Id` is used if the corresponding variant type case takes no arguments or if arguments are to be ignored, and the form `Stable_Id Tuple_Extractions` is used if it takes arguments, which are then matched according to the `Tuple_Extractions` (§9.1.21).

A variant type case pattern contributes type of the variant type case to type of parameter, when used in function parameter list.

9.1.5.3 Generalized Algebraic Datatype Patterns

Generalized algebraic datatype patterns are equivalent to variant type case patterns (§9.1.5.2), only their type inference is a bit more challenging.

9.1.5.4 Open Variant Type Case Patterns

Open variant type case patterns are by all means equivalent to variant type case patterns (§9.1.5.2), except that they work with open variant types instead of (closed) variant types.

9.1.5.5 Throwable & Raiseable Type Case Patterns

Throwable and raiseable type case patterns are equivalent to open variant patterns (§9.1.5.4).

9.1.5.6 Case Class & Case Object Patterns

If `Stable_Id` from (§9.1.5) resolves to a value that is a case class object or a case object, then the pattern is a *case class pattern* or a *case object pattern* respectively.

To match a case class/object pattern, the object's `unapply` (which is always present, unlike with constructor patterns – §??) is applied to the input value:

- If the result value is of type `Option[`T_1`, ..., `T_n`]`, and is not `None`, then the pattern matches if the extractions matches the wrapped tuple.
- If the result value is of type `Option[`T`]`, and is not `None`, then the pattern matches if the extractions matches the wrapped value.
- If the result value is of type `Boolean`, then the pattern matches if the value is **`yes`**.

A case class pattern contributes the class associated with the class object to type of parameter, when used in function parameter list. A case object pattern contributes type of the case object.

9.1.5.7 Active Patterns

Grammar:

```
Active_Pattern_Ids
    ::= '(|' id {'|' id} ['|' '_'] '|)'
```

If `Stable_Id` from (§9.1.5) resolves to an *active pattern case name* C_i, then the pattern is an active pattern. C_i is associated with an *active pattern function* f in one of the following forms:

- `(|`C`|) input`
  Single case. The function accepts one argument (the value being matched) and can return any type.

- `(|`C`|_|) input`
  Partial. The function accepts one argument and must return a value of type `Option[`T`]`, where T is any type.

- `(|`C_1`|...|`C_n`|) input`
  Multi-case. The function accepts one argument and must return a value of type T_1 **or** ... **or** T_n.

- `(|`C_1`|...|`C_n`|_|) input`
  Multi-case partial. The function accepts one argument and must return a value of type `Option[`T_1 **or** ... **or** T_n`]`.

- `(|`C_1`|) arg`$_1$`... arg`$_n$ `input`
  Single case with parameters. The function accepts $n + 1$ arguments, where the last argument (`input`) is the value to match, and can return any type.

- `(|`C_1`|_|) arg`$_1$`... arg`$_n$ `input`
  Partial with parameters. The function accepts $n + 1$ arguments, where the last argument (`input`) is the value to match, and must return a value of type `Option[`T`]`.

- `(|`C_1`|...|`C_n`|_|) input`
  Multi-case partial with parameters. The function accepts $n + 1$ arguments, where the last argument (`input`) is the value to match, and must return a value of type `Option[`T_1 **or** ... **or** T_n`]`.

When an active pattern function accepts more than 1 argument, the `Pattern_Params` are interpreted as expressions that are passed to the active pattern function.

The active pattern function f is applied at runtime to the pattern input, along with any arguments. The pattern matches if the active pattern function returns v, v, or `Some(`v`)` when applied to the pattern input. If the pattern argument `Tuple_Extractions` is present, then it is matched against v. If this does not match, the whole active pattern does not match.

Example 9.1.4 This example shows how to define and use a partial active pattern function:

```
let (|Positive|_|) input = if input > 0 then Some input else None end
let (|Negative|_|) input = if input < 0 then Some -input else None end

match 3
when Positive n then "positive, n = #{n}"
when Negative n then "negative, n = #{n}"
otherwise "zero"
end match
```

Example 9.1.5 This example then shows how to define and use a multi-case active pattern function:

```
let (|A|B|C|) input =
  if input < 0 then A
  else if input = 0 then B
  else C
  end if

match 3
  when A then "negative"
  when B then "zero"
  when C then "positive"
end match
```

Example 9.1.6 An example of a parameterized active pattern function:

```
let (|Is_Multiple_Of|_|) n input =
  if input mod n = 0
  then Some input / n
  else None
  end if

match 16
when Is_Multiple_Of 4 m then printfn "x = 4 * %d" % n
else printfn "not a multiple of 4"
end match
```

An active pattern function may be executed multiple times against the same pattern input during resolution of a single complete pattern match. The precise number is left to implementations, but it should be as close as possible to 1. Implementations of active pattern functions should be referentially transparent.

An active pattern contributes the type of `input` to type of parameter, when used in function parameter list.

9.1.5.8 Pattern Matching Values

If `Stable_Id` from (§9.1.5) resolves to a value that is not a class object or otherwise special for pattern matching, and the value is of a class that implements `Pattern_Matching[`*T*`]`, the pattern matches its input, if the result of applying `=~` to the input value with the resolved value as receiver is **`yes`**.

It is highly suspicious if the resolved value is not a constant value, and a warning should be issued.

A pattern matching value pattern contributes the type T of the implemented `Pattern_Matching[`T`]`. If the value implements `Pattern_Matching[`T_i`]` multiple times with different type arguments T_i, then the contributed type is union of all T_i.

9.1.5.9 Equatable Patterns

If `Stable_Id` from (§9.1.5) resolves to a value that is not a class object or otherwise special for pattern matching, and the value is of a class that implements `Equatable`, the pattern matches its input, if the two values are structurally equal as defined by `Equatable` (not necessarily physically equal).

It is highly suspicious if the resolved value is not a constant value, and a warning should be issued.

An equatable pattern contributes the type `Equatable` to type of parameter, when used in function parameter list. Note however that the value has to pre-exist, so the only possible sources are constant values defined in outer scopes, or one of the parameters.

9.1.5.10 Extractor Patterns

If `Stable_Id` from (§9.1.5) resolves to a value that is a class object, then the pattern is *an extractor pattern.*

To match a constructor pattern, the class object's `unapply` is applied to the input value:

- If the result value is of type `Option[(`T_1`, ..., `T_n`)]`, and is not `None`, then the pattern matches if the extractions match the wrapped tuple.
- If the result value is of type `Option[`T`]`, and is not `None`, then the pattern matches if the extractions match the wrapped value.
- If the result value is of type `Boolean`, then the pattern matches if the value is **yes**.

An extractor pattern contributes the class associated with the class object to type of parameter, when used in function parameter list.

9.1.6 Polymorphic Variant Patterns

Grammar:

```
Tag_Pattern
    ::= Tag_Name [Pattern]
```

Polymorphic variant type case patterns `` `$t$ $p$ `` matches all polymorphic variants whose tag is equal to t, and whose arguments can be extracted by tuple extractions p, if p is present.

9.1.7 Pattern Binders

Grammar:

```
Bound_Pattern
    ::= Pattern 'as' Var_Pattern
```

A pattern binder *p* **as** *x* consists of a variable pattern *x* (§9.1.3) 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.

A pattern binder contributes the type contributed by the pattern *p* to type of parameter, when used in function parameter list.

Example 9.1.7 In the following example, `person` binds to the whole `Person` object.

```
let f (someone: Person) = match someone
  when Person("John Galt", _, _) as person then ...
end match
```

9.1.8 Disjunctive "Or" Patterns

Grammar:

```
Or_Pattern
    ::= Pattern '|' Pattern
```

A disjunctive pattern matches an input value against one of two patterns: `pattern1 | pattern2`.

At runtime, the pattern input is matched against the first pattern, and if that fails, the pattern input is matched against the second pattern.

If any of the sub-patterns binds variables (§9.1.7), then both sub-patterns must bind the same set of variables of the same types, but variables are only bound by the pattern that did not fail the match (if any).

As a convenience, a variable binding that is present in any of the two patterns with a defined type (§9.1.10) does not need to repeat the same defined type in its second occurrence, it is automatically inferred. If that is not the case, the inferred type for both binding occurrences[2] must be the same, as defined earlier.

[2]For the same bound name.

Matching this pattern is short-circuited – if the first sub-pattern matches the input, then the second sub-pattern is not matched at all.

Disjunctive patterns contribute union of types contributed by both sub-patterns to type of parameter, when used in function parameter list.

9.1.9 Conjunctive "And" Patterns

Grammar:

```
And_Pattern
    ::= Pattern '&' Pattern
```

A conjunctive pattern matches an input value against two patterns sequentially: `pattern1 & pattern2`.

At runtime, the pattern input is matched against the first pattern. If that does not fail, then the input is matched against the second pattern in environment enriched by variables bound by the first pattern (if such bindings were defined in the first pattern, obviously). If the input matches the second pattern, the pattern matches the input.

Matching this pattern is short-circuited – if the first sub-pattern does not match the input, then the second sub-pattern is not matched at all.

Conjunctive patterns contribute intersection of types contributed by both sub-patterns to type of parameter, when used in function parameter list.

9.1.10 Typed Patterns

Grammar:

```
Typed_Pattern
    ::= '(' Type_Bound_Pattern ')'
Type_Bound_Pattern
    ::= Pattern ':' Type
Type_Pattern
    ::= ['is' | ':?'] Type
```

A typed pattern of the form `(`*p*`: `*T*`)` consists of a pattern *p* and a type pattern *T*. A typed pattern may also be of the forms *T*, **`is`** *T* and `:?` *T*, all of which are equivalent.

Aml does not distinguish type-annotated patterns from dynamic type-test patterns – all such patterns are dynamic type-test patterns, with implications from a type-annotated pattern.

At runtime, the pattern input is matched, if the input value's type is known to be a subtype (or equivalent) to the type *T*, or if it dynamically is of such a subtype (the input is evaluated either way).

If a pattern *p* is present in the pattern's form, then the pattern matches the input if it matches *p*, where *p* is typed with *T*. If *p* is a variable pattern (§9.1.3) or a wildcard pattern (§9.1.4), then *p* matches only if the input value's type is a subtype of *T*, as otherwise, the binding would not type-check (in case of variable pattern), or the pattern would be rendered useless (in case of wildcard pattern).

As a special case, if the pattern *p* is a variable pattern, then the bound variable is typed with *T*.

A typed pattern contributes the type *T* to type of parameter, when used in function parameter list.

9.1.11 List Patterns

Grammar:

```
List_Pattern
    ::= '%[' [Seq_Extractions] ']'
```

List patterns use `Seq_Extractions`, shared among multiple pattern kinds, but described in this section.

The simplest list pattern is the *empty list pattern*, which is of the form `%[]`, and matches empty lists (or sequences – will be using that from now on), including the constant empty list `List.Nil`.

More elaborate forms of list patterns include sequential extractions (`Seq_Extractions`), which are sub-patterns. There are three principal kinds of extractions:

1. *Mandatory extractions*. These render the minimal length of a sequence - the count of mandatory extractions is the minimal length of a sequence. Mandatory extractions are any patterns.

2. *Optional extractions*. These add to the maximal length of a sequence - the count of optional extractions added to the count of mandatory extractions is the maximal length of a sequence. Optional extractions (`Optional_Extract`) come in 4 forms:

   (a) `?`*p*, which may bind variables only if there is a variable pattern bound to the matched value. The type of such variable pattern is `Option[`*T*`]`, and if a value *v* is present for the extraction, `Some(`*v*`)` is bound to the variable, `None` otherwise.

   (b) `?`*x*`:` *T*, which are specific forms of the first form, where the variable pattern is *x*, combined with a typed pattern (§9.1.10).

   (c) `?_:` *T*, which is a specific form of the first form, where the variable pattern is a wildcard pattern, combined with a typed pattern (§9.1.10).

(d) `?(`*p* `=` *e*`)`, which is different from the previous forms. Moreover, *p* is required to bind the input value to a variable, but when there is no value to match it with, the expression *e* is evaluated and bound to the variable pattern from *p*. Thus, `None` would never be bound, and the type of the bound variable is not wrapped in `Option[`*T*`]`.[3]

3. *Rest extraction.* There may be only up to one rest extraction. When a rest extraction is present, the maximal length of a sequence is virtually infinite, overriding any other finite maximal length. Rest extractions come in just 2 forms:

   (a) `*_` & `*_:` *T*, which do not bind any elements, but optionally require the elements to be of the given type *T*. This form is only useful to match lists whose elements that are matched by this form are of no importance, and also to generally match lists of length *n* and more without binding the extra elements.

   (b) `*`*x*, `*`*x*`:` *T*, which bind the matched elements into a new sublist[4], which is bound to the variable *x*.

These extractions may appear only in the following order:

1. Any number of mandatory extractions. Those are matched to elements from the beginning of a sequence.
2. Any number of optional extractions. Those are matched from unmatched elements following the previous.
3. Zero or one rest extraction. This matches all unmatched elements.
4. Any number of mandatory extractions. Those are matched to elements from the end of a sequence and must not overlap with already matched elements. After these mandatory extractions are matched, the previous two kinds of extractions may be matched.

A list pattern contributes `Sequence[`*T*`]`[5] to type of parameter, when used in function parameter list, where *T* is inferred from its sub-patterns.

Example 9.1.8 An example of a list pattern.

```
let list_size list =
  match list
  when %[] then 0
```

[3] If *p* uses wildcard pattern instead of variable pattern to bind the input value to a variable, *e* might be evaluated for side-effects.

[4] Using a `slice` operation on the full list.

[5] Usually `Linear_Sequence[`*T*`]`, but in fact is abstract over any sequence, including indexed sequences like arrays.

```
    when %[_] then 1
    when %[_; _] then 2
    when %[_; _; _] then 3
    otherwise list.size  -- duh
    end match
```

Note that in the example, the type that would be inferred for `list` is `Sequence[Any]`.

9.1.12 "Cons" Patterns

Grammar:

```
Cons_Pattern
    ::= Pattern '::' Pattern
```

A *"cons" pattern* is a different form of a list pattern (§9.1.11), matching sequence's "head" (first element) by the first pattern and sequence's "tail" (all elements but the first one) by the second pattern.

The pattern `%[`p_1`; ...; `p_n`]` can be seen as a shorthand for a series of "`::`" and empty list patterns: p_1 `::` ... `::` p_n `::` `%[]`.

If p_2 is a variable pattern, then p_1 `::` p_2 is equivalent to `%[`p_1`; *`p_2`]`.

A single cons pattern p_1 `::` p_2 matches non-empty sequences, whose heads match p_1, and whose tail matches p_2.

Cons patterns contribute the same type as list patterns to parameters.

9.1.13 Array Patterns

Grammar:

```
Array_Pattern
    ::= '%[|' [Seq_Extractions] '|]'
```

Array pattern is very much like a list pattern (§9.1.11), except that it matches only indexed sequences.

An array pattern contributes `Indexed_Sequence[`T`]` to type of parameter, when used in function parameter list, where T is inferred from its sub-patterns.

Example 9.1.9 An example of an array pattern.

```
let vector_length vector =
  match vector
  when %[| v1 |] then v1
  when %[| v1; v2 |] then Math.sqrt (v1^2 + v2^2)
  when %[| v1; v2; v3 |] then Math.sqrt (v1^2 + v2^2 + v3^2)
  otherwise raise "Unsupported size of %d." % vector.size
  end match
```

Note that in the example, the type that would be inferred for `vector` is `Array_Like[Number_Like]`, but it could go even further:

```
Array_Like[Number_Like] with constraint { 1 <= size <= 3 }
```

9.1.14 Dictionary Patterns

Grammar:

```
Dict_Pattern
    ::= '%{' [Dict_Extractions] '}'
```

Dictionary patterns use `Dict_Extractions`, shared with multimap patterns (§9.1.15), but described only here.

