GFR-0

The Gear Language Specification

Version 0.1

Markéta Lisová

June 3, 2015

Contents

Preface

Gear is a Ruby-like programming language which enhances advanced object-oriented programming with elements of functional programming. Everything is an object, in this sense it's a pure object-oriented language. Object blueprints are described by classes. Classes can be composed in multiple ways – classic inheritance and/or trait composition, along with prototype-oriented inheritance.

Gear is also a functional language in the sense that every function is also an object, and generally, everything is a value. Therefore, function definitions can be nested and higher-order functions are supported out-of-the-box. Gear also has support for pattern matching, which can emulate the algebraic types used in other functional languages.

Gear has been developed since 2012 in a home environment out of pure enthusiasm for programming and out of a desire for a truly versatile language. This document is a work in progress and will stay that way forever. It acts as a reference for the language definition and some core library classes.

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

The vast majority of Gear's syntax is inspired by *Ruby*. Gear uses keyword program parentheses in Ruby fashion. There is **`class ... end`**, **`def ... end`**, **`do ... end`**, **`loop ... end`**. Ruby itself is inspired by other languages, so this relation is transitive and Gear is inspired by those languages as well (for example, Ada).

Gear is inspired by *Ada* in the way that user identifiers are formatted: `Some_Constant_Name` and — unlike in Ada, but quite similar to it — `some_method_name`. Also, some control structures are inspired by Ada, such as loops, named loops, return expressions and record types. Pretty much like in Ada, Gear's control structures can be usually ended the same way: **`class ... end class`**, etc.

Scala influenced the type system in Gear. Syntax for existential types comes almost directly from it. Also, along with other C-like languages, Scala influenced Gear to allow to choose between Ada-style program parentheses (e.g. **`begin ... end`**) and C-style parentheses (i.e. `{ ... }`) in many places throughout the syntax. Moreover, the structure of this mere specification is inspired by Scala's specification.

From *F#*, Gear borrows some functional syntax (like function composition) and F# also inspired the feature of Units of Measure (§3.5.6 & §12).

Clojure inspired Gear in the way functions can get their names. Gear realizes that turning function names into sentences does not always work, so it is possible to use dashes, plus signs and slashes inside of function names. Therefore, `call/cc` is a legit function identifier. Indeed, binary operators are required to be properly surrounded by whitespace or other non-identifier characters.

Status of This GFR

This GFR is active and mandatory for every proper Gear implementation, without exceptions. The status of this GFR is not likely to change.

Chapter 1

Introduction

Chapter 2

Program Structure

The Gear Programming Language System consists of a couple of command line tools:

- `gear`, runs Gear interactively in a read-eval-print loop.
- `gearc`, compiles Gear modules or scripts.
- `gearopt`, compiles native optimized Gear modules or scripts.
- `gearrun`, runs Gear modules or scripts.
- `geardoc`, generates documentation for Gear modules.
- `geardbg`, debugger for Gear modules or scripts.
- `gearp`, inspects compiled Gear modules or scripts.
- `gearf`, Gear module bundler and builder.
- `gearprof`, Gear module or script profiler.

The inputs to the Gear tools consist of:

- Source code files.
  - Files with extension `.gear`, implementation files.
  - Files with extension `.gearm`, implementation files with permanent macro definitions.
  - Files with extension `.gearx`, script files.
  - Files with name `module.gear`, module definition files.
- Compiled files.

  - Files with extension `.gearb`, bytecode function files.
  - Files with extension `.gearpsi`, bytecode PSI files.

- Semi-source files.
  - Files with extension `.gearp`, protocol files.
  - Files with name `module.dependencies.gear`, module dependencies.
  - Files with name `module.dependencies.lock`, locked module dependencies.

- Script fragments, used in interactive environments.
- Compilation and runtime parameters passed to the command line tools.
- Pragma and annotation applications (§14.2) within source files and their compiled forms.

The `Gear/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 (§18). 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.
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[1] 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[2], but whether these functions

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

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

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.

Chapter 3

Lexical Syntax & Analysis

Contents

Gear programs are written using the Unicode character set; Unicode supplementary characters are supported as well. Gear 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)
                  | ? Dollar sign ? | '§' | '_'

var_id   ::= lower id_rest
plain_id ::= upper id_rest
           | var_id
id       ::= ['@' | '@@'] (plain_id
             | '`' plain_id '`'
             | '``' {any_id}+ '``')
id_char  ::= letter_char
           | digit_char
           | connecting_char
           | combining_char
           | formatting_char
           | '+' | '-' | '/' | '_'
id_rest  ::= {id_char} [('?' | '!')]
any_id   ::= ? any Unicode char except for backquote: '`' ?
```

There are more kinds of identifiers. An identifier can start with a letter, which can be followed by an arbitrary sequence of letters, digits, underscores and operator characters. The identifier may be prefixed with one or two *at* "`@`" signs, creating an instance variable or a class object variable identifier, respectively. These forms are called *plain identifiers*. An identifier may also start with an operator character, followed by arbitrary sequence of operator characters, forming operator identifiers, which can only be used in expressions that directly involve operators (§9.3.10).

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

Gear 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.

[1]English, not mistakes.

The "`$`" character is reserved.

A set of identifiers is reserved for language features instead of for user identifiers. However, unlike in some other languages, keywords are not being recognized inside of paths, except for a few specific cases.

The following names are the reserved words. Identifiers that match any of the specified reserved word carry the extra property of being a possible keyword, which is used in syntactic grammar.

Syntax:

```
Identifier_Keyword ::= ? id, one of: ?
```

`abstract`	`advice`	`advice-execution`		`after`
`alias`	`and`	`annotation`	`arguments`	`around`
`as`	`aspect`	`assert`	`atomic`	`attribute`
`begin`	`before`	`bitfield`	`break`	`broken`
`case`	`catch`	`class`	`clone`	`cloned`
`constant`	`constraint`	`constructor`	`declare`	`def`
`delta`	`destructor`	`digits`	`dispatch`	`do`
`done`	`eager`	`else`	`elsif`	`end`
`ensure`	`enum`	`execution`	`exhausted`	`extends`
`final`	`for`	`for-some`	`foreign`	`function`
`functor`	`get`	`goto`	`handler`	`if`
`immutable`	`implements`	`implicit`	`in`	`include`
`interface`	`intersection`	`invariant`	`invoke`	`is`
`joinpoint`	`label`	`lazy`	`let`	`loop`
`macro`	`match`	`measure`	`member`	`memoize`
`message`	`method`	`module`	`monitored`	`multi`
`mutable`	`native`	`next`	`nil`	`no`
`not`	`object`	`of`	`opaque`	`operator`
`optional`	`or`	`origin`	`otherwise`	`outer`
`override`	`parallel`	`per-self`	`per-target`	`pointcut`
`pragma`	`prefer`	`prepend`	`private`	`property`
`protected`	`protocol`	`public`	`pure`	`raise`
`raiseable`	`raising`	`range`	`rec`	`recursive`
`record`	`redo`	`ref`	`refine`	`refinement`
`release`	`reraise`	`rescue`	`retain`	`rethrow`
`retry`	`return`	`returning`	`requires`	`reverse`
`sealed`	`self`	`set`	`seq`	`sig`
`skip`	`soft`	`step`	`struct`	`super`
`switch`	`synchronized`	`tailcall`	`target`	`then`
`this`	`throw`	`throwable`	`throwing`	`to`
`trait`	`transparent`	`type`	`undefined`	`unless`
`unowned`	`until`	`union`	`upgrade`	`use`
`val`	`var`	`yes`	`weak`	`when`
`while`	`with`	`xor`	`yield`	`yielding`

Not every reserved word is a keyword in every context, this behavior will be further explained by

syntax definitions. For example, the bitfield reserved word is only recognized as a keyword inside an enumeration definition context, in a specific place. In case of ambiguous syntax views of a sequence of source tokens, the ones that include keywords are preferred.[2] In any case, reserved words enclosed in backticks (“`”) are not treated as keywords.

3.2 Symbolic Keywords

Syntax:

```
Symbolic_Keyword ::= ? one of: ?
let!  yield!  return!  as?  as!  is!  not!
|  .  .?  .!  :  (  )  [  ]  <  >
%[|  |]  %[  %(  {  %{  }  %/  /
’  #  ..  ..<  ...  :=  ;  ;;  ->
=>  =>>  *  **  &
_  <@  @>  <@@  @@>
```

Symbolic or partially symbolic character sequences can be treated as keywords, if a lexical or syntactic grammars specifies that. Some lexical grammars extend some of these sequences with “flags”.

3.3 Symbolic Operators

Syntax:

```
op_id       ::= Symbolic_Op
Symbolic_Op ::= {op_char}+
              | ’`’ {op_char}+ ’`’
op_char     ::= ? Unicode char of classes Sm and So ?
```

Symbolic operators are sequences of characters as shown above, except where the sequence of characters is a symbolic keyword (§3.2) and is used by the corresponding grammar as such.[3]

3.4 Newline Characters

Syntax:

[2]This may happen with selection sequences that would include e.g. `prefix.type`. To treat `type` as another selection, enclose it in backticks: `prefix.\`type\``.

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

```
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 ::= ';;'
```

Gear 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 Gear 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, Gear 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.

3.5 Literals

There are literals for numbers (including integer, floating point and complex), characters, booleans, strings, symbols, regular expressions and collections (including tuples, lists, dictionaries and bags).

Syntax:

```
Literal ::= integer_literal
          | floating_point_literal
          | complex_literal
          | rational_literal
          | character_literal
          | boolean_literal
          | unit_literal
          | string_literal
          | symbol_literal
          | regular_expression_literal
          | Collection_Literal
          | 'nil'
          | 'undefined'
```

3.5.1 Integer Literals

Syntax:

```
integer_literal ::= sign (
                      decimal_numeral
                    | hex_numeral
                    | octal_numeral
                    | binary_numeral)
decimal_numeral ::= digit {['_'] digit}
hex_numeral     ::= hex_prefix | hex_digit {['_'] hex_digit}
digit           ::= '0' | ··· | '9'
hex_digit       ::= '0' | ··· | '9' | 'a' | ··· | 'f'
octal_numeral   ::= oct_prefix oct_digit {'_' oct_digit}
oct_digit       ::= '0' | ··· | '7'
binary_numeral  ::= bin_prefix bin_digit {['_'] bin_digit}
bin_digit       ::= '0' | '1'
hex_prefix      ::= '0x'
bin_prefix      ::= '0b'
oct_prefix      ::= '0o'
sign            ::= ['+' | '-'] | ()
```

Integers are usually of type `Number`, which is a class cluster of all types that can represent numbers. Unlike Java, Gear 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).

3.5.2 Floating & Fixed Point Literals

Syntax:

```
float_literal   ::= sign decimal_numeral '.' decimal_numeral
                    [exponent_part_e] [float_type]
                  | sign decimal_numeral exponent_part_e [float_type]
                  | sign decimal_numeral float_type
                  | sign hex_prefix hex_numeral '.' hex_numeral
                    [exponent_part_p [float_type]
                     | hex_exp float_type]
                  | sign hex_prefix hex_numeral exponent_part_p
                    [float_type]
                  | sign hex_prefix hex_numeral hex_exp float_type
                  | sign oct_prefix octal_numeral '.' octal_numeral
                    [exponent_part_e] [float_type]
                  | sign oct_prefix octal_numeral exponent_part_e
                    [float_type]
                  | sign oct_prefix octal_numeral float_type
                  | sign bin_prefix binary_numeral '.' binary_numeral
                    [exponent_part_e] [float_type]
                  | sign bin_prefix binary_numeral exponent_part_e
                    [float_type]
                  | sign bin_prefix binary_numeral float_type
exponent_part_e ::= int_exp sign decimal_numeral
exponent_part_p ::= hex_exp sign decimal_numeral
int_exp         ::= 'e'
hex_exp         ::= 'p'
float_type      ::= 'h' | 'f' | 'd' | 'q' | 'df'
```

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. Literals that have `float_type` of "`df`" are (*d*ecimal) *f*ixed 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 have to be lower-case in Gear sources, but functions that parse floating point numbers do support them being upper-case for compatibility.

3.5.3 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 Gear.

3.5.4 Rational Number Literals

Syntax:

```
real_number_literal ::= integer_literal | float_literal
rat_suffix_literal  ::= real_number_literal 'r'
rational_literal    ::= real_number_literal
                        ['/' rat_suffix_literal]
```

Rational number literals are of type `Number`, and have a container type `Number.Rational`, which has further methods of creating instances of itself. There are also methods in `Number`, such as `//`, which results in a rational number. Rational numbers have sort of increased accuracy of operations, especially when component types are integral types.

Example 3.5.1 Some literal notations and methods that result in rational numbers.

```
val a := 1 / 3r
val b := 1 // 3
val c := Rational(1, 3)
```

3.5.5 Imaginary Number Literals

Syntax:

```
imaginary_literal   ::= rational_literal 'i'
complex_literal     ::= [rational_literal ('+' | '-') ]
                        imaginary_literal
                    | imaginary_literal ('+' | '-')
                      rational_literal
number_literal      ::= imaginary_literal
```

```
                    | complex_literal
                    | rational_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 (§3.4).

3.5.6 Units of Measure

Gear has an addition to number handling, called *units of measure* (§12). Number instances can be annotated with a unit of measure to ensure correctness of arithmetic operations.

Syntax:

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

Example 3.5.2 Some number literals annotated with units of measure:

```
var min_motorway_speed := 90 [<km/hour>]
(* in the end, same as: *)
var distance := 90 [<km>]
var time := 1 [<hour>]
min_motorway_speed := distance // time
```

3.5.7 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 (§3.5.10) 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 left curly brace: { *)
    | '\}' (* Literal right curly brace: } *)
    | '\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 *)
    | '\n' (* Byte value in octal *)
    | '\nn' (* Byte value in octal *)
    | '\nnn' (* Byte value in octal *)
    | '\unnnn' (* Unicode character U+nnnn, hex digits *)
    | '\Unnnnnnnn' (* Unicode character U+nnnnnnnn, hex digits *)
    | '\' named_char ';' (* Named character entity *)
```

The named character reference (named_char) is defined by HTML5 (http://www.w3.org/TR/html5/syntax.html#named-character-references);

The octal byte values are consumed with the longest match.

3.5.8 Boolean Literals

Syntax:

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

Both literals are members of type Boolean. The **no** literal has also a special behavior when being compared to **nil**: **no** equals to **nil**, while not actually being **nil**. Identity equality is indeed different, and **no** does not match in pattern matching (§11) as **nil** and vice versa. The implication is that both **nil** and **no** are false conditions in **if**-expressions.

3.5.9 Unit Literal

Syntax:

```
unit_literal ::= '()'
```

Unit literal is of the type Unit (also known as Gear/Language.Unit, and has alias of Gear/Language.Tuple_0).

3.5.10 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     ::= '"""' {printable_char} '"""'
string_element      ::= printable_char | char_escape_seq
int_string_element  ::= string_element | interpolated_expr
interpolated_expr   ::= '#{' Expr '}'
str_flags           ::= 'm' | 'i'
```

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 string 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 suspicious.

3.5.11 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. Gear does not require their ids to be constant in time.

3.5.12 Regular Expression Literals

Syntax:

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

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

3.5.13 Collection Literals

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

Syntax:

```
Collection_Literal ::= Tuple_Literal
                     | List_literal
                     | Dictionary_Literal
                     | Bag_Literal
Collection_Flags   ::= printable_char {printable_char}
CF                 ::= Collection_Flags
Tuple_Literal      ::= '(' [Exprs] ')'
Collection_Exprs   ::= Expr {semi Expr}
List_Literal       ::= '%' [CF] '['  [Collection_Exprs]  ']'
Array_Literal      ::= '%' [CF] '[|' [Collection_Exprs] '|]'
```

```
Dictionary_Literal ::= '%' [CF] '{'  [Collection_Exprs]  '}'
Bag_Literal        ::= '%' [CF] '('  [Collection_Exprs]  ')'
Dict_Exprs         ::= Dict_Expr {semi Dict_Expr}
Dict_Expr          ::= Simple_Expr1 '=>' Expr
                   | id ':' Expr
```

Tuple literals are members of the `Tuple` type family. List literals are members of the `List_Like` type, usually `Linked_List` with alias of `List`. Array literals are members of the `Array_Like` type, usually `Array`. Dictionary literals are members of the `Dictionary` type with alias of `Map`, usually `Hash_Dictionary` with alias of `Hash_Map`. Bag literals are members of the `Bag` type, usually `Hash_Bag` or `Hash_Set`. Collection flags may change the actual class of the literal, along with some other properties, described in the following text.

List literal collection flags:

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

Array literal collection flags:

1. Flag `i` = **i**mmutable, makes the array frozen.
2. Flag `w` = **w**ords, the following expressions are treated as words, converted to strings for each word separated by whitespace.

Dictionary literals collection flags:

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

Bag literal collection flags:

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

2. Flag `s` = **s**et, the collection is a set instead of a bag (a specific bag, such that for each item, its tally is always 0 or 1, thus each item is in the collection up to once).

3. Flag `l` = **l**inked, makes the collection linked, so either a member of `Linked_Hash_Bag` in case of a regular bag, or `Linked_Hash_Set` in case of a set.

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

The type of elements of a collection is inferred from combination of the expected type of the whole literal, expected type of each element and type argument of the collection literal.

- If an expected type is given, its type arguments take precedence over types of elements. Elements may be subject to implicit conversions (§9.9). This inference works with an implicit conversion from a builder type to the target collection type – there must exists such a builder type that has a build method returning the type that is the expected type or a type that is convertible to the expected type.[4]

- If a type argument is present, it takes precedence over both types of elements and the expected type of the literal, which is then used to implicitly convert the whole parameterized collection type.

- If the expected type is undefined and no type argument is given, then the elements type is inferred as the least upper bound of all expected types of each element. If no element is given, then `Any` is inferred.[5]

- If the collection literal is a dictionary, then the least upper bound is computed for key and value types independently.

Example 3.5.3 The following lines show how type inference works with collection literals.

```
/* Linked_List[String] */
val a: List[String] := %[]
val b := %[][String]
val c := %["hello"; "world"]

/* Array[String] */
val a: Array[String] := %[||]
val b := %[||][String]
val c := %[|"hello"; "world"|]

/* Hash_Dictionary[String, String] */
```

[4] If a custom type is used, it has to statically conform to one of the protocols that the particular collection requires, so that the matching type argument can be retrieved and used for the collection builder.

[5] `Any` is inferred so that the shortest type argument-less collection literal would be more useful than just allowing elements of type `Nothing`, for which just one instance exists.

```
val d: Hash_Dictionary[String, String] := %{}
val e := %{}[String, String]
val f := %{"hello" => "world"}

/* Hash_Dictionary[Symbol, String] */
val d: Hash_Dictionary[Symbol, String] := %{}
val e := %{}[Symbol, String]
val f := %{hello: "world"}

/* Hash_Bag[String] */
val g: Bag[String] := %()
val h := %()[String]
val i := %("hello"; "world")
```

3.6 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. There is a whitespace (including newlines) required after `(*` and before `*)` in the outermost multi-line comment.

A single-line comment in Gear is a multi-line comment, written all on one line. That's all. No special syntax for that.

Documentation comments are multi-line comments that start with `(*!`. A whitespace is required after the opening token.

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 tokens and comments are not discarded during lexical analysis, but instead after lexical filtering, by rules described in (§18).

3.7 Conditional Compilation

Syntax:

```
Pragma           ::= If_Directive
                   | Else_Directive
                   | End_If_Directive
If_Directive     ::= 'pragma' 'if' id {'.' id}
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.

3.8 Hidden Tokens

Some hidden tokens are inserted by lexical filtering (§18), or are used to replace existing tokens.

Chapter 4

Identifiers, Names & Scopes

Names in Gear 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 `Object`.

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

---

[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.

Every binding has a *scope*[3] in which the bound entity can be referenced using a simple name (unqualified). Scopes are nested, inner scopes inherit the same bindings, unless shadowed. A binding in an inner scope *shadows* bindings of lower precedence in the same scope (and nested scopes) as well as bindings of the same or lower precedence in outer scopes. Shadowing is a partial order, and bindings can become ambiguous – fully qualified names can be used to resolve binding conflicts. This restriction is checked in limited scope during compilation[4] and fully in runtime.

If at any point of the program execution a binding would change (e.g., by introducing a new type in a superclass that is closer in the inheritance tree to the actual class than the previous binding), and such a change would be incompatible with the previous binding, it is an error. Also, if a new binding would be ambiguous[5], then it is an error.

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

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

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

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

1. defines an entity with name x in the same scope as the identifier x, and
2. shadows all other bindings that define entities with name x in that name scope.

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

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

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

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

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

Chapter 5

Types

Contents

Syntax:

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

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

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

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

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

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

5.1 About Gear's Type System

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

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

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

And now, move on to Gear. Gear uses hybrid typing. In its core, it uses static typing. But, it allows to opt-in for dynamic typing using a special type `Dynamic`. Unlike in Ruby, you can overload methods (not just override!). You can constrain variables, constants, properties, arguments and result types to particular types. But you don't have to. Types in Gear were heavily inspired by Scala's and Ruby's type systems, but modified for this hybrid environment that Gear provides. Unlike in Ruby, you can have pure interfaces (called protocols[1]), or interfaces with default method implementations (similar to Java 8). Unlike in Java, you can have traits, union types, record types and much more. Unlike in Java, you may easily modify classes, even from other modules (*pimp my library!* and *open-class principle*). You may even easily add more classes if needed, and possibly shadow existing ones. Implicit conversions are also made available dynamically at runtime, not just during compilation.

While Gear is so dynamic, it also needs to maintain stability and performance. Therefore, it "caches" its bindings and tracks versions of each type[2]. If a *cached binding* would change, it is ok – as long as the new binding would be compatible with the old one. Practically, the code that executes first initiates the binding – first to come, first to bind. Bindings are also cached, so that the Gear interpreter does not need to traverse types all the time – it only does so if the needed binding does not exist (initial state with dynamic typing), or if the cached version does not match the current version of the bound type. This mechanism is also used for caching methods, not only types.

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

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

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

5.2 Paths

Syntax:

```
Path             ::= Stable_Id
                   | [id '.'] 'self'
                   | 'self' '[' ('cloned' | 'origin') ']'
                   | 'outer' Class_Qualifier
Stable_Id        ::= [Path '.'] id
                   | [id '.'] 'super' [Class_Qualifier] '.' id
                   | 'outer' Class_Qualifier '.' id
                   | Module_Path '.' Stable_Id
Class_Qualifier  ::= '[' (id | Simple_Type) ']'
Module_Path      ::= [Root] Module_Selector {'.' id}
Module_Selector  ::= [Vendor] id
Root             ::= '@Root' '.'
Vendor           ::= id '/'
```

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

A path is one of the following:

- The empty path ϵ (which can not be written explicitly in user programs).
- **`this`**, which references the directly enclosing class.
- C**`.self`**, where C references a class or a trait. The path **`self`** is taken as a shorthand for C**`.self`**, where C is the name of the class directly enclosing the reference. A special path `Function`**`.self`**, where `Function` is defined by Gear'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).
- **`self[cloned]`**, which references the directly enclosing class of a clone (a cloned instance, see §7.3.3).
- **`self[origin]`**, which references the directly enclosing class of an original object (an instance being cloned, see §7.3.3).
- $p.x$, 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.

5.3 Value Types

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

5.3.1 Singleton Type

Syntax:

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

A singleton type is of the form p`.type`, where p is a path pointing to a value. The type denotes the set of values consisting of 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`.

5.3.2 Cloned Type

Syntax:

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

A cloned type is of the form p`.cloned.type`, 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.

5.3.3 Literal-Based Singleton Type

Syntax:

```
Simple_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[3], the "`.type`" part of the type can be omitted.

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

5.3.4 Type Projection

Syntax:

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

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

Example 5.3.1 A sample code that shows off what type projections are good for:

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

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

This is due to the fact that the `class B` is defined as a class instance member of `class A`, not as a class object member (either via object definition (§7.4) or some form of metaclass access (§7.1.4)). Therefore, `a1.B` refers to type member `B` of the instance `a1`, but not of `a2`. Moreover, `A.B` is not defined here.

5.3.5 Type Designators

Syntax:

```
Simple_Type ::= Stable_Id
```

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

Specifically, the unqualified type name t where t is bound in some class C is taken as a shorthand for C`.self.type#`t. If t is bound in some object[4] or module C, it is taken as a shorthand for C`.type#`t. If t is not bound in a class, object or module, then t is taken as a shorthand for ϵ`.type#`t.

A qualified type designator has the form p`.`t, where p is a path (§5.2) and t is a type name. Such a type designator is equivalent to the type projection p`.type#`t.

[4]Also in class object methods.

Example 5.3.2 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" *)
(* predefined import of Gear/Language *)
Number = Gear/Language.type#Number

object An_Object {
  type t
  t = An_Object.type#t (* bound by object *)
}

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

5.3.6 Parameterized Types

Syntax:

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

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

Say the type parameters have lower bounds $L_1, \ldots, L_n$ and upper bounds $U_1, \ldots, U_n$. The parameterized type is well-formed if each actual type parameter *conforms to its bounds*, so that $\sigma L_i <: T_i <: \sigma U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$[5]. Also, U_i must

[5]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`]`.

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 parameter may be given a name n, which is done by prepending it with an id immediately followed by a colon, which itself may be prepended with a tilde "~". The first case is a shorthand for adding an annotation `@[named :`n`]`, the second is a shorthand for adding the previous annotation and also `@[purely_named]`.

Example 5.3.3 Given the generic type definitions:

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

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

trait Function_1[T, R]; end
```

the following parameterized types are well-formed:

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

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

Function_1[named: List[String], String]
```

Example 5.3.4 Given the type definitions of the previous example, the following types are malformed:

```
Tree_Map[I]               (* wrong number of parameters *)
Tree_Map[List[I], Number] (* type parameter List not within bound *)

F[Number, Boolean]        (* Number is not a type constructor
F[Tree_Map, Number]       (* Tree_Map takes two parameters,
                             F expects a type constructor taking one *)

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

5.3.7 Tuple Types

Syntax:

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

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

A tuple type $T_1 * \ldots * T_n$ is an infix type (§5.3.10) representing the exact same tuple type as $(T_1, \ldots, T_n)$.

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 parameters T_i is annotated to be named n (see §5.3.6, then an additional method named n is added, returning the same value as i, of the type T_i. If that type parameter is furthermore annotated with `@[purely_named]`, then the i is removed.

5.3.8 Annotated Types

Syntax:

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

An annotated type $a_1 \ldots a_n T$ attaches annotations (§14) $a_1, \ldots, a_n$ to the type T.

5.3.9 Compound Types

Syntax:

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

A compound type T_1 **`with`** ... **`with`** T_n **`with refinement`** $\{R\}$ represents values with members as given in the component types $T_1, \ldots, T_n$ and the refinement $\{R\}$. A refinement $\{R\}$ contains declarations and definitions (§7.3.7).

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

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

If a compound type does not contain a concrete class type, then `Object` is implied in case the type is used as a concrete class[6].

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

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

The **`refinement`** keyword may also be omitted from a compound type that consists of just the refinement, but only in contexts in which a type is expected, i.e.: parameter type declaration, value or variable type declaration, result type declaration, type argument application or type parameter declaration; but never stand-alone. If used as such, the constructs **`refinement`** {R} and {R} are then equal.

5.3.10 Infix Types

Syntax:

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

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

Infix type may also result from an infix expression (§9.3.10), 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.

5.3.11 Function Types

Syntax:

```
Type          ::= Function_Args {'->' Function_Args}
                  '->' Type
Function_Args ::= Infix_Type
                | '(' [Function_Arg {',' Function_Arg}] ')'
Function_Arg  ::= ['*' | '**' | '&' | ['~'] id ':'] Param_Type
```

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

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

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

- "`out`" maps to `@[out]`.
- "**`in`**" maps to no annotation, as it is implied, and if the parameter is `out`-only, the function is still applicable.
- "`*`" maps to `@[variadic]`.
- "`**`" maps to `@[capturing_named]`.
- "`&`" maps to `@[captured_block]`.
- "*id*" maps to `@[named :`*id*`]`.
- "*~id*" maps to `@[named :`*id*`] @[purely_named]`.

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

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

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

Function types are covariant in their result type and contravariant in their argument types (§6.9).

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

Function arguments may be optionally annotated with more requirements:

- Parameter prefixed with "`*`" is a requirement of a repeated parameter.
- Parameter prefixed with "*id*" is a requirement of a parameter named *id*.
- Parameter prefixed with "*~id*" is a requirement of a parameter purely named *id*.
- Parameter prefixed with "`**`" is a requirement of a captured named parameters.

- Parameter prefixed with "`&`" is a requirement for the passed block. It does not tell whether the function must capture the passed block, it only restricts the requirements for the particular block, if any. The actual passed block may have more or even less positional or named parameters (extra ones on the block side are given **`undefined`**, unless the type is not nullable (§5.3.13) – that is an error; and extra ones on the type side are simply discarded), but the result type of the passed block must conform (§5.10.2).

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

Example 5.3.5 The following definition of functions `g` and `h` conform to a definition of a function `f`, and there are many more such functions that conform to `f`. And vice versa, `f` conforms to both `g` and `h`, although the latter two are more specific than the first one (§9.9.3).

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

5.3.12 Existential Types

Syntax:

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

An existential type has the form T **`for-some`** `{` Q `}`, where Q is a sequence of type declarations (§6.7). Let $t_1[tps_1] >: L_1 <: U_1 \ , \ldots , \ t_n[tps_n] >: L_n <: U_n$ be the types declared in Q (any of the type parameter sections $[tps_i]$ might be missing). The scope of each type t_i includes the type T and the existential clause Q. The type variables t_i (occurring in the sequence Q) are said to be *bound* in the type T **`for-some`** `{` Q `}`. Type variables that occur in a type T, but which are not bound in T, are said to be *free* in T.

A *type instance* of T **`for-some`** `{` Q `}` is a type σT, where σ is a substitution over $t_1, \ldots, t_n$, such that for each i, $\sigma L_i <: \sigma t_i <: \sigma U_i$. The set of values denoted by the existential type T **`for-some`** `{` Q `}` is the union of the set of values of all its type instances. In other words, a type instance of an existential type is a type application (§9.3.6) of the type T **`for-some`** `{` Q `}`, where the applied type arguments conform to the bounds, or are free in T **`for-some`** `{` Q `}`.

A *skolemization* of T `for-some` { Q } is a type instance σT, where σ is the substitution $[t_1 := t'_1, \ldots, t_n := t'_n]$ and each t'_i is a fresh abstract virtual type with lower bound σL_i and upper bound σU_i[7]. 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 (§6.9) of t in T by U and by replacing every contravariant occurrence of t in T by L.

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

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

A wildcard type must appear as a type argument of a parameterized type. Let T := $p.c$[*targs*, T, *targs*′] be a parameterized type, where *targs*, *targs*′ may be empty and T is a wildcard type `_` >: L <: U. Then T is equivalent to the existential type

$p.c$[*targs*, t, *targs*′] `for-some` { `type` t >: L <: U }

where t is a fresh type variable. Wildcard types may also appear as parts of compound types (§5.3.9), function types (§5.3.11) or tuple types (§5.3.7). Their expansion is then the expansion in the equivalent parameterized type.

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

Example 5.3.6 Assume the class definitions

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

Here are some examples of existential types:

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

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

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

which is equivalent to Java's

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

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

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

Example 5.3.7 The type `List[List[_]]` is equivalent to the existential type

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

Example 5.3.8 Assume a covariant type

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

The type

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

is equivalent (by simplification rule 4 above[8]) to

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

which is in turn equivalent (by simplification rules 2 and 3 above[9]) to

```
List[Number]  .
```

[8]As `T` appears in covariant position in `List`, its upper bound can replace the type variable in `List`.

[9]The type variable `T` is unused, and after dropping it, the quantification is empty.

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

5.3.13 Nullable Types

Syntax:

```
Annot_Type    ::= {Annotation} Simple_Type Nullable_Mod
Nullable_Mod  ::= '?' | '!'
```

A nullable type has the form T`?`, where "`?`" denotes a nullable type. Although **`nil`** as the singleton member of the `Nothing` type is a subtype of every type, Gear types are implicitly not-nullable, meaning that the language complains when the **`nil`** value is assigned to a variable whose type is not bound to be nullable. And it does so by issuing compile-time warnings and runtime warnings[10] when the **`nil`** value is assigned to a type that is not nullable, including parameter assignment and result value assignment. This is due to the fact that the type `Nothing` conforms to any type that descends from `Any`, which is, well, any type.

A block of code (or whole files or modules) may opt-in to behaviour that raises errors in runtime, instead of issuing warnings, using **`pragma`** `error_on_nil` inside the code block, file or module.

If an implicit conversion from `Nothing` to T is available, then nullability is not a problem.

Nullable types in this form can appear everywhere where a type is expected.

A nullable type may be seen as a syntax sugar for the following union type `Nullable[`T`]` (§8.2):