The simplest dictionary pattern is the *empty dictionary pattern*, which is of the form `%{}`, and matches empty dictionaries.

More elaborate forms include dictionary extractions (`Dict_Extractions`), which are sub-patterns. There are two principal kinds of dictionary extractions:

1. Key-based extractions (using `Dict_Extraction` & optionally followed by or solely a `Capture_Extract`).
2. Sequence-based extractions (reusing `Seq_Extractions`).

Key-based extractions are of these forms:

1. Specific key extraction: *k* `=>` *p*, where *k* is either a literal value, or a value expression. Such a sub-pattern retrieves an element from the input dictionary that is mapped by the given key, and matches it against the pattern *p*.
2. Key-wildcard extraction: `_ =>` *p*, where such sub-pattern matches the first not yet matched value from the dictionary, which matches *p* at the same time (thus values that do not match the pattern are ignored and matched in the following sub-patterns, if any). If the input dictionary preserves some ordering of elements, then this kind of sub-pattern match is also deterministic.

3. Capturing extraction: `**`*v*, `**`*v*`:` *T*, `**_` & `**_:` *T*. Such sub-pattern matches all remaining elements from the input dictionary, and if *v* is present in it, binds a sub-dictionary to the variable *v*. If the type *T* is also given, all remaining elements in the input dictionary have to conform to that type. Such sub-pattern may only be present once in a dictionary pattern, and allows input dictionaries to contain more keys than just the explicitly required (or wildcarded) ones.

Dictionaries may also be matched as sequences with sequential extractions, where input dictionaries should preserve some ordering of elements. Dictionary patterns may also be used to match sequences, provided that all keys of the pattern are integral numbers.

A dictionary pattern contributes `Dictionary[`*K*`,`*T*`]` to type of parameter, when used in function parameter list, where *K* and *T* are inferred from its sub-patterns automatically, or `Any`.

9.1.15 Multimap Patterns

Grammar:

```
Multimap_Pattern
    ::= '%{{' [Dict_Extractions] '}}'
```

Multimap patterns are very much like dictionary patterns (§9.1.14), but the elements that are mapped to keys are always sequences and not (almost arbitrary) values.

9.1.16 Bag Patterns

Grammar:

```
Bag_Pattern
    ::= '%(' [Seq_Extractions] ')'
```

A bag pattern is similar to list patterns (§9.1.11), except that it matches elements of the input bag in non-deterministic order, unless the input bag actually preserves some ordering of elements.

A bag pattern does not distinguish between its two major forms, bags and sets. Each element's tally can still be obtained from the input bag.

A bag pattern contributes `Bag[`*T*`]` to type of parameter, when used in function parameter list, where *T* is inferred from its sub-patterns automatically, or `Any`.

9.1.17 Record Patterns

Grammar:

```
Record_Pattern
    ::= '{' Field_Patterns '}'
```

A record pattern enables values of record types to be decomposed into a number of record fields.

The record pattern { f_1 = p_1; ...; f_1 = p_n } matches records that define exactly the fields f_1 to f_n, and such that the value associated with f_i matches the pattern p_i, for $i = 1, \ldots, n$. If there are multiple record types that have the same fields, then at least one of f_i have to specify the field with a path to the intended type[6]: $T.f$, where T is a path to the record type, and f is the field.

The record pattern { f_1 = p_1; ...; f_1 = p_n; _ } matches records that define at least the fields f_1 to f_n, with the same rules as the previous record pattern, where the extra fields, if any, are simply discarded.[7]

Record patterns or the forms { f_1 = p_1; ...; f_1 = p_{n-1}; _ = p_n } and { _ = p_1 } match the extra fields into a new record that is then matched against p_n and p_1 respectively.

Record patterns match tuples too, if all record labels are decimal numbers.

9.1.18 Nested Pattern Matches

Grammar:

```
Nested_Pattern
    ::= Pattern 'with' (Pattern - Nested_Pattern) '=' Expr
```

A nested pattern match of the form p_1 **with** p_2 = e_2 matches first against the pattern p_1, evaluates e_2 and matches its result against p_2. Values bound from p_1 are available inside e_2. The pattern binds the combined set of bound values from both p_1 and p_2, if any. The pattern fails to match if either p_1 or p_2 does not match. The pattern p_1 may itself be or contain another nested pattern match, but p_2 must not.

A nested pattern contributes the type of the pattern p_1 to type of parameter, when used in function parameter list.

Example 9.1.10 An example of a nested pattern match.

```
match xs
  when [x; y]
```

[6]The first field should be used for this.

[7]Such a pattern is more prone to ambiguous record types. In such a case, the record type has to be resolved by specifying the path to it in at least one of the fields.

```
    with Some z = f x y
    then x + y + z
  otherwise 0
end
```

If `xs` is a list with two elements for which `f x y` returns `Some z`, then the whole pattern matches, and the match expression evaluates to `x + y + z`, otherwise it evaluates to `0`.

9.1.19 Structure Patterns

Grammar:

```
Struct_Pattern
    ::= '(' 'structure' id [':' Pkg_Type] ')'
```

A structure pattern (**structure** *n*: *T*) matches a structure with type *T* and binds it to the value name *n*. The type *T* may be omitted, if it can be inferred from the enclosing pattern.

9.1.20 Tuple Patterns

Grammar:

```
Tuple_Pattern
    ::= Pattern {',' Pattern}+
      | Tuple_Extractions
```

A *tuple pattern* has multiple forms, simplest of which is a sequence of 2 or more sub-patterns, which directly influence the required parity of any input tuple, not allowing any optional elements. Elements of the tuple are matched by their position in the tuple, which corresponds to position of each sub-pattern.

More elaborate form uses tuple extractions (`Tuple_Extractions`), described in (§9.1.21). Input tuples for such patterns may be of multiple parities, and also those tuples can have their elements extracted by the elements' labels, instead of position, as in the first form.

A tuple pattern contributes a tuple type to type of parameter, when used in function parameter list. The concrete tuple type uses annotations to mark all optional elements of the required argument as well as labels of those elements, as per sub-patterns.

9.1.21 Extractions in Patterns

Extractions are of the syntactic form `Tuple_Extractions`, as they are primarily designed to extract data from tuples and reused by other kinds of patterns.

Tuple extractions combine two approaches to extractions: positional and labelled.

Positional extractions are very much like sequential extractions, delimited by a comma "`,`" instead of a newline or semicolon. After all positional extractions are evaluated against the input tuple, labelled extractions are used to extract any remaining elements, by their given labels.

Positional extractions are used to extract elements from the sequence of all non-labelled elements of the matched tuple, and then separately labelled extractions are used to extract elements from the map (from labels to elements) of all labelled elements of that tuple. Both extractions must succeed in order for the pattern that uses tuple extractions to match. In some cases, positional and labelled elements of the matched tuple may overlap – that has only one consequence: the overlapping elements may be matched more than once.

The sequence of matched elements depends on which kinds of extractions are present:

- If only positional extractions are present, all elements are matched, stripped of their labels.
- If only labelled extractions are present, then elements without labels are not matched. If a capture extraction is present, then its value contains all elements, where the key is the element's position for elements without label, and the label for the rest that has a label.
- If both extractions are present, then both are performed. Positional extractions are performed only on elements without any label and labelled extractions are similarly only performed on labelled elements.

9.1.22 Infix Operation Patterns

Grammar:

```
Infix_Pattern
    ::= Pattern (Infix_Op - ('and also', 'or else')) Pattern
```

An infix operation pattern $p_1\ op\ p_2$ is a syntax sugar for the named pattern (§9.1.5) $op(p_1, p_2)$. The precedence, associativity and binding direction of operators in patterns is the same as in expressions (§8.3.10).

An infix operation pattern $p\ op\ (q_1, \ldots, q_n)$ is a shorthand for the named pattern $op(p, q_1, \ldots, q_n)$.

9.1.23 Grouped Patterns

Grammar:

```
Grouped_Pattern
    ::= '(' Pattern ')'
```

Patterns can be grouped together to achieve the desired associativity.

Grouped patterns contribute the least upper bound of all types contributed by the grouped patterns to type of parameter, when used in function parameter list.

9.1.24 Annotated Patterns

Grammar:

```
Annotated_Pattern
    ::= {Annotation}+ Pattern
```

Annotated patterns do not generally have any special matching rules, unless their annotations specify a different behaviour.

9.2 Irrefutable Patterns

A pattern p is *irrefutable* for input value of type T, if one of the following applies:

1. p is a variable pattern (§9.1.3),
2. p is a typed pattern $x\texttt{:}\ T'$ (§9.1.10), and $T\ \texttt{<:}\ T'$,
3. p is a named pattern $c(p_1 \ldots p_n)$ (§9.1.5), the type T is an instance of a class c, the primary constructor (§11.6) of type T has argument types $T_1, \ldots, T_n$, and each p_i is irrefutable for type T_i.

An irrefutable pattern is a pattern that always matches its input (hence irrefutable).

9.3 Pattern Matching Expressions

For grammar, refer to (§8.6.3).

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 (§6.7.4) 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` 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 not`** 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 (§15) 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 (§11.5), 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` being raised at runtime.

A when clause that is not the first appearing may be prefixed with **`next`** on the preceding line, in which case control falls through to its code from the previous when clause, but only if the prefixed when clause does not bind any variables that are not present in the preceding when clause. A bound variable is present in the preceding when clause if its inferred or bound type is equivalent to the inferred or bound type of a bound variable in the prefixed clause with the same name. The variables in the prefixed when clause are persisted from the preceding when clause.

9.4 Pattern Matching Anonymous Functions

For grammar, refer to (§8.2.20).

An anonymous pattern matching function can be defined by the form

```
function
  when p_1 then b_1
  ...
  when p_k then b_k
  else b_{k+1}
end
```

The expression is taken to be equivalent to the anonymous function:

```
fun (x_1: S_1, ..., x_j: S_j) ->
  match (x_1, ..., x_j)
    when p_1 then b_1
    ...
    when p_k then b_k
    else b_{k+1}
  end
```

Here, each x_i is a fresh name. As was shown in (§8.2.20), 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.

```
object (Function[S_1, ..., S_j, T])
  method apply (x_1: S_1, ..., x_j: S_j): T = match (x_1, ..., x_j)
    when p_1 then b_1
    ...
    when p_k then b_k
    else b_{k+1}
  end
end
```

Example 9.4.1 Here is a method which uses a fold-left operation `/:` to compute the scalar product of two vectors:

```
let 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:

```
fun (x, y) ->
  match (x, y)
    when (a, (b, c)) then a + b * c
  end
```

An anonymous pattern matching function can also be defined with a single atomic pattern and more simple syntax.

Chapter 10

Type Definitions

Contents

10.1 Type Definitions

Grammar:

```
Type_Definition
    ::= 'type' Typedef {'and' ['type'] Typedef} 'end'
        -- see additional rules below
Type_Declaration
    ::= 'type' Typedcl {'and' ['type'] Typedcl} 'end'
        -- see additional rules below
      | 'type' Type_Name [Type_Abstract]
Type_Name
    ::= [Type_Context] id [Type_Param_Clause] [Given_Clauses]
Postfix_Type_Name
    ::= [Type_Context] Prefix_Type_Param_Clause id [Given_Clauses]
Typedef
    ::= Type_Name [Type_Information]
Typedcl
    ::= Type_Name [Type_Abstract]
      | Postfix_Type_Equation
Type_Information
    ::= [Type_Equation [Type_Re-export] | Type_Abstract]
      | [Type_Representation]
Type_Equation
    ::= '=' Type_Expr
Postfix_Type_Equation
    ::= 'postfix' Postfix_Type_Name '=' Type_Expr
Type_Abstract
    ::= ['>:' Type_Expr] ['<:' Type_Expr] {'<%' Type_Expr} {':' Type_Expr}
Type_Re-export
    ::= '=' '(' 'include' Long_Id ')'
      | Type_Representation
Type_Representation
    ::= New_Type_Representation
New_Type_Representation
    ::= '=' 'new' Type_Expr
```

Additional grammar rules for type definitions and declarations. The connecting "`['type']`" may be needed in some context to resolve ambiguities (e.g. when colliding with intersection types), and is required when followed by either one of `Type_Class_Dcl`, `Type_Family_Dcl` or `Type_Instance_Def`[1].

[1]So that the code will read always as **`type class`** or **`type family`** no matter what, because a **`class`** alone is a different from **`type class`**.

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 (§6.7.2).

The scope of a type parameter extends over the bounds `>:` *L* `<:` *U*, the type parameter clause *tps* itself and any type parameter clauses that *tps* is nested in.

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

Example 10.1.1 The following are legal type declarations and aliases:

```
type Integer_List = List[Integer] end
type T <: Comparable[T]
type Two['a] = Tuple_2['a, 'a] end
type My_Collection[+'x] <: Iterable['x]
```

The following are illegal:

```
type Abs = Comparable[Abs] end -- recursive type alias

type S <: T  -- S, T are bounded by themselves
type T <: S

type T >: Comparable[T.That]  {- can't select from T,
                              T is an abstract type -}
type My_Collection <: Iterable
  {- type constructor members must explicitly state
  their type parameters -}
```

10.1.1 Type Aliases

A *type alias* **type** *t* `=` *T* **end** 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* **end**. The scope of a type parameter extends over to the right hand side *T* and the type parameter clause *tps* itself.

10.1.2 Postfix Type Aliases

A *postfix type alias* **type postfix** *p t* = *T* **end** defines *t* to be a postfix alias for the type *T* parameterized by prefix type parameters *p*. This enables the use of *t* as a postfix type.

10.1.3 New Types

A *new type* **type** *t* = **new** *T* **end** defines a new type *t* whose representation is the same as that of an already existing type *T*. It differs from a *type alias* in that it creates a distinct type that must be used to explicitly coerce to and from the aliased type.[2] Type conversions from and to the aliased type ought to be runtime overhead-free, unless used with the `Dynamic` type.

10.2 Aspect Definitions

Grammar:

```
Type_Definition
    ::= ...
      | Aspect_Definition
Aspect_Definition
    ::= 'aspect' [Aspect_Clause] Aspect_Def
Aspect_Def
    ::= Type_Name Aspect_Tmpl_Env
Aspect_Tmpl_Env
    ::= '=' 'object' Aspect_Stat {semi Aspect_Stat} 'end'
Aspect_Stat
    ::= Advice_Def
      | Pointcut_Def
      | Trait_Field
Pointcut_Def
    ::= 'pointcut' id
        ['(' [id [':' Type_Expr]] {',' [id [':' Type_Expr]]} ')']
        '=' Pointcut_Type 'end'
Pointcut_Type
    ::= 'invoke' (id | regular_expression_literal)
      | 'get' (id | regular_expression_literal)
```

[2] This is not unlike Haskell's **newtype**.

```
      | 'set' (id | regular_expression_literal)
      | 'handler' Type
      | 'returning' [Id_Or_Type]
      | 'signalling' [Id_Or_Type]
      | 'throwing' [Id_Or_Type]
      | 'raising' [Id_Or_Type]
      | 'yielding' [Id_Or_Type]
      | 'advice' 'execution' [(id | regular_expression_literal)]
      | 'self' Id_Or_Type
      | 'target' Id_Or_Type
      | 'arguments' {'(' Id_Or_Type {',' Id_Or_Type} ')'}+
      | 'arguments' '(' ')'
      | ('if' | 'unless') '(' Expr ')'
      | 'not' Pointcut_Ref
      | Pointcut_Ref 'and' Pointcut_Ref
      | Pointcut_Ref 'or' Pointcut_Ref
      | '(' Pointcut_Ref ')'
      | 'retain' [Id_Or_Type]
      | 'release' [Id_Or_Type]
Pointcut_Ref
    ::= id ['(' id {',' id} ')']
      | Pointcut_Type
Advice_Def
    ::= 'advice' Advice_Spec (Pointcut_Ref | Pointcut_Type) Block_Argument
Id_Or_Type
    ::= id [':' Type_Expr]
      | Type_Expr
Advice_Spec
    ::= 'before'
      | 'after'
        [('returning' | 'throwing' | 'raising' | 'yielding' | 'signalling')
         [Id_Or_Type]]
      | 'around'
Aspect_Clause
    ::= 'per-self' [Pointcut_Ref]
      | 'per-target' [Pointcut_Ref]
```

Only for `Block_Argument` inside of `Advice_Def`:

```
Simple_Expr
    ::= ...
      | 'joinpoint' ['.' Selection]
```

10.3 Enumeration Type Definitions

Grammar:

```
Type_Representation
    ::= ...
      | '=' 'enum' Enum_Field {semi Enum_Field}
        [Record_Extensions]
Enum_Field
    ::= {Annotation} 'case' id ['=' Expr]
```

Enums (short for Enumerations) are types that contain constants.

An enum constant can be enhanced with custom parameters, which are then passed to the appropriate custom constructor. Enum definitions should only appear in three forms, called *constant enum constructors*

- Initialized with a constant. Each enum constant then have a method `value`, which corresponds to the passed literal. Each literal passed has to be of the same type, and numeric and character literals get an auto-incremented value for every following enum constant definition that omits its explicit value. At least the first enum constant has to be initialized with a scalar literal. The comparison operators (e.g. "<") are automatically added and their implementations reflect the order in which the enum constants appear.
- Initialized with nothing, then each enum constant is by itself the ordering, no implicit literal value is added. The comparison operators are automatically added and their implementations reflect the order in which the enum constants appear.
- Initialized with custom expression. The argument expression can be extracted with pattern matching as a contained single value, or if the argument expression is a tuple, as a tuple. It is highly recommended for every enum constant to be immutable, but it is not mandatory. This form can't extend any parent class and has the same implicit ordering as the previous form.

10.4 Variant Type Definitions

Grammar:

```
Type_Representation
    ::= ...
      | Variant_Type_Def
Variant_Type_Def
```

```
    ::= '=' 'variant' [nl] (Variant_Type_Representation)
        [Record_Extensions]
Variant_Type_Representation
    ::= Variant_Field {semi Variant_Field}
Variant_Field
    ::= Type_Constr_Dcl
Type_Constr_Dcl
    ::= {Annotation} 'case' id ['of' Type_Constr_Params]
Type_Constr_Params
    ::= Types
```

Variant types are similar to a simplified form for a series of case classes (§11.6.3) and case objects (§11.10.1), with shared interface and a common ancestor. Pattern matching is possible on variant types, as it would be done on case classes and case objects.

Variant type is the algebraic sum type. Unlike unions (§6.3.11), variant type is a disjoint union – its values are pairwise disjoint. The contained values however need not be pairwise disjoint.

Unlike enumerations (§10.3), variant types are not just constants, they instead define classes of values.

The form *id* **of** *T* is called *non-constant variant constructors*. The objects that represent non-constant variant constructors can be memoized by the runtime, so that there is at most one singleton object per every arguments combination, but the runtime does not need to guarantee that such an object will be physically identical at different times. The contained values can be extracted with pattern matching, similar to enumerations, using tuple patterns (§9.1.20).

Note. Non-constant variant constructors are internally type-parameterized with the same type parameters as the variant type.

The form *id* is called *constant variant constructors*. The objects that represent constant variant constructors are singletons. Unlike with regular union types in Aml, constant variant constructors are not pre-existing types, but rather new types.

Variant types may be parameterized, therefore polymorphic. Overloading on type parameters is possible.

Record extension members, if given, are added to the type object, unless they are specified with **member** keyword, then they are added to the type instances. Implementations and declarations of interfaces are automatically added to the type instances.

Example 10.4.1 An example of a variant type.

```
type B-Tree['t] = variant
```

```
    case Empty
    case Node of 't * B-Tree['t] * B-Tree['t]
    with member is_member? x b-tree =
      match b-tree
      when Empty then no
      when Node (y, left, right) then
        if x = y then yes
        else if x < y then is_member? x left
        else is_member? x right
        end
      end
    and member insert x b-tree =
      match b-tree
      when Empty then Node (x, Empty, Empty)
      when Node (y, left, right) then
        if x <= y then
          Node (y, insert x left, right)
        else
          Node (y, left, insert x right)
        end
      end
  end
```

Unlike enumeration types (§10.3), variant types do not enumerate a limited set of values, but instead represent classes of values with non-constant variant constructors and enumerate a limited set of singleton values with constant variant constructors[3].

10.4.1 Generalized Algebraic Datatype Definitions

Grammar:

```
Variant_Type_Def
    ::= ...
      | '=' 'variant' [nl] (Gadt_Type_Representation)
        [Record_Extensions]
Gadt_Type_Representation
    ::= Gadt_Field {semi Gadt_Field}
Gadt_Field
    ::= {Annotation} 'case' id ':' [Types '->'] Type_Expr
        -- ':' instead of 'of' is the important distinction here
```

[3]Thus a variant type that has only constant variant constructors is identical to an enumeration type, which is a situation to be avoided.

10.4.2 Extensible Variant Types

Grammar:

```
Type_Representation
    ::= ...
      | '=' 'open' 'variant' [nl] [Variant_Type_Representation]
        [Record_Extensions]
Struct_Spec
    ::= ...
      | 'type' Type_Name
        '+=' Type_Constr_Dcl {semi Type_Constr_Dcl}
Struct_Def
    ::= ...
      | 'type' Type_Name
        '+=' Type_Constr_Def {semi Type_Constr_Def}
Type_Constr_Def
    ::= Type_Constr_Dcl
      | 'case' id '=' Long_Id
```

10.5 Record Type Definitions

Grammar:

```
Type_Representation
    ::= ...
      | '=' Record_Type_Representation
Record_Type_Representation
    ::= Record_Components
        [Record_Extensions]
Record_Components
    ::= '{' Record_Component {semi Record_Component} '}'
Record_Component
    ::= {Annotation} {Record_Modifier} id ':' Type_Expr
Record_Extensions
    ::= [nl] 'with' Record_Extension {'and' Record_Extension}
Record_Extension
    ::= Tmpl_Member
      | Tmpl_Ifc_Impl
      | Tmpl_Ifc_Dcl
      | {Annotation}+ Record_Extension
Record_Modifier
    ::= Access_Modifier
      | Mutability
```

```
Type_Constr_Params
    ::= ...
      | Record_Components
```

10.6 Structure, Signature & Functor Type Definitions

Grammar:

```
Struct_Type
    ::= Struct_Type_Path
      | 'sig' [Struct_Spec {semi Struct_Specification}] 'end'
      | 'functor' {'(' [id ':' Struct_Type] ')'}+ '->' Struct_Type
      | Struct_Type {'->' Struct_Type}+
      | Struct_Type 'with' Struct_Constraint {'and' Struct_Constraint}
      | '(' Struct_Type ')'
      | 'signature' 'of' Struct_Expr
Struct_Specification
    ::= ['implicit'] 'val' id [Type_Param_Clause] ':' Type_Expr
      | Type_Declaration
      | Objects_Specification
      | 'structure' id ':' Struct_Type {'and' id ':' Struct_Type}
      | 'signature' id
      | 'signature' id '=' Struct_Type {'and' id '=' Struct_Type}
      | 'structure' 'rec' id ':' Struct_Type {'and' id ':' Struct_Type}+ 'end'
      | ['public'] 'open' Struct_Path
      | ['public'] 'open!' Struct_Path
      | ['public'] Use_Clause
      | 'include' Struct_Type
      | {Annotation}+ Struct_Specification
Struct_Expr
    ::= Struct_Path
      | 'struct' [Struct_Items] 'end'
      | 'functor' {'(' [id ':' Struct_Type] ')'}+ '->' Struct_Expr
      | Struct_Expr {'(' [Struct_Expr] ')'}+
      | '(' Struct_Expr ')'
      | '(' Struct_Expr Struct_Ascription ')'
      | '(' 'val' Expr [':' Pkg_Type] ')'
      | 'let' Struct_Defs 'in' Struct_Expr 'end'
Struct_Items
    ::= Struct_Item {semi Struct_Item}
Struct_Item
    ::= Struct_Def
```

```
      | Do_Binding
Struct_Defs
    ::= Struct_Def {semi Struct_Def}
Struct_Def
    ::= [Access_Modifier] Val_Def
      | [Access_Modifier] (Type_Definition - Type_Abstract)
      | Objects_Definition
      | [Access_Modifier] 'structure' id [Struct_Ascription]
        '=' Struct_Expr
      | [Access_Modifier] 'structure' 'rec' id [Struct_Ascription]
        '=' Struct_Expr
        {'and' id [Struct_Ascription] '=' Struct_Expr}
      | ['public'] 'open' Struct_Path
      | ['public'] 'open!' Struct_Path
      | ['public'] Use_Clause
      | 'include' Struct_Expr
      | 'local' Struct_Defs 'in' Struct_Defs 'end'
      | {Annotation}+ Struct_Def
Struct_Ascription
    ::= ':' Struct_Type
      | ':>' Struct_Type
Struct_Constraint
    ::= 'type' Simple_Type Type_Equation
      | 'struct' Struct_Path '=' Extended_Struct_Path
        -- substituting inside signature:
      | 'type' Simple_Type ':=' Type_Expr
      | 'struct' id ':=' Extended_Struct_Path
Local_Bindings
    ::= ...
      | Struct_Defs
Simple_Type
    ::= ...
      | Structural_Type
Structural_Type
    ::= '(' 'structure' Infix_Type ')'
      | '(' 'structure' 'of' Expr ')'
      | '##' ? no whitespace ? Stable_Id
      | '##(' Infix_Type ')'
      | '##{' Expr '}'
Pkg_Type
    ::= Extended_Struct_Path '.' id ['with' Pkg_Constraint {'and' Pkg_Constraint}]
Pkg_Constraint
    ::= 'type' Extended_Struct_Path '.' id Type_Equation
Struct_Path
    ::= Long_Id
Extended_Struct_Path
```

```
    ::= Extended_Struct_Name {'.' Extended_Struct_Name}
Extended_Struct_Name
    ::= id {'(' [Extended_Struct_Path] ')'}
Long_Id
    ::= ...
      | Extended_Struct_Path
```

10.6.1 Structures

The motivation for structures is to package together related definitions, such as data type definitions and associated operations over that types, and to avoid running out of names. Such a package is called a *structure* and is introduced by the **`struct ... end struct`** construct, which contains an arbitrary sequence of definitions (their order is insignificant). A structure is also introduced by the **`module ... end module`** construct, unifying modules with structures.

Components of a structure can be overloaded and polymorphic.

A structure is usually given a name with the **`structure`** or **`module`** binding.

Example 10.6.1 A structure packaging together a type of priority queues and their operations:

```
structure Priority_Queue =
struct
  type Priority = Number.Integer
  type 'a Queue = variant
    case Empty
    case Node of Priority * 'a * 'a Queue * 'a Queue
  end
  let empty = Empty
  let rec insert queue priority element =
    match queue
    when Empty then Node (priority, element, Empty, Empty)
    when Node (p, e, left, right) then
      if priority <= p then
        Node (priority, element, insert right p e, left)
      else
        Node (p, e, insert right priority element, left)
      end
    end
  class Queue_is_Empty = object inherit Raiseable; end
  let rec remove_top = function
    when Empty then raise Queue_is_Empty
    when Node (_, _, left, Empty) then left
```

```
    when Node (_, _, Empty, right) then right
    when Node (_, _, Node(l_prio, l_elt, _, _) as left,
                   Node(r_prio, r_elt, _, _) as right) then
      if l_prio <= r_prio then
        Node (l_prio, l_elt, (remove_top left), right)
      else
        Node (r_prio, r_elt, left, (remove_top right))
      end
    end
  let extract = function
    when Empty then
      raise Queue_is_Empty
    when Node (priority, element, _, _) as queue then
      (priority, element, remove_top queue)
    end
end
```

Outside the structure, its components can be accessed as object properties[4].

Example 10.6.2 Accessing structure components.

```
Priority_Queue.insert Priority_Queue.empty 1 "hello"
```

10.6.2 Signatures

Signatures are interfaces of structures. A signature specifies which components of a structure are accessible from outside of the structure, and with which type. They can be used to hide some components of a structure, like local function definitions.

Example 10.6.3 A signature that specifies three priority queue operations `empty`, `insert` and `extract`, but hides the function `remove_top`. Moreover, it makes the `Queue` type abstract, by not providing any actual representation as a concrete type.

```
signature A_Priority_Queue =
sig
  type Priority = Number.Integer end -- concrete type
  type 'a Queue                      -- abstract type
  val empty: 'a Queue
  val insert: 'a Queue -> Number.Integer -> 'a -> 'a Queue
  val extract: 'a Queue -> Number.Integer * 'a * 'a Queue
  class Queue_is_Empty : object inherit Raiseable; end
end
```

[4]Since everything in Aml is an object, structures are also objects.

Restricting the `Priority_Queue` structure by this signature results in another view of the `Priority_Queue`, where the `remove_top` function is not accessible and the actual representation of priority queues is hidden, using opaque ascription "`:>`", where transparent ascription "`:`" would not hide anything from the structure, including actual concrete type representations.