```
type Nullable [T] is union of
  T
  Nothing
end type
```

When a method is applied on a variable containing **`nil`**, the standard `Member_Not_Found` may occur, if the method is really not found in the class `Nothing` or in any of implicit conversions from `Nothing` (e.g., `Nothing` has an implicit conversion to the option type `None`).

A strictly not-nullable type has the form T`!`, where "`!`" denotes a not-nullable type. Gear types are implicitly not-nullable, but strictly not-nullable types have the extra property that **`nil`** and **`undefined`** do not cause a warning when assigned to a variable or value that is of a strictly not-nullable type, but raise a type error. In case the referenced value changes in runtime by other means than assignment (native access or maybe due to hot code load), the same error has to be raised.

A compiler may decide to issue extra warnings when assignment of a **`nil`** or **`undefined`** can be detected.

[10] In debug mode.

A strictly not-nullable may be seen as some implicitly wrapped-on-assignment / unwrapped-on-read box that checks for `value = nil` and raises an error when such condition is true. But Gear does not have any special methods for object dereference.[11]

When used in a compound type, use nullable modifier only on the first component. When used in a constrained type, use nullable modifier only on the component type of the constrained type.

5.3.14 Unions

Syntax:

```
Const_Type_Def ::= id [Type_Param_Clause] 'is' Union
Union          ::= 'union' 'of' Type {semi Type}
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] is Union[A, B]
```

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` except `Undefined`.

[11] Yet, and probably never will.

- `X or Undefined` 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.

5.3.15 Intersection Types

Syntax:

```
Const_Type_Def ::= id [Type_Param_Clause] 'is' 'intersection' 'of'
                   Type {semi Type}
Infix_Type     ::= Type {'and' Type}
```

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 difference between intersection types and compound types (§5.3.9) is that intersection types are commutative and associative – independent on the order in which the member types appear.

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` except `Undefined`.
- `X and Undefined` is equivalent to `Undefined` 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.

5.3.16 Constrained Types

Syntax:

```
Postfix_Type     ::= Compound_Type
                     'with' 'constraint' Constraint_Block
Constraint_Block ::= '{' Block '}'
```

A constrained type constructs a subset of allowed members of its component type. Such a subset is defined as members for which the constraint block returns boolean **yes**. Constraints defined within the component type are implicitly available in such block as local variables. In case of concurrent changes in the member, the type may not match anymore – a warning is issued when a constraint is applied to a member that is not frozen nor immutable.

Constrained types, in face of overloading, trigger eager argument expression evaluation, so that the constrained type of a parameter can even be checked.[12]

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

5.3.17 Mutable & Immutable Types

Gear offers two annotations that can constrain a type to either *mutability* or *immutability*. Implicitly, types are not constrained in such a way, therefore a type could represent both mutable and immutable values at the same time.

Mutable types. A mutable type is expressed with the annotation `@[mutable]`, and is not transitive. It constrains a type to mutable values (i.e. not frozen), but not values referred to by the values it represents.

Immutable types. An immutable type is expressed with the annotation `@[immutable]`, and is transitive. It constrain a type to immutable values (i.e. frozen) and also requires that all values that are referred to by those immutable values to be also immutable, unless they are explicitly typed as mutable.

Note. See also section about storage of mutable and immutable values: (§6.6).

A **mutable** type can be seen as a constrained type, where the constraint is:

T `with constraint { not self'is_frozen? }` ,

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

where **self** would be a constraint published by the type T, referring to the **self** of a value of the type T.

An **immutable** type can be seen as a constrained type, where the constraint is:

T **with constraint** { **self**'is_frozen? } ,

where **self** would be a constraint published by the type T, referring to the **self** of a value of the type T.

If a mutable or immutable type is seen as a constrained type, but T is already a constrained type, then the condition presented by the already constrained type T can be seen without loss of generality as joined with the mutable/immutable constraint by **and**.

5.3.18 Foreign Types

Syntax:

```
Simple_Type  ::= Foreign_Type
Foreign_Type ::= 'foreign' lang_name '{'
                 printable_char_or_space {',' printable_char_or_space}
                 '}'
lang_name    ::= id
```

A *foreign type* is a tool that helps Gear programs integrate with other languages written for the same VM, the Gear 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 Gear types.

Syntax note. If an opening delimiter "{" appears within the foreign language type name, then it is properly paired with the next "}" by Gear and not treated as closing delimiter, but this is done without any knowledge of the foreign language, only presuming that these delimiters are paired.

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

Example 5.3.9 A foreign type aliases for a few foreign types:

```
type Java_Object is foreign java {java.lang.Object} end type
type Java_int is foreign java {int} end type
type CS_Object is foreign `c#` {System.Object} end type
type Rb_Proc_Status is foreign ruby {Process::Status} end type
```

5.4 Non-Value Types

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

5.4.1 Method Types

A method type is denoted internally as $(Ps) \mapsto R$, where (Ps) is a sequence of types $(p_1 : T_1, \ldots, p_n : T_n)$ for some $n \geq 0$ and R is a (value or method) type. This type represents named or anonymous methods that take arguments of types $T_1, \ldots, T_n$ and return a result of type R. Types of parameters are possibly annotated with conformance restricting annotations (§14).

Method types associate to the right:[13]
$(Ps_1) \mapsto (Ps_2) \mapsto R$ is treated as $(Ps_1) \mapsto ((Ps_2) \mapsto R)$.

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

Another special case are types of methods without any result type. They are written here as $(Ps) \mapsto ()$. Methods that have this result type do not have an implicit return expressions and an attempt to return a value from it results in a compile-time error.[14]

Method types do not exist as types of values. If a method name is used as a value, its type is implicitly converted to a corresponding function type (§5.3.11).

Extra properties of parameters are as follows: a `*` for variadic parameters, `**` for any named parameters and `&` for a captured block parameter, or nothing for regular parameters.

Example 5.4.1 The declarations

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

produce the typings

```
a : () ↦ Integer
b : (@[named :x] Integer) ↦ Boolean
c : (@[named :x] Integer) ↦ (y: String, z: String) ↦ String
d : (@[named :x] @[purely_named] Integer) ↦ Integer
```

[13] Like in Haskell or Scala.

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

```
e : (@[named :x] @[variadic] Integer) ⟼ Integer
f : (@[named :x] Integer) ⟼ Unit
g : (@[named :x] Integer) ⟼ (@[named :y] Integer) ⟼ Integer
h : (@[named :x] Integer) ⟼ (@[named :y] Integer) ⟼ Integer
```

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

5.4.2 Polymorphic Method Types

A polymorphic method type is the same as a regular method type, but enhanced with a type parameters section. It is denoted internally as [*tps*]⟼ T, where [*tps*] is a type parameter section [$\pm a_1$ >: L_1 <: U_1 , ..., $\pm a_n$ >: L_n <: U_n] for some $n \geq 0$ and T is a (value or method) type. This type represents (only[15]) named methods that take type arguments $S_1, \ldots, S_n$, for which the lower bounds $L_1, \ldots, L_n$ conform (§5.10.2) to the type arguments and the type arguments conform and the upper bounds $U_1, \ldots, U_n$ and that yield results of type T. No explicit lower bound implies `Nothing` to be the corresponding lower bound, no explicit upper bound implies `Object` to be the corresponding upper bound. As usual, lower bound must conform to the corresponding upper bound.

Example 5.4.2 The declarations

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

produce the typings

```
empty : [A >: Nothing <: Any] () ⟼ List[A]
union : [A >: Nothing <: Comparable[A]] (Set[A],
        Set[A]) ⟼ Set[A]
```

5.4.3 Type Constructors

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

[15]Not anonymous.

5.5 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[16] are the base types of its upper bound.
   - The base types of a parameterized type C`[`$T_1, \ldots, T_n$`]` are the base types of type C, where every occurence of a type parameter a_i of C has been replaced by the corresponding parameter type T_i.
   - The base types of a compound type T_1 `with ... with` T_n `with {` R `}` are set of base classes of all T_i's.
   - The base types of a type projection S`#`T are determined as follows: If T is an alias or an abstract type, the previous clauses apply. Otherwise, T must be a (possibly parameterized) class type, which is defined in some class B. Then the base types of S`#`T are the base types of T in B as seen from the prefix type S.
   - The base types of an existential type T `for-some {` Q `}` are all types S `for-some {` Q `}`, where S is a base type of T.
2. The notion of a type T *in class* C *seen from some prefix type* S makes sense only if the prefix type S has a type instance of class C as a base type, say S'`#`C`[`$T_1, \ldots, T_n$`]`. Then we define it as follows.
   - If S `=` ϵ`.type`, then T in C seen from S is T itself.
   - Otherwise, if S is an existential type S' `for-some {` Q `}`, and T in C seen from S' is T', then T in C seen from S is T' `for-some {` Q `}`.
   - Otherwise, if T is the i^{th} type parameter of some class D, then:

[16] E.g. type members.

  - 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:

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

   (b) All bindings of the type's refinement (§5.3.9), 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.

5.6 Any-Value Type

Syntax:

```
Simple_Type ::= 'Any'
(* also: *)
Simple_Type ::= 'Gear/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 Gear in the VM.

With respect to overloading resolution (§9.9.1), 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 (§5.3.9), 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 Gear, although explicit inheritance is not allowed from it. Other languages that integrate with Gear on the same VM may have `Any` as their root type as well, disobeying the rule that is set only for Gear. This way, Gear preserves its unified type system.

5.7 Structural Inferred Auto Type

Syntax:

```
Simple_Type ::= 'Auto'
(* also: *)
Simple_Type ::= 'Gear/Language.Auto'
```

This type does represent a single concrete value type: a compound type with usually just the refinement part. However, it does not explicitly state what the refinement is. Instead, the refinement is inferred from the context in which it is used. The inference algorithm is very different from local type inference (§9.9.4).

Structural constraints may be appended to this type as with any other type by using a new compound type (§5.3.9), where `Auto` is used as a class type (never a trait type) and its parts of the inferred refinement are merged with any given refinement from the compound type.

The `Auto` 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 `Inferred`).

Elements that determines the inferred parts of the `Auto` type:

- Any methods that are invoked on the value that has this type assigned. These are parts of the refinement.
- Any classes or traits that are required of the value, except when there is a definitive **`else`**-style clause that allows the value to be of any other type as well. Those requirements are then presented in a union type, with the refinement being compound with each member of the union type.

- If the value is passed as an argument anywhere, where the expected type is determined to be not `Any` or locally type inferred, then the aforementioned handling applies as if the expected type was required of the value.
- If a method is dynamically added to the value before being required, then it is not included in the refinement.
- The `Auto` type is also automatically generalized. If a method that uses it has already type parameters, then these are appended after them, in order of appearance, with method result type being the last one.
- If the value is passed as an argument, with an expected type to be locally type inferred, then it itself becomes typed with an unknown type and a type parameter is appended to the containing function.

There is one important limitation: the `Auto` type must not cause circular references between functions. In that case, the type has to be specified in another way. This inference happens only during compilation, unlike local type inference, which takes place in runtime.

This kind of type inference is useful when a method only cares about actually used capabilities of it's parameters. In that way, it is similar to protocols, but only includes those capabilities that are actually used. Note that in runtime, test of structural types (refinements) is somewhat slower than simple type test, depending on the amount of members present in the refinement of the resulting compound type. This is common to compound types, where the test needs to check all components and not just one.

Example 5.7.1 An automatically generalized swap function.

```
def swap (x: Auto, y: Auto): Auto
  (y, x)
end
```

This is inferred and generalized into the following polymorphic function:

```
def swap [A, B] (x: A, y: B): (B, A)
  (y, x)
end
```

This inference and generalization is possible, because no other requirements are imposed on the values `x` and `y` and the result type is unaffected either.

Example 5.7.2 An automatically inferred function.

```
def print (x: Auto): Unit
  IO.console.print x.to_string
end
```

This is inferred into the following function:

```
def print (x: { def to_string (): String end }): Unit
  IO.console.print x.to_string
end
```

This happens because the only requirement on `x` is that it has a method `to_string` that accepts no arguments (could be variadic either), which is a requirement from the function body, and that method returns a `String`, which is a known requirement of parameter type of the `IO.console.print` function. Note that the refinement type is equivalent to:

```
Any with refinement { def to_string (): String end }
```

Example 5.7.3 Another automatically inferred function.

```
def print (x: Auto): Unit
  IO.console.print x
end
```

This is inferred into the following function:

```
def print (x: String): Unit
  IO.console.print x
end
```

Here, the only requirement on the value `x` comes from a known requirement of parameter type of the `IO.console.print` function. In fact, if `IO.console.print` had several overloaded variants, then the type of `x` would be inferred to be a union type of all the overloaded variants' requirements on it.

By default, when a type annotation is missing in a function declaration, then it is either inherited (based on overriding), or taken to be `Auto`. A pragma (§14.2) exists to modify this default behaviour: to use other type in place of missing type annotation, use **`pragma`** `missing_type` T, where T can be also `Any`, or any other type, like `Object` and even `Dynamic` (§5.8).

Note that `Auto` type inference does not track requirements of values from their usage within other methods.

5.8 Dynamic Type

Syntax:

```
Simple_Type ::= 'Dynamic'
(* also: *)
Simple_Type ::= 'Gear/Language.Dynamic'
```

This type does not represent a single concrete value type[17], but any concrete type, very much like `Any` or `Auto`.

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, 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 (§9.5.1). Nonetheless, this does not imply that a value of another type can not 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 (§5.3.9), where `Dynamic` is used as a trait type.[18]

With respect to overloading resolution (§9.9.3), this type is almost[19] 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 (§9.5.1).

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`.

5.9 Variadic Arguments Type

Syntax:

```
Simple_Type ::= 'Variadic_Arguments'
(* also: *)
Simple_Type ::= 'Gear/Language.Variadic_Arguments'
```

This type is recognized only as a parameter type (be it a function, constructor…), and with a further constraint that it has to be applied to a solely parameter of a whole parameter list. Such parameter list must be the last parameter list of a definition or declaration, and there might be more parameter lists present preceding it.

When applied to a parameter, the parameter then consumes a tuple of every consecutive argument list with all arguments in a tuple, possibly with named members, if the arguments were passed in

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

[18]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.

[19]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.

as named arguments. If a block argument is also given, it is present in the last tuple element, under the last nested tuple element, named "`&`". Any possible arguments combination is applicable to this type, it consumes virtually anything. The types of the tuples per each argument list are to be inspected at runtime dynamically.

The parameter that receives this tuple of tuples is typed as if it was:

$$((T_1, \ldots, T_j),\ (T_{j+1}, \ldots, T_k), \ldots, (T_l, \ldots, T_n)) \quad ,$$

where each T has a type that is retrieved as a function type `() ->` U, where U is the type available at function application time. The runtime is however not required to absolutely convert all arguments into functions (that would not be very efficient), but it can (and should) presume that all arguments are passed by-name, and then the function can evaluate them as appropriate. The function notion is there mostly for user programs that do not have access to the native representation, so that it can decide which arguments to treat as by-name and which by-value, as there is no implicit "function application" of by-name parameters.

This type does not represent by itself any concrete value type, instead, it represents a whole class of them (two-dimensional tuples of variable lengths, basically).

If no arguments are passed to a function that accepts this parameter type, a `Unit` value (`()`) is used. The `Unit` value is also used for matched arguments list that carry no argument expressions. As a special case, when only one argument list is matched, the type is `Tuple_1`, a type that is generally avoided throughout Gear (e.g., `(`e **`as`** T`)` is not of type `Tuple_1[`T`]`, but just T), as its contents can be implicitly unwrapped, but here it is needed to keep the semantics consistent.

Note. This type is used e.g. within the `Language.Object.new` method, so that it can select the appropriate constructor (and perhaps does so natively on each Gear VM).

5.10 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.

5.10.1 Type Equivalence

Equivalence ($\equiv$) 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`** $\equiv$ q**`.type`**.

- Two compound types (§5.3.9) are equivalent, if the sequences of their components are pairwise equivalent, occur in the same order and their refinements are equivalent.
- Two constrained types (§5.3.16) are equivalent, if their base types are equivalent and their constraint blocks read the same.
- Two refinements (§5.3.9 & §7.3.7) 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 (§5.4.1) are equivalent, if they are *override-equivalent* (§6.10.9).
- Two polymorphic method types (§5.4.2) 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 (§5.3.12) 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 (§5.4.3) 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 (§8.2) are equivalent, if they have the same number of member types and those types are pairwise equivalent.
- Two intersection types (§5.3.15) are equivalent, if they have the same number of member types and those types are pairwise equivalent.

5.10.2 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 (§8.4) 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 (§8.4) 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[20].
- A unit of type N conforms to unit of type U if N extends U or if $N \equiv U$.
- A parameterized type T`[<`$N_1, \ldots, N_n$`>]` conforms to T`[<`$U_1, \ldots, U_n$`>]` if the following conditions hold for $i = 1, \ldots, n$.
  - For the i^{th} type parameter of T, $N_i <: U_i$.
- A compound type T_1 `with ... with` T_n `with {` R `}` conforms to each of its component types T_i.
- Conformance of intersection types is defined in (§5.3.15).
- A constrained type T `with constraint` b conforms to T. A type T `with constraint` b_1 conforms to T `with constraint` b_2, if any value e that conforms T `with constraint` b_1 by being a member of T and passing the test presented by b_1, also conforms to T `with constraint` b_2 by passing the test presented by b_2.[21]
- If $T <: U_i$ for $i = 1, \ldots, n$, and every binding d of a type or value x in R exists a member binding of x in T which subsumes d, then T conforms to the compound type U_1 `with ... with` U_n `with {` R `}`.

[20]Quite obviously.

[21]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 (§5.3.12) 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 (§5.3.12) 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 (§14), 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 $(T_1, \ldots, T_n) \Rightarrow R$ (name it f) conforms to a function type $(T'_1, \ldots, T'_m) \Rightarrow R'$ (name it f'), if types of arguments that are applicable to f' are also applicable to f (§9.3.5), and if R conforms to R'. This includes reordering of named arguments/parameters and handling of repeated/optional parameters, and also variances – since although f might be applied to whatever f' might be applied to, f' might not be applied to whatever f might be applied to (and vice versa).

- Polymorphic function traits `Function_`n and `Partial_Function` follow the rules for conformance of function types, as defined above. The repeated parameter, optional parameters, named parameters and all capturing parameters are derived from their conformance

restricting annotations (§14). 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 (§8.2) 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: (§8.2).

- 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. Gear 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, Gear is free to pick up any of them.

5.10.3 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 (§8.2) out of all types in the set.

Gear 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 Gear's `Language` module.

5.10.4 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.

5.11 Reified Types

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

- 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 Gear 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.

5.12 Types Representing Emptiness

In Gear, 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. It represents a missing object in places where an object is expected.

   The type has just one inhabitant, and that is the frozen **`nil`** value. In a Gear VM, this type is usually an immediate value.

2. The `Undefined` type. It represents a missing object in places where an object is expected, but its source is not defined. This applies to non-defined instance variables, undefined variables, or undefined mappings in, e.g., `Map` implementations.

   The type has just one inhabitant, and that is the frozen **`undefined`** value, which is equal to, but not identical to **`nil`**. In a Gear VM, this type is usually an immediate value.

   The semantical difference between a **`nil`** and **`undefined`** in context of, e.g., `Map` implementations is that **`nil`** is returned if the map contains the given key, but has **`nil`** assigned to it on purpose, whereas **`undefined`** means there was no mapping defined.

   Gear forbids access to undefined local variables during compilation, unlike e.g. JavaScript.

3. The `None` case object. It is a subtype of `Option[`T`]` and represents the exact same thing as `Nothing`, but has the extra semantical meaning of "being ready to handle missing values", which non-nullable types do not represent – those expect a value to be present and cause errors when it is not present.

   The `None` case object is primarily designed to be used with pattern matching, and also plays an important internal part in Gear's implementation of pattern matching.

   Implicit conversion (§9.9) from `Nothing` and `Undefined` to `Option[`T`]` is defined in `Language.Predef`, and results in `None`.

4. The `Unit` type. It is basically a `Tuple_0` type.

   It's semantical meaning is that no value is even expected. It is used for functions that should never return any value, 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 Gear VM, this type is usually an immediate value.

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

The following lines show some common relations between empty values.

```
nil = undefined  (* yes *)
nil == undefined (* no *)
nil = None       (* no *)
undefined = None (* no *)
() = undefined   (* no *)
() = nil         (* no *)

Nothing === nil         (* yes *)
Nothing === undefined   (* yes *)
Nothing === ()          (* no *)
Undefined === nil       (* no *)
Undefined === undefined (* yes *)
Undefined === ()        (* no *)
Unit === nil            (* no *)
Unit === undefined      (* no *)
Unit === ()             (* yes *)

val nothing := nil
match nothing {
  when Some(x) then () (* does not match *)
  when None then ()    (* matches *)
}

val nothing := get_undefined (* a function that returns `undefined' *)
match nothing {
  when Some(x) then () (* does not match *)
  when None then ()    (* matches *)
}

val unit = ()
match unit {
  when Some(x) then () (* does not match,
                          also compilation warning, because Unit
                          is not compatible to Option[T] *)
  when None then ()    (* does not match, also not compatible *)
  when Unit then ()    (* matches *)
}
```

Chapter 6

Basic Declarations & Definitions

Contents

Syntax:

```
R_Modifier  ::= 'weak' | 'soft' | 'unowned'
M_Modifier  ::= 'constant' | 'mutable' | 'immutable'
S_Modifier  ::= M_Modifier [R_Modifier]
              | R_Modifier
Dcl         ::= [S_Modifier] 'val' Val_Dcl
              | [S_Modifier] 'var' Var_Dcl
              | 'def' Def_Dcl
              | 'message' Fun_Dcl 'end' ['message']
              | 'function' Fun_Dcl 'end' ['function']
              | 'type' Type_Dcl
Def         ::= Pat_Var_Def
              | 'def' Fun_Def
              | 'method' Fun_Def 'end' ['method']
              | 'method' Fun_Alt_Def
              | 'function' Fun_Def 'end' ['function']
              | 'function' Fun_Alt_Def
              | 'type' Type_Def 'end' ['type']
              | Tmpl_Def
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).

6.1 Value & Variable 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 (§11):

- 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.

6.2 Value Declarations & Definitions

Syntax:

```
L_Modifier   ::= S_Modifier - ('constant', 'mutable')
Dcl          ::= [S_Modifier] 'val' Val_Dcl
Val_Dcl      ::= val_ids ':' Type
Pat_Var_Def  ::= [S_Modifier] 'val' Pat_Def
               | [L_Modifier] 'let' ['rec'] Pat_Var_Nest
                 {nl 'and' Pat_Var_Nest}
Pat_Var_Nest ::= (Pattern2 | var_dcl) ':='
                 (Expr | Pat_Var_Def In_Sep Expr)
Pat_Def      ::= Pattern2 {',' Pattern2} ':=' Exprs
               | [[var_ids ','] '*' id [':' Type] ','] var_ids
                 ':=' Mul_Exprs
               | [var_ids ','] '*' id [':' Type]
                 ':=' Mul_Exprs
var_ids      ::= var_dcl {',' var_dcl}
var_dcl      ::= var_bind | fun_bind
var_bind     ::= id [':' Type]
fun_bind     ::= id [Type_Param_Clause] Param_Clauses [':' Type]
val_ids      ::= id {',' id}
```

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

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

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

The value type T may be always omitted, in that case the type is inferred and bound to the name. If a type T is omitted, the type of expression e is assumed. If a type T is given, then e is expected to conform to it (§5.10.2).

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

A *lazy value* is of the form

lazy val x: T := e

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

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

A *constant value definition*, or a *let binding*, is of the form

let x: T := e

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

1. The modifier **constant** is implicitly added, and can not be overwritten. Therefore, it would be redundant in the syntax. The **constant val** construct is somewhat similar[3].
2. They can't be used in a declaration, only in a definition.
3. They allow for recursive definitions. Normally, the defined value or variable name is not available for recursive use inside of a defined function, unless it is pre-declared (or already overloaded on site, which is then both declared and defined), but using **let rec** does both and makes the name available for recursion.

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

[3] Somewhat only because of the differences between **let** bindings and **val** definitions.

4. Mutual recursion is also possible using **`let rec`**, where the definitions are concatenated with "**`and`**".

5. More value definitions can be nested inside the expression that defines the bound value.

The type T may be omitted.

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

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

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

A value definition can alternatively have a pattern (§11.1) as left-hand side (the name). If p is a pattern other than a simple name or a name followed by a colon and a type, then the value definition **`val`** p `:=` e is expanded as follows:

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

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

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

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

3. If p has no bound variables:

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

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

Example 6.2.1 The following are examples of value definitions.

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

The last three definitions have the following expansions:

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

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

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

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

The following shorthands are recognized:

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

A value definition **val** $p_1, \ldots, p_n$:= e is a shorthand for the sequence of value definitions **val** p_1 := e; ...; **val** p_n := e. 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 (§6.10).

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

6.3 Variable Declarations & Definitions

Syntax:

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

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

A variable definition **var** x`:` T `:=` e defines x as a name of the value that results from evaluation of expression e. The type T can be omitted, in that case the type of expression e is assumed, but not bound to the variable – the variable is only bound to `Object` then. If the type T is given, then e is expected to conform to it (§5.10.2), as well as every future value of the variable.

Variable definitions can alternatively have a pattern (§11.1) as their left-hand side. A variable definition **var** p `:=` e, where p is a pattern other than a simple name followed by a colon and a type, is expanded in the same way (§6.2) as a value definition **val** p `:=` e, except that the free names in p are introduced as mutable variables instead of values.

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

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

`0`	if T is `Integer` or one of its subrange types,
`0.0f`	if T is `Float` or `Half_Float`,
`0.0d`	if T is `Double`,
`0.0q`	if T is `Quadruple`,
`0.0df`	if T is `Decimal` or `Real`,
`0 / 1r`	if T is `Rational`,
`0 + 0i`	if T is `Complex`,
no	if T is `Boolean`,
`()`	if T is `Unit`,
nil	for all other types T.

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

The following shorthands are recognized:

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

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

A variable definition **var** $x_1, \ldots, x_n$: T := e is a shorthand for the sequence of variable definitions **var** x_1: T := e; ...; **var** x_n: T := e. Multiple such sequences may appear in a single variable definition, then every appearing T defined type of the preceding variables without an explicit type.

A variable definition **var** p_1 *ps*: $T, \ldots, p_n$ *ps*: T := e is a shorthand for the sequence of value definitions **var** p_1: *ps* -> $T, \ldots, p_n$: *ps* -> T := e , where *ps* are parameter sections (§6.10).

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

6.4 Property Declarations & Definitions

Syntax:

```
Prop_Dcl   ::= 'property' ['(' Prop_Specs ')'] id
               [':' Type]
Prop_Specs ::= Prop_Spec {',' Prop_Spec}
Prop_Spec  ::= ([Access_Modifier] ('get' | 'set'))
             | ['optional'] 'weak'
             | 'unowned'
             | ['optional'] 'soft'
Prop_Def   ::= 'property' ['(' Prop_Specs ')'] id
               [':' Type] Prop_Impls
Prop_Impls ::= '{' Prop_Impl {semi Prop_Impl} '}'
             | [nl] 'with' Prop_Impl {[nl] 'and' Prop_Impl}
Prop_Impl  ::= ([Access_Modifier] 'get' Prop_Get_Impl)
             | ([Access_Modifier] 'set' Prop_Set_Impl)
             | ([Access_Modifier] 'val' ':=' Expr)
             | ([Access_Modifier] 'var' ':=' Expr)
```

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

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

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

2. Properties defined with only **get** are stored in immutable instance variables (§6.2).

3. Properties defined with **set**[5] are stored in mutable instance variables (§6.3).

4. Properties defined with **weak** are stored as weak references. A property **property** x`:` T is stored in an instance of type `Weak_Reference[`T`]`.

5. Properties defined with **unowned** are stored as unowned references. A property **property** x`:` T is stored in an instance of type `Unowned_Reference[`T`]`.

6. Properties defined with **soft** are stored as unowned references. A property **property** x`:` T is stored in an instance of type `Soft_Reference[`T`]`.

7. The getter and setter, including both implicit and explicit versions, automatically wrap (assignment) or unwrap (evaluation) the corresponding reference type. The default values defined with **val** or **var** do the same.

8. If the property is declared as **optional**, then the property is stored in an instance variable of type `A_Reference[Option[`T`]]`, where `A_Reference` is one of `Weak_Reference`, `Unowned_Reference` or `Soft_Reference`, and is again automatically wrapped and unwrapped as appropriate, but inside the `Option[`T`]` type. Should the reference be cleared, then `None` is set as the value.

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

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

Assignment to properties is translated automatically into a setter function call and reading of properties does not need any translation due to implicit conversions (§9.9).

6.4.1 Property Implementations

```
Prop_Get_Impl ::= '(' ')' '->' '{' Block '}'
                | '{' Block '}'
                | 'do' Block 'end'
                | ':=' Expr
                | '_'
Prop_Set_Impl ::= '(' Param ')' '->' '{' Block '}'
```

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

```
| '{' '|' Param '|' [':' Type [semi]]
  [semi] Block '}'
| 'do' '|' Param '|' [':' Type [semi]]
  [semi] Block 'end'
| '(' Param ')' [':' Type [semi]] ':=' Expr
| '_'
```

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

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

6.5 Reference Types

Syntax:

```
R_Modifier ::= 'weak' | 'soft' | 'unowned'
```

Reference type modifiers are the syntax category `R_Modifier` (stands for Reference Modifier). The implicit modifier would be `strong`, but it does not have a corresponding keyword.[6]

Reference type modifiers are applied to variables, whose `id` begins with "`@`" (class instance variables) or with "`@@`" (class object variables). Such variable may appear as a part of pattern matching, in that case, the modifier is applied as well. By default, all other declared or defined variables are inferred to be `strong`. If a variable is declared with a non-`strong` reference type, then it does not have to be explicitly defined with the same reference type, and it is an error if it is defined with a different reference type.

Gear provides automatic wrapping and unwrapping of variables that are declared or defined non-`strong`. 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.

6.6 Mutable & Immutable Storage

Syntax:

```
M_Modifier ::= 'constant' | 'mutable' | 'immutable'
```

Mutability modifiers are the syntax category `M_Modifier` (stands for Mutability Modifier). The implicit modifier would be `unspecified`, but there is not a corresponding keyword.

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

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 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.

An exception to immutability is presented by the `R_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.

6.6.1 Strong Reference

This reference does not have a direct representation in Gear, 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.

6.6.2 Weak Reference Type

This reference type is represented in Gear 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 behaves as a 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 Gear VM implementation.

6.6.3 Unowned Reference Type

This reference type is represented in Gear 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. 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 Gear VM implementation, and may reuse internal representation of weak references.

6.6.4 Soft Reference Type

This reference is represented in Gear by the type `Soft_Reference[`T`]`.

Like weak references, an *unowned reference* 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 behaves as a nullable type. A value does not need to be released when there are only soft references referring to it – there are other facilities which may trigger clearing of soft references, including out-of-memory scenarios and explicit clearing triggers. Soft references are suitable for implementations of various caches.

Soft references are not required to be provided by proper Gear VM implementations. If a Gear VM uses garbage collection instead of automatic reference counting, it is suggested to provide support for soft references.

Support for soft references is to be queried with the following method:

```
Gear/Language.Soft_Reference.is_supported?
```

6.7 Type Declarations & Aliases

Syntax:

```
Dcl      ::= 'type' Type_Dcl 'end' ['type']
Type_Dcl ::= id [Type_Param_Clause]
             ['>:' Type] ['<:' Type]
Def      ::= 'type' Type_Def 'end' ['type']
Type_Def ::= id [Type_Param_Clause]
             (':=' | 'is') [['alias'] 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 (§5.10.2).

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

To illustrate nested scoping, these declarations are all equivalent:

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

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

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

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

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

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

Example 6.7.1 The following are legal type declarations and aliases:

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

The following are illegal:

```
type Abs := Comparable[Abs] end (* recursive type alias *)

type S <: T end  (* S, T are bounded by themselves *)
```

```
type T <: S end

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

6.8 Type Parameters

Syntax:

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

Type parameters appear in type definitions, class definitions and function definitions. In this section we consider only type parameter definitions with lower bounds `>:` L and upper bounds `<:` U, whereas a discussion of context bounds `:` T and view bounds `<%` T is deferred to chapter about implicit parameters and views (§10).

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* (§6.9), i.e. an optional prefix of either `+` or `-`. The type parameter may be preceded by one or more annotation applications (§9.5.2 & §14).

The names of all type parameters must be pairwise different in their enclosing type parameter clause. The scope of a type parameter includes in each case the whole type parameter clause. Therefore it is possible that a type parameter appears as part of its own bounds or the bounds of other type parameters in the same clause. However, a type parameter may not be bounded directly or indirectly by itself.

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

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

higher-order type parameters are thus often irrelevant, they may be denoted with a "_", which is nowhere visible.

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

Example 6.8.1 The following are some well-formed type parameter clauses:

```
[S, T]
[@[specialized] S, T]
[Ex <: Raiseable]
[A <: Comparable[B], B <: A]
[A, B >: A, C >: A <: B]
[M[X], N[X]]
[M[_], N[_]] (* equivalent to previous clause *)
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
[E, <Length_Unit <: length>]
[+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' *)
```

A type parameter name may optionally be prefixed with an asterisk "`*`". Such type parameter is a *variadic type parameter*, and may only appear as the last type parameter in a sequence of type parameters, and its resulting value is a tuple of all passed type arguments that were passed after the preceding type parameters. If such type parameter is used as the type of a function parameter, that parameter's type is a tuple type, as defined by the variadic type parameter, unless the function parameter is a repeated parameter, then the number of captured argument and their types are bound with the variadic type parameter (influencing local type inference (§9.9.4)).

Note. If variance annotation is present with a variadic type parameter, place a space between the two:

```
[+T, - *A]
```

6.9 Variance of Type Parameters

Variance annotations indicate how instances of parameterized types relate with respect to subtyping (§5.10.2). 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[...T...]: 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[...T...].

Example 6.9.1 In the following example, variance of positions is annotated with $^+$ (for positive) or $^-$ (for negative):

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

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

References to the type parameters in object-private or object-protected values, types, variables, or methods (§7.2) of the class are not checked for their variance position. In these members the type parameter may appear anywhere without restricting its legal variance annotations.

Example 6.9.2 The following variance annotation is legal.

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

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

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

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

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

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

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

Example 6.9.3 The following variance annotation is illegal, since a appears in contravariant position in the parameter of `append`:

```
abstract class Sequence [+A] {
  def append (x: Sequence[A]): Sequence[A] end
```

```
    (* error: illegal variance, 'A' occurs in contravariant position *)
}
```

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

```
abstract class Sequence [+A] {
  def append [B >: A] (x: Sequence[B]): Sequence[B] end
}
```

Example 6.9.4 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.

6.10 Function Declarations & Definitions

Syntax:

```
Dcl           ::= 'def' Fun_Dcl 'end' ['def']
                | 'message' Fun_Dcl 'end' ['message']
                | 'function' Fun_Dcl 'end' ['function']
                | 'operator' Op_Dcl 'end' ['operator']
                | 'attribute' Att_Dcl 'end' ['attribute']
Fun_Dcl       ::= Fun_Sig ':' Type
Att_Dcl       ::= Att_Sig ':' Type
Op_Dcl        ::= Op_Sig ':' Type
Def           ::= 'def' Fun_Def 'end' ['def']
                | 'def' Fun_Alt_Def
                | 'method' Fun_Def 'end' ['method']
                | 'method' Fun_Alt_Def
                | 'function' Fun_Def 'end' ['function']
                | 'function' Fun_Alt_Def
                | 'operator' Op_Def 'end' ['operator']
                | 'operator' Op_Alt_Def 'end' ['operator']
                | 'attribute' Att_Def 'end' ['attribute']
                | 'attribute' Att_Alt_Def 'end' ['attribute']
Fun_Def       ::= Fun_Sig [':' Type] [Fun_Dec] [semi Fun_Stats]
Fun_Alt_Def   ::= Fun_Sig [':' Type] ':=' Expr
Att_Def       ::= Att_Sig [':' Type] [Fun_Dec] [semi Fun_Stats]
```

```
Att_Alt_Def   ::= Att_Sig [':' Type] ':=' Expr
Op_Def        ::= Op_Sig [':' Type] [Fun_Dec] [semi Fun_Stats]
Op_Alt_Def    ::= Op_Sig [':' Type] ':=' Expr
Fun_Dec       ::= [semi] 'declare' Fun_Dec_Exprs [semi] 'begin'
Fun_Dec_Exprs ::= Fun_Dec_Expr {semi Fun_Dec_Expr}
Fun_Sig       ::= Function_Path [Type_Param_Clause] Param_Clauses
Att_Sig       ::= Function_Path [Type_Param_Clause] Att_Clauses
Op_Sig        ::= Op_Path [Type_Param_Clause] Param_Clauses
Function_Path ::= id
                | 'self' '.' id
                | id '.' id
Op_Path       ::= op_id
                | 'self' '.' op_id
Param_Clauses ::= {Param_Clause} ['(' 'implicit' Params ')']
Att_Clauses   ::= Param_Clause1 {Param_Clause}
                  ['(' 'implicit' Params ')']
Param_Clause  ::= '(' [Params] ')'
Param_Clause1 ::= '(' Param ')'
Params        ::= Param {',' Param}
Param         ::= {Annotation} [Param_Extra] id [id]
                  [':' Param_Type] ['=' Expr]
                | Pattern2
Param_Extra   ::= ['lazy' | 'eager'] [M_Modifier] [Param_Rw]
                  ['*' | '**' | '&' | '~']
Param_Rw      ::= 'val' | 'var' | 'ref'
Param_Type    ::= ['=>' | '=>>'] (['optional'] Type | id | '_')
```

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

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

The type of the function body is expected to conform (§5.10.2) 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.

6.10.1 Curried Function Definitions

Multiple parameter clauses render curried functions.

Type parameters of the whole function are linked to each value parameter clause. If local type inference selects a type argument that does not appear in a previous value parameter clause, it is still used in the following value parameter clauses where it does appear.

6.10.2 Function Parameters

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

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

Value parameters within a single parameter list (a parameter list is a parameter clause) are divided into three virtual sections, each of which may be empty:

1. Positional parameters (§6.10.6): mandatory parameters (§6.10.6.1), optional parameters (§6.10.6.2), variadic parameters (§6.10.6.3).
2. Purely named parameters (§6.10.7).
3. Block capturing parameter.

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

1. Positional parameters:
   (a) n mandatory positional parameters (§6.10.6): x: T, where $n \geq 0$.
   (b) n optional positional parameters (§6.10.6.2): x: T := e, where $n \geq 0$.
   (c) n variadic parameters (§6.10.6.3): $\star x$: T, where $0 \leq n \leq 1$.
   (d) n post mandatory positional parameters (§6.10.6): x: T, where $n \geq 0$.
2. Purely named parameters:
   - n named parameters (§6.10.7): ~x: T and/or ~x: T := e, where $n \geq 0$.
   - n capturing named parameter (§6.10.7): $\star\star x$: T, where $0 \leq n \leq 1$.
3. n captured block parameters (§6.10.8): &x: T, where $0 \leq n \leq 1$.

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

`default$`n

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

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

Example 6.10.1 In the method

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

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

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

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

6.10.3 External & Internal Parameter Names

Syntax:

```
Param ::= ... id [id] ...
```

The *external parameter name* is the one which may be used in function applications (§9.3.5) to specify the target parameter. The *internal parameter name* is the one that the function body uses. If the parameter name appears as "a b", then a is the external parameter name and b is the internal parameter name. If the parameter name appears as a, it is both the external and the internal parameter name. If the parameter name is prefixed with a tilde "~", then such a parameter may only be specified by its external name in function applications, and is said to be a purely named parameter (§6.10.7).

If a function applies itself as a part of its own definition, then again, the external parameter name is effective, never the internal parameter name.

[7] Without any explicit arguments.

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

Each internal parameter name may appear only once across every parameter list. Each external parameter name may only appear once in each parameter list, but may generally appear across multiple parameter lists multiple times, provided that its internal parameter name does not.

If an external parameter name is specified to be just an underscore "_" (the only acceptable form of such parameter name is then "_ b", where b is the internal parameter name), then no named argument will ever correspond to it. It is an error if this is combined with a purely named parameter, since there would be no way to create a corresponding argument to it. The other implication of this is that such external parameter names may only be used as the first parameters in each parameter list, never appearing after parameters with external names, and also arguments expressions must include as many positional arguments as there are parameters without external name for the function to be applicable. Such parameters are effectively *purely positional*.

6.10.4 Parameter Evaluation Strategies

Note: This section applies (from the other side of the wall) to function applications (§9.3.5) as well, but it's pointless to have it duplicated over there.

Gear utilizes five parameter (resp. argument) evaluation strategies. Every strategy defers evaluation of argument values until the function application is resolved, using only arguments' expected types – see (§9.9.3).

Call-by-value, strict. Also known as *call-by-object*, *call-by-object-sharing* or *call-by-sharing*, is the default evaluation strategy, applied to all parameters, unless otherwise specified. Such arguments are evaluated prior function invocation and are not re-evaluated. This includes explicit parameter values (§6.10.5).

Call-by-reference, strict. Also known as *pass-by-reference*, the function can modify the original argument variable. Those are only indirectly supported in Gear, by usage of wrapper objects. Those are described in (§6.10.4.1).

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 (§6.10.4.2).

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 (§6.10.4.3).

Call-by-future, non-deterministic. This is a rather experimental feature of Gear, depending on whether the Gear VM is capable of parallelization and whether there are defined standard tools for "system" parallelization. These are described in (§6.10.4.4).

6.10.4.1 By-Reference Parameters

Syntax:

```
Param_Rw ::= 'ref'
```

The name of a parameter may be prefixed with a "**ref**" keyword, in which case the parameter's type T is boxed into `Reference_Cell[`T`]`. See (§9.2.12) for more details on reference cells.

6.10.4.2 By-Name Parameters

Syntax:

```
Param_Extra ::= ['lazy'] ['val']
                ['*' | '~']
Param_Type  ::= '=>' (Type | id | '_')
```

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

The by-name modifier is disallowed for implicit parameters (§10.2). The by-name modifier implies **`val`** parameter and is disallowed for **`var`** parameters.

A by-name parameter bound to a wildcard type "_" matches any type of by-name argument.

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

6.10.4.3 By-Need Parameters

Syntax:

```
Param_Extra ::= 'lazy' [M_Modifier] [Param_Rw]
                ['*' | '~']
```

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

The by-need modifier is disallowed for implicit parameters (§10.2). The by-need modifier implies **`val`** parameter and is allowed for **`var`** parameters.

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

If a parameter is prefixed with **eager** instead, it can't be prefixed with **lazy**, or be by-need, by-name or by-future.

6.10.4.4 By-Future Parameters

Syntax:

```
Param_Extra ::= ['val'] ['*' | '~']
Param_Type  ::= '=>>' (Type | id | '_')
```

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

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

The by-future modifier is disallowed for implicit parameters (§10.2). The by-future modifier implies **val** parameter and is disallowed for **var** parameters.

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

By-future parameter prefix is omitted from examples of valid parameter definitions, but it's allowed occurrence is the same as of by-name parameters (§6.10.4.2).

6.10.5 Explicit Parameters

Syntax:

```
Param ::= Literal
        | Pattern2
```

The parameter may be specified by its literal value. Such parameters may only appear where positional mandatory parameters (§6.10.6.1) may appear. The type of the parameter is the inferred type of the literal value. Methods with explicit parameters are considered more specific during overloading resolution to methods with the same parameter types (§9.3.5).

The recommendation for usage of these parameters are:

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

Example 6.10.2 Sample methods that use explicit parameters:

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

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

The parameter's explicit value may also be specified as a pattern, then the type of the pattern is being contributed from the pattern as defined for each pattern type (§11). The pattern may or may not bind variables, which are then treated as parameters to the function, eagerly evaluated. Note that parameters in a single parameter list that are not explicit render together a tuple extracting pattern, thus explicit parameters using patterns are a *generalization of "regular" parameters*.

6.10.6 Positional Parameters

Positional parameters are identified by their ordered position within their section. Each such parameter has an external name that can be used to pass an argument into it by its name. The external name might not be the same as the internal name of the parameter: if there are two identifiers for the parameter name, the first one is external, the second one is internal. If there is one identifier for the parameter name, it is both external and internal. An internal name is the name under which the parameter is made available inside the function body, external name is the name under which an argument may be passed into the parameter. It is an error if the function body references a parameter by it's external name, if its internal name is different.

Positional parameters can optionally contain default values. Positional parameters with default values are *optional parameters* (the other without default values are *required parameters*), and may appear only in the middle of the parameters section. During application, they are picked from left to right, and are omitted in application from right to left and assigned default values from left to right.

Example 6.10.3 Positional mandatory parameters vs. optional parameters. Consider the following function declaration:

```
def f (a, b := 0, c := 0, d := 0, e): Unit end
```

Then, a minimal number of passed arguments is 2, one for each positional mandatory parameter. If 3 arguments are passed, the second is assigned to `b`, and `c` and `d` are assigned with their default values. If 4 arguments are passed, the second is assigned to `b` and the third is assigned to `c`. If 5 arguments are passed, the second is assigned to `b`, the third is assigned to `c` and the fourth is assigned to `d`. If a parameter is not assigned with an argument, then **`undefined`** is assigned to it.

Positional parameters may also contain a special parameter that captures all extra arguments: *variadic parameter*. It's only legal position is right before the last positional mandatory parameter, and after any optional parameters, due to the order in which arguments are assigned to parameters.

6.10.6.1 Mandatory Parameters

Positional mandatory parameters are of the forms:

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

Positional mandatory parameters may not have any modifiers, except for by-name (§6.10.4.2).

6.10.6.2 Optional Parameters

Optional parameters are of the forms:

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

Optional parameters may not have any modifiers, except for by-name modifier (§6.10.4.2). Optional parameters have a *default value expressions* and may appear between positional parameters, being followed by any number of positional parameters (including no more positional parameters at all), or being followed by repeated parameters and then positional parameters (§9.3.5.5). Optional parameters are disallowed for repeated parameters.

Optional parameters add the annotation (§14) `@[optional_param]` to the corresponding parameter type of the function trait.

If a parameter has a nullable type T (either by being a nullable type, or allowing otherwise `Nothing` with a union type), and appears where an optional parameter may appear, it is taken as an optional parameter with default value of **`undefined`** implicitly.

If the keyword **`optional`** is given before the parameter type, then the parameter type T is wrapped in type `Option[`T`]`, and instead of **`undefined`**, a missing argument is given as `None`, and a present argument x is given wrapped in `Some(`x`)`.

6.10.6.3 Variadic Parameters

Variadic parameters are of the forms:

```
*x: T
```

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

Between optional parameters and the tailing positional parameters may be a value parameter prefixed by "`*`", e.g. `(..., *x: T)`. The type of such a *variadic parameter* inside the method is then a list type `Sequence[T]`. Methods with variadic parameters take a variable number of arguments of type T between the optional parameters block and the last positional parameters block, including no arguments at all (an empty list is then its value). If the type T is defined by a variadic type parameter, then the type of the parameter inside the method is T (which is a tuple type – §5.3.7).

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

Variadic parameter may be passed in a function application by name iff the type of the named argument is compatible with `Sequence[T]`, or the variadic type parameter T, if it is variadic.

Example 6.10.4 The following method definition computes the sum of the squares of a variable number of integer arguments.

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

  result
end
```

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

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

Furthermore, assume the definition:

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

The following application of the method `sum` is not resolved:[9]

```
sum xs (* Error: method match not found, wrong arguments *)
```

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

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

```
sum *xs
```

6.10.7 Purely Named Parameters

Purely named parameters are of the forms:

```
~x: T
var ~x: T
val ~x: T
~x: T := e
var ~x: T := e
val ~x: T := e
~x: => T
val ~x: => T
~x: => T := e
val ~x: => T := e
~x: optional T := e
var ~x: optional T := e
val ~x: optional T := e
~x: => optional T := e
val ~x: => optional T := e
```

Capturing named parameter are of the form:

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

Purely named parameters are parameters that may only be assigned arguments using their external name, never by their position, which may be arbitrarily reordered by the language.

Named parameters are a way of allowing users of the method to write down arguments in any order, provided that their name is given at function application (§9.3.5 & §9.3.5.5). Named parameters may have a default value expression. Named parameters are disallowed for variadic parameters. Named parameters inside the method are then accessible the same way as a positional parameters.

Purely named parameters have the same definition of external and internal parameter names as positional parameters: if there are two identifiers for the purely named parameter name, then the first one is the external and the second one is the internal parameter name. If there is only one identifier, it is both the external and internal parameter name.

Purely named parameters do not impose any rules on the position where each of their kind may appear, exactly because their order of appearance is insignificant. Therefore, purely named parameters with default values may appear anywhere, just like the capturing named parameter (which may appear exactly once, or not at all).

Capturing named parameter is capturing any other applied named arguments that were not captured by their explicit declaration (§9.3.5.5). It is declared after the section of purely named parameters, prefixed by "`**`", e.g. `(..., **`x`: `T`)`. The type of such a captured named parameter inside the method is then a dictionary type `Dictionary[Symbol, `T`]`. Methods with capturing named parameter take a variable number of named arguments of type T mixed with other named arguments and before the captured block parameter. capturing named parameter are disallowed for repeated parameters and by-name parameters.

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

If a capturing named parameter is supposed to capture arguments of multiple types, one can use union types (§8.2). This does not ensure any particular relation between the argument's key and value's type other than that provided by the union though.

If a purely named parameter has a nullable type T (either by being a nullable type, or allowing otherwise `Nothing` with a union type), it is taken as having default value of **`nil`** implicitly.

6.10.8 Captured Block Parameter

Captured block parameters are of the forms:

`&`x
`&`x`: `T

Captured block parameter is a way to capture an applied block that is otherwise passed in implicitly as a function into **`yield`** expressions. The forms of captured block parameters explicitly denote the case without the block's function type, since block parameters receive any arguments and those missing are implicitly set to **`nil`**. The function type of the block may be used to further constrain the applied block argument, but is not used during method resolution (§9.3.5). The captured block parameter may be used also to capture function arguments, e.g. anonymous functions (§9.2.18), then the type is used during method resolution.

It is an error if a block parameter type T is provided and it is not a function type (§5.3.11). It is also an error if the applied block argument does not accept the arguments declared by the type T, or if the block would not return a value conforming to the result type required by the type T. The applied block argument may accept more arguments than required by T, however, these will be set implicitly to **`nil`**. Also, the applied block argument may itself require less constrained parameter types, in which case the arguments applied to it must (and will) always conform (§5.10.2) to the block's parameter requirements. Whether the function type T has a result type or not is irrelevant.

If a block parameter type T is given, then the applied block argument must accept parameters, such that the parameter constrains declared by the function type T conform to the parameter constrains declared by the applied block: the parameters of the applied block must be the same or

less restrictive than those declared by T – they must be pairwise contravariant or invariant, never covariant (§6.9).

If a function has multiple parameter lists, the captured block parameter may only appear in the last one, unless the last one is an implicit parameters list, in which case the last allowed parameter list for a captured block parameter to appear in is the one directly preceding the implicit parameters list.

6.10.9 Method Signature

Two methods M and N have the same signature, if they have the same name, the same type parameters (if any), the same parameters with equivalent types, and equivalent 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[10] 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.

6.11 Method Types Inference

Parameter Type Inference. Functions that are members of a class C may define parameters without type annotations. The types of such parameters are inferred as follows. Say, a method m in a class C has a parameter p which does not have a type annotation. We first determine methods m' in C that might be overridden (§7.1.8) by m, assuming that appropriate types are assigned to all parameters of m whose types are missing. If there is exactly one such method, the type of the parameter corresponding to p in that method—seen as a member of C—is assigned to p. It is an error if there are several such overridden methods m'. If there is none[11] (m does not override any m' known at compile-time), then the parameters are inferred to be of type `Any`.

Example 6.11.1 Assume the following definitions:

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

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

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

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

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

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

Example 6.11.2 Assume the following definitions:

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

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

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

6.12 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 (§9.9.1) 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).

6.13 Function & Method Preference Declarations

Syntax:

```
Dcl ::= 'def' 'prefer' Preference_Id 'with' Preference_Dcl
        'to' Preference_Dcls 'end' ['def']
      | 'prefer' 'message' Preference_Id 'with' Preference_Dcl
        'to' Preference_Dcls 'end' ['message']
      | 'prefer' 'function' Preference_Id 'with' Preference_Dcl
        'to' Preference_Dcls 'end' ['function']
      | 'prefer' 'operator' Preference_Op_Id 'with' Preference_Dcl
        'to' Preference_Dcls 'end' ['operator']

Preference_Id      ::= id
                     | '(' id {',' id} ')'
                     | regular_expression_literal
Preference_Op_Id   ::= op_id
                     | '(' op_id {',' op_id} ')'
Preference_Dcl     ::= Pref_Args_Dcl
                     | Pref_Result_Dcl
                     | Pref_Args_Dcl ('and' | 'or') Pref_Result_Dcl
Preference_Dcls    ::= Preference_Dcl {',' Preference_Dcl}
Pref_Args_Dcl      ::= 'arguments' Pref_Args_Sections
Pref_Args_Sections ::= Pref_Args_Section {Pref_Args_Section}
Pref_Args_Section  ::= '(' Pref_Arg_Dcl {',' Pref_Arg_Dcl} ')'
Pref_Arg_Dcl       ::= ['*' | '**' | '&'] [id ':'] Type
                     | Literal
                     | Stable_Id
Pref_Result_Dcl    ::= 'returning' [Type]
```

Preference declaration is a simple tool to resolve ambiguities in function and method overloading resolution (§9.9.3).

Preference declaration declares that a function or method of a given name, multiple given names, or a name matching a regular expression, which passes the filter of arguments and/or result type, is more specific than the overloaded alternative that passes the filter of arguments and/or result type.

Arguments filter does not include implementation details, such as evaluation strategy or external/internal parameter name distinction, which has no influence on overriding. It also ignores optional parameters and thus optional arguments. It says that if the overloaded alternative, stripped of the ignored implementation details, has equivalent parameters as the given filter, it passes the filter.

Result type filter is passed by overloaded alternative, if the result types are equivalent.

A preference declaration can use arguments filter, result type filter, or both. It is an error if a

preference declaration prefers two or more alternatives to each other.

If there are multiple preference declarations with equivalent filters of the preferred alternative, then the alternatives to which this alternative is preferred are unified, i.e., their order is insignificant, and the alternative is still preferred just once to the other alternatives.

Note. Preference declarations take into account all parameter lists, not individual parameter lists, of each overloaded alternative. A preference declaration may also appear in a function or method body, to resolve ambiguities of local functions.

Inheritance. Preference declarations are inherited to subtypes, without regard of visibility. This does not affect overloading resolution, which only counts with alternatives that are visible in the scope.

6.14 Use Clauses

Syntax:

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

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*[12] 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 (§7.2). The most general form of an import expression is a list of *import selectors*

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

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

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

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[13], 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.

Example 6.14.1 Consider the object definition:

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

Then the block

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

is equivalent to the block

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

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

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

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.

Chapter 7

Classes & Objects

Contents

Syntax:

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

Classes (§7.3) & objects (§7.4) are both defined in terms of *templates*.

7.1 Templates

Syntax:

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

A template defines the type signature, behaviour and initial state of a trait, class of objects or of a single object. Templates for part of instance creation expressions (constructors, see §7.1.3 & §7.3.2), 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 (§7.1.6).

Each trait reference mt_i that is not prepended must denote a trait (§7.3.6). 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

[1]Including protocols, which are also traits.

new supertype must have at least one constructor that does not take parameters and is accessible to the subclass (§7.2).

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

The *least proper supertype* of a template is the class type or compound type (§5.3.9) consisting of all its parent class types.

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

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

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

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

Example 7.1.1 Consider the following class definitions:

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

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

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

[2]The classes are open in Gear, a single class may have its statements spread across multiple source files (§7.1.1).

7.1.1 Open Templates

Unlike in Java, Scala or any other language that does not feature open classes, and similar to Ruby's open classes or C#'s partial classes, Gear 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 a parent constructor invocation). However, it is possible for the auxiliary templates to define additional traits applied or prepended to the template.

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

- Eager loaded auxiliary template, using the construct `include 'path/to/file'`. This way the auxiliary template is loaded along with the leading template.
- Autoload (§7.1.2) of the auxiliary template.
- On-demand load of the auxiliary template via independent `VM.load 'path/to/file'`, after the leading template is defined. Gear

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

```
@[open-template :main]
@[open-template :auxiliary]
```

7.1.2 Autoloading

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

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

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

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

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

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

7.1.3 Constructor Invocations

Syntax:

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

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

A type argument list can be only given if the class c takes type parameters. If no explicit arguments are given, an empty list `()` is implicitly supplied 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.

7.1.4 Metaclasses & Eigenclasses

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

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

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

In Gear, 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. Gear 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].

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

Table 7.1: 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 Gear, or directly using the `eigenclass` method. This method can possibly trigger creation of an eigenclass, if the receiver of the `eigenclass` message did not previously have its own (singleton) eigenclass (because it was a terminal object whose eigenclass was a regular class, or the reciever was an eigenclass itself).

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

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

Syntax:

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

Example 7.1.2 The following code shows how metaclasses are nested in case of `Object` type. Don't try this at home though.

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

Example 7.1.3 The following code shows what **`self`** references when inside of a class definition, but outside of any defined methods.

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

Example 7.1.4 Direct access to the eigenclass of any object, here a class' eigenclass:

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

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

Example 7.1.5 Alternative direct access to the eigenclass of any object, here a class' eigenclass:

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

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

Example 7.1.6 Indirect access to the eigenclass using a singleton method definition:

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

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

Example 7.1.7 Indirect access to the eigenclass using a class object definition:

```
class D
```

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

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

Example 7.1.8 Alternative indirect access to the eigenclass using a class object definition:

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

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

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.

7.1.5 Class Linearization

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

Definition 7.1.9 Let base classes of a class C be the list of every superclass of C with every trait that these classes include and/or prepend and every protocol that these classes implement. Let C be a class with base classes C_1 `with` C_2 `with` ... `with` C_n. The *linearization* of C, $\mathscr{L}(C)$ is defined as follows:

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

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

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

Example 7.1.10 Consider the following class definitions.[4]

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

```
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 (§9.3.5).

7.1.6 Inheritance Trees & Include Classes

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

Every class has a link to its superclass. In fact, the link is made up of an include class structure, which itself holds an actual link to the superclass. That superclass has its own link to its superclass and this chain goes forever until the `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 7.1.11 A sample trait schema:

[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.

```
class C extends D with Some_Trait
end

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

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

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

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

Nested includes. When a trait M itself includes one or more traits $M_1, \ldots, M_n$, then these included traits are inserted between the included trait M and the superclass, unless they are already respectively included by any of the superclasses. If M with included $M_1, \ldots, M_n$ is prepended to C, then $M_1, \ldots, M_n$ are inserted before the superclass of C, if not already included in any of the superclasses. It is an error if nested includes form a dependency cycle: any *auto-included* trait must not require to include a trait that triggered the auto-include. The order in which traits are included in another trait may change when included in a class, i.e. if the class includes two traits A and B that themselves include the same two traits D and E in reverse order (A includes 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 7.1.12 Take the following trait and class definitions, where S is the superclass of the class C:

```
trait D end
trait E end

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

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

Then traits are auto-included in the following order:

- `C` → `[S]`
  First, the superclass is added.

- `C` → `[A]` → `[S]`
  Then, trait A is included, unless already included in S.

- `C` → `[A]` → `[E]` → `[S]`
  Including of trait A triggers auto-include of traits included in A. Start with the first one in chain of A: E.

- `C` → `[A]` → `[E]` → `[D]` → `[S]`
  Then, A has D in its chain.

- `C` → `[B]` → `[A]` → `[E]` → `[D]` → `[S]`
  Finally include B. B triggers auto-include, but both of its included traits are already included in the chain, so nothing more happens.

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

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

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

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

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

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

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

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

[7]Trait cycle dependency detected.

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

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

Traits & Metaclasses. Since traits may contain "class" methods as well as instance methods, all the operations with include classes are mirrored on the respective metaclasses.[10]

7.1.7 Class Members

A class C defined by a template C_1 `with` ... `with` C_n { *stats* } can define members in its statement sequence *stats* and can inherit members from all parent classes. Gear 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, Gear uses the following definition of *matching* on members:

Definition 7.1.13 A member definition M *matches* a member definition M', if M and M' bind the same name, and one of the following conditions holds.

1. Neither M nor M' is a method definition.

2. M is override-matching M' (§6.10.9).

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

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

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

[9]Trait composition design flaw detected.

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

This definition also determines the overriding relationship between matching members of a class C and its parents (§7.1.8). First, a concrete definition always overrides an abstract definition. Second, for definitions M and M', which are both concrete or both abstract, M overrides M' if M appears in a class that precedes (in the linearization of C) the class in which M' is defined.

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

7.1.8 Overriding

A member M of a class C that matches a member M' of a base class of C is said to *override* that member. In this case the binding of the overriding member M must conform (§5.10.2) 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 (§7.2) in C, then M should be labeled **`abstract override`**.
- If M and M' are both concrete value definitions, then either none of them is marked **`lazy`**, or both must be marked **`lazy`**.

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

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

7.1.9 Inheritance Closure

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

- If T is in $\mathscr{S}$, then every type T' which forms syntactically a part of T is also in $\mathscr{S}$.
- If T is a class type in $\mathscr{S}$, then all parents of T (§7.1) are also in $\mathscr{S}$.

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

7.1.10 Early Definitions

Syntax:

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

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

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

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

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

7.2 Modifiers

Syntax:

```
Modifier         ::= Local_Modifier
                   | Access_Modifier
                   | 'override' [Class_Qualifier]
Local_Modifier   ::= 'implicit'
                   | 'lazy'
```

```
                  | 'final'
                  | 'sealed'
                  | 'abstract'
Access_Modifier  ::= 'public'
                  | ('protected' | 'private') [Access_Qualifier]
Access_Qualifier ::= '[' (id | 'self') ']'
Mutability       ::= 'mutable' | 'immutable'
```

Access modifiers may appear in two forms:

Accessibility flag modifier. Such a modifier appears in a template (§7.1) or a module definition (§13.1.1) alone on a single line. All subsequent members in the same class-level block than have accessibility of this modifier applied to them, if allowed to (does not apply to destructors). Only access modifiers can be used this way.

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

Example 7.2.1 An example of an accessibility flag modifier:

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

Example 7.2.2 An example of directly applied modifier:

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

By default[11], the **public** access modifier affects every member of the class type, except for instance variables and class instance variables, which are object-private (**private[self]**).

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

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

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

- 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 (§7.4) and any member of the directly enclosing class, including inner classes. They are not inherited by subclasses and they may not override definitions in parent classes.

  The modifier may be *qualified* with an identifier C (e.g. **private[**C**]**) that must denote a class or a module enclosing the declaration or definition. Members labeled with such a modifier are accessible respectively only from code inside the module C or only from code inside the class C and the class object C (§7.4).

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

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

  Class-private and object-private members must not be **abstract**, since there is no way to provide a concrete implementation for them, as private members are not inherited. Moreover, modifiers **protected** & **public** can not be applied to them (that would be a contradiction[12]), and the modifier **override** can not be applied to them as well[13].

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

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

[13] Otherwise, if a private member could override an inherited member, that would mean there is an inherited

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

  – the defining class

  – all classes that have the defining class as a base class

  – all class objects of any of those classes

  A **protected** access modifier can be qualified with an identifier C (e.g. **protected**[C]) that must denote a class or module enclosing the definition. Members labeled with such a modifier are *also*[14] accessible respectively from all code inside the module C or from all code inside the class C and its class object C (§7.4).

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

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

  – 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 Gear is virtual, so Gear 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;

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

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

it is *concrete* if a full definition is given. This behavior can be turned off only in tests, if needed, and is implicitly turned on.

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

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

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

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

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

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

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

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

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

- The **mutable** mutability modifier guarantees that all member values of the type that it is applied to will not be immutable, hence not frozen. It transitively applies to all subtypes, and is disallowed when any of the parents of the type is declared to be immutable. It is an error to freeze values that are declared mutable.

- The **immutable** mutability modifier guarantees that all member values of the type that it is applied to will be immutable, hence frozen. It transitively applies to all subtypes, and is disallowed when any of the parents of the type is declared to be mutable. When applied to classes, it does so by implicitly freezing all values right before all their constructors are finished[15].

- 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 7.2.3 The following code illustrates the use of qualified and unqualified private:

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

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

A rule for access modifiers in scope of overriding: Any overriding member may be defined with the same access modifier, or with a less restrictive access modifier. No overriding member can

[15]Freezing is re-entrant and irreversible. If a constructor freezes a value explicitly, nothing is wrong.

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]`** $\longrightarrow$ `{ protected, protected[`C`], public }`
- **`protected`** $\longrightarrow$ `{ protected[`C`], public }`
- **`protected[`**C**`]`** $\longrightarrow$ `{ protected[`D`], public }`

  This is only for the case where C is accessible from within D.
- **`protected[`**C**`]`** $\longrightarrow$ `{ public }`
- **`public`** $\longrightarrow$ `{ public }`

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

7.3 Class Definitions

Syntax:

```
Tmpl_Def            ::= [Mutability] 'class' Class_Def
Class_Def           ::= id [Type_Param_Clause]
                        [Class_Param_Clauses] Class_Tmpl_Env
Class_Param_Clauses ::= {Annotation} [Access_Modifier]
                        {Class_Param_Clause}
                        ['(' 'implicit' Class_Params ')']
Class_Param_Clause  ::= '(' [Class_Params] ')'
```

```
Class_Params        ::= Class_Param {',' Class_Param}
Class_Param         ::= Param
Class_Tmpl_Env      ::= Tmpl_Env_With_Pars ['class']
                      | Tmpl_Env_Brackets
                      | Tmpl_Env_No_Pars ['class']
Tmpl_Env_With_Pars  ::= 'extends' [Early_Defs] Class_Parents
                        semi ['begin'] [Template_Body] 'end'
Tmpl_Env_Brackets   ::= ['extends' [Early_Defs]]
                        '{' [Template_Body] '}'
Tmpl_Env_No_Pars    ::= ['begin' [Template_Body]] 'end'
```

A class definition defines the type signature, behavior and initial state of a class of objects (the instances of the defined class) and of the class object, which is the class instance itself, with behavior defined in its metaclass (§7.1.4).

The most general form of a class definition is

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

for $n \geq 0$.

Here,

> c is the name of the class to be defined.
>
> *tps* is a non-empty list of type parameters of the class being defined. The scope of a type parameter is the whole class definition including the type parameter section itself. It is an error to define two type parameters with the same name. The type parameter section [*tps*] may be omitted. A class with a type parameter section is called *polymorphic*, a class without a type parameter section is called otherwise *monomorphic*. Type arguments are reified in Gear, i.e. type arguments are preserved in runtime[16], generating a new concrete subtype of the original generic class. This ensures type safety in a dynamic environment of Gear in runtime.
>
> *as* is a possibly empty sequence of annotations (§14). If any annotations are given at this point, they apply to the primary constructor of the class.
>
> m is an access modifier (§7.2), such as **private** or **protected** (**public** is implied otherwise), possibly with a qualification. If such an access modifier is given, it applies to the primary constructor of the class.
>
> $(ps_1)\ldots(ps_n)$ are formal value parameter clauses for the *primary constructor* of the class. The scope of a formal value parameter includes all subsequent parameter sections and the

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

template t. However, a formal value parameter may not form part of the types of any of the parent classes or members of the class template t – only parameters from *tps* may form part of the types of any of the parent classes or members of the class template t. It is illegal to define two formal value parameters with the same name. If no formal parameter sections are given, an empty parameter section `()` is implied.

If a formal parameter declaration x`:` T is preceded by a **`val`** or **`var`** keyword, an accessor (getter) definition (§6.2) for this parameter is implicitly added to the class. The getter introduces a value member x of class c that is defined as an alias of the parameter. Moreover, if the introducing keyword is **`var`**, a setter accessor x`_=(`e`)` is also implicitly added to the class. The formal parameter declaration may contain modifiers, which then carry over to the accessor definition(s). A formal parameter prefixed by **`val`** or **`var`** may not at the same time be a call-by-name parameter (§6.10.4.2).

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 (§7.1) of the form

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

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

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

Example 7.3.1 The following example illustrates **`val`** and **`var`** parameters of a class `C`:

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

Example 7.3.2 The following class can be created only from its class object.