```
structure Abstract_Priority_Queue =
  (Priority_Queue :> A_Priority_Queue)
```

The same restriction could also be performed during the original definition of the structure:

```
structure Priority_Queue :> A_Priority_Queue =
struct
  ...
end
```

10.6.3 Functors

Functors are functions from structures to structures. They are used to express parameterized structures: a structure A is parameterized by structures $B_1 \ldots B_n$ is a functor F with formal parameters $B_1 \ldots B_n$, along with the expected signatures for each B_i. The functor F can be applied to many possible implementations of each B_i.

Example 10.6.4 A structure implementing sets as sorted lists, parameterized by a structure providing the type of the set elements and an ordering function over this type, to keep the sets sorted:

```
type Comparison_Result =
variant
  case Less
  case Equal
  case Greater
end

signature Ordered_Type =
sig
  type T
  val compare: T -> T -> Comparison_Result
end

structure Set_Functor =
functor (Element_Type: Ordered_Type) ->
  struct
    type Element = Element_Type.T end
```

```
    type Set = Element List end
    let empty = %[]
    let rec add x s = match s
      when %[] then %[x]
      when %[head, *tail] then
        match Element_Type.compare x head
        when Equal then s
        when Less then x :: s
        when Greater then head :: add x tail
        end
      end
    let rec is_member? x s = match s
      when %[] then no
      when %[head, *tail] then
        match Element_Type.compare x head
        when Equal then yes
        when Less then no
        when Greater then is_member? x tail
        end
      end
  end
```

10.7 Range, Floating & Fixed Point Subtype Definitions

Grammar:

```
Type_Representation
    ::= ...
      | '=' FP_Type_Def
      | '=' FP_Subtype_Def
      | '=' FP_Range_Def
FP_Type_Def
    ::= (FiP_Type_Def | FlP_Type_Def)
FP_Subtype_Def
    ::= Type (FiP_Subtype_Def | FlP_Subtype_Def)
FP_Range_Def
    ::= FP_Range
FlP_Type_Def
    ::= FP_Digits [FP_Range]
FlP_Subtype_Def
    ::= FP_Digits [FP_Range]
      | FP_Range
```

```
FiP_Type_Def
    ::= FP_Delta [FP_Range]
      | FP_Delta FP_Digits [FP_Range]
FiP_Subtype_Def
    ::= FP_Delta [FP_Digits] [FP_Range]
      | FP_Digits [FP_Range]
      | FP_Range
FP_Digits
    ::= 'digits' Expr
FP_Delta
    ::= 'delta' Expr
FP_Range
    ::= 'range' Expr ('..' | '..<') Expr
      | 'range' Type
```

The described syntaxes are for definitions of 4 special types of values:

1. Range subtypes.
2. Floating point types.
3. Ordinary fixed point types.
4. Decimal fixed point types.

Range subtypes. A range subtype (the `FP_Range_Def` syntax category) is a type defined by a lower and upper bounds. The expected type of both bounds is `Magnitude`. Such a range type may be used in combination with the following subtypes to constrain them, or standalone as a regular range value. The lower bound must be lower than or equal to the upper bound. The lower bound may be negative infinity, the upper bound may be positive infinity. The range subtype itself is a subtype of `Magnitude`, or more precisely, a subtype of the least upper bound of types of the bounds, selecting a range of its values. For floating and fixed point types, it has to be a range of `Real` values.

Floating point types.
A floating point type (the `FlP_Type_Def` syntax category) is a way to define an appropriate representation of a floating point number, based on the required accuracy instead.

> The *requested decimal precision*, which is the minimum number of significant decimal digits required for the floating point type, is specified by the value of the expression given after the keyword **`digits`**. Such expression is expected to be of `Integer` type.
>
> The bounds of the range specification are expected to be `Real` type; the types do not need to be the same.[5]

[5] E.g., one bound can be an integer, the other a real number.

The requested decimal precision shall be positive and not greater than an implementation-defined precision limit in `Number.Max_Base_Digits`. If the range specification is omitted, the requested decimal precision shall be not greater than `Number.Max_Digits`.

A floating point type definition is illegal if the implementation does not support a floating point type that satisfies the requested decimal precision and range.

A subtype of a floating point type is compatible to the parent type if the digits of the subtype are not greater than the digits of the parent type, and its range fits to the range of the parent type.

Example 10.7.1 Examples of floating point types and subtypes:

```
type Coefficient =
  digits 10 range -1.0 ..< 1.0
end

type Mass =
  digits 7 range 0.0 ..< 1.0e+35
end

-- a subtype with a smaller range
type Probability =
  Real range 0.0 ... 1.0
end
```

Fixed point types. An ordinary fixed point type (the first branch of the `FiP_Type_Def` syntax category) is a way to define a decimal type, based on the given delta. A decimal fixed point type (the second branch of the `FiP_Type_Def` syntax category) is a way to define a decimal type, based on the given delta and number of needed digits.

The error bound of a fixed point type is specified as an absolute value, called the *delta* of the fixed point type.

For a type defined by the fixed point type definition, the delta of the type is specified by the value of the expression given after the keyword **`delta`**; this expression is expected to be of a `Real` type. For a type defined by the decimal fixed point definition, the number of significant decimal digits is specified by the expression given after the keyword **`digits`**; this expression is expected to be of `Integer` type.

The expressions given after the reserved keywords **`delta`** and **`digits`** shall result in positive values.

The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type.

For ordinary fixed point type, the small is an implementation-defined power of 2 not greater than the delta, unless annotation `@[fixed_point_small s]` is applied to the type, defining the small to be *s*, where *s* is not greater than the delta.

For decimal fixed point type, the small equals the delta; the delta shall be a power of 10. If a range specification is given, both bounds of the range shall be in the range $-(10^{digits} - 1) * delta \ \ldots \ (10^{digits} - 1) * delta$.

A fixed point type definition is illegal if the implementation does not support a fixed point type with the given small and specified range or digits.

An ordinary fixed point type definition defines an ordinary fixed point type, whose base range includes at least all multiples of the small that are between the bounds defined by the range specification, if it is given, or between negative infinity to positive infinity[6], if the range specification is not given.

A decimal fixed point type definition defined a decimal fixed point type, whose base range includes at least the range $-(10^{digits} - 1) * delta \ \ldots \ (10^{digits} - 1) * delta$.

If a decimal fixed point type definition does is not given a range specification, then an implicit range $-(10^{digits} - 1) * delta \ \ldots \ (10^{digits} - 1) * delta$ is specified for it.

A subtype of a decimal fixed point type is compatible to the parent type if the digits of the subtype are not greater than the digits of the parent type, and its range fits to the range of the parent type.

Example 10.7.2 Examples of fixed point types and subtypes:

```
type Volt =
  delta 0.125 range 0.0 ..< 255.0
end

type Fraction =
  delta Number.Fine_Delta range -1.0 ..< 1.0
end

type Money =
  delta 0.01 digits 15
end

type Salary =
  Money digits 10
end
```

[6]Using the `Decimal` types.

10.8 Type Classes & Type Families

Grammar:

```
Typedef
    ::= ...
      | Type_Class_Dcl
      | Type_Family_Dcl
      | Type_Instance_Def
Typedcl
    ::= ...
      | Type_Class_Dcl
      | Type_Family_Dcl
Type_Class_Dcl
    ::= 'class' Type_Name Type_Class_Dcl_Body
Type_Family_Dcl
    ::= 'family' Type_Name [Type_Family_Closed_Def]
Type_Instance_Def
    ::= Type_Class_Instance_Def
      | Type_Family_Instance_Def

Type_Class_Dcl_Body
    ::= 'with' [Type_Class_Spec {'and' Type_Class_Spec}]
      | Type_Equation
Type_Class_Spec
    ::= Struct_Spec
      | Struct_Def  -- to define "default" implementations if instance omits it
Type_Class_Instance_Def
    ::= ['class'] 'instance' Type_Instance_Id Type_Class_Instance_Body
Type_Class_Instance_Body
    ::= 'with' Template_Stat {'and' Template_Stat}
      | Type_Equation

Type_Family_Closed_Def
    ::= 'with' [Type_Family_Instance_Def {'and' Type_Family_Instance_Def}]
      | Type_Equation
Type_Family_Instance_Def
    ::= ['family'] 'instance' Type_Instance_Id Type_Family_Instance_Body
Type_Family_Instance_Body
    ::= Type_Equation
      | Type_Representation

Type_Instance_Id
    ::= [Type_Context] id Type_Args [Given_Clauses]
      | [Type_Context] Atomic_Type_Arg id [Given_Clauses]
```

Chapter 11

Classes & Objects

Contents

11.1 Blueprints of Objects

Grammar:

```
Objects_Definition
    ::= Objects_Binding {'and' Objects_Binding}
Objects_Binding
    ::= Class_Definition
      | Object_Definition
      | Trait_Definition
      | Protocol_Specification
      | Extension_Definition
      | Refinement_Definition
Objects_Specification
    ::= Objects_Spec {'and' Objects_Spec}
Objects_Spec
    ::= Class_Specification
      | Object_Specification
      | Trait_Specification
      | Protocol_Specification
      | Extension_Specification
      | Refinement_Specification

Class_Definition
    ::= Class_Modifiers 'class' Type_Name
        [Access_Modifier] [Param_Clauses] [':' Class_Type]
        ('=' Class_Expr | '+=' Trait_Expr)
Object_Definition
    ::= Object_Modifiers 'val' Type_Name [':' Trait_Type]
        ['=' | '+='] Standalone_Object_Expr
Trait_Definition
    ::= Trait_Modifiers 'trait' Type_Name [':' Trait_Type]
        ('=' | '+=') Trait_Expr
Extension_Definition
    ::= Trait_Modifiers 'extension' [id 'for'] Parameterized_Type
        [Given_Clauses] [':' Class_Body_Type]
        ('=' | '+=') Extension_Expr
Refinement_Definition
    ::= 'refinement' Parameterized_Type [':' Trait_Type]
        '=' Trait_Expr

Class_Specification
    ::= Class_Modifiers 'class' Type_Name [Param_Clauses] ':' Class_Type
Object_Specification
    ::= Object_Modifiers 'val' Type_Name ':' Object_Type
Trait_Specification
```

```
    ::= Trait_Modifiers 'trait' Type_Name ':' Trait_Type
Protocol_Specification
    ::= 'protocol' Type_Name ':' Class_Body_Type
Extension_Specification
    ::= Trait_Modifiers 'extension' [id 'for'] Parameterized_Type
        [Given_Clauses] ':' Class_Body_Type
Refinement_Specification
    ::= 'refinement' Parameterized_Type ':' Trait_Type

Class_Modifiers
    ::= [Access_Modifier] ['implicit'] ['open']
        ('final' | 'sealed' | 'abstract' | 'case')
Trait_Modifiers
    ::= [Access_Modifier] ['open']
Object_Modifiers
    ::= [Access_Modifier] ['implicit'] ['open'] ['case']
```

11.2 Class Definitions & Specifications

11.2.1 Class Definitions

Grammar:

```
Class_Expr
    ::= -- only in non-recursive occurrences:
        Parameterized_Type
      | Function_Declarations [Expr semi] Class_Expr
        -- in both recursive and non-recursive occurrences:
      | '(' Class_Expr [':' Class_Type] ')'
      | 'fun' [Param_Clauses] [Guard] '->' [Function_Declarations] Class_Expr
      | 'object' Class_Body 'end'
      | 'begin' Class_Expr 'end'
      | Class_Use_Expr
      | Class_Binding_Expr
      | Expr semi Class_Expr
Class_Body
    ::= [Instance_Self_Type] {[nl] Class_Field}
Instance_Self_Type
    ::= '(' id [':' Type_Expr] ')'
        -- here, id can be also 'self', which is the default
Class_Use_Expr
    ::= Class_Use_Expr_As
      | Class_Use_Aspect
```

```
      | Class_Use_Refinement
      | Use_Clause In_Sep Class_Expr
        -- no Class_Use_Droppable intentionally
Class_Use_Expr_As
    ::= 'use' Expr ('as' | 'as!' | 'as?')
        [id ':'] Type In_Sep Class_Expr
Class_Use_Aspect
    ::= 'use' 'aspect' Stable_id In_Sep Class_Expr
Class_Use_Refinement
    ::= 'use' 'refinement' Stable_Id In_Sep Class_Expr
Class_Binding_Expr
    ::= ['implicit'] Val_Def In_Sep Class_Expr
      | 'let' 'structure' id {'(' [id ':' Struct_Type] ')'} [':' Struct_Type]
        '=' Struct_Expr In_Sep Class_Expr

Class_Field
    ::= Class_Structure_Field
      | Class_Member_Field
      | Class_Extra_Field
      | Class_Interface_Fields
Class_Structure_Field
    ::= 'inherit' Long_Id [Arguments] ['as' id] [Cond_Modifier]
      | 'inherit' 'val' (Expr - Binding_Expr) 'as' id [Cond_Modifier]
      | Class_Structure_Include
Class_Structure_Include
    ::= 'include' Long_Id [Cond_Modifier]
Class_Template_Member_Field
    ::= [Class_Member_Modifiers] Ival_Def
      | [Class_Member_Modifiers] Property_Def
      | [Class_Member_Modifiers] Method_Def
      | [Class_Member_Modifiers] Operator_Def
      | [Class_Member_Modifiers] Attribute_Def
      | ['abstract'] [Class_Member_Modifiers] Ival_Spec
      | ['abstract'] [Class_Member_Modifiers] Property_Spec
      | ['abstract'] [Class_Member_Modifiers] Message_Dcl
      | ['abstract'] [Class_Member_Modifiers] Operator_Dcl
      | ['abstract'] [Class_Member_Modifiers] Attribute_Dcl
Class_Member_Field
    ::= Class_Template_Member_Field
      | [Class_Member_Modifiers] Constructor_Def
      | Destructor_Def
Class_Extra_Field
    ::= [Access_Modifier] 'invariant' Expr
      | Objects_Definition
      | Objects_Specification
      | [Class_Member_Modifiers] Type_Definition
```

```
      | Alias_Def
      | Access_Modifier {nl Class_Field}+
      | Do_Binding
      | {Annotation}+ Class_Field
      | Nested_Annotation
      | 'local' Struct_Defs 'in' Class_Field {[nl] Class_Field} 'end'
      | Class_Conditional_Field(Class_Field)
Class_Interface_Fields
    ::= 'interface' Annot_Type {'and' Annot_Type}
        'with' {[nl] Trait_Concrete_Field}
        'end'
Class_Conditional_Field(Nested)
    ::= ('if' | 'unless') Boolean_Condition ('then' | semi) Nested
        [[semi] Else Nested] 'end'

Ival_Def
    ::= Ivar_Modifiers 'val' id [':' Type_Expr] '=' Expr
Ival_Spec
    ::= Ival_Modifiers 'val' id [':' Type_Expr] {',' id [':' Type_Expr]}
      | Ival_Modifiers 'val' '(' id {',' id} ')' [':' Type_Expr]
Ival_Modifiers
    ::= ['implicit'] [Access_Modifier] [Storage_Modifier]
Class_Member_Modifiers
    ::= ['implicit'] [Access_Modifier] ['final'] ['override' [Class_Qualifier]]
Alias_Def
    ::= 'alias' Alias_Id '=' Alias_Id
Alias_Id
    ::= symbol_literal
      | id
      | op_id
```

11.2.2 Class Specifications

Grammar:

```
Class_Type
    ::= Class_Function_Type
      | Class_Body_Type
Class_Function_Type
    ::= Function_Arg_Type '->' Class_Type
      | Function_Arg_Type '->' Signalling_Types '->' Class_Body_Type
Class_Body_Type
    ::= Parameterized_Type
      | 'object' ['(' Type_Expr ')'] {[nl] Class_Field_Spec} 'end'
```

```
Class_Field_Spec
    ::= Class_Structure_Field_Spec
      | Class_Member_Field_Spec
      | Class_Extra_Field_Spec
      | Class_Interface_Fields_Spec
Class_Structure_Field_Spec
    ::= 'inherit' Long_Id [Cond_Modifier]
      | 'inherit' 'val' 'of' Type_Expr
      | Class_Structure_Include
Class_Member_Field_Spec
    ::= ['abstract'] [Class_Member_Modifier] Ivar_Spec
      | ['abstract'] [Class_Member_Modifier] Property_Spec
      | ['abstract'] [Class_Member_Modifier] Message_Dcl
      | ['abstract'] [Class_Member_Modifier] Operator_Dcl
      | ['abstract'] [Class_Member_Modifier] Attribute_Dcl
      | [Class_Member_Modifiers] Constructor_Dcl
Class_Extra_Field_Spec
    ::= Objects_Specification
      | [Class_Member_Modifiers] Type_Declaration
      | Alias_Def
      | Access_Modifier {nl Class_Field_Spec}+
      | {Annotation}+ Class_Field_Spec
      | Nested_Annotation
      | Class_Conditional_Field(Class_Field_Spec)
Class_Interface_Fields_Spec
    ::= 'interface' Annot_Type {'and' Annot_Type} 'end'
```

11.3 Standalone Object Definitions & Specifications

11.3.1 Standalone Object Definitions

Grammar:

```
Standalone_Object_Expr
    ::= 'object' Object_Body 'end'
Object_Body
    ::= [Instance_Self_Type] {[nl] Object_Field}
Object_Field
    ::= Class_Structure_Field
      | Class_Template_Member_Field
      | Class_Extra_Field
```

```
      | Class_Interface_Fields
```

11.3.2 Standalone Object Specifications

Grammar:

11.4 Trait Definitions & Specifications

11.4.1 Trait Definitions

Grammar:

```
Trait_Expr
    ::= 'object' Trait_Body 'end'
Trait_Body
    ::= [Instance_Self_Type] {[nl] Trait_Field}