```
class Sensitive private () {
  ...
}
object Sensitive {
```

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

7.3.1 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, *_]
```

7.3.2 Constructor & Destructor Definitions

Primary constructor. The primary constructor is a special function, more special in its syntax than any other function in Gear. 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 (§7.3). Technically, a primary constructor is happy being left with nothing but the parameter definitions, since these are always automatically mapped to instance values[17] and also implicitly adds accessor methods to the class with appropriate visibility, defined explicitly by accessor modifiers (§7.2) or implicitly as **`public`**.

Explicit primary constructor fields. These are expressions inside the class template (§7.1) 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). **Syntax:**

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

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

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 Gear have multiple entry points, but with great power comes great responsibility: user must ensure that every constructor path defines all necessary members. It is an error if a constructor invocation leads to an instance with abstract members. If a constructor does not explicitly invoke another constructor, an invocation of the implicit constructor is inserted implicitly as the first invocation in the constructor.

Syntax:

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

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

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 (either implicit one, which is **`nil`**, or explicit ones used in their definitions), 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[18]. 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

[18]This is a relaxation from Swift by Apple, which only allows to call a designated constructor of the immediate superclass.

the dynamic code. Also note that the constructor of the superclass may be dealing with an incomplete object[19], 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.

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 (§7.2). 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. Runtime keeps track of all declared instance variables and their defined-ness during object construction.[20]

[19]As the constructors may further edit the new object later.

[20]This is again relaxation from Swift by Apple. The object might be fully initialized earlier, or later.

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.

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.

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 7.3.3 An example of a designated constructor of class C.

```
class C
  constructor (param)
    val @resource := param
    super()
  end constructor
end class
```

Example 7.3.4 An example of a pair of constructors of class C.

```
class C
  constructor (param)
    val @resource := param
    super()
  end constructor
  convenience constructor := self(42)
end class
```

Constant instance variables. Any constructor may modify a constant instance variable during its evaluation, even multiple times. This behavior 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 7.3.5 An example of an explicit destructor of class C.

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

7.3.3 Clone Constructor Definitions

Syntax:

```
CCtor_Def     ::= ['convenience'] 'clone' CCtor_Fun_Def
CCtor_Fun_Def ::= [Type_Param_Clause] [Param_Clauses]
                    (':=' Constr_Expr | Clone_Block)
Clone_Block   ::= [semi Constr_Block3] 'end' ['clone']
                | Constr_Block2
```

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

Clone constructor implicitly returns the new cloned object, unless returning explicitly a different object (which would possibly invoke its regular constructors). The original object is available

with the **self[origin]**, **this[origin]** and **super[origin]** constructs (the latter being only useful as prefix in function applications), the new cloned object is available with the **self[cloned]**, **this[cloned]** and **super[cloned]** constructs. The **self[cloned]** construct is only legal in a body of the clone constructor. Clone constructors are invoked in the context of the original object, but the original object still has to be accessed with the defined construct.

Clone constructors have the ability to modify immutable instance variables, but only of the new cloned object (**self[cloned]**) – the original object is already constructed.

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

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

The `Object` class has a default root clone constructor, which carries the modifier **protected** (§7.2). Therefore, any custom clone constructor has to be explicitly protected in a less restrictive way, or public, in order to allow direct object cloning from outside of the class that defines it.

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

Designated clone constructor. A designated clone constructor can only invoke inherited clone constructors from superclass.

Convenience clone constructor. A convenience clone constructor is a clone constructor marked with the keyword **convenience**. They can only invoke clone constructors from the same class.

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

Example 7.3.6 An example of a default clone constructor of class C, performing a deep copy.

```
class C
  clone
    (* super is invoked implicitly here *)
```

```
    self[cloned].@resource := self[origin].@resource.clone
  end constructor
end class
```

7.3.4 Upgrade Constructor Definitions

Syntax:

```
UCtor_Def     ::= ['convenience'] 'upgrade' 'constructor'
                  UCtor_Fun_Def
UCtor_Fun_Def ::= '(' id [':' 'self' '.' 'type'] ')'
                  (':=' Constr_Expr | Constr_Block)
```

7.3.5 Case Classes

Syntax:

```
Tmpl_Def ::= [Mutability] '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 (§7.3).

A case class definition of c[*tps*](ps_1)...(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 (§11.1.14), and are defined as follows:

```
object c {
  def apply[tps](ps_1)...(ps_n): c[tps] :=
      c[Ts].new(xs_1)...(xs_n)
  def unapply[tps](x: c[tps]?): Option[e] :=
      unless x = nil
        Some(x.xs_11, ..., x.xs_1k)
      else
        None
      end
}
```

Here, *Ts* stands for the vector of types defined in the type parameter section *tps*, each xs_i denotes the parameter names of the parameter section ps_i, $xs_{11}, \ldots, xs_{1k}$ denote the names of all parameters in the first parameter section xs_1, and e denotes the type that `Option` is parameterized with. If a type parameter section is missing in the class c, it is also missing in the `apply` and `unapply` methods. The definition of `apply` is missing also if the class c is marked **`abstract`**.[21]

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 (§5.3.7). 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 (§6.10.7), 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}
  ...
  self[cloned].@xs_{1,n} := xs_{1,n}
end clone
```

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

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.

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

Method `equals: (Object) ↦ Boolean` (equivalent to operator "=") is structural equality, where two instances are equal if they both belong to the case class in question and they have equal (with respect to `equals`) elements.

Method `hash_code: () ↦ Number.Integer_Unsigned` computes a hash code.

Method `to_string: () ↦ String` returns a string representation which contains the name of the case class and its elements.

7.3.6 Traits

Syntax:

```
Tmpl_Def       ::= [Mutability] 'trait' Trait_Def
Trait_Def      ::= id [Type_Param_Clause]
                   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 (§9.3.3). 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 7.3.7 The following trait defines the property of being comparable to objects of some type. It contains an abstract operator `<` and default implementations of the other comparison operators `<=`, `>` and `>=`. Operators are methods, too. The trait also requires the self-type to be `T`.

```
trait Comparable[T <: Comparable[T]] {
  requires T
  operator < (that: T): Boolean end
  operator <=(that: T): Boolean := self < that or self = that
  operator > (that: T): Boolean := that < self
  operator >=(that: T): Boolean := that <= self
}
```

7.3.7 Refinements

There are two separate branches of refinements in Gear, both are quite similar, but used for different purposes. Refinements are kind of a trait (§7.3.6). 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 (§5.3.9).
>
> Second, refinements that are basically traits designed to be locally prepended to classes. The second branch is described here.

Syntax:

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

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

Refinements need to be "activated" in the scope for it to take effect (§9.3.4).

If a refinement refines a parameterized type, then the refinement only activates for this parameterized type (§5.3.6).

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`**.

7.3.8 Protocols

Syntax:

```
Tmpl_Def     ::= 'protocol' Pro_Def
Pro_Def      ::= id [Type_Param_Clause]
                 Pro_Tmpl_Env
Pro_Tmpl_Env ::= 'extends' Trait_Parents
                 semi [Template_Body] 'end' ['protocol']
               | ['extends'] '{' [Template_Body] '}'
               | ['begin' [Template_Body]] 'end' ['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 (§7.3.6) 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 (§7.1.10). If this is needed, again, use a trait instead.
- Protocols can make use of the **optional** and **requires** keywords as modifiers in their template (§9.8), 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.

7.3.9 Interfaces

Syntax:

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

Interfaces are filtered versions of classes and traits with only declarations and without early definitions or any definitions at all, except for auxiliary constructor definitions that become empty constructor declarations (without function bodies). Interfaces can be generated from classes or traits by simple transformations and manually edited as needed. Their only purpose is to be used in *module interfaces*, so that implementation is not distributed along, but only declarations in interfaces and protocols are.

7.4 Object Definitions

Syntax:

```
Object_Def      ::= id Object_Tmpl_Env
Object_Tmpl_Env ::= Tmpl_Env_With_Pars ['object']
                  | Tmpl_Env_Brackets
                  | Tmpl_Env_No_Pars ['object']
```

Object definitions define singleton instances. If no superclass is given, `Gear/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 `Gear/Language.Class[]`, without any type arguments[22] – it is sufficient to write down `Class`, if that name is imported to resolve to `Gear/Language.Class`, which it usually is. If the class definition is not connected to a class, then rules from compound types apply (§5.3.9), 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: Gear 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 (§7.1) 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 (§7.3.6) that are to be included in a class object of class C: the self type would have to be `Class[`C`]`, not just C.

7.4.1 Case Objects

Syntax:

[22] This is to prevent the automatic extension of `Gear/Language.Object`.

```
Tmpl_Def ::= [Mutability] 'case' 'object' Object_Def
```

Case objects are pretty much similar to case classes (§7.3.5). Case objects can not be class objects at the same time. Their template (§7.1) is the singleton object form of an object definition (§7.4).

A case object definition implicitly adds some methods to the object instance, making it suitable as an extractor object (§11.1.14), and are defined as follows:

```
object C {
  def unapply(x: C.type?): Boolean := (x == self)
}
```

Chapter 8

Type Definitions

Contents

8.1 Aspects

Syntax:

```
Tmpl_Def        ::= 'aspect' [Aspect_Clause] Aspect_Def
Aspect_Def      ::= id ['extends' Class_Parents] Aspect_Tmpl_Env
Aspect_Tmpl_Env ::= '{' Aspect_Body '}'
                  | 'begin' Aspect_Body 'end' ['aspect']
Aspect_Body     ::= Template_Stat
                  | Aspect_Stat
Aspect_Stat     ::= Advice_Def
                  | Pointcut_Def
Pointcut_Def    ::= 'pointcut' id
                    ['(' [id [':' Type]] {',' [id [':' Type]]} ')']
                    'is' Pointcut_Type 'end' ['pointcut']
Pointcut_Type   ::= 'invoke' (id | regular_expression_literal)
                  | 'get' (id | regular_expression_literal)
                  | 'set' (id | regular_expression_literal)
                  | 'handler' Type
                  | 'returning' [id [':' Type] | Type]
                  | 'throwing' [id [':' Type] | Type]
                  | 'raising' [id [':' Type] | Type]
                  | 'yielding' [id [':' Type] | Type]
                  | 'advice-execution'
                    [(id | regular_expression_literal)]
                  | 'self' (id | Type)
                  | 'target' (id | Type)
                  | 'arguments' '(' id [':' Type] | Type
                    {',' (id [':' Type] | Type)} ')'
                  | 'if' (Infix_Expr | '(' Infix_Expr ')')
                  | 'not' Pointcut_Ref
                  | Pointcut_Ref 'and' Pointcut_Ref
                  | Pointcut_Ref 'or' Pointcut_Ref
                  | '(' Pointcut_Ref ')'
                  | 'retain' [id [':' Type] | Type]
                  | 'release' [id [':' Type] | Type]
Pointcut_Ref    ::= id ['(' id {',' id} ')'] | Pointcut_Type
Advice_Def      ::= 'advice'
                    Advice_Spec (Pointcut_Ref | Pointcut_Type)
                    Block_Expr
Advice_Spec     ::= 'before'
                  | 'after' [('returning' | 'throwing' | 'raising'
                             | 'yielding') [id [':' Type] | Type]]
                  | 'around'
Aspect_Clause   ::= 'per-self' [Pointcut_Ref]
                  | 'per-target' [Pointcut_Ref]
```

Only for `Block_Expr` inside of `Advice_Def`:

```
Simple_Expr1    ::= 'joinpoint' ['.' Selection]
```

8.2 Union Cases

Syntax:

```
Const_Type_Def ::= id [Type_Param_Clause] 'is' Case_Union
Case_Union     ::= 'case' 'union' 'of'
                   id ['of' Type] {semi id ['of' Type]}
                   [semi Template_Body]
```

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.

Types contained in a union type of form **`case union`** t are given a case name, with the syntax form of a **`of`** T, where a is the case name. The case name can be used as extractor in pattern matching:

```
type Message is case union of
  Result of String
  Request of Integer * String
  def name := match self
    when Result(nm) then nm
    when Request(_, nm) then nm
  end match
end type
```

Such a union type may contain the optional template body, and is called *case union type*. A case union type is a simplified form for a series of case classes and case objects.

8.3 Enumeration Types

Syntax:

```
Const_Type_Def     ::= id Is ['bitfield']
                       'enum' '{' Enum_Field {semi Enum_Field} '}'
                       [Record_Extensions]
Enum_Field         ::= id ':=' scalar_literal
```

```
                    | id 'of' Named_Exprs
                    | id
```

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

An enum constant can be enhanced with custom parameters, which are then passed to the appropriate custom constructor. Enum definitions should only appear in four forms:

- Initialized with a scalar literal. 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 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. The signature of the enum must extend a scalar type, and is the only one that allows for the **`bitfield`** keyword to be used with it.

- 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.

8.4 Variant Types & Algebraic Data Types

Syntax:

```
Const_Type_Def ::= id [Type_Param_Clause] Is 'case' 'enum' [nl]
                   (Variant1 | Variant2)
                   [Record_Extensions]
Variant1       ::= Variant_Field {semi Variant_Field}
Variant2       ::= '{' Variant_Field {semi Variant_Field} '}'
Variant_Field  ::= id ['of' Types]
```

Variant types are similar to a simplified form for a series of case classes (§7.3.5) and case objects (§7.4.1), with shared interface and a common ancestor. Variant types are also an implementation of *algebraic data types*, and case classes and case object may be used to achieve similar properties as

well. Pattern matching is possible on variant types, as it would be done on case classes and case objects.

Unlike enumerations (§8.3), 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 (§11.1.13).

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.

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 8.4.1 An example of a variant type.

```
type B-tree[T] is case enum
{
  Empty
  Node of T * B-tree[T] * B-tree[T]
}
with def is_member? (x, b-tree) :=
  match b-tree
  when Empty then no
  when Node(y, left, right) then
    if x = y then yes
    elsif x < y then is_member? x, left
    else is_member? x, right
    end if
  end match
and def 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 if
  end match
end type
```

8.5 Record Types

Syntax:

```
Const_Type_Def    ::= Record_Name Is 'record' (Record1 | Record2)
Record1           ::= Record_Components [Record_Extensions]
Record2           ::= '{' Record_Components '}' [Record_Extensions]
Record_Name       ::= id [Type_Param_Clause]
Record_Components ::= Record_Component {semi Record_Component}
Record_Component  ::= ['val' | 'var'] id ':' (Type | id)
                    | 'case' id Record_Cases
                      {'when' id
                      ('then' | nl)
                      Record_Components} 'end' ['case']
Record_Cases      ::= semi Record_Case {semi Record_Case} 'end' ['case']
                    | '{' Record_Case {semi Record_Case} '}'
Record_Case       ::= 'when' ['.'] Stable_Id ['(' [Extractions] ')']
                      ('then' | semi) Record_Components
Record_Extensions ::= [nl] 'with' Record_Extension {'and' Record_Extension}
Record_Extension  ::= Tmpl_Member
                    | Tmpl_Ifc_Impl
                    | Tmpl_Ifc_Dcl
```

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.

8.6 Range, Floating & Fixed Point Subtype Definitions

Syntax:

```
Type_Def        ::= FP_Type_Def
                  | FP_Subtype_Def
                  | FP_Range_Def
FP_Type_Def     ::= id [Type_Param_Clause] (':=' | 'is')
                    (FiP_Type_Def | FlP_Type_Def)
FP_Subtype_Def  ::= id [Type_Param_Clause] (':=' | 'is')
                    Type
                    (FiP_Subtype_Def | FlP_Subtype_Def)
```

```
FP_Range_Def    ::= id [Type_Param_Clause] (':=' | 'is')
                    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' Infix_Expr
FP_Delta        ::= 'delta' Infix_Expr
FP_Range        ::= 'range' Infix_Expr
                    ('..' | '...' | '..<') Infix_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.[1]

[1] 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 8.6.1 Examples of floating point types and subtypes:

```
type Coefficient is digits 10 range -1.0 ..< 1.0
end type

type Mass is digits 7 range 0.0 ..< 1.0e+35
end type

(* a subtype with a smaller range *)
type Probability is Real range 0.0 ... 1.0
end type
```

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[2], 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 8.6.2 Examples of fixed point types and subtypes:

```
type Volt is delta 0.125 range 0.0 ..< 255.0
end type

type Fraction is delta Number.Fine_Delta range -1.0 ..< 1.0
end type

type Money is delta 0.01 digits 15
end type

type Salary is Money digits 10
end type
```

[2]Using the `Decimal` types.

Chapter 9

Expressions

Contents

Syntax:

```
Expr         ::= Value_Expr
               | Loop_Expr
               | Rescue_Expr
               | Raise_Expr
               | Throw_Expr
               | Catch_Expr
               | Return_Expr
               | Assign_Expr
               | Update_Expr
               | Yield_Expr
               | Infix_Expr
               | Annot_Expr
               | Cast_Expr
               | Use_Expr
               | Delayed_Expr
               | Ref_Expr
               | Jump_Expr
               | Anon_Fun
               | Anon_Class
               | Expr semi Expr
Value_Expr   ::= Simple_Expr
               | Cond_Expr
               | Match_Expr
               | New_Expr
               | Def
               | {Local_Modifier} Tmpl_Def
               | '(' Expr ')'
               | Quasiquote_Expr
               | Quote_Expr
               | Metaclass_Access
               | Workflow_Expr
               | Type_Expr
Type_Expr    ::= '(' Type ')'
               | Annot_Type - Nullable_Mod
Simple_Expr ::= Block_Expr
               | Simple_Expr1
               | Method_Expr
               | TT_Expr
Result_Expr ::= Anon_Params '->' Block
               | ['memoize'] Expr
               | ['memoize'] 'tailcall' Argument_Exprs
Exprs        ::= Expr {',' Expr}
```

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

discussed subsequently.

9.1 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 Gear is in fact yet another kind of an expression, and those expressions themselves always have a type and a value.

9.2 Data Expressions

9.2.1 Simple Constant Expressions

Syntax:

```
Simple_Expr1 ::= integer_literal
               | floating_point_literal
               | complex_literal
               | rational_literal
               | character_literal
               | boolean_literal
               | string_literal
               | symbol_literal
               | regular_expression_literal
```

Typing of literals is as described in (§3.5); their evaluation is immediate.

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

9.2.2 The Nil and Undefined Values

Syntax:

```
Simple_Expr1 ::= 'nil'
               | 'undefined'
```

The **nil** value is of type Nothing, and is thus compatible with every type that is nullable (§5.3.13).

The **undefined** value is of type Undefined, a subtype of Nothing, and is thus also compatible with every type that is nullable (§5.3.13).

The **nil** represents a "no object", and is itself represented by an object.

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** represents also "no object", being itself represented by an object, but also represents "undefined value", which could be any of the following:

- Undefined dynamically accessed instance variable or class object variable. This is a common case.
- Undefined values for keys in implementations of Map. This is also a common case, if the **nil** object is to be considered a legit value.
- Undefined type argument.
- Rarely, undefined value or variable. Could happen if there is an extreme error in a macro, or basically a bug in the language.

Defined values and variables may not be undefined by assigning them with the **undefined** object directly by user code, but it is important to note that this may happen on special occasions indirectly, most notably during hot code loading, e.g. if the definition of an instance variable goes missing from the new code version.

See (§5.12) for more details on "nothing" values.

9.2.3 Tuple Expressions

Syntax:

```
Simple_Expr1 ::= '(' [Named_Exprs] ')'
Named_Exprs  ::= Named_Expr {',' Named_Expr}
Named_Expr   ::= [['~'] id ':'] Expr
```

A tuple expression `(`$e_1, \ldots, e_n$`)` is an alias for the class instance creation `Tuple_`n`(`$e_1, \ldots, e_n$`)`, where $n \geq 2$. The empty tuple `()` is the unique value of type Unit. A tuple with only one value is only the value itself, without being wrapped in a tuple. Also `Tuple_1` is a special type (pretty much like Any or Auto), which helps to make this possible even in dynamic use cases.

Sub-expressions in a tuple expression that are prepended with an id (in form of `~`x_i`:` e_i) are named x_i – this adds annotation `@[named :`x_i`]` to the type of the sub-expression. If the annotation is explicitly added, the behaviour is the same. Named sub-expressions, either in the syntax sugar or explicit form, can only appear at the end of the sequence of sub-expressions, thus no unnamed sub-expressions can appear after a named sub-expression.

Tuple expressions have the runtime type of `Gear/Language.Tuple_`n`[`$T_1, \ldots, T_n$`]`, where each T_i is the known type of each e_i, including the annotations that name sub-expressions.

Tuple expressions might be automatically converted into full argument expressions, as defined in (§9.3.5).

9.2.4 List Expressions

Syntax:

```
Words        ::= Word {{? Unicode whitespace ?}+ Word}
Word         ::= {id_char}+
Simple_Expr1 ::= List_Expr
List_Expr    ::= '%' List_Flags '[' [Expr {semi Expr}] ']'
               | '%w' List_Flags '[' [Words] ']'
List_Flags   ::= [['i' | (* immutable *)
                   'm']  (* mutable *)
                  ['l']] (* double linked *)]
```

An expression of the form `%[`e_1`; ...; `e_n`]` is a list expression.

9.2.5 Array Expressions

Syntax:

```
Simple_Expr1 ::= Array_Expr
Array_Expr   ::= '%' Array_Flags '[|' [Expr {semi Expr}] '|]'
               | '%w' Array_Flags '[|' Words '|]'
Array_Flags  ::= [['i' | (* immutable *)
                   'm']] (* mutable *)
```

An expression of the form `%[|`e_1`; ...; `e_n`|]` is an array expression.

9.2.6 Dictionary Expressions

Syntax:

```
Simple_Expr1    ::= Dictionary_Expr
Dictionary_Expr ::= '%' Dict_Flags '{' [Dict_Mapping1 {semi Dict_Mapping1}] '}'
                  | '%' Dict_Flags '{' [Dict_Mapping2 {semi Dict_Mapping2}] '}'
```

```
Dict_Mapping1   ::= Simple_Expr1 '=>' Expr
Dict_Mapping2   ::= ['~'] id ':' Expr
Dict_Flags      ::= [['i' | (* immutable *)
                      'm']  (* mutable *)
                     ['l']] (* linked *)
```

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

9.2.7 Multimap Expressions

Syntax:

```
Simple_Expr1  ::= Multimap_Expr
Multimap_Expr ::= '%' Dict_Flags '{{' [Dict_Mapping1 {semi Dict_Mapping1}] '}}'
                | '%' Dict_Flags '{{' [Dict_Mapping2 {semi Dict_Mapping2}] '}}'
```

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

9.2.8 Bag Expressions

Syntax:

```
Simple_Expr1 ::= Bag_Expr
Bag_Expr     ::= '%' Bag_Flags '(' [Expr {semi Expr}] ')'
               | '%' Bag_Flags '(' [Expr {semi Expr}] ')'
Bag_Flags    ::= [['i' | (* immutable *)
                   'm']  (* mutable *)
                  ['l']  (* linked *)
                  ['s']] (* set instead of bag (tally-less bag) *)
```

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

9.2.9 Record Expressions

Syntax:

```
Simple_Expr1       ::= Record_Expr
Record_Expr        ::= 'record' [nl] '{' [nl] Record_Fields_Init [nl] '}'
Record_Fields_Init ::= Record_Field_Init {semi Record_Field_Init}
Record_Field_Init  ::= Stable_Id '=>' Expr
```

An expression of the form

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

is a record expression.

9.2.10 Record Clone Expressions

Syntax:

```
Simple_Expr1       ::= Record_Clone_Expr
Record_Clone_Expr  ::= 'record' [nl]
                       '{' [nl] Expr 'with' Record_Fields_Init [nl] '}'
Record_Fields_Init ::= Record_Field_Init {semi Record_Field_Init}
Record_Field_Init  ::= Stable_Id '=>' Expr
```

An expression of the form

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

is a record clone expression.

9.2.11 Delayed Expressions

Syntax:

```
Delayed_Expr ::= 'lazy' [nl] Expr
```

Delayed expressions of the form **lazy** e are a syntax sugar for:

```
Gear/Language.Lazy.new(() -> { e })
```

The `Gear/Language.Lazy` instance is not automatically unboxed, unlike lazy value or variable definitions. It is an error to define a lazy value or variable with a delayed expression, as that would be only redundant. The delayed expression is evaluated by sending the instance a "`value`" message[1], or by using the prefix operator "`!`". The delayed expression is force re-evaluated by sending the instance a "`value!`" message, overwriting any previous value[2], or by using the prefix operator "`!!`", with the same side-effect.

9.2.12 Ref Expressions

Syntax:

```
Ref_Expr ::= 'ref' [nl] Expr
```

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

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

[1]This is the behaviour of call-by-need parameters.

[2]This is the behaviour of call-by-name parameters.

The `Gear/Language.Reference_Cell` instance is not automatically unboxed. The contained value can be retrieved from a reference cell using the prefix operator "`!`", or by sending the instance a "`value`" message. The contained value can be overwritten by using the infix operator "`<-`", or by sending the instance a "`value_=`" message.

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

Example 9.2.1 An example of how to manipulate reference cells.

```
(* initialize a reference cell *)
let a := ref 42
let b := Gear/Language.Reference_Cell.new(42)
let c := new Gear/Language.Reference_Cell(42)

(* retrieve contained value from reference cell *)
let d := !a
let e := a.value

(* change the value the reference cell contains *)
a <- 64
a.value := 128
```

9.2.13 Workflows

Syntax:

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

Workflow ::= 'let!' (Pattern2 | var_dcl) ':=' Expr semi Workflow
           | Pat_Var_Def semi Workflow
           | 'def' Fun_Def semi Workflow
           | 'method' Fun_Def 'end' ['method'] semi Workflow
           | 'method' Fun_Alt_Def semi Workflow
           | 'function' Fun_Def 'end' ['function'] semi Workflow
           | 'function' Fun_Alt_Def semi Workflow
           | 'type' Type_Def 'end' ['type'] semi Workflow
           | Tmpl_Def semi Workflow
           | 'yield!' Expr
           | Yield_Expr
           | 'return!' Expr
           | Return_Expr
           | 'if' Condition ('then' | semi) Workflow 'end' ['if']
           | 'if' Condition ('then' | semi) Cond_Block
             {[semi] 'elsif' Condition ('then' | semi) Cond_Block}
```

```
            Else Workflow 'end' ['if']
          | 'match' Simple_Expr1 semi Wf_When {semi Wf_When}
            [semi Else Workflow] 'end' ['match']
          | 'match' Simple_Expr1 '{' Wf_When {semi Wf_When}
            [semi Else Workflow] '}'
          | 'case' Simple_Expr1 semi Wf_Case {semi Wf_Case}
            [semi Else Workflow] 'end' ['case']
          | 'case' Simple_Expr1 '{' Wf_Case {semi Wf_Case}
            [semi Else Workflow] '}'
          | 'switch' Simple_Expr semi Wf_Switch {semi Wf_Switch}
            [semi Else Workflow] 'end' ['switch']
          | 'switch' Simple_Expr '{' Wf_Switch {semi Wf_Switch}
            [semi Else Workflow] '}'
          | 'begin' Workflow
            'catch' [nl] Wf_When {semi Wf_When} [semi Else Workflow]
            ['ensure' semi Block_Stat {semi Block_Stat}] 'end'
          | '{' Workflow '}' 'catch'
            '{' Wf_When {semi Wf_When} [semi Else Workflow] '}'
            'ensure' '{' Block_Stat {semi Block_Stat} '}'
          | 'begin' Workflow
            'rescue' [Pattern [Guard]] semi Workflow
            {'rescue' [Pattern [Guard]] semi Workflow }
            ['ensure' semi Block_Stat {semi Block_Stat}] 'end'
          | '{' Workflow '}' 'rescue'
            'rescue' [Pattern [Guard]] '{' Workflow '}'
            {'rescue' [Pattern [Guard]] '{' Workflow '}' }
            'ensure' '{' Block_Stat {semi Block_Stat} '}'
          | Rethrow_Expr (* only between 'catch' .. 'end' *)
          | Reraise_Expr (* only between 'rescue' .. 'end' *)
          | ('while' | 'until') Condition
            'loop' Workflow 'end' ['loop']
          | ('while' | 'until') Condition
            '{' Workflow '}'
          | Workflow ('while' | 'until') Condition
          | 'for' Val_Dcls 'in' ['reverse'] Expr
            ['step' Expr] 'loop' Workflow 'end' ['loop']
          | 'for' Val_Dcls 'in' ['reverse'] Expr
            ['step' Expr] '{' Workflow '}'
          | Workflow semi Workflow
          | Expr

Short_Workflow ::= 'for' Generator_Expr
                 | [Label_Dcl] 'for' Val_Dcls 'in' ['reverse'] Expr
                   ['step' Expr] Wf_For_Loop

Wf_When       ::= 'when' Pattern [Guard] ('then' | semi) Workflow
```

```
Wf_Case       ::= 'when' Case_Patterns ('then' | semi) Workflow
Wf_Switch     ::= 'when' Switches ('then' | semi) Workflow
Wf_For_Loop   ::= 'loop' Wf_Loop_Block 'end' ['loop']
                | '{' Wf_Loop_Block '}'
Wf_Loop_Block ::= {Wf_Yield | Block_Stat | Loop_Ctrl_Expr}
Wf_Yield      ::= Yield Expr
```

9.2.14 Collection Comprehensions

Syntax:

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

Collection comprehensions extend the syntax of collection "literals"[3], 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.

9.2.15 Sequence Comprehensions

Syntax:

```
Collection_Literal ::= Seq_Literal
Eagerness          ::= ['eager' | 'lazy']
Seq                ::= Eagerness 'seq'
Seq_Literal        ::= Seq '{' Collection_Gen '}'
Collection_Literal ::= Seq_Literal
```

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

Apart from collection comprehensions (§9.2.14), Gear 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 **`seq`** `{` *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 Gear Standard Runtime Library offers more functions to manipulate and create sequences.

A sequence comprehension expression **`seq`** `{` *e* `}` is equivalent to **`lazy seq`** `{` *e* `}`, whereas **`eager seq`** `{` *e* `}` is implicitly cached, but eagerly evaluated – and thus can't use `Method_Expr`, because there is no known range of keys or indexes.[4]

9.2.16 Literal Conversions & Collection Builders

9.2.17 Generator Expressions

Syntax:

```
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
                 | Guard
                 | Pattern1 ':=' Expr
Generator      ::= [Label_Dcl] Pattern1 'in' Expr [Guard]
Guard          ::= Cond_Modifier1
```

[4]The GSRL offers functions to overcome this issue.

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 (§11). 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 (§11.1.21) 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((`x_1`, ..., `x_n`) -> {` 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((`x_1`, ..., `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`** `{`x `@` p **`in`** e`}` **`yield`** `{` **`val`** x' `@` p' `:=` e'`; (`x`,` x'`) }`

Generators in generator expression can optionally have a label l assigned, so that expressions like **`break`** l could work.[5] 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.

[5]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.

Example 9.2.2 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 9.2.3 Generator expressions can be used to express vector and matrix algorithms concisely.