Trait_Field
    ::= Trait_Structure_Field
      | Trait_Concrete_Field
Trait_Concrete_Field
    ::= Trait_Member_Field
      | Trait_Extra_Field
      | Class_Structure_Include
      | Class_Interface_Fields
Trait_Structure_Field
    ::= 'inherit' Long_Id ['as' id] [Cond_Modifier]
Trait_Member_Field
      | [Class_Member_Modifiers] Property_Def
      | [Class_Member_Modifiers] Method_Def
      | [Class_Member_Modifiers] Operator_Def
      | [Class_Member_Modifiers] Attribute_Def
      | ['abstract'] [Class_Member_Modifiers] Ivar_Spec
      | ['abstract'] [Class_Member_Modifiers] Property_Spec
      | ['abstract'] [Class_Member_Modifiers] Message_Dcl
      | ['abstract'] [Class_Member_Modifiers] Operator_Dcl
      | ['abstract'] [Class_Member_Modifiers] Attribute_Dcl
Trait_Extra_Field
    ::= [Access_Modifier] 'invariant' Expr
      | Objects_Definition
      | Objects_Specification
```

```
      | [Class_Member_Modifiers] Type_Definition
      | Alias_Def
      | Access_Modifier {nl Trait_Field}+
      | Do_Binding
      | {Annotation}+ Trait_Field
      | Nested_Annotation
      | Class_Conditional_Field(Trait_Field)
```

11.4.2 Trait Specifications

Grammar:

```
Trait_Type
    ::= Parameterized_Type
      | 'object' ['(' Type_Expr ')'] {[nl] Trait_Field_Spec} 'end'

Trait_Field_Spec
    ::= Trait_Structure_Field_Spec
      | Trait_Concrete_Field_Spec
Trait_Structure_Field_Spec
    ::= 'inherit' Long_Id [Cond_Modifier]
Trait_Concrete_Field_Spec
    ::= Trait_Member_Field_Spec
      | Trait_Extra_Field_Spec
      | Class_Structure_Include
      | Class_Interface_Fields_Spec
Trait_Member_Field_Spec
      | ['abstract'] [Class_Member_Modifiers] Ivar_Spec
      | ['abstract'] [Class_Member_Modifiers] Property_Spec
      | ['abstract'] [Class_Member_Modifiers] Message_Dcl
      | ['abstract'] [Class_Member_Modifiers] Operator_Dcl
      | ['abstract'] [Class_Member_Modifiers] Attribute_Dcl
Trait_Extra_Field_Spec
    ::= [Access_Modifier] 'invariant' Expr
      | Objects_Definition
      | Objects_Specification
      | [Class_Member_Modifiers] Type_Definition
      | Alias_Def
      | Access_Modifier {nl Trait_Field_Spec}+
      | Do_Binding
      | {Annotation}+ Trait_Field_Spec
      | Nested_Annotation
      | Class_Conditional_Field(Trait_Field_Spec)
```

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 §11.4.4 & §11.6.2), 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 (§11.4.7).

Each trait reference mt_i that is not prepended must denote a trait (§11.7). 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 (§11.5).

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 (§??) 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 **use**, i.e. **use self as val** *x* **in** *t*, **use self as val** *x*: *T* **in** *t* or **use self as** *T* **in** *t*, where *x* is an alias for **self**, *T* is the prescribed type for *x* (or **self** if no *x* given), and *t* is the rest of the template in which the formal parameter definition appears. 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

[1] Including protocols, which are also traits.

[2] The classes are open in Aml, a single class may have its statements spread across multiple source files (§11.4.3).

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*.

Example 11.4.1 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
```

11.4.3 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, Aml 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 *main 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 the 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 a file name scheme `t.aux/a.aml`, where *t* is the name of the template and *a* is arbitrary auxiliary name. This way the auxiliary template is loaded along with the leading template, on-demand, without incrementing the template's version.
- Autoload (§??) of the auxiliary template.
- On-demand load of the auxiliary template via independent `VM.load 'path/to/file'`, after the leading template is defined.

11.4.4 Constructor Invocations

Grammar:

```
Base_Call
    ::= Annot_Type {'(' [Argument] ')'}
```

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 if a superclass has a designated parameterless constructor, 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.

A constructor invocation may choose any immediate superclass constructor, including designated and convenience constructors. In any case, such constructor must exist. The only exception to this rule is presented by the `Object` class, which does not have a superclass, and therefore no superclass constructor to invoke.

Primary constructor evaluation happens in the following order in respect to the constructor invocation:

1. Default values are evaluated.
2. Early definitions are evaluated. Up to this point, this is true for any constructor.
3. Instance variables defined by the primary constructor are evaluated.
4. Superclass constructor is invoked.
5. Explicit primary constructor body—if one exists—is evaluated.

11.4.5 Metaclasses & Eigenclasses

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

Everything is an object in Aml. Every object has a class that defines the structure (i.e. the instance variables) and behaviour 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 Aml, 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. Aml 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 11.1: 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 Aml, 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.

Grammar:

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

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

Example 11.4.2 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
      self = Metaclass[Metaclass[Metaclass[Object]]]
    end
  end
end
```

Example 11.4.3 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 11.4.4 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 11.4.5 Alternative direct access to the eigenclass of any object, here a class' eigenclass accessed via an attribute:

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

Example 11.4.6 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 11.4.7 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 11.4.8 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.

Note. If a metaclass of an object is edited, and such an object is returned from a function, then it might be a good idea to add the edits to its metaclass to the result type of such function, if appropriate, so that outside code may be able to non-dynamically use the edits.

11.4.6 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 11.4.9 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 11.4.10 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 (§8.3.4).

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

11.4.7 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 `Any` class is encountered, which has no superclass.

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

If a trait M is already (included in or prepended to)[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.

If a trait M is polymorphic, it is only already (included in or prepended to) any of the superclasses, if the same trait's type instantiation is the one in question, and the same applies if the trait's name is overloaded, with the addition that a monomorphic trait is only already (included in or prepended to) any of the superclasses, if the monomorphic trait is. Therefore, a single trait name may be included multiple times, but only different type instantiations, or the monomorphic one.

Example 11.4.11 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

[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.

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

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 *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 11.4.12 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].

[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.

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]

11.4.8 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. Aml 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, Aml uses the following definition of *matching* on members:

Definition 11.4.13 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' (§7.10.13).

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 11.4.14 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 (§11.4.9). 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.

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

11.4.9 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 (§6.7.2) 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 (§11.5) in C, then M should be labeled **`abstract override`**.
- If M and M' are both concrete value definitions, then one of the following must hold for them:
  - both M and M' are explicitly strict,
  - both M and M' are explicitly lazy,
  - M' is explicitly strict and M is not explicit in eagerness, meaning it inherits strictness from M',
  - M' is explicitly lazy and M is not explicit in eagerness, meaning it inherits laziness from M',
  - none of M and M' explicitly defines eagerness.

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.

11.4.10 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 (§??) are also in $\mathscr{S}$.

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

11.5 Modifiers

Grammar:

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

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

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

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

- The **`private`** access modifier can be used with any declaration or definition in a class. Such members can be accessed only from within the directly enclosing class, the class object (§11.10) 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* (§11.10).

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

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

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

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

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

  A **`protected`** access modifier can be qualified with an identifier *C* (e.g. **`protected`**`[`*C*`]`) that must denote a class or module enclosing the definition. Members labeled with such a modifier are *also*[13] accessible respectively from all code inside the module *C* or from all code inside the class *C* and its class object *C* (§11.10).

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

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

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

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

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

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

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

- The **`override`** modifier applies to class member definitions and declarations. It is never mandatory, unlike in Scala or C# (in further contrast with C#, every method in Aml is virtual, so Aml 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). If the optional class qualifier is present after the **`override`** modifier, then a supertype of the given simple name or identified by the stable id has to provide a member that this modifier is applied to, either abstract or concrete, and it is an error if it does not.

- 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 behaviour 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 11.5.1 The following code illustrates the use of qualified and unqualified private:

```
module Outer_Mod.Inner_Mod
  class Outer = object
    class Inner = object
      private[self] method e = ()
      private method f = ()
      private[Outer] method g = ()
```

```
      private[Inner_Mod] method h = ()
      private[Outer_Mod] method i = ()
    end
  end
end
```

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.

11.6 Class Definitions

A class definition defines the type signature, behaviour 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 behaviour defined in its metaclass (§11.4.5).

The most general form of a class definition is

class *c* [*tps*]
  as *m*(*ps*$_1$)...(*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 Aml, i.e. type arguments are preserved in runtime[14], generating a new concrete subtype of the original generic class. This ensures type safety in a dynamic environment of Aml in runtime.

as is a possibly empty sequence of annotations (§15). If any annotations are given at this point, they apply to the primary constructor of the class.

m is an access modifier (§11.5), 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*$_1$)...(*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

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

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 (§??) 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 `set_`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 (§7.10.5.2).

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 (§??) of the form

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

for ($n \geq 0$), which defines the base classes, behaviour 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 11.6.1 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 11.6.2 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
  }
}
```

11.6.1 Polymorphic & Monomorphic Class Overloading

As defined earlier, a class with a type parameter section [*tps*] is called *polymorphic*, a class without a type parameter section is called otherwise *monomorphic*. Furthermore, classes may be distinguished from each other by the count of their type parameters, in which case overloading resolution on their names apply – and classes that are monomorphic are "more specific" than classes that are polymorphic, in case the class name appears without a type application.

Note. If full type inference is desired for a polymorphic class *c*, then constructs such as the following can be used:

```
-- for polymorphic class with 1 type parameter
c[_]

-- for polymorphic class with 2 type parameters
c[_, _]

-- for polymorphic class with 3 type parameters
c[_, _, _]

-- for polymorphic class with 1 or more type parameters,
   when there is only one such polymorphic class
c[*_]
```

Note. Partial type inference is also possible:

```
-- for polymorphic class with 2 type parameters
c[_, String]
```

```
-- for polymorphic class with 3 type parameters
c[_, Number, _]

-- for polymorphic class with 2 or more type parameters,
   when there is only one such polymorphic class
c[String, *_]
```

11.6.2 Constructor & Destructor Definitions

Primary constructor. The primary constructor is a special function, more special in its syntax than any other function in Aml. 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 (§11.6). Technically, a primary constructor is happy being left with nothing but the parameter definitions, since these are always automatically mapped to instance values[15] and also implicitly adds accessor methods to the class with appropriate visibility, defined explicitly by accessor modifiers (§11.5) or implicitly as **`public`**.

Explicit primary constructor fields. These are expressions inside the class template (§??) that refer to the formal value parameters, and they become an implicit part of the primary constructor. The evaluation happens after early definitions are evaluated and before explicit primary constructor body is evaluated, but combinations of the two approaches should be avoided for the sake of readability – the only expressions that the formal value parameters should appear in are simple instance variable definitions.

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).

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 Aml have multiple entry points, but with great

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

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.

Grammar:

```
Constructor_Def
    ::= ['convenience'] 'constructor' [Param_Clauses] '=' [Function_Declarations] Expr
Destructor_Def
    ::= 'destructor' '=' [Function_Declarations] Expr
```

Implicit constructor. An implicit constructor is an automatically generated constructor. This is what Java and C# call "default constructor". An implicit constructor "does nothing" but invokes the super-constructor and initializes all members specific to the constructed object to their default values (if possible), so that an explicit empty constructor would not have to be specified. Implicit constructor can only be generated if all members have a default value and the superclass has a parameterless designated constructor.

Designated constructor. A designated constructor is such a constructor that initializes all members of the class. The primary constructor always has to be also a designated constructor. A designated constructor can only call a constructor of the immediate superclass[16]. A single class may have multiple designated constructors, but either way all instance members have to be initialized and at least one designated constructor has to exist. Note that new instance members may be added dynamically in runtime – but those are also initialized at the time they are added by the dynamic code. Also note that the constructor of the superclass may be dealing with an incomplete object[17], therefore a designated constructor must make sure that all instance members are initialized (either with their default values, or ones set up by the constructor), before it delegates up to the constructor of the superclass.

Convenience constructor. A convenience constructor is any other constructor than the designated constructors. It can only call a constructor of the same class (either convenience or designated). Convenience constructors are annotated with **`convenience`** keyword. Any convenience constructor must ultimately call a designated constructor of the same class.

[16]This is a relaxation from Swift by Apple, which only allows to call a designated constructor of the immediate superclass.

[17]As the constructors may further edit the new object later.

Inheritance of constructors. Constructors are not automatically inherited. Constructors whose signature matches signature of a superclass constructors have to be prefixed with the **`override`** modifier (§11.5). The following rules apply to inheritance of constructors:

Rule 1 The subclass must define default values for all new introduced instance variables for any constructor inheritance to take place.

Rule 2 If the subclass does not define any designated constructors (and does not define primary constructor parameters, including empty parentheses), it automatically inherits all of its superclass designated constructors.

Rule 3 If the subclass provides implementation of all of its superclass designated constructors—either by inheriting them as per rule 2, or by providing a custom implementation as a part of its own definition—then it automatically inherits all of the superclass convenience constructors.

Phases of object construction. The following lists present the order in which object construction occurs.

Phase 1

1. Somebody calls the **`new`** instance method of `Class` with arguments for the desired constructor.
2. A constructor is picked up with a 1:1 mapping to the arguments of **`new`**.
3. A memory is allocated for the new object. The memory is not yet initialized, and so is not the object.
4. Constructors are applied. Runtime keeps track of all declared instance variables and their defined-ness during object construction.[18]
5. As soon as every instance variable is properly defined, the **`self`** object becomes available for regular use.
6. After this point, the object is considered fully initialized, and the phase is complete. Instance values may no longer be modified after this point.

Phase 2

1. This phase begins as soon as all declared instance variables are defined.
2. As the control flows on, either continuing up the inheritance chain, or descending from a parent constructor, other constructors have now the ability to access the new object (as **`self`**). This includes both designated and convenience constructors, whichever gets control back earlier goes first.

[18]This is again relaxation from Swift by Apple. The object might be fully initialized earlier, or later.

3. At this phase, instance methods may be also called, because the object is already constructed. Also, modifications of immutable instance variables are still allowed – up to the point where the bottommost constructor finishes its evaluation.

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 11.6.3 An example of a designated constructor of class *C*.

```
class C
begin
  constructor (param) =
    val @resource = param in
    super

end class
```

Example 11.6.4 An example of a pair of constructors of class *C*.

```
class C
begin
  constructor (param) =
    val @resource = param in
    super

  convenience constructor = self (42)
end class
```

Constant instance variables. Any constructor may modify a constant instance variable during its evaluation, even multiple times. This behaviour passes on to any methods that are invoked during the object's construction, but should be used with caution.

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. To further emphasize that, the `this` and **`self`** keywords are disallowed in a destructor body, and invoking any method that does use the **`self`**-value is forbidden. The VM is required to raise an error in case **`self`** would be attempted to be retained again, to prevent zombie objects apocalypse.

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 11.6.5 An example of an explicit destructor of class *C*.

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

Process of object destruction.

1. An object becomes eligible for destruction (see below).
2. Strong retain count drops to zero.
3. All weak references within the object are removed. Includes weak references, unowned references and soft references. If any of the referenced objects then become eligible for destruction, their destruction happens in the same order as soon as this object's destruction finishes.
4. The object's destructor is applied. When that finishes, the destructor from its superclass is applied and so on.
5. After the root destructor is applied, strong reference count to the destructed object is checked. If it is non-zero, a runtime error is raised (`Object_Resurrection_Error`).
6. All strong references within the object are removed. If any of the referenced objects then become eligible for destruction, their destruction happens in the same order as soon as this object's destruction finishes.
7. Memory is reclaimed.

Objects eligible for destruction. An object *eligible for destruction* must meet the following conditions:

- Strong reference count has dropped to zero.
- There are no soft references to the object.
- If there are soft references to the object, the runtime is allowed to drop all of them in case of low memory. Definition of such condition is implementation-specific.
- The object is eligible for destruction regardless of any weak references remaining.

11.6.3 Case Classes

Grammar:

```
Class_Definition ::= ['monitored'] '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 (§11.6).

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 (§9.1.5.6), and are defined as follows:

```
object c
  method apply[tps](ps_1)...(ps_n): c[tps] :=
      c[Ts].new(xs_1)...(xs_n)
  method unapply[tps](x: c[tps]?): Option[e] :=
      unless x = nil
        Some(x.xs_11, ..., x.xs_1k)
      else
        None
      end
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`**.[19]

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 result type.
- For $n = 1$, `Option[`Tp_1`]` is the result 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 result 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 (§6.3.8). 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(xs_1)...(xs_n)
```

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 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 (§7.10.7), 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
```

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

```
  ...
  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. All instance variables are shallow-copied into the new cloned object.

11.7 Trait Definitions

Grammar:

```
Trait_Definition ::= 'trait' Trait_Def
Trait_Def        ::= id [Type_Param_Clause] [Given_Clauses]
                     Trait_Tmpl_Env
Trait_Tmpl_Env   ::= Tmpl_Env_With_Pars ['trait']
                   | Tmpl_Env_Brackets
                   | Tmpl_Env_No_Pars ['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 (§8.3.2). 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 11.7.1 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]] {
  operator < (a: 'T) (b: 'T): Boolean end
  operator <=(a: 'T) (b: 'T): Boolean := not (a < b) and not (a > b)
  operator > (a: 'T) (b: 'T): Boolean := b < a
  operator >=(a: 'T) (b: 'T): Boolean := b <= a
}
```

11.8 Refinement Definitions

There are two separate branches of refinements in Aml, both are quite similar, but used for different purposes. Refinements are kind of a trait (§11.7). 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 (§??).
>
> Second, refinements that are basically traits designed to be locally prepended to classes. The second branch is described here.

Grammar:

```
Refinement_Definition ::= 'refinement' Refinement_Def
Refinement_Def        ::= id 'refine' Type {'and' Type}
                          Refinement_Tmpl_Env
Refinement_Tmpl_Env   ::= Tmpl_Env_With_Pars ['refinement']
                        | Tmpl_Env_Brackets
```

Such a refinement does not declare any type parameters, since those are already declared on the types 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.

If multiple types are refined at once, then special care has to be taken when overriding existing members of the refined types – those members have to be all present in every refined type, as well as all members of the refined types that are accessed from within the refinement. Then, such a refinement is equivalent to a series of refinements, one per each refined type.

Multiple refinements bearing the same name may co-exist in any module, as long as each refinement per each refined type does not define multiple members of the same signature.

Refinements need to be “activated” in the scope for it to take effect (§8.3.3), similar to aspects.

If a refinement refines a parameterized type, then the refinement only activates for this parameterized type (§6.3.6).

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`**.

11.9 Protocol Definitions

Grammar:

```
Protocol_Definition ::= ['monitored'] 'protocol' Pro_Def
Pro_Def             ::= id [Type_Param_Clause] [Given_Clauses]
                        Pro_Tmpl_Env
Pro_Tmpl_Env        ::= Tmpl_Env_With_Pars ['protocol']
                      | Tmpl_Env_Brackets
                      | Tmpl_Env_No_Pars ['protocol']
```

Protocols express the contracts that other classes have to implement, and are added to classes with the keyword "`implements`" or as a part of the type's signature.

Protocols are basically traits (§11.7) 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 (§**??**). If this is needed, again, use a trait instead.
- Protocols can make use of the **`optional`** and `requires` keywords as modifiers in their template (§8.8), as any other traits, but their usage is preferred within protocols.

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

11.10 Object Definitions

Grammar:

```
Object_Definition ::= 'object' Object_Def
Object_Def        ::= id Object_Tmpl_Env
Object_Tmpl_Env   ::= Tmpl_Env_With_Pars ['object']
                    | Tmpl_Env_Brackets
                    | Tmpl_Env_No_Pars ['object']
Expr              ::= 'object' Object_Tmpl_Env
```

Object definitions define singleton instances. If no superclass is given, `Aml/Language.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 a class of the same name exists defined in source code after the object definition, then the object must explicitly extend `Aml/Language.Class[]`, without any type arguments[20] – it is sufficient

[20] This is to prevent the automatic extension of `Aml/Language.Object`.

to write down `Class`, if that name is imported to resolve to `Aml/Language.Class`, which it usually is. If the class definition is not connected to a class, then rules from compound types apply (§??), 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: Aml 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 (§??) of one of the following forms:

Singleton object form. This is a form of objects that are not class objects (not instances of `Metaclass[`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 `Metaclass[`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 (§11.7) that are to be included in a class object of class C: the self type would have to be `Class[`C`]`, not just C.

11.10.1 Case Objects

Grammar:

```
Object_Definition ::= ['monitored'] 'case' 'object' Object_Def
```

Case objects are pretty much similar to case classes (§11.6.3). Case objects can not be class objects at the same time. Their template (§??) is the singleton object form of an object definition (§11.10).

A case object definition implicitly adds some methods to the object instance, making it suitable as an extractor object (§9.1.5.6), and are defined as follows:

```
object c {
  def unapply(x: c.type?): Boolean := (x == self)
}
```

Chapter 12

Implicit Parameters, Views & Multiple Dispatch

Contents

12.1 The Implicit Modifier

Grammar:

```
Local_Modifier  ::= 'implicit'
Implicit_Params ::= 'implicit' ':' Atomic_Pattern
                  | '(' 'implicit' ':' (Atomic_Pattern | Params) ')'
```

Template members and parameters labelled with **`implicit`** modifier can be passed to implicit parameters (§12.2) and can be used as implicit conversions called views (§12.3).

If the member marked with **`implicit`** is a class *c*, it makes the constructors of the class available for view applications of expression *e* (§12.3) using the instance creation expression `(`**`new`** *c*`(`*e*`))` (§??). Such constructor must take exactly one non-implicit argument as its first parameter (constructors with no or more than one non-implicit parameters are not made available for view applications).[1]

Example 12.1.1 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] = object
  inherit Semi_Group ['a]
  message unit: 'a end
  message add (x: 'a, y: 'a): 'a end
end object

object Monoids
  implicit object String_Monoid
    inherit Monoid[String]
    method unit: String = ""
    method add (x: String, y: String): String = x + y
  end
  implicit object Int_Monoid
    inherit Monoid[Integer]
    method unit: Integer = 0
    method add (x: Integer, y: Integer): Integer = x + y
  end
end object
```

[1] The reason for this is simple: the Aml system can't make up the extra arguments and the constructor must be applied to the viewed expression.

12.2 Implicit Parameters

An implicit parameter (**implicit:** *p*) of a method marks the variables that bind arguments by *p* as implicit, both for the outside world, and for the method's scope. Any function (including constructors) can have at most one explicitly defined implicit parameter, and it must be the last parameter defined. Any function can have up to two implicit parameters, but the second one is managed by the Aml system. Such a second implicit parameter is called *shadow implicit parameter* and can't be written explicitly in user programs, but may be examined in runtime using reflection.

A method with implicit parameter can be applied to arguments just like normal method. In this case the **implicit** label has no effect. However, if such a method misses argument for its implicit parameter, such argument will be automatically provided, if possible. If a tuple extraction is used as the implicit parameter, then an argument will be automatically provided for each defined tuple element, but those extractions can only define positional non-optional extractions at the top level.

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 (§12.1) 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 (§7.13).
- 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. The same for intersection type.
- 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 (T_1 **or** ... **or** 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 (§8.9.3). If the parameter has a default argument and no implicit argument can be found, the default argument is used.

Example 12.2.1 Assuming the classes from Example 12.1.1, 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.

12.3 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[2], 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]Starting with implicit definitions in the scope.

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 there are several eligible views, the most specific one is chosen according to overloading resolution with an expected type inherited from the original selection. 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 there are several eligible views, the most specific one is chosen according to overloading resolution with an expected type inherited from the original selection. 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 (§8.9.3) is applied if there are several possible candidates.[3]

If there are multiple implicit scopes searched in order, we mean that the following implicit scope is searched if:

- The previous scope did not contain any eligible implicit value.
- The previous scope did contain multiple eligible implicit values, but overloading resolution could not select a unique most specific one.

12.4 View & Context Bounds

Grammar:

```
Type_Param ::= (id | '_') [Type_Param_Clause]
               ['>:' Type] ['<:' Type]
               {'<%' Type} {':' Type}
```

[3]The overloading resolution here is for the case of function applications, and the shape takes into account just one argument and corresponding parameter pair.

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 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:

```
let f [A <% T_1 ... <% T_m : U_1 : U_n] (ps): R = ...
```

Then the method definition above is equivalent to

```
let 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 12.4.1 The following example shows a method with a context-bound parameter using a partially applied type, and the resulting translation below it.

```
let f [T : T -> String] (t: T) = ...
let f [T] (t: T) (implicit: w_1: T -> String) = ...
```

12.5 Multi-Methods & Multiple Dispatch

Aml offers multiple ways to achieve multiple dispatch and the related multi-methods:

1. Application to values that are typed `Dynamic`.
2. Type classes.
3. Dynamic value dispatch (§12.5.3).

12.5.1 Application to Dynamic type

This is the easiest way to achieve multiple dispatch, where the overloaded function is selected based on not only the receiver, but also on any other argument. Those arguments that are typed with `Dynamic` (§6.6) affect the resolved overloaded alternative.[4]

Example 12.5.1 Multiple dispatch with `Dynamic` type, Asteroids colliding with Spaceships.

```
structure program =
struct
  abstract class Thing = object end
  class Asteroid = object inherit Thing end
  class Spaceship = object inherit Thing end

  method collide_with_impl (x: Asteroid, y: Asteroid): Unit
    print_line "Asteroid hits an Asteroid"
  end method

  method collide_with_impl (x: Asteroid, y: Spaceship): Unit
    print_line "Asteroid hits a Spaceship"
  end method

  method collide_with_impl (x: Spaceship, y: Asteroid): Unit
    print_line "Spaceship hits an Asteroid"
  end method

  method collide_with_impl (x: Spaceship, y: Spaceship): Unit
    print_line "Spaceship hits a Spaceship"
  end method

  method collide_with (x: Thing, y: Thing): Unit
    let a: Dynamic = x in
```

[4] This is similar to what C# since its version 4.0 does.

```
    let b: Dynamic = y in
    collide_with_impl (a, b)
  end method

  method run (*args: String): Unit
    let asteroid = Asteroid.new in
    let spaceship = Spaceship.new in
    collide_with (asteroid, spaceship)
    collide_with (spaceship, spaceship)
  end method
end struct
```

This code upon running `program.run` prints the following to the console:

```
Asteroid hits a Spaceship
Spaceship hits a Spaceship
```

12.5.2 Type Classes

Another approach to multiple dispatch involves implicit parameters and type classes. This is probably the most verbose approach.

Example 12.5.2 Multiple dispatch with type classes (via traits and implicit objects), matrices and vectors.

```
structure program =
struct
  class Matrix = object end
  class Vector = object end

  trait Mult_Dep['a, 'b, 'c] = object
    message apply (a: 'a, b: 'b): 'c end
  end

  structure Mult_Dep =
  struct
    implicit object mmm
      inherit Mult_Dep[Matrix, Matrix, Matrix]
      ...
    end
    implicit object mvv
      inherit Mult_Dep[Matrix, Vector, Vector]
      ...
    end
```

```
    implicit object mim
      inherit Mult_Dep[Matrix, Integer, Matrix]
      ...
    end
    implicit object imm
      inherit Mult_Dep[Integer, Matrix, Matrix]
      ...
    end
  end struct

  let multiply ['a, 'b, 'c]
      (a: 'a, b: 'b) (implicit: dep: Mult_Dep['a, 'b, 'c]): 'd =
    dep.apply (a, b)

  let run (*args: String): Unit = begin
    -- ok:
    let r1: Matrix = multiply(
        (Matrix with {}).new, (Matrix with {}).new) in
    let r2: Vector = multiply(
        (Matrix with {}).new, (Vector with {}).new) in
    let r3: Vector = multiply(
        (Matrix with {}).new, 2) in
    let r4: Matrix = multiply(
        2, (Matrix with {}).new) in

    {- error, no implicit value found for type
         Mult_Dep[Matrix, Vector, Matrix]:  -}
    let r5: Matrix = multiply(
        (Matrix with {}).new, (Vector with {}).new) in
    ()
  end
end struct
```

Example 12.5.3 Multiple dispatch with type classes (literal ones this time), matrices and vectors.

```
structure program =
struct
  class Matrix = object end
  class Vector = object end

  type class Mult_Dep['a, 'b, 'c]
  with message apply (a: 'a, b: 'b): 'c end
  end type
```

```
    type class instance Mult_Dep[Matrix, Matrix, Matrix]
    with method apply (a: Matrix, b: Matrix): Matrix = ...
    end type

    ...

    let multiply ['a, 'b, 'c]
        given Mult_Dep['a, 'b, 'c]
        (a: 'a, b: 'b): 'c =
      apply (a, b)

end struct
```

12.5.3 Dynamic Value Dispatch

The last approach taps into a whole new kind of methods.

It uses two-phase dispatch:

1. A dispatch function is defined with `def` **`dispatch`** *x*`...` (or any alternative syntax using the **`dispatch`** keyword). This function returns a value or a tuple of values, called the *dispatch value*, which determines the multi-method to use, but its result type is independent of this value – the type of the dispatch value is specified with an element of the `Mul_Val` syntactic category, or automatically inferred if not given.
2. A multi-method is installed with `def` **`multi`** *x*`...` (or any alternative syntax using the **`multi`** keyword).

Dynamic value dispatch functions and methods are different from regular functions and methods: their overloading depends not on the argument expressions only, but most importantly, on the dispatch value returned by the dispatch function. Even their internal representation in the VM is different. The dispatch function maps the argument expressions to a single value, or a tuple value, based on which a multi-method is selected, using regular overloading resolution, utilizing only the first parameter list.

The result type of a dispatch function (or method) is determined based on the dispatch value by the result type of the corresponding installed multi-method, therefore, it may trigger early argument expressions evaluation during overloading resolution (and indeed evaluation of the dispatch function itself),[5] pretty much like constrained types do. The dispatch function may constrain the types that multi-methods are allowed to result into by its own result type, thus the result types of the multi-methods must be covariant with the result type of the dispatch function.

[5]Yet the dispatch function may opt-in for lazy argument evaluation, but still, the argument expression may still be evaluated, and very likely it will be.

The dispatch function can itself be overloaded on its parameter lists and result value, by other regular functions and methods, or even other dispatch functions, in which case the multi-methods are shared for each dispatch function.

Multi-methods expect only values in their first parameter list, therefore, the syntax in them never generates parameter names, which is different from regular methods; therefore, the first parameter list contains only explicit parameters. The types of parameters[6] in the following parameter lists are inferred from the dispatch function, which passes the arguments to multi-methods as they are, and it is in fact an error if the multi-method specifies any other types for those parameters.

Example 12.5.4 Multiple dispatch with dynamic value dispatch, Asteroids colliding with Spaceships.

```
object program
  abstract class Thing = object end
  class Asteroid = object inherit Thing end
  class Spaceship = object inherit Thing end

  multi method collide_with (a: Thing, b: Thing) =
    method (dispatch Dynamic * Dynamic use type) = (a, b)
    and method (when :? Asteroid, :? Asteroid)
        = print_line "Asteroid hits an Asteroid"
    and method (when :? Asteroid, :? Spaceship)
        = print_line "Asteroid hits a Spaceship"
    and method (when :? Spaceship, :? Asteroid)
        = print_line "Spaceship hits an Asteroid"
    and method (when :? Spaceship, :? Spaceship)
        = print_line "Spaceship hits a Spaceship"
    and method (when _, _) = print_line "Found a UFO"

  method run (*args: String): Unit =
    let asteroid = Asteroid.new in
    let spaceship = Spaceship.new in
    begin
      collide_with asteroid, spaceship
      collide_with spaceship, spaceship
    end
end
```

Example 12.5.5 Multiple dispatch with dynamic value dispatch, area of shapes. This example shows a possible direction of extensions to the multi-methods, by providing more implementations for different symbols.

[6] Including extra properties, e.g. "by-name" status.