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

The comprehension is translated to:

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

9.2.18 Anonymous Functions

Syntax:

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

```
Bindings        ::= '(' Binding {',' Binding} ')'
Binding         ::= (id | '_') [':' Type]
Nameless_Params ::= Nameless_Param {',' Nameless_Param}
Nameless_Param  ::= '_' [':' Type]
```

The anonymous function `(`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.[6]

If the expected type of an anonymous function is of the form `Function_`n`[`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 (§11.4), where the parameters are inferred from the expected type.

The anonymous function is evaluated as the following expression:

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

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

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

A parameter of an anonymous function may optionally be preceded by an **`implicit`** modifier. In that case the parameter is labeled **`implicit`** (§10); however the parameter section itself does not count as an implicit parameter section in the sense of (§10.2). Such a parameter **`implicit`** x_i is then added transparently to the block b as **`implicit val`** y_i `:=` x_i, where y_i is a fresh name. Also, therefore arguments to anonymous functions always have to be given explicitly.

Example 9.2.4 Examples of anonymous functions:

```
(* identity function *)
x -> { x }

(* curried function composition *)
f -> g -> x -> { f(g(x)) }
```

[6] In future versions of Gear, a syntax where curly brackets are not required to be surrounding an anonymous function's body may be allowed.

```
(* a summation function *)
(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 }
```

9.2.18.1 Placeholder Syntax for Anonymous Functions

Syntax:

```
Simple_Expr1 ::= '%' digit {digit}
```

An expression (`Expr`) may contain embedded percent 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.[7]

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, provided that the percent symbol does not appear as the expression part of a typed expression.

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 `(`$u'_1, \ldots, u'_n$`) -> { `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 9.2.5 The anonymous functions in the left column use placeholder syntax. Each of these is equivalent to the anonymous function to its right.

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

```
%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     z -> { if z then x else y }
%1.map(f)               x -> { x.map(f) }
%1.map(%1 + 1)          x -> { x.map(y -> { y + 1 }) }
if %1 <= 1 then 1       x -> { if x <= 1 then 1
else %1 * %0(%1 - 1)           else x * Function.self(x - 1) }
```

Note that `Function.self` in the last example refers to the anonymous function itself.

9.2.18.2 Method Values & Partial Applications

Syntax:

```
Method_Expr       ::= '&' Simple_Expr2
                    | '&(' Simple_Expr3 ')'
                    | '&(' op_id ')' [Papply_Poetry1 | Papply_Parens1]
                    | '&(' nary_op_id ')' [Papply_Poetry1 | Papply_Parens1]
Simple_Expr2      ::= (Simple_Expr1 - Poetry_Args) [Papply_Parens]
Simple_Expr3      ::= (Simple_Expr1 - Poetry_Args)
                      [Papply_Poetry | Papply_Parens]
Papply_Parens     ::= Papply_Parens1 [Block_Arg | '&_' ['as' Type]]
                    | Block_Arg
                    | '&_' ['as' Type]
Papply_Parens1    ::= {'(' [Papply_Args_Expr] ')'}+
Papply_Poetry     ::= Papply_Poetry1
                      [Block_Expr2 | ',' '&_' ['as' Type]]
Papply_Poetry1    ::= Papply_Args_Expr
Papply_Args_Expr  ::= (Papply_Pos_Args [',' Papply_Named_Args])
                    | Papply_Named_Args
Papply_Pos_Args   ::= Papply_Pos_Arg {',' Papply_Pos_Arg}
Papply_Pos_Arg    ::= Pos_Arg | '_' ['as' Type]
Papply_Named_Args ::= Papply_Named_Arg {',' Papply_Named_Arg}
                      {',' '**' Expr}
                    | '**' Expr {',' '**' Expr}
Papply_Named_Arg  ::= Named_Arg
                    | ['~'] id ':' '_' ['as' Type]
```

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 (§9.9.5). If e is a parameterless method or call-by-name parameter of type `=>` T, `&`e represents the function of type `() ->` T, which evaluates e when it is applied to the empty parameter list `()`.

Example 9.2.6 The method values in the left column are each equivalent to the anonymous functions (§9.2.18) on their right.

```
&(Math.sin)             x -> { Math.sin(x) }
&(Array.range)          (x1, x2) -> { Array.range(x1, x2) }
&(List.map_2)           (x1, x2) -> x3 -> { List.map_2(x1, x2)(x3) }
&(List.map_2(xs, ys))   x -> { List.map_2(xs, xy)(x) }
&(42.`*`)               x -> { 42 * x }
val vs := 1 .. 9; &(vs.fold)
                        x1 -> x2 -> { vs.fold(x1)(x2) }
&((1 .. 9).fold(z))     { val eta1 := z
                          val eta2 := (1 .. 9)
                          x -> eta2.fold(eta1)(x) }
&(Some(1).fold(`???`))  { val eta1 := () -> { `???` }
                          val eta2 := Some(1)
                          x -> { eta2.fold(eta1())(x) } }
```

Note that if e resolves to a parameterless method of type `() -> `T or if e has a method type `()` $\mapsto$ T, it is evaluated to type T (§9.9.2) – 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.

Joined parameters. A partial application generates an anonymous function, in which all skipped parameters are joined into a single fresh parameter list, containing all the skipped parameters, with preserved order and typing. Such an anonymous function may again be a subject to further partial application, if so desired.[8]

Parameters appearing in parameter lists that were not provided any arguments are not considered skipped, and due to the behaviour of automatic currying, are only a regular part of the result value of the anonymous function.

Parameters that are optional, variadic, or purely named with default values, are never considered as skipped, and thus do not make it into the fresh new parameter list.

Implicit partial application. A function application (§9.3.5) using poetry-style argument passing is implicitly partially applied per each argument given.

[8] A proper IDE should show all the possible fresh parameter lists per each overloaded alternative.

The runtime is allowed to optimize such serial partial applications to fit as many arguments given as possible into a single argument list of any overloaded alternative. If the following conditions are all met, then the partial application is also fully applied:

1. Function application has all arguments needed for a single overloaded alternative, so that it can be applied to the arguments.
2. An implicit parameter list can be provided automatically with arguments, if any is present in the alternative. This is different from regular function applications, in the sense that regular function application signals an error if the implicit arguments can't be assembled.
3. The result type is compatible to the expected type. If the expected type is `Gear/Language.Auto`, then it is always compatible.
4. It is not in a method value, which prevents full application not just in case of implicit partial applications.

This rule practically means that methods with more parameters in a single parameter list are preferred to methods with fewer parameters in a single parameter list, when it comes to poetry-style argument passing. However, it is possible to split poetry-style argument passing with parentheses, to delimit separate applications in consecutive function applications.

If poetry-style arguments include *named arguments*, and any overloaded alternative or non-overloaded member has the same external parameter name across multiple parameter lists, then their order of appearance in those parameter lists is preserved in the order of named arguments. Moreover, it is possible to create a partial application that skips not just parameters, but even whole parameter lists, and the order and typing of missing arguments is preserved in parameters of the generated anonymous function.

Example 9.2.7 Basic examples of implicit partial applications using poetry-style arguments passing.

```
def papplied (a, b, c): Number := a + b + c

(* resolves to partially applied `papplied`,
   `a` reified to `1` *)
let papplied_1 (b, c) := papplied 1

(* resolves to partially applied `papplied_1`,
   `b` reified to `2` *)
let papplied_2 (c) := papplied_1 2

(* finally applies `papplied`,
   resolves to evaluated `1 + 2 + 3` *)
let fully_applied := papplied_2 3
```

```
(* resolves to partially applied `papplied`,
   `a` reified to `1` and `b` reified to `2` *)
let papplied_3 (c) := papplied 1, 2
```

Example 9.2.8 Examples of implicit partial applications in face of overloaded members.

```
def papplied (a, b, c): Number := a + b + c
def papplied (a): Number := a + 3

(* resolves to partially applied `papplied`,
   `a` reified to `1` *)
let papplied_1 (b, c) := papplied 1

(* resolves to partially applied `papplied`,
   `a` reified to `1` *)
let papplied_2 () := papplied 1
(* full application *)
let papplied_full := papplied_2()

(* finally applies `papplied`,
   resolves to evaluated `1 + 3` *)
let papplied_3 := papplied 1

(* finally applies `papplied`,
   resolves to evaluated `1 + 2 + 3` *)
let papplied_4 := papplied 1, 2, 3
```

Example 9.2.9 Example of skipping parameters and parameter lists.

```
def papplied (a, b)(c): Number := a + b + c

(* resolves to partially applied `papplied`,
   skipping first parameter list *)
let papplied_1 (a, b) := papplied ~c: 3

(* resolves to partially applied `papplied`,
   skipping `a` in its first parameter list *)
let papplied_2 (a)(c) := papplied ~b: 2
```

9.2.19 Anonymous Classes

Syntax:

```
Anon_Class      ::= ['class' [Class_Param_Clauses] 'extends']
```

```
                    [Early_Defs] Anon_Class_Tmpl
Anon_Class_Tmpl ::= Class_Parents 'with' '{' [Template_Body] '}'
```

Anonymous classs are a mechanism to implement an abstract class or override a concrete class "ad hoc", in place where needed, without needing to create a new constant (although as an expression, the anonymous class definition can indeed be assigned to a constant and gain its name). Anonymous classes can't be type constructors (§5.4.3).

A minimal anonymous class expression is of the form c `with` `{` t `}`, where c is the class that the anonymous class inherits from (can be even `Object`), and t is the template of the anonymous class. The anonymous class inherits all traits mixed into this parent class, and can itself include or prepend more traits (via the `Class_Parents` syntax element).

Optionally, the anonymous class may define its own primary constructor parameters, in which case the form of the anonymous class is `class` $(ps_1)\ldots(ps_n)$ `extends` c `with` `{` t `}`, where ps_1 to ps_n are the primary constructor parameters. Superclass constructor arguments may be specified in any case.

9.3 Application Expressions

9.3.1 Designator Expressions

Syntax:

```
Simple_Expr1 ::= Path
               | '(' Anon_Class ')' '.' Selection
               | Value_Expr '.' Selection
Selection    ::= ['?'] (id | digit {digit})
```

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 (§4). 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 (§5.2) of type T, the selection $r.x$ refers to a member m of r that is identified in T by the name x.

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

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

```
{ val y := e; if Nothing !== y then y.x else nil }
```

for some fresh name y; also called *safe navigation* or *safe selection*; `!===` is the negated result of `===` used in (§9.6.3).

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 backticks, e.g. `def `1` () := 1`. The expression e must not be a number literal.

9.3.2 Attribute Selection Expressions

Syntax:

```
Simple_Expr1 ::= Simple_Expr ''' Selection
```

An attribute is a special function that has at least one parameter list with a single parameter, and optionally more regular parameter lists following the first one.

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. (§9.3.5).

9.3.3 Self, This, Super, Outer & Module

Syntax:

```
Simple_Expr1 ::= [id '.'] 'self' ['[' ('cloned' | 'origin') ']']
                   ['.' Selection]
                 | [id '.'] 'this' ['[' ('cloned' | 'origin') ']']
                   '.' Selection
                 | [id '.'] 'super' ['[' ('cloned' | 'origin') ']']
                   [Class_Qualifier]
                   ['.' Selection]
                 | 'outer' Class_Qualifier
                   ['.' 'super' [Class_Qualifier]]
                   ['.' Selection]
                 | 'outer' Class_Qualifier
                   '.' 'this' '.' Selection
                 | 'module' [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 **this** is the same as **self**, except that function resolution searches from the class that this expression appears in, possibly skipping overrides in subtypes of the actual class of **self**, and continues as usual up the inheritance chain. The **this** expression is interchangeable with **self** in the following paragraphs, although use of **self** is preferred.

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

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

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

The **super** prefix may be followed by a qualifier **[**T**]**, as in C**.super[**T**].**m. In this case, the reference is to the type or method m in the parent class or trait of C, whose simple name is T. 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, but, when the slow path of stable id is taken, it may refer to a lazily imported type name. It is obviously an error if the slow path is taken, no name is found during compilation and there are no lazy imports in the scope, or no name is found using the lazy import.

9.3.4 Use Expressions

Syntax:

```
Use_Expr    ::= Use_Expr_As | Use_Aspect
Use_Expr_As ::= 'use' Simple_Expr ('as' | 'as!' | 'as?')
```

```
                [id ':'] Type [Block_Expr]
Use_Aspect  ::= 'use' 'aspect' Path [Block_Expr]
```

Use expressions of the form **use** e **as** a: T are similar to typed expressions (§9.5.1). 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 `Block_Expr` is syntactically given, only in the scope of that block expression. If a block is given, then the return value of the block is the value of this expression, otherwise, the value retrieved by evaluation of `Simple_Expr` is the value of this expression. Conversions described in typed expressions (§9.5.1) apply in these expressions as well, including the differences between **as** and **as!**.

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

9.3.5 Function Applications

Syntax:

```
Simple_Expr1   ::= (Simple_Expr1 - Poetry_Args) Argument_Exprs
Argument_Exprs ::= ['.'] Parens_Args {Parens_Args} [Block_Arg]
                 | Poetry_Args [Block_Expr2]
                 | Block_Arg

Parens_Args ::= '(' [Args_Expr] ')'
Poetry_Args ::= Args_Expr
Args_Expr   ::= (Pos_Args [',' Named_Args])
              | Named_Args
Pos_Args    ::= Pos_Arg {',' Pos_Arg}
Pos_Arg     ::= ['*'] Expr
              | '$'
              | TT_Expr
              | Slice_Expr
Named_Args  ::= Named_Arg {',' Named_Arg} {',' '**' Expr}
              | '**' Expr {',' '**' Expr}
Named_Arg   ::= ['~'] id ':' (Expr | TT_Expr | Slice_Expr)
              | ['~'] id ':' '$'
Block_Arg   ::= 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 (§9.3.8) into it. If f has a method type $(p_1\colon T_1, \ldots, p_n\colon T_n) \mapsto U$, the type of each argument expression e_i is typed with the corresponding parameter type T_i (§9.3.5.2) as expected type. Let S_i be type of argument e_i (for

$i = 1, \ldots, m$). If f is a polymorphic method, local type inference (§9.9.4) is used to determine type arguments for f. If f is of a value type, the application is taken to be equivalent to f`.apply(`$e_1, \ldots, e_m$`)`, i.e. the application of an `apply` method defined by f.

If none of e_i is purely a named argument, and f accepts a single parameter of `Any`, `Object` or `Tuple_`n type (where $n = m$), then the function application is taken as f`((`$e_1, \ldots, e_m$`))`. With respect to overloading resolution (§9.9.3), a member that requires argument expressions to tuple conversion is always less specific than a member that does not require such conversion.

Distinctions. Function applications using the `Poetry_Args` syntactic category are automatically partially applied, see (§9.2.18.2).

Note. Target type expression (§9.3.11) can't be the first argument in a function application that uses poetry-style arguments passing, due to resulting ambiguity between an argument expression and a chained function application or selection.

9.3.5.1 Argument Evaluation Strategies

Gear defers evaluation of arguments up to the point of function application, and happens then as specified in parameter evaluation strategies (§6.10.4). The type that each argument is type-checked against the corresponding parameter type (defined as follows) is the expected type of the argument expression, i.e. not its actual concrete type, which is known only after its evaluation. If the expected type is undefined, then `Any` is assumed. Typed expressions (§9.5.1) may be used to give the argument expression a concrete expected type. When the argument expression is evaluated, it is evaluated as if it were in the scope of the function application (which it is), so that visibility rules from that scope apply.

9.3.5.2 Corresponding Parameters

The argument expressions $a_1, \ldots, a_n$ are said to be corresponding to parameters $p_1, \ldots, p_n$ based on the following mapping definition.

1. Argument unpacking is applied.

    (a) *Sequence arguments* are prefixed with an asterisk "`*`", and are evaluated to a `Sequence[_]`-compatible type. The values are unpacked and injected in place of the sequence argument in order defined by the sequence. This triggers early evaluation of the argument, which is only done once in face of overloading resolution and/or the following function application. Sequence arguments may only appear among positional arguments.

(b) *Mapping arguments* are prefixed with a double asterisk "`**`", and are evaluated to a `Sequence[(Symbol, _)]`-compatible type; `Map[Symbol, _]` is one such an acceptable type which does not require further conversion to a sequence. The key and value pairs (tuples) are unpacked from the sequence and appended to the sequence of other named arguments.

2. Condition check: if there are named arguments, each argument name must be unique per argument list. It is an error if there are duplicate argument names, either before or after argument unpacking.

3. Positional arguments are mapped to their corresponding parameters, as defined by positional parameters (§6.10.6). Positional parameters will never map to purely named parameters.

4. Condition check: after the previous step, if there are any other unassigned arguments, the remaining arguments must be named, and optionally followed by a block argument.

5. The remaining named arguments are mapped based on their name to parameters of the same external name (§6.10.3). If the corresponding parameter is a variadic parameter, then rules for the parameter apply (§6.10.6.3). Named arguments are never corresponding to positional parameters without an external name (§6.10.3).

6. If there are any extra named arguments, and a capturing named parameter is present, those extra named arguments are mapped to it.

7. If there is a *block argument* passed as positional and appearing last in the argument list, it is mapped to the captured block parameter, if that is present, and if not, then the argument is available for use with the **`yield`** keyword (§9.3.9). It not an error if the function member does not use **`yield`**, but that should be avoided. A block argument is either a block expression, or an argument prefixed with an ampersand "`&`". In a consecutive function application, there may be only one block argument per each argument list as allowed by syntax, unlike with captured block parameter.

8. If there are any parameters without an assigned argument, those parameters must have a default value, which is evaluated prior to the function application (provided that the function was selected as unique in overloading resolution, if any). It is an error if there are any parameters left without a corresponding argument or default value. It is not an error if a variadic parameter is left with an empty sequence, as it is not an error if a capturing named parameter contains an empty mapping.

The type of each argument expression a_i is typed with the corresponding parameter type T_j as expected type, where the mapping from i to j is defined by corresponding parameters and is correct.

9.3.5.3 Applicable Function

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

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

- A mapping defined by corresponding parameters exists and is correct.
- For every named argument x_i`:` a'_i, the type S_i is compatible (§9.9) with the parameter type T_j, whose name p_j matches x_i, or if f defines a capturing named parameter and x_i does not match name of any p_j, then the type S_i is compatible with the parameter type T_j, whose name p_j matches the name of the capturing named parameter.
- For every positional argument a_i, the type S_i is compatible (§9.9) with its corresponding T_i.
- The given block or the last argument prefixed with "`&`" is of a type compatible (§9.9) with the type of the captured block parameter, if such parameter is defined.
- If the expected type of the function application is defined, the result type R is compatible (§9.9) to it.
- Every formal parameter p_j`:` T_j which is not specified by either a positional or a named argument has a default value (§6.10.6.2 & §6.10.7).

If f is a polymorphic method, it is applicable if local type inference (§9.9.4) can determine type arguments, so that the instantiated method is applicable. If f is of a value type, it is applicable if it has a method member named `apply`, which is applicable. Note that if explicit type parameters are given to the polymorphic method, type application (§9.3.6) happens prior to function application.

If a function application appears to be an argument to another function application (let's call it a nested function application), the expected type of the nested function application is used to determine, whether the outer function is applicable, but the nested function application is not evaluated until time specified by argument evaluation strategy (§9.3.5.1) corresponding to the argument. Local type inference may indeed occur for the nested function application, if it involves a polymorphic method, but again, only using the available expected types.

If a corresponding parameter p_j to an argument expression e_i is of a constrained type (§5.3.16), the argument expression is early evaluated, so that it can be determined whether the argument's type conforms to the constrained type, but only if the expected type of e_i does conform to the type of p_j (otherwise, it is clear that it won't conform, thus there is no need to evaluate the expression).

Example 9.3.1 Assume the following function, which computes the sum of variable number of arguments:

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

Then

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

all yield 10 as result. On the other hand,

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

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

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

would be interpreted as

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

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

9.3.5.4 Tail-call optimization

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

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

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

9.3.5.5 Named & Optional Arguments

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

- No named argument appears left of a positional argument in the argument list.
- No positional argument appears right of a named argument. A bit of an exception is the captured block argument, which appears to be positional, but is treated specially.
- The names p_i of all named arguments are pairwise distinct per argument list.
- Every formal parameter $p_j: T_j$, which is not specified by a positional argument, has a default argument.
- Every formal parameter $\sim p_j: T_j$, which is not specified by a named argument, has a default argument.
- If there are more named arguments than named parameters (excluding the capturing named parameter), a capturing named parameter is defined. (If it is not, the function is not applicable.)

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

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

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

9.3.5.7 Curried Functions & Partial Applications

A curried function can appear in two distinct forms:

> *Implicitly curried form*, which is defined by using multiple parameters lists.
>
> *Explicitly curried form*, which is defined by using function types as result types of functions, or simply by returning a function from within a function. Those have no special treatment, e.g. in regard of overloading resolution.

Each form has some implications on function applications.

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

Example 9.3.2 The following are examples of consecutive function applications:

```
f(a, b)(c, d)
f(a, b).apply(c, d)
f(a, b).(c, d) (* short form of the previous *)
```

The following are not consecutive function applications:

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

An implicitly curried function requires a consecutive function application for all of its parameters lists, excluding the implicit parameters list. If the implicitly curried function is intended to be *partially applied* (not providing all the parameters lists with arguments lists), then a method value (§9.2.18.2) can be used. This also applies to the implicit parameters list – if providing it is to be deferred, a method value that encloses arguments lists up to the implicit parameters list can be used, but then the implicit arguments list has to be provided later in order to evaluate the curried function.

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

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

9.3.5.8 Function Compositions & Pipelines

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

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

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

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

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

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

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

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

  (* right-binding *)
  operator |> (a: T): R :=
    self(a)

  (* left-binding *)
  operator <| (a: T): R :=
    self(a)

end trait
```

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

9.3.5.9 Memoization

How to memoize a function's result is described in (§9.6.4.1).

A memoized function's body is not evaluated, if it was once called with the same arguments (based on equality, not identity), and if that result value is still memoized. If so, the memoized result value is immediately returned without evaluation of the function's body, which can speed up execution of some functions significantly. Such functions should however be referentially transparent in best-case scenario (§6.10 & §9.8) 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 (§6.10) may opt-in to implicit memoization[9], as well as functions declared as **`transparent`** (§9.8). Functions declared as **`opaque`** (§9.8) should not be memoized.

Parameters of memoization[10] may be controlled, even on per-function basis, with use of specialized annotations and pragmas.

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

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

9.3.5.10 Application Shortcut

A function application f`.(`$e_1, \ldots, e_m$`)` b is a syntax sugar for a function application `(`f`())(`$e_1, \ldots, e_m$`)` b, which in turn is a shortcut for function application f`().apply(`$e_1, \ldots, e_m$`)` b, as appropriate.[11]

This is especially useful for situations where a prefix appears to the function application and f denotes a member that evaluates to a function, or maybe to a collection, but does not itself accept any arguments.

A function application $p.f$`.(`$e_1, \ldots, e_m$`)` b is a syntax sugar for a function application `(`$p.f$`)(`$e_1, \ldots, e_m$`)` b, which in turn is a shortcut for function application $p.f$`().apply(`$e_1, \ldots, e_m$`)` b, where p is some prefix expression.

9.3.5.11 Receiver Universal Shortcut

A function application $r.f$`(`$e_1, \ldots, e_m$`)`, where at least one of e_i is literally `$` or contains it as a sub-expression, is a syntax sugar for function application, where each `$` is replaced with `r.\`$\`()`.

A function application r`(`$e_1, \ldots, e_m$`)`, where at least one of e_i is literally `$`, is a syntax sugar for function application, where each `$` is replaced with `r.\`$\`()` and `r` is a variable reference.

In case of nested function applications, this shortcut is applied absolutely only to the outermost application, to prevent any confusion. It can be combined with application shortcut.

Example 9.3.3 Combining receiver universal shortcut with application shortcut.

```
(* the expression *)
prefix.receiver.($)
(* is equivalent to the expression *)
prefix.receiver.(prefix.receiver.`$`())
(* which is in turn equivalent to *)
prefix.receiver().apply(prefix.receiver().`$`())
```

Note. The receiver universal shortcut is defined for Gear's collection types as an alias to `size` – to be used in slicing, primarily.

9.3.6 Type Applications

Syntax:

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

[11] E.g. curried functions do not need any explicit `.apply()` applications, as these are implied, but would work anyway, therefore the second expanded form is equivalent to the first expanded form.

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

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

Type applications can be omitted if local type inference (§9.9.4) 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 partial type application, where the provided type arguments are taken as constant types.

9.3.7 Instance Creation Expressions

Syntax:

```
New_Expr ::= 'new' [Early_Defs] Class_Parents
               ['begin' [Template_Body] 'end']
           | 'new' [Early_Defs] Class_Parents
               ['{' Template_Body '}']
           | 'new' '{' Template_Body '}'
```

Unlike languages like Java, Scala, C# and similar, Gear does not have dedicated language construct for creating new instances of classes. Instead, all such attempts are made through the `Class#new` method (not to be confused with "static" `Class.new`, which is for dynamic creation of ad-hoc classes), which passes all the arguments to the appropriate constructor (and is defined using `Variadic_Arguments`).

It is indeed possible to write custom T`#new` implementations, where T is the class—or even trait!—for which the custom implementation is, and that implementation may decide to use `Class#new` or not (typical use case: a factory method without the usual corporate factory names).

There is one more way to create an instance of a class, and that is to use a syntax construct of the `New_Expr` syntax category. This construct has multiple forms for different use cases. Note that there is also a syntax construct for anonymous classes (§9.2.19), which is related, but has slightly different semantics – most notably, anonymous classes can be reused and bound to a name, because they do not create any new instance (just yet), and can define class parameters, including type parameters.

A simple instance creation expression is of the form **`new`** c, where c is a constructor invocation (§7.1.3). Let T be the type of c, then T must denote a type instance of a non-abstract subtype of `Any`. T must also conform to the *self type* of the class denoted by T. The expression is evaluated by invoking `Class#new`, where the method is invoked with **`self`** being T, thus skipping any custom overrides of the method. Note that c might be an instantiated type variable, in which case it

might not be entirely possible to guarantee that the expression will be evaluated successfully – no matching and accessible constructor might be available in the instantiated type.

A general instance creation expression is of the form `new` t for some class template t. Such an expression is equivalent to the following block:

`{ class` a `extends` t`; new` a `}` ,

where a is a fresh name of an *anonymous class* (§9.2.19) bound to the name a within the block (and unbound when the block exits).

A shorthand instance creation expression is of the form `new { `S` }`, where `{ `S` }` is a class body, which is equivalent to the general instance creation expression `new Object { `S` }`. This is for creating structural types, ad-hoc.

Example 9.3.4 Consider the following shorthand instance creation expression:

```
new { def name() := "Dagny Taggart" }
```

This is a shorthand for the general instance creation expression

```
new Object { def name() := "Dagny Taggart" }
```

Which is in turn a shorthand for the block

```
{ class a extends Object { def name() := "Dagny Taggart" }; new a }
```

where a is a freshly created name.

9.3.8 Blocks

Syntax:

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

A block expression `{ `s_1`; ...; `s_n`; `e` }` is constructed from a sequence of block statements $s_1, \ldots, s_n$ and a final expression e. The statement sequence may not contain two definitions

or declarations that bind the same name in the same namespace, except for local function definitions, which then create overloaded local function definitions (behaving pretty much like regular overloaded functions). The final expression may be omitted, in which case the unit value `()` is assumed.

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

The type of a block `{` s_1`;` ...`;` s_n`;` e `}` is T **`for-some`** `{` Q `}`, where T is the type of e and Q contains existential clauses (§5.3.12) for every value or type name which is free in T and which is defined locally in any of the statements $s_1, \ldots, s_n$. We say that the existential clause *binds* the occurence of the value or type name. Specifically,

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

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

9.3.8.1 Block Expression as Argument

A block expression may be used as the very last argument in a function application (§9.3.5), 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 (§9.6.4.1) 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 (§9.2.18), where the return expression is interconnected with the anonymous function itself.

9.3.8.2 Variable Closure

Gear 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 behavior and store the reference as either **`weak`**, **`soft`** or **`unowned`** reference, to break the retain cycle.[12]

If a block expression is used as an argument in a function application (§9.3.5), 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 (§9.3.3). Write access to variables is provided in a manner equivalent to `out` arguments.

Variable closure is applied to anonymous functions (§9.2.18) 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.

9.3.9 Yield Expressions

Syntax:

```
Yield_Expr ::= [id '.'] 'yield' [Parens_Args | Poetry_Args]
```

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

1. A way to invoke the block argument (§9.3.5 & §9.3.8.1) 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 (§9.2.13).

4. A part of generators (§9.2.17) and collection comprehensions (§9.2.14).

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

5. In other cases, it is an error to use **`yield`**.

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 Gear'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 Gear's `Language` module.

9.3.10 Prefix & Infix Operations

Syntax:

```
nary_op_id   ::= '?' {op_char}
Infix_Expr   ::= Prefix_Expr
               | Infix_Expr [Infix_Op Infix_expr]
               | Infix_Expr nary_op_id Infix_Expr {':' Infix_Expr}
               | Range_Expr
Simple_Expr1 ::= '(' op_id ')' Poetry_Args
               | '(' nary_op_id ')' Poetry_Args
Infix_Op     ::= op_id | id
Prefix_Expr  ::= [op_id | 'not'] Simple_Expr1
Range_Expr   ::= Infix_Expr ('..' | '...' | '..<') Infix_Expr
                 [FP_Delta [FP_Digits]]
Slice_Expr   ::= Infix_Expr ('..' | '...' | '..<') [Infix_Expr]
               | ('..' | '...' | '..<') Infix_Expr
```

Expressions can be constructed from operands and operators.

9.3.10.1 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 method application e`.`*op*`()` (i.e., prefix operators are right-binding). If such method does not exist, it is equivalent to the function application *op*`(`e`)`, and if such function does not exist either, implicit conversions on e are attempted.

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 (§9.3.10.3). For example, the source input sequence `-sin(x)` is read as `-(sin(x))`, which is in turn read

as `sin.apply(x).`-`()`, whereas the function application `negate sin(x)` would be parsed as `negate(sin(x))`, using poetry-style arguments. To further emphasize the precedence of prefix operators, `-a + b` is read as `-(a) + b`, and `-(a + b)` as just that.

Prefix operations may alternatively appear in the form (*op*) e, which is equivalent to a function application form, but still searches for implementation of the operator in e first.

9.3.10.2 Postfix Operations

Apart from standard function applications (§9.3.5) that may be viewed as postfix, Gear does not include support for postfix operations.

9.3.10.3 Infix Operations

An infix operator can be an arbitrary identifier, usually an operator identifier. Infix operators have static *precedence*, *associativity* and *binding direction* 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 (§3.4) is followed.

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 with poetry-style arguments passing (unless e_1 is a tuple expression with more than 1 elements, then it's regular-style arguments passing).

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* (§9.3.10.4). 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 `&`.
- **or**, 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* and *binding direction* of an operator is determined by the operator's last character. Operators ending in a colon ":" are right-associative, unless they are n-ary (§9.3.10.5); and operators ending in a greater-than sign ">" are right-binding, if they both consist of more than one operator character. All other operators are left-associative and left-binding. No operator is right-associative and right-binding at the same time, as right-binding operators are in fact left-associative.

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

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

- If e_0 in $e_0\ op_1\ e_1$ does not implement the operator, then the infix operation is equivalent to function application $op_1(e_0,\ e_1)$, and if such function does not exist either, then implicit conversions are attempted on e_0. This is also applied recursively to consecutive infix operations.

Alphanumeric Infix Operators vs. Function Application. Infix operations take precedence over function applications (§9.3.5) that use "poetry"-style arguments passing.

Example 9.3.5 Distinction between infix operations and function applications with "poetry"-style arguments passing.

```
(* 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 *)
a.shift-left(b)
(* or alternatively *)
shift-left(a, b)
(* and not to the less obvious *)
a(shift-left(b))
```

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.

Standard Operators. Always consult GFR-2 for complete reference. 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 identity *)
a /== b      (* not identical *)
not (a == b) (* quite the same as 'not identical' *)
```

Standard arithmetic operators:

```
a + b     (* addition *)
a +. b    (* real addition *)
a - b     (* subtraction *)
```

```
a -. b    (* real subtraction *)
a * b     (* multiplication *)
a *. b    (* real 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 *)
a < b    (* less than *)
a <= b   (* less than or equal *)
a > b    (* greater than *)
a >= b   (* greater than or equal *)
```

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` less than all arguments *)
a <= (...)  (* `a` less than or equal all arguments *)
a > (...)   (* `a` greater than all arguments *)
a >= (...)  (* `a` greater than or equal all arguments *)
```

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) *)
not a    (* bitwise not (prefix, unary; when applied to numbers) *)
```

Standard boolean (logical) operators:

```
a && b   (* boolean and *)
a and b  (* boolean and *)
a || b   (* boolean or *)
a or b   (* boolean or *)
a xor b  (* boolean xor *)
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".

9.3.10.4 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 "`/=`"[13].

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.

[13]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`".

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 (§9.9) to a value with a member named "+=", applicable to a value of type of r.
2. The assignment 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 variable l is assignable, defined, not immutable. This implies that it is defined as a **var** 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.[14] 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.

Example 9.3.6 Right-associativity of assignment operations, using mutable variables as left-hand sides.