```
object program
  multi method area (obj: Map[Symbol, Any]): Real =
    method (dispatch Symbol use match) = obj :shape
    and method (for :Rectangle) = (obj :width) *. (obj :height)
    and method (for :Circle) = Math.π * (obj :radius)^2

  method run (*args: String): Unit =
    let rect = %{
        :shape => :Rectangle
        :width => 4 [<cm>]
        :height => 13 [<cm>]} in
    let circle = %{
        :shape => :Circle
        :radius => 12 [<cm>]} in
    begin
      print_line area rect
      print_line area circle
      print_line area %{}[Symbol, Any]  -- error, no such method
    end
end
```

Chapter 13

Units of Measure

Contents

13.1 Units of Measure

Syntax:

```
Type_Representation
    ::= ...
      | '=' 'abstract' 'measure'
      | '=' 'measure' ['(' Long_Id ')'] [semi] measure_def

measure_def
    ::= '_' -- anonymous measure, inferrable by compiler
      | measure_simple
measure_simple
    ::= measure_sequence
      | measure_simple '*' measure_simple  -- product, e.g. "'U * 'V"
      | measure_simple '/' measure_simple  -- quotient, e.g. "'U / 'V"
      | '/' measure_simple  -- reciprocal, e.g. "/'U"
      | '1'  -- dimensionless
measure_sequence
    ::= measure_power [measure_sequence]
measure_power
    ::= measure_atom
      | measure_atom '^' integer_literal  -- power of measure, e.g. "m^2"
measure_atom
    ::= Type_Var  -- variable measure, e.g. "'U"
      | Long_Id  -- named measure, e.g. "ft"
      | '(' measure_simple ')'  -- parenthesized measure, e.g. "(N m)"

measure_literal
    ::= '_'  -- anonymous measure, inferrable by compiler
      | measure_literal_simple
measure_literal_simple
    ::= measure_literal_sequence  -- implicit product, e.g. "m s^-3"
      | measure_literal_simple
        '*' measure_literal_simple  -- product, e.g. "m * s^4"
      | measure_literal_simple
        '/' measure_literal_simple  -- quotient, e.g. "m/s^3"
      | '/' measure_literal_simple  -- reciprocal, e.g. "/s"
      | '1'  -- dimensionless
measure_literal_sequence
    ::= measure_literal_power [measure_literal_sequence]
measure_literal_power
    ::= measure_literal_atom
      | measure_literal_atom '^' integer_literal  -- power of measure, e.g. "m^2"
measure_literal_atom
    ::= Long_Id  -- named measure, e.g. "ft"
```

```
        | '(' measure_literal_simple ')'  -- parenthesized measure, e.g. "(N m)"
```

Numbers in Aml 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.

Chapter 14

Top-Level Definitions

Contents

14.1 Compilation Units

14.1.1 Modules

Grammar:

```
Implementation_File
    ::= Implementation_Packaging
Signature_File
    ::= Signature_Packaging
Script_File
    ::= Implementation_File
Script_Fragments
    ::= Script_Fragment {';;' Script_Fragment}
Script_Fragment
    ::= Script_Stat
Implementation_Packaging
    ::= {Pragma} 'module' Struct_Path Impl_Packaging
      | Anonymous_Implementation_Packaging
Signature_Packaging
    ::= {Pragma} 'module' 'signature' Struct_Path Sig_Packaging
Impl_Packaging
    ::= semi [Impl_Pkg_Seq] ['end']
        -- 'end' is only optional if the module is alone in the file
Impl_Pkg_Seq
    ::= Impl_Pkg_Stat {semi Impl_Pkg_Stat}
Impl_Pkg_Stat
    ::= Use_Clause
      | Expr
      | Struct_Def
Sig_Packaging
    ::= semi [Sig_Pkg_Seq] 'end'
Sig_Pkg_Seq
    ::= Sig_Pkg_Stat {semi Sig_Pkg_Stat}
Sig_Pkg_Stat
    ::= Struct_Spec
      | Use_Clause
Anonymous_Implementation_Packaging
    ::= Impl_Pkg_Seq
        -- the name of the module is derived from the file's name
Script_Stat
    ::= Impl_Pkg_Stat
      | Sig_Pkg_Stat
```

Module definitions are objects that have one main purpose: to join related code and separate

it from the outside. Aml'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 another identifier.

An *implementation file* consists of a sequence of packagings, import clauses, and class and object definitions, which are preceded by a module clause. A *signature file* consists of a sequence of packagings, import clauses, and structure elements specifications, including interfaces and protocols.

Implicitly imported into every compilation unit are, in that order:

1. the module `Aml/Language`
2. the submodule `Aml/Language.Predefined`

Members of a later import in that order hide members of an earlier import.

The implicit import of these modules can be disabled in each scope by a pragma.

The implicitly added code looks like the following code listing, with all its implications:

```
use global.Aml/Language.{_}
use global.Aml/Language.Predefined.{_}
```

The vendor specified in module's `src/module.aml` is inherited by every source file in the same module and it's submodules (unless explicitly overwritten). Modules imported from within the module object defined in `src/module.aml` also make those modules available without fully qualifying their name in other source files of the same module.

14.1.2 Packagings

A module is a special object which defines a set of member classes, objects and another modules. Like open templates (§11.4.3), modules are introduced by multiple definitions across multiple source files.

A packaging **`module`** *p* { *stats* } or **`module`** *v*/*p* *stats* **`end`** injects all definitions in *stats* as members into the module whose qualified name is *p*. Members of a module are called *top-level* definitions. If a definition in *stats* is labelled **`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 (§7.13), or refer to it via its fully qualified name.

The special *vendor-less* **`global`** "module" (defined as a part of `Long_Id` in §6.2) can only be specified as the first element of each packaging name, and can be used to refer to the "global" scope of names, from within any nested scope.

Example 14.1.1 Given the packagings

```
module An_Org/A
  module B
    ...
  end
end
module An_Org/C end
module Another_Org/D end
```

all members of the module `An_Org/A.B` are visible under their simple names to the modules `An_Org/A.B` and `An_Org/A`, but not the others: module `An_Org/C` has the same vendor, but is located outside of the packaging of module `An_Org/A.B`, and module `Another_Org/D` is completely out of the packaging game.[1]

The fully qualified names of these modules are as follows:

- `An_Org/A`
- `An_Org/A.B`
- `An_Org/C`
- `Another_Org/D`

Selections $p.m$ from p as well as imports from p work as for objects. 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 **`global.`**`Aml/Language.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 **`global.`**`Aml/Language.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. This should however be only used in script files, not inside modules, where all code has to be packaged.

[1] The packaging game is too strong for the module `D`.

Chapter 15

Annotations, Pragmas & Macros

Contents

Grammar:

```
Expr
    ::= ...
      | Pragma_Expr
Class_Expr
    ::= ...
      | Class_Pragma_Expr
Annotation_Definition
    ::= 'annotation' Class_Def
Macro_Dcl
    ::= 'macro' Fun_Sig 'end'
Macro_Def
    ::= 'macro' Fun_Sig '=' [Fun_Dec] Expr
Annotation
    ::= '@[' Long_Id [Arguments] {semi Long_Id [Arguments]} ']'
Nested_Annotation
    ::= Inner_Annotation
      | Standalone_Annotation
Inner_Annotation
    ::= '[@' Long_Id [Arguments] {semi Long_Id [Arguments]} ']'
Standalone_Annotation
    ::= '[@@' Long_Id [Arguments] {semi Long_Id [Arguments]} ']'
Pragma_Expr
    ::= 'pragma' Long_Id [Arguments] (Do_Block | In_Sep Expr ['end'])
      | 'pragma' Long_Id '(' [Arguments] ')' (Do_Block | In_Sep Expr ['end'])
Class_Pragma_Expr
    ::= 'pragma' Long_Id [Arguments] In_Sep Class_Expr
      | 'pragma' Long_Id '(' [Arguments] ')' In_Sep Class_Expr
```

Annotations, pragmas & macros are a way to provide metadata to both the compiler and runtime of Aml, possibly affecting the resulting bytecode and abstract syntax trees.

15.1 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.

15.2 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.

15.3 Macros

Macros are a way to directly manipulate with abstract syntax trees. Unlike in languages such as C, macros in Aml are written using the same language. The only essential restriction here is that while compiling a Aml 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 (§8.7) 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
    - `Aml/Language.Reflection.Macros.Whitebox.Context` or
    - `Aml/Language.Reflection.Macros.Blackbox.Context`.
2. A parameter list with the parameters of the macro definitions, with the described parameter type translation applied.

Example 15.3.1 The following code is an example implementation of a simple **`assert`** macro.

```
-- import a blackbox context
use Aml/Language.Reflection.Macros.Blackbox.Context

-- define the macro
macro assert (condition: Boolean, message: String_Like): Unit =
    Asserts.assert_impl

-- implement the macro
object Asserts
  method assert_impl
```

```
      (c: Context)
      (cond: c.Expr[Boolean], msg: c.Expr[String_Like]): Unit =
    <@ unless ${cond}
         raise ${msg}
       else
         ()
       end @>
end
```

Macros can be applied as an intermediate step by IDEs, so that their resulting transformations could be viewed prior to proceeding with compilation.

15.3.1 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 behaviour. 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, functional dependency 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 functional dependency materialization. On the contrary, whitebox macros defer type inference of undetermined type parameters (`Undefined`) 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.

15.3.2 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 16

Automatic Inference

Chapter 17

Design Guidelines & Code Conventions

Contents

This chapter introduces the way programs *should* be written and laid out in Aml.

17.1 Introduction

17.1.1 Purpose of Having Code Conventions

They are important to programmers for many reasons:

- Maintenance of a software is a huge part of its lifetime costs.
- Software is usually not maintained for its whole lifetime by its original author. If you think that is not your case, imagine a future world, where an apocalypse happened. People that survived it might need to update your software for the new environment requirements. Or maybe in a different scenario, mankind reaches another terraformed planet and need to update your software for the new planet's requirements, probably different time and so on. Still not convinced?
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If your software is shipped including its source code, you might want to not feel embarrassed about it, and rather ship it clean.

17.2 Module Structure & File Names

This section lists commonly used module structure and file suffixes and names.

A module is a directory located in your system at a location unspecified by this specification. In the following sections, it will be referred to as *module root*.

Module directory name. A directory holding a module must have the same name as the module, contained in a directory of the vendor name. If it is standalone, so that no other modules of the same name share the same parent directory, the vendor part including the leading dot can be omitted.

Example 17.2.1 Such directories may look like:

```
/Aml/Language
/Example/Your_Module
    /source  -- or `src`
        /aml
            /Some_Submodule
            /macros
                /some_macro.aml
```

```
            /some_source_file.aml
            /module.aml
            /build.aml
            /module.dependencies.aml
            /module.dependencies.lock
        /cpp14  -- or whichever the Aml VM is implemented in
            /native_method.hpp
            /native_method.cpp
    /compiled  -- or `com`
        /aml
            /Some_Submodule
            /Example/Your_Module.amlb
            /Example/Your_Module/native/  -- optimized code
            /Example/Your_Module.amlpsi
        /cpp14
            /native_method.a
    /binaries (* or `bin` *)
        /an_executable.a
        /run_your_module  {- some executable file
                          using #! to run an Aml VM -}
    /resources  -- or `res`
        /logo.png
    /.gitignore
    /.git
```

Sources. A common name for this directory is `src`, or `source`. Inside of this directory, all Aml sources may be located, or alternatively, it can be forked into more directories, each for a specific language.

The directory name consists of a lowercase name of the source language, optionally followed by a platform specifier. A build tool used to create the software may then choose the sources appropriately.

Aml source files have the suffix `.aml`. Readme files are not source files, having their name starting with any-case `readme`, followed by e.g. `.md` for a Markdown-formatted readme.

Aml source directory (`/src` or `/src/aml`) might also contain a build file (or multiple build files, and even along with the language-specific directories in `/src` directly – some may be generated during compilation to provide skeletons for implementation of native methods and functions), which, for Aml, is named `build.aml`, and can make use of other files as needed.

Macros should be placed in a `/macros` directory inside of a directory with Aml sources, so e.g. `/src/macros` or `/src/aml/macros`. Those files may also use other files as needed. Macro definitions may appear outside of that directory too.

A module file may be placed in a Aml sources directory, and is named **`module.`**`aml`. This file is important in the fact that it defines the module and vendor name for the module, and the

vendor name for all nested submodules. Along with this file, a **`module.`**`dependencies.aml` and **`module.`**`dependencies.lock` files may be present, to define the module's dependencies on other modules, where the latter is a generated file, supposed to be checked into your VCS[1].

This directory may be omitted from a release version of your software (usually useful for proprietary software).

Submodules can be placed in a directory of the same name as the submodule's inside the directory where Aml sources are.

Compiled files. A common name for this directory is `com`, or `compiled`.

Aml compiled files (bytecode) have the suffix `.amlb` for bytecode and `.amlpsi` for extracted Program Structure Information. Readme files might be copied into it, if needed. Note that Aml VM will refuse to execute module's entry point if the module is not compiled including all native method implementations.

Aml native compiled files have suffix defined by the implementation. Could be `.dylib`, `.so`, `.dll`, or whichever the implementation uses.

Note that files compiled from other languages, if not bound to Aml environment via native method definitions, may be still used with `FFI` mechanisms.

Compiled files do not need to be checked into a VCS, unless they can't be generated from sources (usually applies to files compiled from external sources).

Binary files. A common name for this directory is `bin`, or very less commonly `binaries`. It is intended to contain any necessary binary files that are not to be compiled during build of the software, but rather just included in it as they are. May include executable binary files, or even shell scripts that start up a Aml VM with a specific entry point.

Resources. A common name for this directory is `res`, or `resources`, the latter may be preferred. Such directory is supposed to hold any files that the module needs, may it be images, sounds or whatever. Its organization is up to the module's maintainer.

17.3 File Organization

A file in Aml is basically a function, but it is a good idea to keep it organized in sections, if it represents a definition rather than a script. Each section should be separated by blank lines and preceded with a documentation comment if needed (usually when the documented section is a part of a public API).

[1] Version Control System.

Files longer than 1989 lines are cumbersome and should be avoided. Aml has a mechanism of open classes that allows programmers to split longer class definitions into multiple files, one defining the basic parts of the class and locations of its other sources and the others should define the implementation, logically separated.

17.3.1 Aml Source Files

Each Aml source file contains a definition of an implicit function, which is defined by the whole file. Such function may define types, classes, traits, and also do stuff.

A Aml file whose purpose is to define classes should contain at most one such top-level class (and may include inner classes as needed, indeed). When a private class or interface is associated with a public class, these can be put into the same file. The public class should be the first class defined in the file.

A typical Aml source file has the following ordering:

1. Module identification, e.g.:

   ```
   module Heroes/Cool_App
   ```

2. Imports, e.g.:

   ```
   use Them/Cool_Library.{Some_Class as A_Class, _}
   ```

3. Pragmas for the whole file, e.g.:

   ```
   pragma Profile :Prague
   ```

4. The class or type definition(s).

17.3.2 Class Definition Organization

The following list describes the parts of a class definition (or a trait), in the order that they should appear.

1. **Class documentation**
   A class documentation should introduce the class and its purpose to the documentation reader.

2. **Class signature statement**
   According to Aml's syntax, this defines the class' name, parents (including traits), type parameters, primary constructor's parameters and annotations for the class itself and its primary constructor. The primary constructor can also define some (or all) of the class' instance variables. The order of each element is defined by the syntax (§11.6).

3. **Class implementation comment**
   This comment should contain any class-wide information that is considered essential to understand the implementation and wasn't appropriate for the class documentation.

4. **Type declarations**
   Define these early, so that they can be used in the following code.

5. **Superclass instance method invocations**
   Use this section to call superclass instance methods. Imagine methods like `has_many` from RoR's `ActiveRecord`[2].

6. **Class instance methods**
   These methods are defined for the class object and may appear in a separate object definition. Their order should be in decreasing visibility (§11.5). Furthermore, factory methods should precede all other methods.

7. **Auxiliary constructors**
   Their order should be in decreasing visibility (§11.5).

8. **Methods and other members**
   These members should be grouped by functionality rather than by scope or accessibility. This section includes everything else, including method definitions, message declarations, inner classes.

17.4 Indentation

Tabs with two space displayed width are recommended to be used as the unit of indentation. The displayed width may be customized. If further indentation is required beyond indentation of the scope (e.g. to align type names in successive variable definitions), spaces has to be used for the extra indent (so-called "smart tabs"), but never for the base indentation.

17.4.1 Line Length

Avoid lines longer than 80 characters and try not to exceed 100 characters per line top. Use line breaks to achieve shorter lines.

[2]RoR stands for Ruby on Rails.

17.4.2 Line Wrapping

When an expression will not fit into a single line, break it according to the following general principles. Note though that Aml is also possibly used in a REPL environment, therefore it's syntax uses line breaks as significant whitespace in many places. This can however be prevented by means of preventing an expression end – usually by wrapping an expression that is supposed to be continued on the next line in parentheses.

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line. Spaces have to be used for this.
- If the previous rules lead to confusing code or to code that's squished up against the right margin, just indent two tabs instead.
- If the main expression on the line is an assignment or a return statement, don't forget to wrap the assigned expression in parentheses, if it's to be wrapped on the next line. Otherwise, a compile-time error would be raised, as the following operator applications or function applications could not be chained.
- Don't break before a closing parenthesis of a function application.

Example 17.4.1 Here are some examples of breaking function applications:

```
-- preferable
function_1 (long_expression_1, long_expression_2, long_expression_3,
            long_expression_4, long_expression_5)
val a = function_1 (long_expression_1,
                    function_2 (long_expression_2,
                                long_expression_3))
-- acceptable
function_1 (long_expression_1, long_expression_2,
    long_expression_3, long_expression_4,
    long_expression_5)
val a = function_1 (
    long_expression_1,
    function_2 (
        long_expression_2,
        long_expression_3))
```

Example 17.4.2 Following are examples of breaking an expression with operators. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
-- prefer
long_name_1 = long_name_2 * (long_name_3 + long_name_4 - long_name_5)
              + 4 * long_name_6
-- avoid
long_name_1 = long_name_2 * (long_name_3 + long_name_4
                             - long_name_5) + 4 * long_name_6
```

Example 17.4.3 Following are examples of indenting method definitions. The first is the conventional case, the second would shift the second and third line to the far right if it used conventional indentation, so instead it indents two tabs. Also, if a visibility modifier is causing the line to be too long, consider using it as a pragma before the method definition instead (but remember that definitions following it will have the same visibility).

```
-- conventional indentation
method some_method (an_arg: Integer, another_arg: Object,
                    yet_another: String,
                    and_still_another: Object): Unit = {
  ...
}

(* indent two levels to avoid very deep indents *)
private method self.very_long_method_name (an_arg: Integer,
    another_arg: Object, yet_another_arg: String,
    and_still_another: Object): Unit = {
  ...
}
```

Example 17.4.4 Line wrapping conditional statements should generally use the two tabs rule, since conventional one tab indentation makes seeing the body difficult. For example:

```
-- don't use this indentation
if ((condition_1 && condition_2)
  || (condition_3 && condition_4)
  || not (condition_5 && condition_6))
  do_something_about_it  -- this line is easy to miss
end

-- use this indentation instead (notice the extra spaces)
if (       (condition_1 && condition_2)
    ||     (condition_3 && condition_4)
```

```
    || not (condition_5 && condition_6))
  do_something_about_it
end

-- or use this
if (     (condition_1 && condition_2)
  ||     (condition_3 && condition_4)
  || not (condition_5 && condition_6))
then
  do_something_about_it (* this line is now harder to miss *)
end
```

Example 17.4.5 Aml does not have any ternary operators, but the following is an equivalent expression:

```
alpha = if a_long_boolean_expression
          beta
        else
          gamma
        end

-- or alternatively
alpha = if a_long_boolean_expression
  beta
else
  gamma
end
```

17.5 Comments

Aml programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are delimited by block `(* ... *)`. Documentation comments are delimited by `(*! ... *)` or alternatively `(** ... *)`[3]. Documentation comments can be extracted by Aml toolchains.

Implementation comments are meant for commenting-out code (to temporarily disable it) or for comments about the particular implementation. Documentation comments are meant to describe the specification of the code, independently on the actual implementation.

Comments should be used to provide additional information that is not readily available in the code itself, or can't be, e.g. information how the corresponding module is build.

[3] As a convenience for those used to this syntax from other languages. Aml toolchain prefers the first version.

Discussion of non-trivial or non-obvious design decisions is appropriate, but should not duplicate information that is already present (and clearly visible) in the code. Such redundant comments are likely to get out of date and deprecated.

Note. If too many comments are necessary to be included in the code, it might be a sign of a poor quality of the code – consider rewriting it to make it clearer and get rid of the extra comments.

Comments should not be enclosed in any decorative boxes made of asterisks or whatever else.

17.5.1 Placement

Preferable is to put declarations at the beginning of scopes (blocks). Variable declarations in the middle of code should be avoided. The only exception is in expressions like generators.

To optimize variable creation, variables do not have to be defined before first used – `Aml` provides laziness by default and explicit laziness.

17.5.2 Initialization

Try to initialize every variable as soon as possible.

17.6 Statements

17.6.1 Simple Statements

Each line should contain at most one statement. Avoid grouping statements on one line, unless there is an obvious reason to do so.

17.6.2 Compound Statements

Compounds statements are those enclosed in a block.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace or keyword should be at the end of the line that begins the compound statement. The closing brace or keyword should begin a line and be indented to the beginning of the compound statement, unless that indent would be too deep.

Part III

The Amlantis System Programming Language

`Aml/System`

Chapter 18

Introduction to The Amlantis System Language

Contents

18.1 About The Amlantis System Language

`Aml/System` is a language that targets a more low-level runtime: the System Runtime. It has syntax equivalent to that of `Aml`.

`Aml/System` is different from `Aml` in their libraries, along with minor language differences, which are described in this part.

The main purpose of `Aml/System` is to be a language in which a dynamic runtime and compiler for `Aml` can be implemented, which would make `Aml` almost self-hosted. This specification does not require implementation of `Aml/System` to be self-hosted in any way, and the same applies to `Aml`.

The System Runtime is nothing but a fancy name for whichever operating system is hosting the implementation of `Aml/System`. It can talk directly to the OS, it can indeed make use of `libc`, `libc++`, `libstdc++`, or any other library that would make the talking easier.

18.2 Stock Implementations

The "stock" implementation of `Aml/System` is a self-hosted one, using bootstrapping.

- `Stage 0 Aml/System` compiler is written in a different language[1], and its sole purpose is to be able to compile a subset of `Aml/System` features – a particular subset needed to compile a `stage 1` compiler.
- `Stage 1 Aml/System` compiler is written in a subset of the `Aml/System` language, and written so that it can compile every feature of `Aml/System` that `stage 2` compiler.
- `Stage 2 Aml/System` compiler is a self-hosted compiler, written in `Aml/System` without limitations (`stage 1` compiler must compile every feature that it uses), and can compile every `Aml/System` program, including its full library and indeed, itself. Which is what makes it self-hosted.
- `Stage 3 Aml/System` compiler is practically the same compiler as `stage 2`, and is just compiled by itself, no extra code is required for this compiler, it's just a name for the output of `stage 2` applied to its own source.

The specification of `Aml/System` only requires a `stage 3` -like compiler, not necessarily self-hosted. The existence of the previously mentioned stages is not mandatory.

The "stock" implementation of `Aml` and `Aml/SE` is an almost self-hosted one, using basically the same language (`Aml/System`), but targeting a different runtime, which it implements.[2]

[1] At the time of writing this, Rust 1.6 was chosen.

[2] Originally, C++14 and Rust were chosen as the most likely candidates, but both failed, thus leading to creation of `Aml/System`.

Chapter 19

Differences between The Amlantis System & Core Languages

Contents

19.1 Dynamic Features

`Aml/System` is a low-level runtime programming language, and as such, it relies heavily on the target platform, unlike `Aml`, which is intended to be platform-agnostic as much as reasonable.

Major platforms at the time of its creation, Mac OS X, Windows, GNU/Linux and BSD, do not support at low-enough-level some dynamic features of `Aml`. This is mostly due to the calling conventions of having classic call stack, not saguaro ones. Of course, if `Aml/System` is ever ported to a platform that does support these, the feature can be incorporated into it as well, provided that code can branch on its availability during compilation and runtime.

Another feature that `Aml/System` handles differently is the type system. While for `Aml`, no particular memory representation is required by this specification, `Aml/System` is different. To support unified type system, some values might need extra wrapping[1] in order to achieve the expected result, and for that reason, it is more reasonable to use more specific types in `Aml/System` than needed in `Aml`.

Another major difference is that the `Aml/System` runtime does not allow for the same level of dynamic name lookups as `Aml` has: all name references are resolved at compile time.

19.2 Classes & Objects Differences

While `Aml` has an elaborate class system with metaclasses, eigenclasses and so on, `Aml/System` only allows class instance members and class members, which are all defined at compile time and never change in runtime.

The same applies to other class-based concepts from `Aml`: once defined, a binding for a name in the same lexical context never changes.

This has some implications on the internal representation of objects and class objects in `Aml/System`:

- Layout of both objects and class objects is predictable, the compiler may (and definitely should) count with that.
- Instance variables will never be added at runtime, ever.
- Methods will also never change at runtime.

[1] Similar to, but not the same as Java's **`int`** vs `java.lang.Integer`.

19.3 Integration of The Amlantis System, Core & Other Languages

`Aml/System` can be used to implement native methods in `Aml`. In fact, the dynamic runtime of `Aml` can even use `Aml/System` while compiling optimized native versions of its otherwise interpreted methods.

While `Aml/System` may use a custom linkage on a given target platform, other languages that can be used to implement native methods in `Aml` have to use a C-style linkage for `ffi`. This includes (but is not limited to) languages like C (obviously), OCaml (**`external`**), C++ (**`extern "C"`**), D (**`extern (C)`**), Haskell (`foreign export ccall`) or Rust (**`extern "C"`**).

Part IV

The Amlantis SE Programming Language

`Aml/SE`

Chapter 20

Introduction to Amlantis SE

Contents

20.1 About The Amlantis SE Language

`Aml/SE` is a language that targets the same runtime as `Aml`. It has different syntax though, more close the Lisp language family. Technically, `Aml/SE` is an extension for the other `Aml` languages, as it serves primarily as a tool for data definitions, which happen to be also executable.

Like Lisps, `Aml/SE` uses *Symbolic Expressions*[1] for the data definitions, which is also similar to the serialization format of OCaml's `Sexp` extension. The syntax is based on Scheme, but uses elements of `Aml`'s syntax – including identifier syntax, which needs to be compatible with `Aml`.

[1] This is also where its name comes from, for now.

Part V

The Amlantis Standard Runtime Library

Part VI

The Amlantis Virtual Machine Runtime

Part VII

The Amlantis Tools

Part VIII

Extensions of Amlantis

Chapter 21

Perl Almost-Compatible Regular Expressions

Contents

Perl-Almost-Compatible Regular Expressions (PACRE) are a subsystem of Aml.

Right from the start, it is important to note that Aml is in no way Perl, and thus the regular expressions as integrated in it will *never* be 100% compatible to what regular expressions and regexes are in Perl (both versions 5 and 6).

Aml aims to implement the most important features from Perl 6's Regexes (`http://doc.perl6.org/language/regexes`), adding some features of its own and discarding or not being able to implement others. When possible, differences in both ways from Perl 6 will be noted in this document.

21.1 Syntax of Regular Expressions

21.1.1 Aml Literals for Regular Expressions

Syntax:

```
regexp_literal     ::= matching_regexp_p5
                     | matching_regexp_p6
                     | subst_regexp_p5
                     | subst_regexp_p6
matching_regexp_p5 ::= '%/' {int_string_element} '/' [regexp_flags]
subst_regexp_p5    ::= '%s/' {int_string_element} '/'
                       {int_string_element} '/' [regexp_flags]
matching_regexp_p6 ::= '%rx/' {int_string_element} '/'
                     | '%rx#' {int_string_element - int_expr1} '#'
                     | '%rx~' {int_string_element} '~'
                     | '%rx[' {int_string_element} ']'
                     | '%rx{' {int_string_element} '}'
                     | '%rx(' {int_string_element} ')'
subst_regexp       ::= '%srx/' {int_string_element} '/'
                       {int_string_element} '/'
                     | '%srx#' {int_string_element - int_expr1} '#'
                       {int_string_element - int_expr1} '#'
                     | '%srx~' {int_string_element} '~'
                       {int_string_element} '~'
regexp_flags       ::= letter_char {letter_char}
```

Part IX

Appendix

Chapter A

The Amlantis Language Syntax Summary

Chapter B

Changelog

Changes in Version 0.9

Dropping the by-future parameter evaluation strategy. Changing syntax of foreign types to use string literals for the type specification. Reworking “emptiness”: dropping the mess around **`nil`** and nullable types. Adding the `monitored` modifier for classes. Dropping the `Aml#` and `Aml#/System` languages. Extending capabilities of the use clauses. Refinements to identifiers syntax. Fixing grammar of aspects. Adding **`public use`**. Dropping the `*.amlm` source file extension. Renaming `Aml/S` to `Aml/SE`.

Changes in Version 0.8

Introducing the `Default[`*T*`]` and `Never_Nil` traits. Describing consistency relation for the gradual typing system. Refining typed expressions to make use of `Type_Convertible[`*S*`]`. Describing contracts. Dropping function preference declarations. Introducing type classes and type families. Adding usage of `monitored` to classes and objects. Removing definition of right-binding operators. Updating reference to Unicode from 7.0 to 8.0.

Changes in Version 0.7

Changing placeholder syntax for existential types from underscore to a question mark. Adding new keywords: `for-all` for universally quantified types, **`family`** and **`instance`** for type classes and type families, **`given`** for type constraints. Removing variadic type parameters. Removing in-place values of enumerations (variants are for that already). Changing syntax of by-name and by-future parameters (from `=>` and `=>>` to `->` and `->>`

respectively). Extending grammar rules of function and class-&-company definitions to include the new keywords, for use with type class constrained types and functional dependencies. Adding ~> as a symbolic keyword (used in functional dependencies). Adding more type constraint types: dependencies and equations. Removing nullable operators. Adding percentage literals.

Changes in Version 0.6

Adding grammar for recursive structures, updating grammar of applicative functors that take more than one structure parameter, updating grammar of generative functors.

Changes in Version 0.5

Changed definition of setter functions, from suffix "`_=`" to prefix "`set_`".

Changes in Version 0.4

The language and its system got renamed from Gear to Amlantis.

Changes in Version 0.3

Major change in concept of function applications, influenced by `OCaml` & `F#`.

Changes in Version 0.2

The language becomes more and more mature.

Version 0.1

This was the initial version of the contemporary Amlantis.