```
(* the assignment *)
a := b := c
(* is equivalent to *)
a := (b := c)

(* the assignment *)
(a := b) := c
(* is equivalent to *)
a := b; a := c

(* this also works *)
a := let b := c
(* as *)
a := (let b := c)
```

[14]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.

9.3.10.5 N-ary Infix Expressions

Syntax:

```
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, 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 two infix expressions and the third function application are equivalent.

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

Example 9.3.7 The most widely used n-ary operator is the ternary conditional operator "?", which could be defined on the `Boolean` type as follows:

```
class Boolean extends Object
  operator ? [A, B <: A, C <: A] (if_true: => B, if_false: => C): A
    if self
      if_true
    else
      if_false
    end
  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:

```
object Conditional
begin
  operator ?: [A, B] (if_true: => A, if_false: => B): A or B
    if if_true.to_boolean
      if_true
    else
      if_false
    end
  end
end
```

```
implicit def to_boolean_wrapper [W] (obj: W): Boolean_Wrapper[W]
  Boolean_Wrapper.new(obj)
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` *)
```

9.3.10.6 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.

9.3.11 Target Type Expressions

Syntax:

```
TT_Expr ::= '.' Simple_Expr1
```

Target type expression e with expected type *et* is a shorthand for the selection and/or application (either function, type or both) $et.e$.

Target type expressions appear throughout the syntax in a few places. Each such place imposes a condition on what is a legal target type expression for it. Every target type expressions requires an expected type to be defined. Based on the expected type, the target type expression is a selection from the expected type. If a target type expression is used as an argument, its type is a refinement that contains the selected member (or members if chained), and optionally a requirement of type parameters and/or application parameters. The refinement is then as usual checked relative to the expected type, and not recursively on any nested members. Implicit conversions may not be applied on the resulting expression, because there would be no way to evaluate an original value, which depends on the expected type.

Note. Target type expressions are limited to a simple selection, or a chain of selections, optionally followed by a type application and/or a function application with arbitrary number of argument lists.

9.3.12 Assignments

Syntax:

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

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

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

Analogously, an assignment f`.`x `:=` e is interpreted as the invocation f`.`x`_=(`e`)`. If f is evaluated to **`nil`**, then the invocation is forwarded to **`nil`**.[15]

An assignment f`.?`x `:=` e is interpreted as the invocation f`.?`x`_=(`e`)`. If f is evaluated to **`nil`**, then the invocation is evaluated to **`nil`**. See (§9.3.1) for more on behavior of the "`.?`" navigation.

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

The interpretation of an assignment x `:= .`e, where x may be any of the left-hand sides described so far in assignments, depends on the type of x: if e is a selection or a name, then the member of x with the name selected by e is used as the assigned value, and if arguments are also applied to e, then they are applied to the selected member as well. It is an error if the type of x does not contain the selected member.

Example 9.3.8 Here are some assignment expressions and their equivalent interpretations.

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

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

9.3.12.1 Multiple Assignments

Syntax:

```
Mul_Assign_Expr ::= Mul_Vars ':=' Mul_Exprs
Mul_Vars        ::= [[val_ids ','] '*' id ','] val_ids
                  | [val_ids ','] '*' id
Mul_Exprs       ::= Mul_Expr {',' Mul_Expr}
Mul_Expr        ::= ['*'] Expr | TT_Expr
```

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

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

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

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

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

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

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

Example 9.3.9 The following examples show how multiple assignment works.

```
(* swap two variables *)
a, b := b, a

(* swap with a pattern and a tuple,
   rebinding the names a and b *)
let (a, b) := (b, a)

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

9.4 Definition Expressions

Syntax:

```
Expr ::= Def
```

9.5 Type-Related Expressions

9.5.1 Typed Expressions

Syntax:

```
Cast_Expr  ::= Infix_Expr ('as' | 'as!' | 'as?') (Type | Simple_Expr1)
Infix_Expr ::= Infix_Expr ('is' ['not'] | 'is!' | 'is' 'not!')
               (Type | Simple_Expr1)
```

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

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

3. Otherwise, the conversion is of the form e`.as_instance_of[`T`]()`.[16]

If a type implements the `as_instance_of[`T`]()` method, it must meet one of the following conditions:

- Have at most one empty parameter list.
- Have exactly one implicit parameter list.[17]

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

1. Type of e is a subtype of T.
2. An implicit conversion c from expression type E of method type `(`E`)` $\mapsto$ T exists in the scope.
3. As a last resort, type of e overrides the method `is_instance_of[`T`]()` and evaluating it results in **`yes`** value.[18]

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 (§9.6.1), in the related branch. Therefore, it is also used in resolution of function applications (§9.3.5). 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 **`with`** 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".

[16]Note that this conversion method is not included in implicit conversions, and has to be used either directly, or via a typed expression.

[17]And no other parameter list.

[18]This method basically tells whether the instance implements `as_instance_of[`T`]()`, thus being able to convert itself into another type without other implicit conversions.

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 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.

Typed expressions do not allow checks against target type expressions (§9.3.11) and conversions to them, as such checks and conversions would make no sense at all.

9.5.2 Annotated Expressions

Syntax:

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

An annotated expression $a_1 \ldots a_n\ e$ attaches annotations $a_1 \ldots a_n$ to the expression e (§14).

9.6 Control Flow Expressions

9.6.1 Conditional Expressions

Syntax:

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

```
Else            ::= 'else' | 'otherwise'
Condition       ::= Simple_Expr1 | Pat_Var_Def
```

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** (§9.5.1), and **no** otherwise. The **nil** and **undefined** objects convert always to **no**. If the e_1 is the single instance "`()`" of type `Unit`, it is an error. The **then**-part e_2 and the **else**-part e_3 are both expected to conform to the expected type of the conditional expression, but are not required to. The type of the conditional expression is the weak least upper bound (§5.10.4) 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 behavior, one has to use the operator identifier in a function application (§9.3.5).

Example 9.6.1 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 conditional expression

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

is evaluated as if it was

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

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

The alternative conditional expression **unless** e_1 **then** e_2 **else** e_3 is evaluated as if it was **if not** e_1 **then** e_2 **else** e_3. Unlike in Ruby, the **elsif**-part is allowed to appear with this conditional expression. However, there is no syntax sugar for the **else unless** keyword tokens sequence.

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

Unlike in some languages, conditional expressions do not require to place parentheses around the conditions – but it is possible to do so, the result is equivalent. That might be useful when the condition is inevitably long and needs to span multiple lines – so that boolean operators may be situated at the beginning of each new line, instead of being at the end of the previous line.

Note. Gear 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 Definition. The conditional variable definition **if** p `:=` e_1 **then** e_2 **else** e_3, where p `:=` e_1 is a variable definition (see syntactic category `Pat_Var_Def`), 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**, the condition is still met 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".

9.6.2 Loop Expressions

Gear 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 Gear.

9.6.2.1 Loop Control Expressions

Syntax:

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

```
Redo_Expr      ::= 'redo' [label_name] [Cond_Modifier1]
Exhausted_Expr ::= 'exhausted' Block_Expr
Broken_Expr    ::= 'broken' Block_Expr
```

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

In the following paragraphs, a *loop identified by the label* l is a loop expression preceded in its syntax with the syntax element `Label_Dcl` (§9.6.4.3). All annotations (§9.5.2) that precede the label declaration are applied to the following loop expression, never to the label.

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

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

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

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

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

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

The standard library provides loop-like methods, where these loop control structures are not available as keywords, but as methods instead (either imported or available on some given loop-control object) – they might be implemented e.g. using the **`throw`** expressions (§9.6.5), 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.[19]

9.6.2.2 Iterable For Expressions

Syntax:

```
Loop_Expr      ::= [Label_Dcl] 'for' Val_Dcls 'in' ['reverse'] Expr
                   ['step' Expr] For_Loop
For_Loop       ::= 'loop' Loop_Block_Expr 'end' ['loop']
                 | 'do' Loop_Block_Expr 'done'
                 | '{' Loop_Block_Expr '}'
```

[19] 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.

```
Loop_Block_Expr ::= {Block_Stat | Loop_Ctrl_Expr}
Val_Dcls        ::= Val_Dcl
                  | Pattern1
```

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

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

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

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

Iterable expression repeats evaluation of the expression e_3 for every value that comes from the `Iterable_Like`'s `Iterator`, unless the loop controls alter this flow (§9.6.2.1).

Iterable expressions can be seen as simple comprehensions over iterating a single iterable value. For more complex iterating expressions, see generators (§9.2.17).

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 reverse`** e_2 **`loop`** e_3 **`end`**

is translated to the invocation

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

An expression

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

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

e_2`.reverse.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`.

9.6.2.3 While & Until Loop Expressions

Syntax:

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

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 (§9.6.1). 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:

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

The real implementation has to handle loop control expressions (§9.6.2.1) 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 (§9.6.2.1).

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.

9.6.2.4 Pure Loops

Syntax:

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

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

A pure loop expression repeats evaluation of the expression e as long as the loop controls don't alter this flow (§9.6.2.1). It is basically equivalent to an iterable expression (§9.6.2.2) 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.

9.6.3 Pattern Matching, Case Expressions & Switch Expressions

Syntax:

```
Match_Expr     ::= Pat_Match_Expr | Case_Expr | Switch_Expr
Pat_Match_Expr ::= 'match' Simple_Expr1 Match_Body
Match_Body     ::= semi When_Clauses 'end' ['match']
                 | '{' When_Clauses '}'
When_Clauses   ::= When_Clause {semi When_Clause}
                   [semi Else Cond_Block]
When_Clause    ::= 'when' Pattern [Guard] ('then' | semi) Cond_Block
Case_Expr      ::= 'case' Simple_Expr1 Case_Body
Switch_Expr    ::= 'switch' Simple_Expr1 Switch_Body
Case_Body      ::= semi Case_Clauses 'end' ['case']
                 | '{' Case_Clauses '}'
Switch_Body    ::= semi Switch_Clauses 'end' ['switch']
                 | '{' Switch_Clauses '}'
Case_Clauses   ::= Case_Clause {['next'] semi Case_Clause}
                   [['next'] semi Else Cond_Block]
Switch_Clauses ::= Switch_Clause {['next'] semi Switch_Clause}
                   [['next'] semi Else Cond_Block]
Case_Clause    ::= 'when' Case_Patterns ('then' | semi) Cond_Block
Switch_Clause  ::= 'when' Switches ('then' | semi) Cond_Block
Case_Patterns  ::= Case_Pattern {',' Case_Pattern}
Switches       ::= Switch {',' Switch}
Case_Pattern   ::= Stable_Id
                 | id
                 | Infix_Expr
Switch_Scalar  ::= (Literal - Collection_Literal)
Switch         ::= Switch_Scalar [('..' | '...' | '..<') Switch_Scalar]
```

Pattern matching is described in (§11). Here, the syntax of expressions that make use of pattern matching is given.

Case expressions

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

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

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

Let T be the type of the selector expression e. The parameter p_i of each invocation of the method "`===`" is typed with T as its expected type and `Boolean` as the result type. It is an error if T does not conform to the actual type of the parameter, as the invocation would not be applicable (§9.3.5). The method "`===`" may be overloaded for multiple parameter types, then overloading resolution (§9.9.3) applies as usual.

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

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

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

The expected type of every block b_i is the expected type of the whole pattern matching expression. The type of the pattern matching expression is then the weak least upper bound (§5.10.2) of the types of all blocks b_i.

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

A case clause or a switch 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 case clause or switch clause.

Switch expressions

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

are extremely simplified pattern matching expressions, though they do not use patterns from pattern matching expressions, but *equality* and/or *range membership* instead. Therefore, switch expressions do not aim at matching the selector expression e, thus decomposing the selector e, but rather tests if it falls into a particular set of values, defined using the equality or range membership:

- number types are matched based on their value,
- string types are matched based on their hash code,
- boolean types are considered `1` for **`yes`** and `0` for **`no`**.

Compiler adds extra code to determine the value to switch on if the switched expression is of an unknown type, or not of one of the mentioned types. Generally, there are two internal kinds of switch instructions:

1. Single entry switch table, for switch expressions without ranges,

2. Triplet entry switch table, for switch expressions with ranges, where the triplet consists of the two marginal values and an indicator whether the range is inclusive (`..`) or exclusive (`...` or `..<`) of the upper limit.

Switch expressions therefore do not involve any other implicit conversions or user code evaluation, other than that required to get the value to switch on. If a switch branch uses a guard, that guard becomes a part of the conditional expression that follows.

Difference from Java etc. Java (and countless other languages) does not allow switching on other values than integral numbers (and strings since Java 1.7), where Gear allows all scalar literals, including floating point numbers. It is quite unreliable to switch on floating point numbers, due to rounding errors, whereas switching on fixed point numbers is reliable. Both switching on floating point numbers and fixed point numbers should only be involved in switch expressions with ranges.

NaN in a switch. The *not a number* (`Number.NaN`) value is treated specially in switch expressions – it simply never matches anything else but the **`else`** branch. The comparisons of values involved are safe from errors that could be raised from regular unsuspecting comparison with a `NaN` value.

9.6.4 Unconditional Expressions

Unconditional expressions change the flow of programs without a condition.

9.6.4.1 Return Expressions

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

Syntax:

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

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

Syntax:

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

A return expression **`return`** e must occur inside the body of some enclosing method, or inside a block nested in the body of the innermost enclosing method. Unlike in Scala, the innermost

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

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

Returning from a nested block is implemented by throwing and catching a specialized exception, which may be seen by **`throw-catch`** expressions (§9.6.5) 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 (§9.3.5.9). Memoization is not available from within anonymous functions and blocks.

9.6.4.2 Structured Return Expressions

Syntax:

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

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

9.6.4.3 Local Jump Expressions

Syntax:

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

Local jumps transfer control from the points of **`goto`** statements to the statements following a **`label`**. Such a jump may only occur inside of the same function, i.e. it is not possible to jump

from one method to another. Also, the jump can't happen to be from outside of a loop into a loop, but the other way around is possible. The only loop expressions that may be jumped out of are the pure loop (§9.6.2.4) and a **`while`** loop, which are not transformed as comprehensions into method calls.

9.6.4.4 Continuations

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

Definitions. The following code shows how a function that create unlimited continuations might be defined.

```
protocol Continuation [-A, +B] extends Function_1[A, B] {...}
protocol Unlimited_Continuation [-A] extends Continuation[A, Unit] {...}
def call/cc [A, B <: A] (&ctx: Unlimited_Continuation[A] -> B?): A end
```

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, Unit]` 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 9.6.2 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 (§9.6.4.1).

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) {
  def map [A1] (f: A -> A1): Shift[A1, B, C] := {
    Shift.new (k: Continuation[A1, B]) -> { cont((x: A) -> k(f(x))) }
  }
  def flat_map [A1, B1, C1 <: B] (f: A -> Shift[A1, B1, C1]):
      Shift[A1, B1, C] := {
    Shift.new (k: Continuation[A1, B1]) -> { f(x).cont(k) }
  }
```

```
}
def reset [A, C] (ctx: => @[CPS_Param[A, C]] A): C end
def shift [A, B, C] (cont: Continuation[A, B] -> C):
    @[CPS_Param[B, C] A end
annotation CPS_Param [-B, +C] {...}
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:

```
val 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 (§9.2.13) do with their passed blocks. Both conversions may happen dynamically at runtime. Therefore, the passed block is eventually a regular positional argument anyway.

Example 9.6.3 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:

```
val ctx := for {
  x in Shift.new {|cont: Integer -> Integer|
    cont(5)
  }
} yield (x + 1)
reset(ctx)
```

Note. Gear 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.

9.6.5 Throw, Catch & Ensure Expressions

Syntax:

```
Catch_Expr    ::= Catch_Expr1 | Catch_Expr2
Catch_Expr1   ::= 'begin' Block
                  'catch' [nl] Catch_Clauses
                  ['ensure' [nl] Block_Stat {semi Block_Stat}] 'end'
Catch_Expr2   ::= '{' Block '}'
                  'catch' '{' Catch_Clauses '}'
                  ['ensure' '{' Block_Stat {semi Block_Stat} '}']
Throw_Expr    ::= 'throw' Expr
Catch_Clauses ::= Catch_Clause {semi Catch_Clause}
                  [semi Else Catch_Block]
Catch_Clause  ::= 'when' Pattern [Guard] ('then' | semi) Catch_Block
Catch_Block   ::= Expr | {Catch_Stat semi} [Result_Expr]
Catch_Stat    ::= Block_Stat | Rethrow_Expr
Rethrow_Expr  ::= 'rethrow' [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 `{` b `}` **`catch`** h, where h is a handler pattern matching anonymous function (§11.4)

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

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

Let T be the expected type of the **`begin-catch`** expression. The block b is expected to conform to T. The handler h is expected to conform to type `Partial_Function[Throwable,` T`]`. The type of the **`begin-catch`** expression is the weak least upper bound (§5.10.2) of the type of b and the result type of h.

A **`begin-ensure`** expression `{` b `}` **`ensure`** `{` e `}` evaluates the block b. If evaluation of b does not cause any value to be thrown, the block e is evaluated. If any value is thrown during evaluation of e, the evaluation of the whole expression is aborted with the thrown value. If no value is thrown during evaluation of e, the result of b is returned as the result of the whole expression, unless e contains an explicit **`return`** (§9.6.4.1) – 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** (§9.6.4.1) – 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 { b } **catch** e_1 **ensure** { e_2 } is a shorthand for {{ b } **catch** e_1 } **ensure** { e_2 }.

9.6.5.1 Raise Expressions

Syntax:

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

A raise expression **raise** e is similar to **throw** e (§9.6.5), 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`.

9.6.5.2 Rescue Expressions

Syntax:

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

```
                ['ensure' semi Block_Stat {semi Block_Stat}] 'end'
Rescue_Expr2 ::= '{' Block '}'
                'rescue' [Pattern [Guard]] '{' Rescue_Block '}'
                {'rescue' [Pattern [Guard]] '{' Rescue_Block '}'}
                ['ensure' '{' Block_Stat {semi Block_Stat} '}']
Fun_Stats    ::= Block
                {'rescue' [Pattern [Guard]] semi Rescue_Block }
                ['ensure' semi Block_Stat {semi Block_Stat}]
Rescue_Block ::= {Rescue_Stat semi} [Result_Expr]
Rescue_Stat  ::= Block_Stat | Retry_Expr | Reraise_Expr
Retry_Expr   ::= 'retry' [label_name] [Cond_Modifier1]
Reraise_Expr ::= 'reraise' [Cond_Modifier1]
```

Rescue expression **rescue** h is similar to catch expression (§9.6.5), with two major differences: First, each **rescue** is followed by **when** clause; second, all **rescue** expressions in the same scope form together a handler h, where rules from catch expressions apply, only that h is expected to conform to type `Partial_Function[Raiseable,` T`]`, where T is the expected type of the whole **begin**-**rescue** expression.

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

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

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

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

9.7 Quoted Expressions

9.7.1 Quasi-quotation

Syntax:

```
Quasiquote_Expr ::= '<@' Any_Expr '@>'
Any_Expr        ::= Expr
                  | Block_Stat {semi Block_Stat}
                  | Template_Stat {semi Template_Stat}
                  | Alias_Expr
                  | Compilation_Unit
```

```
                  | Top_Stat_Seq
Splice_Expr     ::= '#{' Expr '}'
```

Quasi-quote expression "`<@` *e* `@>`" is basically a well-formed piece of Gear 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-quote expressions may optionally be interpolated, so that values or other nodes may be injected into the represented AST. There are the following ways to interpolate the AST:

- Interpolating expression "`#{` *e* `}`", where *e*'s value is expanded into the AST as is. If the expression would require parentheses around it in the source, then parentheses have to be around the interpolating expression.[20]
- 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* `}`".

A quasi-quote without any interpolation is equivalent to a quote (§9.7.2).

Quasi-quotes are useful in combination with macros (§14.3).

An interpolating (splice or unquote) expression (like "`#{` *e* `}`") appearing outside of a quasi-quote expression produces a compile-time warning and is replaced with *e* directly.

9.7.2 Quotation

Syntax:

```
Quote_Expr      ::= '<@@' Any_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]`.

9.8 Statements

Syntax:

[20]Note that the same delimiters are used to interpolate string literals.

```
Block_Stat     ::= Use
                 | {Annotation} ['implicit'] Def
                   [In_Sep Block_Stat ['end']]
                 | {Annotation} {Local_Modifier} Tmpl_Def
                 | Expr
                 | Alias_Expr
                 | Capture_Usage
                 | ()
                 | 'open' Stable_Id
                 | 'assert' Expr
Template_Stat  ::= Use
                 | {Annotation} {Modifier} Def
                   [In_Sep Block_Stat ['end']]
                 | Tmpl_Member [In_Sep Block_Stat ['end']]
                 | Tmpl_Ifc_Impl
                 | Tmpl_Ifc_Dcl
                 | {Annotation} [Opt_Req] {Modifier} Dcl
                 | {Annotation} {Modifier} Prop_Dcl
                 | {Annotation} {Modifier} Prop_Def
                 | 'inherit' Stable_Id [Cond_Modifier]
                 | 'include' Stable_Id [Cond_Modifier]
                 | ['inherit' 'as'] 'prepend' Stable_Id [Cond_Modifier]
                 | 'implements' Stable_Id [Cond_Modifier]
                 | Expr
                 | Alias_Expr
                 | ()
                 | Opt_Req [val_ids | symbol_ids]
                 | Access_Modifier [val_ids | symbol_ids]
                 | {Annotation} {Modifier} Constraint_Dcl
                 | {Annotation} {Modifier} Constraint_Def
                 | {Annotation} 'invariant' Invariant_Def
Fun_Stats      ::= [Fun_Stat {semi Fun_Stat}] Return_Expr
                 | Block
                   {'rescue' [Pattern [Guard]] semi Block }
                   ['ensure' semi Block_Stat {semi Block_Stat}]
Fun_Stat       ::= Block_Stat
Tmpl_Member    ::= {Annotation} {Modifier} ['member'] Def
Tmpl_Ifc_Impl  ::= 'interface' (Annot_Type - Nullable_Mod)
                   'with' Record_Extension {'and' Record_Extension}
                   'end' ['interface']
Tmpl_Ifc_Dcl   ::= 'interface' (Annot_Type - Nullable_Mod)
                   'end' ['interface']
Fun_Dec_Expr   ::= {Annotation} Dcl
                 | {Annotation} ['implicit'] Def
                 | ('transparent'
                   | 'opaque')
```

```
                  | 'pure'
                  | 'native' [Expr]
                  | ('immutable'
                    | 'mutable')
                  | ()
Alias_Expr      ::= 'alias' symbol_literal 'is' symbol_literal
                  | 'alias' id 'is' id
Capture_Usage   ::= 'use' id {',' id}
                    'as' R_Modifier
Constraint_Dcl ::= 'constraint' id ['as' Type] 'end' ['constraint']
Constraint_Def ::= 'constraint'
                    (id ['as' Type] [':='] Constraint_Block
                    | Alias_Expr)
Invariant_Def   ::= '{' Block '}' | 'do' Block 'end' ['invariant']
Opt_Req         ::= 'optional' | 'requires'
symbol_ids      ::= symbol_literal {',' symbol_literal}
```

Statements occur as parts of blocks and templates. Despite their name, they are actually generally expressions as well, except that for some statements, their value is not much of a use, i.e. use clauses, whose value is a **nil**, or the empty statement/expression, whose value is again **nil**.

Function statements is an umbrella term for a series of statements and expressions, so their effective value is more complex.

An expression that is used as a statement can have an arbitrary value type. An expression statement e is evaluated by evaluating e and discarding and releasing the result of the evaluation.

Block statements may be definitions, which bind local names in the block. The only modifier allowed in all block-local definitions is **implicit**. When prefixing a class or object definition, modifiers **abstract**, **final** and **sealed** are also permitted (§7.2).

Evaluation of a statement sequence entails evaluation of the statements in the order they are written. This behavior can be overridden for statement sequences in workflows (§9.2.13).

Statement can be an import via a use clause (§6.14), 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 (§9.3.5.9), 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 Gear 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 Gear VM implementation. Every Gear 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**) 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.

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.

Constraint definitions, of the form **constraint** *id* **as** T `:=` b, define argument-less methods (*id*), whose names are then available as local variables inside of constraint blocks (b). A type T can be used to specify explicitly the result type of such methods. The instance variables of the owning object that are read by such methods are observed for changes, so that constrained types can be consistent. Constraints may be aliased. If at runtime the object changes in a way that breaks the constraint condition, then `Broken_Constraint` is raised upon next read of a variable typed with a constrained type.

If a parameter-less method of the same name as a constraint already exists, it is sufficient to just declare the constraint and let the pre-existing method implement it. Otherwise, a constraint declaration behaves exactly like a method declaration. On the contrary, neither method declaration, nor method definition alone create a constraint.

9.9 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 (§9.9.5) and view applications (§10.3), if necessary.

9.9.1 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 (§9.9.3) is applied to pick a unique member.

Type instantiation. An expression e of a polymorphic type

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

which does not appear as the function part or a type application is converted to a type instance of T by determining with local type inference (§9.9.4) instance types $T_1, \ldots, T_n$ for the type variables $a_1, \ldots, a_n$ and implicitly embedding e in the type application $e[T_1, \ldots, T_n]$ (§9.3.6).

Numeric widening. If e is of a number type which weakly conforms (§5.10.2) to the expected type, it is widened to the expected type.

Numeric narrowing. If the expected type has smaller range than the number type of e, but the value of e fits into the expected type, it is narrowed to the expected type.

Value discarding. If e is of some value type and the expected type is `Unit`, e is converted to the expected type by embedding it in the block `{ `e`; () }`.

View application. If none of the previous conversions applies, view applications are not disallowed by pragmas (implicitly they are allowed), and e's type does not conform to the expected type *et*, an attempt is made to convert e to the expected type with a view application (§10.3). 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 (§9.9.6). Otherwise, `member_not_found` is invoked on the receiver, whose implementation in `Object` is to raise an error.

9.9.2 Method Conversions

The following implicit conversions can be applied to methods which are not applied to some argument list.

Evaluation. A parameterless method m of type `() ->` T is always converted to type T by evaluating the expression to which m is bound.

Implicit application. If the method takes only implicit parameters, implicit arguments are passed following the rules of (§10.2).

Eta expansion. Otherwise, if the expected type *et* is a function type `(`Ts'`) ->` T', eta-expansion (§9.9.5) is performed on the expression e.

Empty application. Otherwise, if e is of a method type `()` $\mapsto$ T, it is implicitly applied to the empty argument list, yielding e`()`.

9.9.3 Overloading Resolution

If an identifier or selection e references several members of a class, the context of the reference is used to identify a unique member, if possible. The way this is done depends on whether or not e is used as a function. Note that even if overloaded resolution picks up a unique member, that

member still may not be applied in regard of the actual expected types of the function application. Let $\mathscr{A}$ be the set of members referenced by e. 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.

9.9.3.1 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 $(p_1 : T_1, \ldots, p_n : T_n)$ `->` b, the shape is `(Any, ..., Any) ->` *shape*(b), where `Any` occurs n times in the argument type.
- For a named argument $n : e$, the shape is `@[named_arg :`n`]` *shape*(e), which is an annotated type.[21]
- For all other expressions, the shape is `Nothing`.

Let $\mathscr{B}$ be the set of alternatives in $\mathscr{A}$ that are *applicable* (§9.3.5) to expressions $(e_1, \ldots, e_n)$ of types $(\textit{shape}(e_1), \ldots, \textit{shape}(e_n))$. 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 (§9.9.1).

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".

[21] This is different from e.g. Scala, since Gear 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 (§5.3.16) or non-trivial pattern[22], 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 9.9.1 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.

$$\begin{aligned} \mathit{weight}(A, B) &= \textstyle\sum_{i=1}^{m} \mathit{pweight}(A_i, B_i) \\ \mathit{pweight}(t, u) &= \mathit{cweight}(t, u) + \mathit{rweight}(t) \end{aligned}$$

$$\begin{aligned} \mathit{cweight}(t, u) &= \begin{cases} 1 & \text{if } t <: u \\ 0 & \text{otherwise} \end{cases} \\ \mathit{rweight}(t) &= \begin{cases} 1 & \text{unless } t \text{ is a variadic or a capturing named parameter} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

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.

If A and B share the same relative weight and A requires argument expressions to tuple conversion and B does not, then B's relative weight is taken to be higher than that of A, and vice versa.

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 9.9.2 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 (§9.9.3.3), 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.

[22]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 (§9.3.5.7) 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 (§9.9.3.3), 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 9.9.3 Assume the following overloaded function definitions:

```
def f (*x: Integer) end              (* 1. *)
def f (x: Integer) end               (* 2. *)
def f (x: Integer, y: Integer) end   (* 3. *)
```

In the application `f(1)`, there are two applicable alternatives in regard to both shape and argument counts – the first two. Applicability test gives relative weight to (1) over (2) of 1, since it has a repeated parameter, and relative weight to (2) over (1) of 2, therefore the second is chosen.

In the application `f(1, 2)`, there are again two applicable alternatives – the first and the last. Applicability test gives relative weight to (1) over (3) of 2, since it has a repeated parameter matching both arguments, and relative weight to (3) over (1) of 4, therefore the second is chosen.

In the application `f(1, 2, 3)`, there is only one applicable alternative (the first), which can be detected (as soon as) based on the shape of its argument expressions.

9.9.3.2 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*`]`.

9.9.3.3 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 (§9.9) 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 9.9.4 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.[23]

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.[24]

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 (§7.2) from the scope where e appears.

9.9.4 Local Type Inference

Local type inference infers type arguments to be passed to expressions of polymorphic type. Say e is of a type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n] \mapsto T$ and no explicit type arguments are given.

Local type inference converts this expression to a type application $e[T_1, \ldots, T_n]$. The choice of the type arguments $T_1, \ldots, T_n$ depends on the context in which the expression appears and on the expected type *et*.

Local type inference is not always able to infer all type arguments, and sometimes may even infer useless ones – in that cases, explicit type arguments are to be used to solve the problems, if even possible.

Case 1: Selections. If the expression appears as a prefix of a selection (or is a function application with empty arguments list; or is any of that as a prefix of multiple such selection) with a name x, then type inference is *deferred* to the whole expression $e.x$. That is, if $e.x$ has type S, it is now treated as having type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n] \mapsto S$, and local type inference is applied in turn to infer type arguments for $a_1, \ldots, a_n$, using the context in which $e.x$ appears. If x is itself polymorphic, the type parameters $a_1, \ldots, a_n$ are virtually added to type parameters of x as fresh names, just for the purpose of deferred type inference.

Case 2: Values. If the expression e appears as a value without being applied to some value arguments in a function application, the type arguments are inferred by solving a constraint system which relates the expression's type T with the expected type *et*. Without loss of generality we can assume that T is a value type; if it is a method type, we apply eta-expansion (§9.9.5) to convert it to a function type (which is a value type and we're home again). Solving means finding a substitution σ of types T_i for the type parameters a_i, such that all of the following conditions hold:

- None of inferred types T_i is a singleton type (§5.3.1).

[23] 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.

[24] This can be fixed e.g. by using typed expressions (§9.5.1).

- All type parameter bounds are satisfied, i.e. $\sigma L_i <: \sigma a_i$ and also $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The expression's type conforms to the expected type, i.e. $\sigma T <: \sigma et$.

It is an error if no such substitution exists. If several substitutions exist, local type inference will choose for each type variable a_i a minimal or a maximal type T_i of the solution space (where a maximal type is closer to `Any` and minimal is closer to `Undefined`). A *maximal* type T_i will be chosen if the type parameter a_i appears contravariantly (§6.9) in the type T of the expression. A *minimal* type T_i will be chosen in all other situations, i.e. if the type variable appears covariantly, non-variantly or not at all in the type T. We call such a substitution σ an *optimal solution* of the given constraint system for the type T.

Case 3: Methods. This case applies if the expression e appears in an application $e(d_1, \ldots, d_m)$. In that case T is a method type $(p_1 : R_1, \ldots, p_n : R_n) \mapsto T'$. Without loss of generality, we can assume that the result type T' is a value type; if it is a method type, we apply eta-expansion (§9.9.5) to convert it to a function type (which is a value type and we're home, yet again). One computes first the types S_j of the argument expressions d_j, using two alternative schemes. Each argument expression d_j is typed first with the expected type R_j, in which the type parameters $a_1, \ldots, a_n$ are taken as unknown type constants. If this fails for the particular argument expression[25], the argument d_j is typed instead with an expected type R'_j, which results from R_j by replacing every type parameter in $a_1, \ldots, a_n$ with *undefined*. If d_j is overloaded, a unique member is selected based on R_j, and if that fails, it is selected based on R'_j; it is an error if a unique member for d_j can not be selected. [26]

If e is a member with multiple parameter lists, then every following passed argument list is considered to be virtually a part of $d_1, \ldots, d_m$. Some argument lists may be missing, e.g. in case of partial application, and `Undefined` is then inferred for every type parameter that appears only in the missing parameter lists, so that it gets inferred again upon future application[27]. Multiple argument lists may be written in multiple forms, e.g. $e(d_1, \ldots, d_a)\ldots(d_l, \ldots, d_m)$, and $e(d_1, \ldots, d_a)$`.apply(...).apply(`$d_l, \ldots, d_m$`)` is accepted as well, as is the shorter form $e(d_1, \ldots, d_a)$`.(...).(`$d_l, \ldots, d_m$`)`.[28]

In a second step, type arguments are inferred by solving a constraint system, which relates the method's type with the expected type *et* and the argument types $S_1, \ldots, S_m$. Solving the constraint system means finding a substitution σ of types T_i for the type parameters a_i, such that all of the following conditions hold:

- None of inferred types T_i is a singleton type (§5.3.1).

[25] E.g., because the conformance can't be determined, or maybe d_j is overloaded and could not select a unique member with expected type R_j.

[26] Selecting a unique overloaded member here is likely to fail, if R_j is polymorphic and the member is overloaded on result type. Using a typed expression is a way to prevent that.

[27] But the type arguments that are inferred in this step are permanent and not inferred again upon future applications.

[28] This way, methods with multiple parameter lists are first-class citizens, unlike in Scala.

- All type parameter bounds are satisfied, i.e. $\sigma L_i <: \sigma a_i$ and also $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The method's result type T' conforms to the expected type, i.e. $\sigma T' <: \sigma et$.
- Each argument type weakly conforms (§5.10.2) to the corresponding formal parameter type (§9.3.5.2), i.e. $\sigma S_j <:_w \sigma R_j$ for $j = 1, \ldots, m$.
- If there is no constraint involving T_i, `Undefined` is inferred.

It is an error if no such substitution exists. If several solutions exists, an optimal one for the type T' is chosen.

All or parts of an expected type *et* may be undefined. The rules for conformance (§5.10.2) are extended to this case by adding the rule that for any type T, the following two statements are always true:

- *undefined* $<: T$
- $T <:$ *undefined*

It is possible that no minimal or maximal solution for a type variable exists, which is an error. Because "<:" is a partial order, it is also possible that a solution set has several optimal solutions for a type. In that case, Gear is free to pick any one of them.

Example 9.9.5 Consider the two methods, where `List` is covariant in its type parameter:

```
def cons [A] (x: A, xs: List[A]): List[A] := x ~> xs
def list_nil [B]: List[B] := List.Nil
```

and the definition

```
val xs := cons(1, list_nil)  .
```

The application of `cons` is typed with an undefined expected type. This application is completed by local type inference to `cons[Integer](1, list_nil)`. Here, one uses the following reasoning to infer the type argument `Integer` for the type parameter `A`:

First, the argument expressions are typed. The first argument `1` has type `Integer`, whereas the second argument `list_nil` is not overloaded, and is itself polymorphic. One tries to type `list_nil` with an expected type `List[A]`. This leads to the constraint system

```
List[?B] <: List[A]  ,
```

where we have labeled `?B` with a question mark to indicate that it is a variable in the constraint system. Because class `List` is covariant, the optimal solution of this constraint is a minimal type,

```
B := Nothing  .
```

This finishes one separate local type inferencing instance.

In the second step, one solves the following constraint system for the type parameter `A` of `cons`:

```
Integer <: ?A              (* for the parameter x *)
List[Nothing] <: List[?A]  (* for the parameter xs *)
List[?A] <: undefined      (* for the result type *)
```

The optimal solution of this constraint system is

```
A := Integer  ,
```

so `Integer` is the type inferred for `A`.

The optimal solution is found based on the following reasoning:

1. Solutions for the first constraint are types `Integer` and its supertypes.
2. Solutions for the second constraint are `Nothing` and all supertypes of `Nothing`, basically any type. Together with the first constraint, only those types from the solution of the first constraint are possible.
3. Solutions for the third constraint are all types. Together with the first two constraint, only those types from the solution of the first constraint are possible.
4. Therefore, the minimal type is chosen from the solution space, as `A` appears covariantly (§6.9) in the type T of the expression (which is `List[A]`), which is in turn `Integer`.

Example 9.9.6 Consider now the definition

```
val ys := cons("abc", xs)  ,
```

where `xs` is defined as type `List[Integer]` as before. In this case local type inference proceeds as follows.

First, the argument expressions are typed. The first argument `"abc"` is of type `String`. The second argument `xs` is first tried to be typed with expected type `List[A]`. This fails, because we can't tell whether `List[Integer]` is a subtype or supertype of `List[A]`, as we don't yet know what `A` is. Therefore, the second strategy is attempted and `xs` is typed with expected type `List[`*undefined*`]`. This succeeds and yields the argument type `List[Integer]`.

In a second step, one solves the following constraint system for the type parameter `A` of `cons`:

```
String <: ?A
List[Integer] <: List[?A]
List[?A] <: undefined
```

The optimal solution of this constraint system is

```
A := Object  ,
```

so `Object` is the type inferred for `A`, as it is the least upper bound of `String` and `Integer`. Notice that if any of those were types not descending from `Object`, `Any` would be inferred instead, as the least upper bound of any type.

9.9.5 Eta-Expansion

Eta-expansion converts an expression of a method type (not a function application) to an equivalent expression of a function type. It is especially useful to prevent re-evaluation of the expression's subexpressions, if the expression is passed using a by-name strategy. It proceeds in two steps.

First, one identifies the maximal subexpressions of e, let's say these are $e_1, \ldots, e_m$. For each of these, one creates a fresh name x_i. Let e' be the expression resulting from replacing every maximal subexpression e_i in e by the corresponding fresh name x_i. Second, one creates a fresh name y_i for every argument type T_i of the method, for $i = 1, \ldots, n$, using named arguments and parameters as defined by the method. The result of eta-expansion is then:

```
{
  val x1 := e1
  ...
  val xm := em
  (y1: T1, ..., yn: Tn) -> e'(y1, ..., yn)
}
```

Example 9.9.7 A few examples of eta-expansion, the original expression and eta-expanded below it:

```
(* expression: *)
&((1 .. 9).fold(z))
(* expands to: *)
{ val eta1 := z
  val eta2 := (1 .. 9)
  x -> eta2.fold(eta1)(x) }
```

9.9.6 Dynamic Member Selection

Gear defines a marker trait `Dynamic_Member_Selecting` that enables dynamic invocations rewriting without resorting to use error handling mechanisms to implement dynamic dispatch.

Instances x of this trait allow method invocations x`.method(args)` for arbitrary method `method` and argument lists `args`, as well as property accesses x`.property` for arbitrary property names `property`.

If an invocation is not implemented by x (i.e. if type checking fails and no other implicit conversion provides a way to proceed with the invocation), it is virtually rewritten according to the following rules:

```
foo.method("blah")      foo.apply_dynamic(:method)("blah")
foo.method(x: "blah")   foo.apply_dynamic_named(:method)((:x, "blah"))
foo.method(1, x: 2)     foo.apply_dynamic_named(:method)(
                            (nil, 1), (:x, "blah"))
foo.property            foo.select_dynamic(:property)
foo.property := 10      foo.update_dynamic(:property)(10)
foo.array(10)           foo.apply_dynamic(:array)(10)
foo.array(10) := 11     foo.select_dynamic(:array).update(10)(11)
```

Chapter 10

Implicit Parameters, Views & Multiple Dispatch

Contents

10.1 The Implicit Modifier

Syntax:

```
Local_Modifier ::= 'implicit'
Param_Clauses  ::= {Param_Clause} '(' 'implicit' Params ')'
```

Template members and parameters labeled with **`implicit`** modifier can be passed to implicit parameters (§10.2) and can be used as implicit conversions called views (§10.3).

If the member marked with **`implicit`** is a class, it makes the primary constructor of the class available for implicit conversions as a function. Such primary constructor may take exactly one non-implicit argument in its first parameter list.

Example 10.1.1 The following code defined an abstract class of monoids and two concrete implementations, `String_Monoid` and `Int_Monoid`. The two implementations are marked implicit and will be used throughout the following discussions.

```
abstract class Monoid [A] extends Semi_Group [A] {
  def unit: A end
  def add (x: A, y: A): A end
}
object Monoids {
  implicit object String_Monoid extends Monoid[String] {
    def unit: String := ""
    def add (x: String, y: String): String := x + y
  }
  implicit object Int_Monoid extends Monoid[Integer] {
    def unit: Integer := 0
    def add (x: Integer, y: Integer): Integer := x + y
  }
}
```

10.2 Implicit Parameters

An implicit parameter list `(implicit` $p_1, \ldots, p_n$`)` of a method marks the parameters $p_1, \ldots, p_n$ as implicit. A method or constructor can have at most one implicit parameter list, and it must be the last parameter list given.

A method with implicit parameters can be applied to arguments just like normal method. In this case the **`implicit`** label has no effect. However, if such a method misses arguments for its implicit parameters (determined by a missing consecutive function application – §9.3.5.7), such arguments will be automatically provided, if possible.

The actual arguments that are eligible to be passed to an implicit parameter of type T fall into two categories.

- First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition (§10.1) 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 (§6.14).
- If there are no eligible identifiers under the previous rule, then, second, eligible are also all **`implicit`** members of some object that belongs to the implicit scope of the implicit parameter's type, T.

The *implicit scope* of a type T consists of the type T itself and all classes and class objects that are associated with the type T. Here, we say a class C is *associated* with a type T, if it is a base class of some part of T. The *parts* of a type T are:

- If T is a compound type T_1 `with` ... `with` T_n, the union of the parts of $T_1, \ldots, T_n$, as well as T itself.
- If T is a parameterized type S`[`$T_1, \ldots, T_n$`]`, the union of the parts of S and $T_1, \ldots, T_n$.
- If T is a singleton type p`.type`, the parts of the type of p.
- If T is a type projection S`#`U, the parts of S as well as T itself.
- If T is a union type `union of (`T_1`; ...;` T_n`)`, the union of all types $T_1, \ldots, T_n$.
- In all other cases, just T itself.

If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of overloading resolution without any application (§9.9.3). If the parameter has a default argument and no implicit argument can be found, the default argument is used.

Example 10.2.1 Assuming the classes from Example 10.1.1, 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.

10.3 Views

Implicit parameters and method can also define implicit conversions called *views*. A *view* from type S to type T is defined by an implicit value, which has function type S `->` T, or `(()` `->` S`)` `->` T, or by a method convertible to a value of one of those function types.

Views are applied in the following situations.

1. If an expression e is of a type T, and T does not conform to the expression's expected type *et*. In this case an implicit v is searched, which is applicable to e and whose result type conforms to *et*, and this process makes T compatible with *et*. The search proceeds as in the case of implicit parameters, where the implicit scopes are the scope of T followed by the scope of *et*, searched in this order. If such a view is found, the expression e is converted to $v(e)$.

2. In a selection $e.m$ with e of a type T, if the selector m does not denote an accessible member of T, including restrictions imposed by access modifiers, i.e. the member m may actually exist in T. In this case, a view v is searched which is applicable to e and whose result contains an accessible member named m. The search proceeds as in the case of implicit parameters, where the implicit scope is just that of T. If 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 (§9.9.3) is applied if there are several possible candidates.[1]

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.

10.4 View & Context Bounds

Syntax:

[1]The overloading resolution here is for the case of function applications, and the shape takes into account just one argument and corresponding parameter pair.

```
Type_Param ::= (id | '_') [Type_Param_Clause]
               ['>:' Type] ['<:' Type]
               {'<%' Type} {':' Type}
             | '<' (id | '_') '>' ['<:' id]
```

A type parameter A of a method or a non-trait class may have one or more view bounds A `<%` T. In this case, the type parameter may be instantiated to any type S, which is convertible by an application of a view to the bound T.

A type parameter A of a method or a non-trait class may have one or more view bounds A `:` T. In this case, the type parameter may be instantiated to any type S for which *evidence* exists at the instantiation point that S satisfies the bound T. Such evidence consists of an implicit value with type T`[`S`]`.

A method or class containing type parameters with view or context bounds is treated as being equivalent to a method with implicit parameters, and if it already contains explicitly some implicit parameters, those are added right after the section of any positional parameters and before the section of any named parameters. Consider the case of a single parameter with a view and/or context bounds, such as:

def f `[`A `<%` T_1 `...` `<%` T_m `:` U_1 `:` U_n`]` `(`*ps*`):` R `:=` `...`

Then the method definition above is equivalent to

def f `[`A`]` `(`*ps*`)(`**implicit** v_1`:` A `->` T_1`, ...,` v_m`:` A `->` T_m`,`
w_1`:` U_1`[`A`], ...,` w_n`:` U_n`[`A`])`: R `:=` `...`

where the v_i and w_j are fresh names for the newly introduced implicit parameters. These parameters are called *evidence parameters*.

If a class or method has several view- or context-bounded type parameters, each such type parameter is expanded into evidence parameters in the order they appear and all the resulting evidence parameters are concatenated in one implicit parameter section. Since traits do not have constructor parameters, such translation is impossible for them.

If the type of a context-bound parameter uses the type parameter in its definition, then the translation is simple – it stays the same and the type argument is still possibly inferred. Such type is then called a *partially applied type*, although all type arguments are necessarily provided in the end. It is an error if the type is a parameterized type and it does not use the type parameter that it is context-bound with. Thus, a context bound A `:` T is in fact a shortcut for A `:` T`[`A`]`.

Example 10.4.1 The following example shows a method with a context-bound parameter using a partially applied type, and the resulting translation below it.

```
def f [T : T -> String](t: T) := ...
def f [T] (t: T)(implicit w1: Function_1[T, String]) := ...
```

10.5 Multi-Methods & Multiple Dispatch

Syntax:

```
Dcl ::= 'def' ['multi' | 'dispatch'] Fun_Dcl 'end' ['def']
      | ['multi' | 'dispatch'] 'message' Fun_Dcl 'end' ['message']
      | ['multi' | 'dispatch'] 'function' Fun_Dcl 'end' ['function']
      | ['multi' | 'dispatch'] 'operator' Op_Dcl 'end' ['operator']
Def ::= 'def' ['multi' | 'dispatch'] Fun_Def 'end' ['def']
      | 'def' ['multi' | 'dispatch'] Fun_Alt_Def
      | ['multi' | 'dispatch'] 'method' Fun_Def 'end' ['method']
      | ['multi' | 'dispatch'] 'method' Fun_Alt_Def
      | ['multi' | 'dispatch'] 'function' Fun_Def 'end' ['function']
      | ['multi' | 'dispatch'] 'function' Fun_Alt_Def
      | ['multi' | 'dispatch'] 'operator' Op_Def 'end' ['operator']
      | ['multi' | 'dispatch'] 'operator' Op_Alt_Def
```

Gear 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 (§10.5.3).

10.5.1 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` (§5.8) affect the resolved overloaded alternative.[2]

Example 10.5.1 Multiple dispatch with `Dynamic` type, Asteroids colliding with Spaceships.

```
object program {
  abstract class Thing extends Object {}
  class Asteroid extends Thing {}
  class Spaceship extends Thing {}

  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
```

[2]This is similar to what C# since its version 4.0 does.

```
    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
    val a: Dynamic := x
    val b: Dynamic := y
    collide_with_impl(a, b)
  end method

  method run (*args: String): Unit
    val asteroid := Asteroid.new
    val spaceship := Spaceship.new
    collide_with(asteroid, spaceship)
    collide_with(spaceship, spaceship)
  end method
}
```

This code upon running `program.run` prints the following to the console:

```
Asteroid hits a Spaceship
Spaceship hits a Spaceship
```

10.5.2 Type Classes

Another approach to multiple dispatch involves implicit parameters and type classes. This is probably the most verbose approach.

Example 10.5.2 Multiple dispatch with type classes, matrices and vectors.

```
object program {
  class Matrix extends Object {}
  class Vector extends Object {}

  trait Mult_Dep[A, B, C] {
    message apply (a: A, b: B): C end
  }
  implicit object mmm
```

```
      extends Mult_Dep[Matrix, Matrix, Matrix] {...}
  implicit object mvv
      extends Mult_Dep[Matrix, Vector, Vector] {...}
  implicit object mim
      extends Mult_Dep[Matrix, Integer, Matrix] {...}
  implicit object imm
      extends Mult_Dep[Integer, Matrix, Matrix] {...}

  method multiply [A, B, C]
      (a: A, b: B)(implicit dep: Mult_Dep[A, B, C]): C
    dep.apply(a, b)
  end method

  method run (*args: String): Unit
    (* ok: *)
    val r1: Matrix := multiply(
        (Matrix with {}).new, (Matrix with {}).new)
    val r2: Vector := multiply(
        (Matrix with {}).new, (Vector with {}).new)
    val r3: Vector := multiply(
        (Matrix with {}).new, 2)
    val r4: Matrix := multiply(
        2, (Matrix with {}).new)

    (* error, no implicit value found for type
         Mult_Dep[Matrix, Vector, Matrix]: *)
    val r5: Matrix := multiply(
        (Matrix with {}).new, (Vector with {}).new)
  end method
}
```

10.5.3 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.

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),[3] 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.

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[4] 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 10.5.3 Multiple dispatch with dynamic value dispatch, Asteroids colliding with Spaceships.

```
object program {
  abstract class Thing extends Object {}
  class Asteroid extends Thing {}
  class Spaceship extends Thing {}

  def dispatch collide_with (a: Thing, b: Thing): Unit
    (a.type, b.type)
  end def

  def multi collide_with (Asteroid, Asteroid)(a, b)
    print_line "Asteroid hits an Asteroid"
  end def

  ...

  method run (*args: String): Unit
```

[3] 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.

[4] Including extra properties, e.g. “by-name” status.

```
    val asteroid := Asteroid.new
    val spaceship := Spaceship.new
    collide_with(asteroid, spaceship)
    collide_with(spaceship, spaceship)
  end method
}
```

Example 10.5.4 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.

```
object program {
  def dispatch area (obj: Map[Symbol, Any]): Real
    obj(:shape)
  end def

  def multi area (:Rectangle)(obj)
    obj(:width) * obj(:height)
  end def

  def multi area (:Circle)(obj)
    Math.π * obj(:radius) ^ 2
  end def

  def run (*args: String): Unit
    val rect := %{
        :shape => :Rectangle
        :width => 4 [<cm>]
        :height => 13 [<cm>]}
    val circle := %{
        :shape => :Circle
        :radius => 12 [<cm>]}
    print_line area(rect) (* 52 cm^2 *)
    print_line area(circle) (* 452.38... cm^2 *)
    print_line area(%{}[Symbol, Any]) (* error *)
  end def
}
```

Chapter 11

Pattern Matching

Contents

11.1 Patterns

Syntax:

```
Pattern        ::= Pattern1 {'|' Pattern1}
Pattern1       ::= ['implicit'] var_id ':' Type_Pat
                 | '_' ':' Type_Pat
                 | ['implicit'] var_id ['@' Pattern3]
                 | Pattern3
Pattern2       ::= var_id ['@' Pattern3]
                 | Pattern3
Pattern3       ::= Simple_Pattern {id Simple_Pattern}
Simple_Pattern ::= '_'
                 | var_id
                 | Literal - Collection_Literal
                 | ['.'] Stable_Id
                 | ['.'] Stable_Id '(' [Extractions] ')'
                 | '(' Extractions ')'
                 | '(' [Patterns] ')'
                 | '%[' Pattern {semi Pattern} [semi '*' (var_id | '_')]
                   {semi Pattern} ']'
                 | '%[' '*' (var_id | '_') {semi Pattern} ']'
                 | '%[|' Pattern {semi Pattern} [semi '*' (var_id | '_')]
                   {semi Pattern} '|]'
                 | '%[|' '*' (var_id | '_') {semi Pattern} '|]'
                 | '%{' Dict_Key '=>' Pattern
                   {semi Dict_Key '=>' Pattern}
                   [semi '**' (var_id | '_') [':' Type]] '}'
                 | '%{' '**' (var_id | '_') [':' Type] '}'
                 | '%(' Pattern {semi Pattern} [semi '*' (var_id | '_')]
                   {semi Pattern} ')'
                 | '%(' '*' (var_id | '_') {semi Pattern} ')'
                 | Pattern '&' Pattern
                 | Record_Pattern
Patterns       ::= Pattern {',' Pattern}
Extractions    ::= Pos_Patterns {',' Named_Patterns} [Block_Pattern]
                 | Named_Patterns [Block_Pattern]
                 | Block_Pattern
Pos_Patterns   ::= Pos_Pattern {',' Pos_Pattern} [',' Rest_Pattern
                   {',' Pos_Pattern}]
                 | Rest_Pattern {',' Pos_Pattern}
Pos_Pattern    ::= Pattern
Rest_Pattern   ::= '*' var_id [':' Type_Pat]
                 | '*' '_' [':' Type_Pat]
Named_Patterns ::= Named_Pattern {',' Named_Pattern} [',' Capt_Pattern]
                 | Capt_Pattern
```

```
Named_Pattern  ::= NPattern1 {'|' NPattern1}
NPattern1      ::= ['implicit'] ['~'] var_id var_id ':' Type_Pat
                 | ['implicit'] '~' var_id ':' Type_Pat
                 | var_id '_' ':' Type_Pat
                 | ['implicit'] var_id ['@' NPattern2]
NPattern2      ::= ['~'] var_id Simple_Pattern {id Simple_Pattern}
                 | '~' var_id
Capt_Pattern   ::= '**' var_id [':' Type_Pat]
                 | '**' '_' [':' Type_Pat]
Block_Pattern  ::= '&' var_id [':' Function_Type]
                 | '&' ':' Function_Type
Dict_Key       ::= Simple_Expr1
                 | (Literal - Collection_Literal)
                 | '_'
Record_Pattern ::= 'record' '{' Field_Patterns '}'
Field_Patterns ::= Field_Pattern {semi Field_Pattern} [semi '_']
                 | '_'
Field_Pattern  ::= Stable_Id '=>' Pattern
```

11.1.1 Variable Patterns

Syntax:

```
Simple_Pattern ::= '_'
                 | var_id
```

A variable pattern x is a simple identifier which starts with a lower case letter. It matches any value and binds the variable name to that value. The type of x is the expected type of the pattern as given from the outside. A special case is the wild-card pattern "_", which is treated as if it was a fresh variable on each occurence, and which does not bind itself to the value, i.e., it is alone equivalent to the **else** clause of `When_Clauses`.

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`.

11.1.2 Typed Patterns

Syntax:

```
Simple_Pattern ::= '_' ':' Type_Pat
                 | var_id ':' Type_Pat
Type_Pat       ::= Type
```

A typed pattern $x\colon\ T$ consists of a pattern variable x and a type pattern T. The type of x is the type T, where each type variable and wildcard is replaced by a fresh, unknown type. This pattern

matches any value matched by the type pattern T (§11.2), and it binds the variable name to that value (unless the variable name is "_").

A typed pattern contributes the type T to type of parameter, when used in function parameter list.

11.1.3 Pattern Binders

Syntax:

```
Pattern2 ::= var_id '@' Pattern3
```

A pattern binder x `@` p consists of a pattern variable x and a pattern p. The type of the variable x is the type T resulting from the pattern p. This pattern matches any value v matched by the pattern p, provided the type of v is also an instance of T, and it binds the variable name to that value.

A pattern binder contributes the type of the pattern p to type of parameter, when used in function parameter list.

Example 11.1.1 In the following example, `person` binds to the whole `Person` object.

```
def f (someone: Person) := match someone
  when person @ Person('John Galt', _, _) then ...
end match
```

11.1.4 Literal Patterns

Syntax:

```
Simple_Pattern ::= Literal - Collection_Literal
```

A literal pattern L matches any value that is equal (in terms of `=`) to the literal L. The type of L must conform to the expected type of the pattern. Literal kinds that are considered legal with this pattern are all non-collection literals.

A dictionary pattern contributes the type of the literal to type of parameter, when used in function parameter list.

11.1.5 List Patterns

Syntax:

```
Simple_Pattern ::= '%[' Pattern {semi Pattern} [semi '*' (var_id | '_')]
                   {semi Pattern} ']'
                 | '%[' '*' (var_id | '_') {semi Pattern} ']'
```

Remember from (§11.1.4) how we said that collection literals are not allowed as patterns? Well, we lied, sort of. Collection literals, or at least the syntax that uses their boundary tokens, is used for list patterns (here), array patterns (§11.1.6), dictionary patterns (§11.1.7) and bag patterns (§11.1.8).

The list pattern enables values of `List_Like` type to be decomposed into a number of elements. The list pattern itself always matches only lists of a specific number of elements, and may match lists of variable number of elements, where all the extra elements are extracted into a new sub-list into a variable that is prefixed with `*` inside the pattern (provided that it has a name – in case of name "_", the sub-list is not created, but discarded).

A list pattern contributes `List_Like[`T`]` to type of parameter, when used in function parameter list, where T is inferred from its sub-patterns automatically, or `Any`.

Example 11.1.2 An example of a list pattern.

```
let list_size (list) :=
  match list
  when %[] then 0
  when %[_] then 1
  when %[_; _] then 2
  when %[_; _; _] then 3
  otherwise list.size
  end match
```

Note that in the example, the type that would be inferred for `list` is `List_Like[Any]`.

11.1.6 Array Patterns

Syntax:

```
Simple_Pattern ::= '%[|' Pattern {semi Pattern}
                     [semi '*' (var_id | '_')]
                     {semi Pattern} '|]'
                 | '%[|' '*' (var_id | '_') {semi Pattern} '|]'
```

The array pattern enables values of `Array_Like` type to be decomposed into a number of elements. The array pattern itself always matches only arrays of a specific number of elements, and may match arrays of variable number of elements, where all the extra elements are extracted into a new sub-array into a variable that is prefixed with `*` inside the pattern (provided that it has a name – in case of name "_", the sub-array is not created, but discarded).

An array pattern contributes `Array_Like[`T`]` to type of parameter, when used in function parameter list, where T is inferred from its sub-patterns automatically, or `Any`.

Example 11.1.3 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 { size >= 1 and size <= 3 }`.

11.1.7 Dictionary Patterns

Syntax:

```
Dict_Key       ::= Simple_Expr1
                 | (Literal - Collection_Literal)
                 | '_'
Simple_Pattern ::= '%{' Dict_Key '=>' Pattern
                     {semi Dict_Key '=>' Pattern}
                     [semi '**' (var_id | '_') [':' Type]] '}'
                 | '%{' '**' (var_id | '_') [':' Type] '}'
```

The dictionary pattern enables values of `Dictionary` type to be decomposed into a number of elements. The dictionary pattern itself always matches only dictionaries of a specific number of elements, and may match dictionaries of variable number of elements, where all the extra elements are extracted into a new sub-dictionary into a variable that is prefixed with `**` inside the pattern (provided that it has a name – in case of name "_", the sub-dictionary is not created, but discarded).

The matching is defined as follows:

- An element from the matched dictionary matches if its key is equal to the key required by the sub-pattern, and the value matches the value pattern of the sub-pattern.
- The sub-pattern form k `=>` p requires the element's key to be equal to k, and the value to be matched by p. The pattern p may bind some variables using pattern binders.
- The sub-pattern form `_ =>` p ignores the element's key, and the value is required to be matched by p. Again, p may bind some variables using pattern binders.
- The sub-pattern form `**` n matches all remaining elements of the dictionary in a sub-dictionary parameterized with the types of the remaining keys and values. This sub-pattern may optionally contain a type pattern, constraining the match on values that are of the given type (not the keys).

- More sub-pattern forms may be added in the future versions as needed.

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`.

11.1.8 Bag Patterns

Syntax:

```
Simple_Pattern ::= '%(' Pattern {semi Pattern} [semi '*' (var_id | '_')]
                     {semi Pattern} ')'
                 | '%(' '*' (var_id | '_') {semi Pattern} ')'
```

The bag pattern enables values of `Bag` type to be decomposed into a number of elements. The bag pattern itself always matches only bags of a specific number of elements, and may match bags of variable number of elements, where all the extra elements are extracted into a new sub-bag into a variable that is prefixed with `*` inside the pattern (provided that it has a name – in case of name "_", the sub-bag is not created, but discarded).

An important thing to know about bag patterns is that the order in which elements are matched is irrelevant, therefore the pattern matching needs to match all unmatched elements left in the bag with each next sub-pattern. Iff the matched bag is sorted[1], only then the order in which elements are possibly matched is determinable. Otherwise, it could be any order, even orders that make no apparent logical sense. And also – searching the bag over and over for a matching element increases the complexity of the pattern match, which is short-circuited only when the matched bag has less than the number of needed elements.

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`.

11.1.9 Record Patterns

Syntax:

```
Simple_Pattern ::= Record_Pattern
Record_Pattern ::= 'record' '{' Field_Patterns '}'
Field_Patterns ::= Field_Pattern {semi Field_Pattern} [semi '_']
                 | '_'
Field_Pattern  ::= Stable_Id '=>' Pattern
```

A record pattern enables values of record types to be decomposed into a number of record fields.

The record pattern `record { `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 $i = 1, \ldots, n$.

[1]This guarantee has to be made by the actual type of the matched bag.

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[2]: $T.f$, where T is a path to the record type, and f is the field.

The record pattern **record** { 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.[3]

11.1.10 Stable Identifier Patterns

Syntax:

```
Simple_Pattern ::= ['.'] Stable_Id
```

A stable identifier pattern is a stable identifier r (§5.2). The type of r must conform to the expected type of the pattern. The pattern matches any value v, such that $r = v$.

To resolve the syntactic overlap with a variable pattern (§11.1.1), a stable identifier pattern may not be a simple name starting with a lower case letter. However, it is possible to enclose such a variable or method name in backquotes, then it is treated as a stable identifier pattern.

A stable identifier pattern and the following extractor patterns contributes the type of the stable identifier to type of parameter, when used in function parameter list.

Example 11.1.4 Consider the following function definition:

```
def f (x: Integer, y: Integer) := match x
  when y then ...
end match
```

Here, `y` is a variable pattern, which matches any value, namely here it would bind simply to `x`. If we wanted to turn the pattern into a stable identifier pattern, this can be achieved as follows:

```
def f (x: Integer, y: Integer) := match x
  when `y` then ...
end match
```

Now, the pattern matches the `y` parameter of the enclosing function `f`. That is, the match succeeds only if the `x` argument and the `y` argument of `f` are equal.

11.1.11 Target Type Patterns

Syntax:

[2]The first field should be used for this.

[3]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.

```
Simple_Pattern ::= '.' Stable_Id
                 | '.' Stable_Id '(' [Extractions] ')'
```

Target type patterns are not patterns by themselves, but rather render a subset of possible patterns.

The key difference is the dot "." that introduces the stable id – it signals the compiler that the stable identifier that follows is to be searched not in the scope, but rather in the type of the value that the pattern is being matched against.

A target type pattern can't contribute to type of parameter, when used in function parameter list, because there would be no target type to base the stable identifier search in. When used in a let binding in a workflow, the translated version uses the statically known target type from the binding, thus it must be known during compilation.

Note. Due to dynamic behaviour of Gear, the stable id might not refer to a statically-known entity in the type of the matched value, in which case the runtime has to find the referenced entity itself. During compilation, the entity is bound at compile time to the version of the value's type known at compile time, so if that version changes, runtime has to switch to the dynamic behaviour, and cache the result as appropriate.

11.1.12 Constructor Patterns

Syntax:

```
Simple_Pattern ::= ['.'] Stable_Id '(' [Extractions] ')'
```

A constructor pattern is of the form $c(p_1, \ldots, p_n)$, for $n \geq 0$. It consists of a stable identifier c, followed by element patterns $p_1, \ldots, p_n$. The constructor c is a simple or qualified name which denotes a case class (§7.3.5). If the case class is monomorphic, then it must conform to the expected type of the pattern, and the formal parameter types of c's primary constructor (§7.3.2) are taken as the expected types of the element patterns $p_1, \ldots, p_n$. If the case class is polymorphic, then its type parameters are instantiated so that the instantiation of c conforms to the expected type of the pattern, unless the type arguments are already given. These types of the formal parameter types of c's primary constructor are then taken as the expected types of the component patterns $p_1, \ldots, p_n$. The pattern matches all objects created from constructor invocations $c(p_1, \ldots, p_n)$, where each element pattern p_i matches the corresponding value v_i. Any extra parameter sections of c's primary constructor do not affect this behavior. The pattern matching is done using the same approach as the extractor pattern (§11.1.14), since case classes have the same `unapply` methods as used in it.

A special case arises when c's formal parameter types contain a variadic parameter, or when c's formal parameter types contain a purely named parameter. This is further discussed in (§11.1.14) for the corresponding `unapply`-family of methods, and (§11.1.15) for the pattern matching behaviour.

If the primary constructor of a case class has any optional parameters (§6.10.6.2), then those may be skipped using a wildcard pattern "_". The same "trick" can be used in extractor patterns for any unwanted extracted values.

11.1.13 Tuple Patterns

Syntax:

```
Simple_Pattern ::= '(' [Patterns] ')'
```

A tuple pattern `(`$p_1, \ldots, p_n$`)` is an alias for the constructor pattern `Tuple_`n`(`$p_1, \ldots, p_n$`)`, where $n \geq 2$. The empty tuple `()` is the unique value of type `Unit`. Sub-patterns $p_1, \ldots, p_n$ may be named, in which case mapping rules from (§11.1.15) apply.

Note. A "tuple pattern" containing just one sub-pattern is not a tuple pattern, but a grouped pattern (§11.1.19).

11.1.14 Extractor Patterns

Syntax:

```
Simple_Pattern ::= ['.'] Stable_Id '(' [Extractions] ')'
```

An extractor pattern x`(`$p_1, \ldots, p_n$`)`, where $n \geq 0$, is of the same syntactic form as a constructor pattern. However, instead of a case class, the stable identifier x denotes an object which has a member method named `unapply` or `unapply_sequence` that matches the pattern.

An `unapply` method in an object x *matches* the pattern x`(`$p_1, \ldots, p_n$`)` if it takes exactly one argument and one of the following applies:

- $n = 0$ and `unapply`'s result type is `Boolean`. In this case the extractor pattern matches all values v for which x`.unapply(`v`)` returns **`yes`**.
- $n = 1$ and `unapply`'s result type is `Option[`T`]`, for some type T. In this case, the only argument pattern p_1 is typed in turn with expected type T. The extractor pattern matches then all values v for which x`.unapply(`v`)` returns a value of form `Some(`v_1`)`, and p_1 matches v_1.
- $n > 1$ and `unapply`'s result type is `Option[(`$T_1, \ldots, T_n$`)]`, for some types $T_1, \ldots, T_n$. In this case, the argument patterns $p_1, \ldots, p_n$ are typed in turn with expected types $T_1, \ldots, T_n$. The extractor pattern matches then all values v for which x`.unapply(`v`)` returns a value of form `Some((`$v_1, \ldots, v_n$`))`, and each pattern p_i matches the corresponding value v_i.

An `unapply_sequence` method in an object x matches the pattern

x`(`$q_1, \ldots, q_a$`,` $p_1, \ldots, p_n$`,` $r_1, \ldots, r_b$`)` ,

if it takes exactly one argument and its result type is of the form

`Option[(`$T_1, \ldots, T_a$`, @[sequence] Sequence[`S`],` $T_{n+1}, \ldots, T_b$`)]`

(if $a = 0$ and $b = 0$, the type `Option[@[sequence] Sequence[`S`]]` is also accepted). This case is further discussed in (§11.1.15).

11.1.15 Pattern Sequences & Mappings

Syntax:

```
Simple_Pattern ::= ['.'] Stable_Id '(' [Extractions] ')'
```

A *pattern sequence* $p_1, \ldots, p_n$ appears in two contexts.

Pattern sequences in constructor patterns. First, in a constructor pattern c`(`$q_1, \ldots, q_a, p_1, \ldots, p_n, r_1, \ldots, r_b$`)`, where c is a case class, which has $a + 1 + b$ primary constructor parameters, with a variadic parameter (§6.10.6.3) of type $*S$ in the middle, so that it has generated an `unapply_sequence` method instead of `unapply`, and therefore the behaviour of extractor pattern may be applied, as defined below.

Pattern sequences in extractor patterns. Second, in an extractor pattern x`(`$q_1, \ldots, q_a, p_1, \ldots, p_n, r_1, \ldots, r_b$`)`, if the extractor object x has an `unapply_sequence` method with a result type conforming to

`Option[(`$T_1, \ldots, T_a$`, @[sequence] Sequence[`S`],` $T_{n+1}, \ldots, T_b$`)]` ,

but does not have an `unapply` method. The expected type for the pattern sequence is in each case the type S.

Presence of a sequence wildcard. The middle pattern in a pattern sequence $p_1, \ldots, p_n$ may be a *sequence wildcard*, which is either of the form `*var_id`, or `*_`. Each element pattern p_i is type-checked with S as expected type, unless it is a sequence wildcard. If a sequence wildcard is present, the pattern matches all values v that are `Sequence[`S`]`, where the sequence starts with elements matching patterns $p_1, \ldots, p_{d-1}$) and ends with elements matching patterns $p_{d+1}, \ldots, p_n$, where d is the index of the sequence wildcard inside the pattern sequence, and the sequence is of length at least $n - 1$[4]. If no sequence wildcard is present[5], the pattern matches all values v that are `Sequence[`S`]`, where the sequence is of length n and consists of elements matching patterns $p_1, \ldots, p_n$.

[4]The sub-sequence captured by the sequence wildcard may indeed be empty.

[5]Which case is not covered by syntax of pattern sequences, but solely by syntax of constructor patterns and extractor patterns, which both refer here.

Example 11.1.5 How to match a sequence with tail, using a pattern sequence.

```
def test_sequence (s: Sequence[Number])
  match s
  when Sequence(a, b, *tail) then do_something(a, b, tail)
  end match
end def
```

This pattern matches sequences of numbers of length at least 2 and does something with the extracted elements. Type of `tail` is `Sequence[Number]` and is of lengths from 0 to whatever size the operating system allows (it would not be very realistic to say "infinity" here, but theoretically, it would be ok).

Example 11.1.6 How to match first and last elements of a sequence, ignoring the middle, using a pattern sequence.

```
def test_sequence (s: Sequence[Number])
  match s
  when Sequence(first, *_, last) then do_something(first, last)
  end match
end def
```

This pattern matches sequences of numbers of length at least 2 and does something with the extracted elements.

Pattern names. Each pattern, which appears inside of a constructor pattern or an extractor pattern, may be given a name, so that the named parameter from a case class constructor, or a simple mapping from the extractor's `unapply` method may be used to define which patterns correspond to which parameter or key. The name for a pattern may be given in two forms. First, if the named pattern is a variable name p (maybe followed by a type binding), then the form is ~p, and the name is the same as the variable name. Second, if a different name is intended for the pattern p, or the pattern is not a variable name, then the form is $a\ \ p$, where a is the name of the pattern. Named patterns may never appear before any non-named patterns, including a sequence wildcard pattern.

Pattern mappings. Patterns are corresponding to elements of a tuple returned from `unapply` or `unapply_sequence` very much like arguments are corresponding to parameters:

1. Patterns without a name p_i are mapped in their order of appearance to elements of the tuple, i.

2. Remaining patterns with a name are mapped to named elements of the tuple.

3. If there are more named elements of the tuple left and there is a pattern prefixed with "`**`" (i.e. "`**`p_i"), then the remaining named elements are mapped to it, with a type of `Map[Symbol,` S`]`, where S is the union of types of all the remaining named elements. Such pattern is considered to be a named pattern, but does not include a pattern name in its syntax, and should appear as the last in a sequence of named patterns. The pattern prefixed with "`**`" may also optionally contain a type pattern, constraining the match to values that are of that type (not the keys).

4. It is an error if a pattern has no element in the tuple left to map to, or if the next remaining tuple element is not named. If the tuple has still more elements to map to, it simply does not match the enclosing extractor or constructor pattern, therefore, the patterns have to be exhausting. It is also an error if a pattern without a name maps to an element of the tuple that is purely named.

If any named elements of the tuple are intended to be ignored, simply use the pattern "a `_`", or, to ignore all remaining named elements of the tuple, use the pattern "`**_`".

Overloading. The `unapply` and `unapply_sequence` methods may be overloaded, in which case each overloaded alternative is tested for a pattern match. If one alternative matches, then the pattern matches. If no alternative matches, then the pattern does not match. If more than one alternative matches, the one that is strictly more times preferred (using declared preference) than any other alternative, that one is chosen. It is an error if no unique alternative could be chosen at this point.

11.1.16 Infix Operation Patterns

Syntax:

```
Pattern3 ::= Simple_Pattern {id Simple_Pattern}
```

An infix operation pattern p *op* q is a syntax sugar for the constructor or extractor pattern *op*`(`p`,` q`)`. The precedence, associativity and binding direction of operators in patterns is the same as in expressions (§9.3.10).

An infix operation pattern p *op* `(`$q_1, \ldots, q_n$`)` is a shorthand for the constructor or extractor pattern *op*`(`$p, q_1, \ldots, q_n$`)`.

11.1.17 Conjunction Patterns

Syntax:

```
Simple_Pattern ::= Pattern '&' Pattern
```

A conjunction pattern ("and" *pattern*) matches only if both patterns match. Moreover, only the first pattern in a sequence of conjunction patterns may bind variable names, but variables from

the other patterns have their scope extended to the following patterns. This behavior is unlike in pattern alternatives, which aren't allowed to bind variable names at all, due to the fact that each alternative may match a completely different structure, whereas a conjunction pattern matches on the same structure.

11.1.18 Pattern Alternatives

Syntax:

```
Pattern ::= Pattern {'|' Pattern}
```

A pattern alternative p_1 | ... | p_n, where $n \geq 2$, consists of a number of alternative patterns p_i. All alternative patterns are type checked with the expected type of the pattern. They may not bind variable names other than wildcards (which are discarded). The alternative pattern matches a value v if at least one of its alternatives matches v. Consequently, if a first such match is successful, the remaining patterns are not tested.

Non-normatively: Usually, each pattern alternative is a literal pattern, and therefore binding a variable name makes no sense, since the value that the pattern matching expression matches against is the already bound variable. Still, if needed, the pattern alternatives may be used together with a pattern binder (§11.1.3), which can be useful when the structure that is being matched contains union types.

11.1.19 Grouped Patterns

Syntax:

```
Simple_Pattern ::= '(' Pattern ')'
(* sub-syntax of: Simple_Pattern ::= '(' [Patterns] ')' *)
```

Patterns can be grouped together to achieve the desired associativity.

11.1.20 Regular Expression Patterns

Syntax:

```
Simple_Pattern ::= regexp_literal
```

A regular expression pattern p (*regexp pattern*) is a variant of literal pattern, designed to match `String_Like` values. Literally, the pattern p matches a value v, if v is of a type that conforms to `String_Like` and its contents match the regular expression. Moreover, sub-patterns bind to variable names of a name of the form `match_`n, where n is either the position of the sub-pattern (unless the sub-pattern is explicitly not captured), or the name of a named sub-pattern.

Regular expression patterns are not designed to match against algebraic data structures.

11.1.21 Irrefutable Patterns

A patter p is *irrefutable* for a type T, if one of the following applies:

1. p is a variable pattern (§11.1.1),
2. p is a typed pattern x`:` T' (§11.1.2), and T `<:` T',
3. p is a constructor pattern c`(`$p_1 \ldots p_n$`)` (§11.1.12), the type T is an instance of a class c, the primary constructor (§7.3) of type T has argument types $T_1, \ldots, T_n$, and each p_i is irrefutable for type T_i.

11.2 Type Patterns

Syntax:

```
Type_Pat ::= Type
```

Type patterns consist of types, type variables and wildcards. A type pattern T is of one of the following forms:

A reference to a class C, p`.`C, p`.type` or T`#`C. This type pattern matches any non-**`nil`** instance of the given class (therefore, it does match the empty tuple `()` with type `Unit`). Note that the prefix of the class, if it is given, is irrelevant for determining class instances, unlike in Scala.

The bottom type `Nothing` (with singleton instance **`nil`**) is the only type pattern that matches **`nil`** (only), but its preferable to match against `Option[`T`]` with implicit conversion of **`nil`** to object `None`.

A singleton type p`.type`. This type pattern matches only the value denoted by the path p (only the single value denoted by the path p, since **`nil`** is not matched).

A parameterized type pattern T`[`$a_1 \ldots a_n$`, <`$u_1 \ldots u_m$`>]`, where the a_i are type variable patterns or wildcards "`_`" and u_i are unit of measure kinds. This type pattern matches all values which match T for some arbitrary instantiation of the type variables and wildcards.

A compound type pattern T_1 **`with`** ... **`with`** T_n, where each T_i is a type pattern. This type pattern matches all values that are matched by each of the type patterns T_i, and in this sense it is equivalent to the pattern T_1 `&` ... `&` T_n.

A union type pattern `(`T_1 **`or`** ... **`or`** T_n`)`, where each T_i is a type pattern. The pattern has to be enclosed in parentheses, so that it does not get mistaken by the compiler for a pattern with the same syntax. This type pattern matches all values that are matched by at least one T_i.

An intersection type pattern `(`T_1 **`and ... and`** T_n`)`, where each T_i is a type pattern. The pattern has to be enclosed in parentheses, so that it does not get mistaken by the compiler for a pattern with the same syntax. This type pattern matches all values that are matched by every T_i.

Types are not subject to any type erasure (§5.11), so it is basically safe to use any other type as type pattern, unlike in Scala.

A *type variable pattern* is a simple identifier which starts with a lower case letter.

11.3 Pattern Matching Expressions

Syntax:

```
Match_Expr     ::= Pat_Match_Expr
Pat_Match_Expr ::= 'match' Simple_Expr1 Match_Body
Match_Body     ::= semi When_Clauses 'end' ['match']
                 | '{' When_Clauses '}'
When_Clauses   ::= When_Clause {semi When_Clause}
                   [semi Else Cond_Block]
When_Clause    ::= 'when' Pattern [Guard] ('then' | semi) Cond_Block
```

A pattern matching expression

`match` e **`{ when`** p_1 **`then`** b_1 **`... when`** p_n **`then`** b_n **`else`** b_{n+1} **`}`**

consists of a selector expression e and a number $n > 0$ of cases. Each case consists of a (possibly guarded) pattern p_i, a block b_i and optionally the default block b_{n+1}, if none of the patterns matched. Each p_i might be complemented by a guard **`if`** `e` or **`unless`** `e`, where e is a guarding expression, that is typed as `Boolean`. The scope of the pattern variables in p_i comprises the pattern's guard and the corresponding block b_i. If the following when clause **`when`** p_{i+1} **`then`** b_{i+1} is preceded by the keyword **`next`**, then the pattern variables in p_i do not comprise the block b_{i+1} and neither the pattern p_{i+1}.

Let T be the type of the selector expression e. Every pattern $e \in \{p_1 \ldots p_n\}$ is typed with T as its expected type.

The expected type of every block b_i is the expected type of the whole pattern matching expression. The type of the pattern matching expression is then the weak least upper bound (§5.10.4) 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 (§14) 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 (§7.2), the compilation of the pattern matching expression can emit warnings, which diagnose that a given set of patterns is not exhaustive, i.e. there is a possibility of a `No_Match` 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.

11.4 Pattern Matching Anonymous Functions

Syntax:

```
Block_Expr ::= '{' When_Clauses '}'
```

An anonymous function can be defined by a sequence of cases

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

which appears as an expression without a prior **match**. The expression is expected to be of a block type, unless it is expected to be a function type, in that case it is converted to `Function_`k`[`$S_1, \ldots, S_k,\ T$`]` automatically for $k \geq 1$, and for $k = 0$ the expression does not match the expected function type in function applications (§9.3.5).[6]

The expression is taken to be equivalent to the anonymous function:

[6]This is due to Gear not knowing the expected types in advance, but this anonymous function expression is able to match any non-empty arguments list, which is simply passed into the implicit pattern matching expression.

```
(x_1: S_1, ..., x_j: S_j) -> {
  match (x_1, ..., x_j) {
    when p_1 then b_1
    ...
    when p_k then b_k
    else b_{k+1}
  }
}
```

Here, each x_i is a fresh name. As was shown in (§9.2.18), this anonymous function is in turn equivalent to the following instance creation expression, where T is the weak least upper bound of the types of all b_i.

```
(Function_j[S_1, ..., S_j, T] with {
  def 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 match
}).new
```

Example 11.4.1 Here is a method which uses a fold-left operation `/:` to compute the scalar product of two vectors:

```
def scalar_product (xs: List[Double], ys: List[Double]) :=
  (0.0 /: xs.zip(ys)) {
    when (a, (b, c)) then a + b * c
  }
```

The when clauses in this code are equivalent to the following anonymous function:

```
(x, y) -> {
  match (x, y) {
    when (a, (b, c)) then a + b * c
  }
}
```

Note that the fold-left operation `/:` is an operator ending in a colon ":", and therefore right-associative, and therefore the expression is interpreted as specified in (§9.3.10.3) for right-associative operations:

```
{
  val x$ := 0.0;
  xs
    .zip(ys)
    .`/:`(x$)((x, y) -> {
      match (x, y) {
        when (a, (b, c)) then a.`+`(b.`*`(c))
      }
    })
}  .
```

Chapter 12

Units of Measure

Contents

12.1 Units of Measure

Syntax:

```
Const_Type_Def ::= Unit_Name 'is' ['abstract'] 'measure'
                   [semi Unit_Convs {semi Unit_Convs}]
Unit_Name      ::= id - Unit_Op ['extends' id - Unit_Op]
Unit_Convs     ::= Unit_Name ':=' Unit_Conv
Unit_Conv      ::= '(' Unit_Conv ')'
                 | Unit_Elem [Unit_Op Unit_Elem]
                 | Unit_Conv Unit_Op Unit_Conv
Unit_Elem      ::= number_literal | id |
Unit_Op        ::= '*' | '/' | '^'
```

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

Example 12.1.1 The following defines the measure `cm` (centimeter).

```
type cm is measure
end type
```

Example 12.1.2 The following defines the measure `ml` (milliliter) as a cubic centimeter (`cm ^ 3`).

```
type ml is measure
  ml := cm ^ 3
end type
```

Example 12.1.3 The following shows possible usage of abstract units of measure.

```
type distance is abstract measure
end type

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

type km extends distance is measure
  km := m * 1000
end type

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

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

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

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

Syntax:

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

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

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

Chapter 13

Top-Level Definitions

Contents

13.1 Compilation Units

13.1.1 Modules

Syntax:

```
Compilation_Unit ::= {'module' Module_Path semi [Top_Stat_Seq]}
                     Top_Stat_Seq
Top_Stat_Seq     ::= Top_Stat {semi Top_Stat}
Top_Stat         ::= {Annotation} {Modifier} Tmpl_Def
                   | Use
                   | Packaging
                   | Module_Object
                   | Expr
                   | ()
```

Module definitions are objects that have one main purpose: to join related code and separate it from the outside. Gear'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.

A compilation unit (a single source file) consists of a sequence of packagings, import clauses, and class and object definitions, which may be preceded (and should be preceded) by a module clause.

A compilation unit

module p_1
...
module p_n
stats

starting with one or more module clauses is equivalent to a compilation unit consisting of the packaging

module p_1
  ...
  module p_n
    stats
  end module
end module

Implicitly imported into every compilation unit are, in that order:

1. the module `Gear/Language`
2. the object `Gear/Language.Predef`

Members of a later import in that order hide members of an earlier import.

The implicitly added code looks like the following code listing, with all its implications:[1]

```
use @Root.Gear/Language.{_}
use @Root.Gear/Language.Predef.{_}
```

The vendor specified in module's `src/`**`module`**`.gear` 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`**`.gear` also make those modules available without fully qualifying their name in other source files of the same module.

[1]The `@Root` is actually redundant, as explained in (§13.1.4).

13.1.2 Packagings

Syntax:

```
Packaging  ::= 'module' Module_Path (Packaging1 | Packaging2)
Packaging1 ::= semi Top_Stat_Seq 'end' ['module']
Packaging2 ::= '{' Top_Stat_Seq '}'
```

A module is a special object which defines a set of member classes, objects and another modules. Like open templates (§7.1.1), 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 labeled **private**, it is visible only for other members in the same module.

Inside the packaging, all members of package p are visible under their simple names. This rule extends to members of the enclosing modules of p (that are of the same *vendor*). However, every other module needs to either import the members with a use clause (§6.14), or refer to it via its fully qualified name.

The special *vendor-less* `@Root` "module" can only be specified as the first element of each packaging name.

Example 13.1.1 Given the packagings

```
module An_Org/A {
  module B {
    ...
  }
}
module An_Org/C {}
module Another_Org/D {}
```

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.[2]

The fully qualified names of these modules are as follows:

- `An_Org/A`
- `An_Org/A.B` (§13.1.4)

[2]The packaging game is too strong for the module `D`.

- `An_Org/C`
- `Another_Org/D`

Notice how these fully qualified names do not use the `@Root.` path prefix, explained in (§13.1.4).

Selections $p.m$ from p as well as imports from p work as for objects. Moreover, unlike in Scala, modules may be used as values, instances of `Module` class, which shares some behavior with `Class` class. It is illegal to have a module with the same fully qualified name (minus the vendor parts) as a class or a trait.

Top-level definitions outside a packaging are assumed to be injected into the `Object` class directly, and therefore visible to each other without qualification. However, as `Object` is actually a simple name for the fully qualified name `@Root.Gear/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 `@Root.Gear/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.

13.1.3 Module Object

Syntax:

```
Module_Object ::= 'module' 'object' Trait_Tmpl_Env
```

A module object **`module object`** p **`extends`** t can specify some properties of the module object, add new traits to it, and adds the members of the template t to the module object p. There can be only one module object per module, but the module object definition is still an open template (§7.1.1). The module object has to have the leading template defined in a file named p`.gear` in the module's root directory, otherwise the module object has an implied empty template.

The module object should not define a member with the same name as one of the top-level objects or classes defined in module p. If there is a name conflict, it is an error.

The module object has also a special role in defining entry points of the module. An entry point is a method of the module object that can be invoked from the outside, to actually run the module as a program (§13.2).

13.1.4 Module References

Syntax:

```
Path ::= Module_Path
```

A reference to a module takes the form of a qualified identifier. Like all other references, module references are relative, i.e., a module reference starting in a name p will be looked up in the closest

enclosing scope that defines a member named p, and continue with the next closest enclosing scope, as long as the modules share the exact same vendor.

The special predefined name `@Root` (which is vendor-less) refers to the outermost "root module", which contains all top-level modules.

A nested module inherits the vendor from its directly enclosing module, unless there is no enclosing module, or the vendor is specified explicitly. If a module inherits the vendor, then it doesn't need to specify it again in its fully qualified identifier, as it is implied that nested modules will have the same vendor.

Example 13.1.2 Consider the following program:

```
module B {
  class C {}
}
module A.B {
  class D {
    val x := @Root.B.C.new
  }
}
```

Here, the reference `@Root.B.C` refers to class `B` in the top-level module `B`. If the `@Root~` prefix had been omitted, the name `B` would instead resolve to the module `A.B`, and, provided that the module does not also contain a class `C`, a runtime error would result (constant not found).

```
module B {
  class C {}
}
module A.B {
  class D {
    use @Root.B.{C as D} (* another way to solve it *)
    val x := D.new
  }
}
```

13.1.5 Module Units

Units are portions of modules that are compiled into separate bytecode files. Usually, there is one unit per module named `default`, however, with a **`pragma`** `Module_Unit :`n, where n is an identifier, all definitions from the source file from that pragma on are a part of a unit named n.

A module with only one unit has usage no different from a module with multiple units, the only difference is in the compilation output.[3]

[3]Therefore, hot code loading may select just a part of a module to be updated at runtime.

13.2 Programs

A *program* is a module that has 1 or more entry points. An entry point is a method of the module object that can be invoked from the outside, to actually run the module as a program. To mark a method as an explicit entry point, use the `entry` pseudo-modifier[4] before the method definition or declaration. An implicit entry point is a method with a name `main` and a method type `(Sequence[String]) ⟼ Unit`. A module does not need to have any entry points at all – that renders it a "library-only" module.

Example 13.2.1 The following example will create a hello world program by defining a module entry point in module `Test`.

Syntax:

```
module Example/Test
module object {
  entry def main (args: Sequence[String]) := {
    Console.print_line "Hello world!"
  }
}
```

This program can be started by the command

```
% gear Test
```

[4]Actually, it is implemented as a method of the class `Module`, therefore not a keyword, but IDEs may opt-in to highlight it as such, due to its importance.

Chapter 14

Annotations, Pragmas & Macros

Contents

Syntax:

```
Annotation     ::= '@[' Simple_Type [NB_Arg_Exprs] ']'
NB_Arg_Exprs   ::= Parens_Args {Parens_Args}
                 | Poetry_Args
Annotation_Def ::= 'annotation' Class_Def
Expr           ::= Pragma
Pragma         ::= 'pragma' Simple_Type [NB_Arg_Exprs]
Def            ::= 'def' 'macro' Fun_Def 'end' ['def']
                 | 'def' 'macro' Fun_Alt_Def
```

Annotations, pragmas & macros are a way to provide metadata to both the compiler and runtime of Gear, possibly affecting the resulting bytecode and abstract syntax trees.

14.1 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.

14.2 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.

14.3 Macros

Macros are a way to directly manipulate with abstract syntax trees. Unlike in languages such as C, macros in Gear are written using the same language. The only essential restriction here is that while compiling a Gear 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 (§9.7) 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
    - `Gear/Language.Reflection.Macros.Whitebox.Context` or
    - `Gear/Language.Reflection.Macros.Blackbox.Context`.
2. A parameter list with the parameters of the macro definitions, with the described parameter type translation applied.

Example 14.3.1 The following code is an example implementation of a simple **`assert`** macro.

```
(* import a blackbox context *)
use Gear/Language.Reflection.Macros.Blackbox.Context

(* define the macro *)
def macro assert (condition: Boolean, message: String_Like): Unit :=
    Asserts.assert_impl
```

```
(* implement the macro *)
object Asserts {
  def assert_impl
      (c: Context)
      (cond: c.Expr[Boolean], msg: c.Expr[String_Like]): Unit :=
    <@ unless #{cond}
         raise #{msg}
       else
         ()
       end @>
}
```

Macros can be applied as an intermediate step by IDEs, so that their resulting transformations could be viewed prior to proceeding with compilation.

14.3.1 Whitebox & Blackbox Macros

Blackbox macros are such macros that exactly follow their type signature, including the result type, and therefore can be treated as blackboxes. Their implementations are irrelevant to understanding their behavior. On the otherhand, *whitebox macros* do not necessarily follow their type signature, which they have, but only as an approximation, which may or may not be precise. Therefore, whitebox macros may be used to create e.g. type providers, 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.

14.3.2 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 15

Design Guidelines & Code Conventions

Contents

This chapter introduces the way programs *should* be written and laid out in Gear.

15.1 Introduction

15.1.1 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.

15.2 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 15.2.1 Such directories may look like:

```
/Gear/Language
/Example/Your_Module
    /source (* or `src` *)
        /gear
            /Some_Submodule
            /macros
                /some_macro.gearm
            /some_source_file.gear
```

```
            /module.gear
            /build.gear
            /module.dependencies.gear
            /module.dependencies.lock
        /cpp14 (* or whichever the Gear VM is implemented in *)
            /native_method.hpp
            /native_method.cpp
    /compiled (* or `com` *)
        /gear
            /Some_Submodule
            /Example/Your_Module.gearb
            /Example/Your_Module/native/ (* optimized code *)
            /Example/Your_Module.gearpsi
        /cpp14
            /native_method.a
    /vendor -> /usr/local/A_Gear_VM/modules
    /binaries (* or `bin` *)
        /an_executable.a
        /run_your_module (* some executable file
                            using #! to run a Gear VM *)
    /protocols (* or `pro` *)
        /Plugin.gearp
    /resources (* or `res` *)
        /logo.png
    /.gitignore
    /.git
```

Sources. A common name for this directory is `src`, or `source`. Inside of this directory, all Gear 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.

Gear source files have the suffix `.gear`. Readme files are not source files, having their name starting with any-case `readme`, followed by e.g. `.md` for a Markdown-formatted readme.

Gear source directory (`/src` or `/src/gear`) 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 Gear, is named `build.gear`, and can make use of other files as needed.

Macros should be placed in a `/macros` directory inside of a directory with Gear sources, so e.g. `/src/macros` or `/src/gear/macros`. Those files may also use other files as needed. A file that contains solely Gear macro definitions may be optionally named with suffix `.gearm`, but that is not a requirement.

A module file may be placed in a Gear sources directory, and is named **`module`**`.gear`. 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.gear` 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 Gear sources are.

Compiled files. A common name for this directory is `com`, or `compiled`.

Gear compiled files (bytecode) have the suffix `.gearb` for bytecode and `.gearpsi` for extracted Program Structure Information. Readme files might be copied into it, if needed. Note that Gear VM will refuse to execute module's entry point if the module is not compiled including all native method implementations.

Note that files compiled from other languages, if not bound to Gear environment via native method definitions, may be still used with `FFI` mechanisms.

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 Gear VM with a specific entry point.

Protocol files. A common name for this directory is `pro`, or `protocols`. It is supposed to hold extracted interface of the module for use in other modules outside of this module. Such files contain just **`interface`** and **`protocol`** definitions, which may result by translation from **`class`** and **`trait`** definitions, and also other type definitions that are not regular classes.

Gear protocol files have the suffix of `.gearp`, but their structure is identical to any other Gear source file.

Module builds that contain just the protocol files are an option to include in vendor directory.

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.

[1]Version Control System.

Vendor directory. A common name for this directory is `ven`, or `vendor`, the latter may be preferred. Such directory is supposed to hold vendor directories (which may be just symlinks to the actual installed vendor modules, or even just virtually displayed in IDE as links to the vendor modules), which in their leave paths have e.g. modules with only protocol files, so that compiler may bind to their identifiers and use that in building of PSI and for type-checking. A vendor directory may also be used to store the whole compiled vendor module, or be a place to download the module dependency on demand as needed and if possible, if the vendor module is not already present in user's system.

Dependencies for a module may be defined in the module file, along with version requirements and possibly a network path to download the module.

A path to a vendor directory follows the module name.

15.3 File Organization

A file in Gear 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).

Files longer than 1989 lines are cumbersome and should be avoided. Gear 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.

15.3.1 Gear Source Files

Each Gear 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 Gear 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 Gear 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).

15.3.2 Class Definition Organization

The following list describes the parts of a class definition (or an interface, a protocol, 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 Gear'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 (§7.3).

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 variables**
   These are variables whose names are prefixed with a double at-sign "`@@`". Their order should be in decreasing visibility (§7.2). Alternatively, if the class object has a separate definition using **`object`** keyword, that should be following the class definition in the same file, preserving the order defined in this list. This also applies to class instance methods.

7. **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 (§7.2). Furthermore, factory methods should precede all other methods.

[2]RoR stands for Ruby on Rails.

8. **Instance variables**
   These are variables whose names are prefixed with a single at-sign "`@`". Their order should be in decreasing visibility (§7.2).

9. **Auxiliary constructors**
   Their order should be in decreasing visibility (§7.2).

10. **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.

15.4 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.

15.4.1 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.

15.4.2 Line Wrapping

When an expression will not fit into a single line, break it according to the following general principles. Note though that Gear 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 15.4.1 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 15.4.2 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 15.4.3 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 *)
def some_method (an_arg: Integer, another_arg: Object,
                 yet_another: String,
                 and_still_another: Object): Unit := {
```

```
  ...
}

(* indent two tabs 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 15.4.4 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 15.4.5 Gear 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
```

15.5 Comments

Gear 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 Gear 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.

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.

15.6 Declarations

15.6.1 One Per Line

One declaration per line is recommended, since it encourages commenting, unless the declarations are a part of a pattern matching extraction.

Multiple declaration expressions on the same line using a semicolon to separate them should be avoided, and should definitely not mix variable and function declarations.

Example 15.6.1 The following shows preferred variable declarations and declarations to be avoided:

[3] As a convenience for those used to this syntax from other languages. Gear toolchain prefers the first version.

```
(* preferred *)
val level: Integer
val size: Integer

(* avoid *)
val level, size: Integer
val level: Integer; size: Integer
```

Example 15.6.2 Another acceptable alternative is to align the type names with spaces:

```
val level:         Integer
val size:          Integer
val current_entry: Object
```

15.6.2 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 – Gear provides the **`lazy`** modifier for variables, which takes care of that.

15.6.3 Initialization

Try to initialize every variable as soon as possible, or use the **`lazy`** modifier on it.

15.7 Class Definitions

When writing Gear classes, the following formatting rules should be followed:

- A space between a method name and the parenthesis of its first parameter list, if any.
- Open brace "`{`" appears at the end of the same line as the declaration statement.
- Open keyword **`begin`** appears on the next line to the declaration statement.
- Closing brace "`}`" or close keyword **`end`** appears on a line alone (or followed by the keyword appropriate for the closed term). The only exception is when the close brace or close keyword should end the declaration immediately (usually in method declarations or definitions of methods that have an empty body, thus returning a `Unit` value).
- The **`declare`** and **`begin`** keywords should appear alone on their lines.
- Members except instance variables and class instance variables are separated by a blank line.

15.8 Statements

15.8.1 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.

15.8.2 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.

15.8.3 Return Statements

A **`return`** statement with a value should not use parentheses unless required by the syntax.

Example 15.8.1 For instance:

```
return

return my_disk.size

return (builder
    .set_title "hello"
    .set_message "world"
    .build)
```

Chapter 16

Memory Models

Contents

Gear offers to its proper implementations these memory models:

- Automatic Reference Counting (§16.1)
- Garbage Collection (§16.2)

Each memory model has its pros and cons, discussed further in the following sections.

16.1 Automatic Reference Counting

A Gear implementation with *automatic reference counting* tracks reference counts (strong reference count, weak reference count) of all heap-allocated values. Whenever strong reference count drops to 0, the referenced value is destructed (by invoking destructors on it and maybe destructing other values it itself referenced), and its memory is reclaimed by the Gear VM. Therefore, specific time of value's destruction may be determined, and the runtime does not need to "pause the world" to perform any memory clean-up tasks, which could cause interrupts in the program's behaviour.

Such implementation does not need to support soft and phantom references.

16.2 Garbage Collection

A Gear implementation with *garbage collection* does not destruct values that are no longer referred to immediately, but does so at unspecified times in the future, by "pausing the world" to detect those values it can safely destruct.

Such implementation can and should support soft and phantom references.

Chapter 17

Automatic Inference

Chapter 18

Lexical Filtering

Chapter A

Gear Syntax Summary