The Onyx programming language

Vlad Faust

Version 0.0.0, August 23, 2020

Table of Contents

Foreword

NXSF (the Onyx Software Foundation) is an international non-profit organization with a mission to develop the Onyx programming language and its ecosystem. More information on NXSF can be found at https://nxsf.org.

This document is developed by the NXSF Onyx Standard Committee. Any feedback or questions on this document should be directed to the NXSF Onyx Standard Committee at onyx@nxsf.org.

A entity willing to become a member of the NXSF Onyx Standard Committee or any other NXSF Committee to obtain voting rights on developing documents should consult to the information found at https://nxsf.org/membership.

The procedures used to develop this document and those intended for its further maintenance are described in the NXSF Directives found at https://nxsf.org/directives.

This document follows the Semantic Versioning 2.0 scheme. Multiple major versions of this document may co-exist simultaneously, but every revision replaces previous minor and patch versions of this document.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. NXSF shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the NXSF list of patent declarations received found at https://nxsf.org/patents.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

Introduction

The Onyx programming language is a statically-typed general purpose programming language with the following design goals:

- Performance in mind.

Chapter 1. Scope

This specification describes the form and establishes the interpretation of programs written in the Onyx programming language. It describes:

- The representation of Onyx programs;
- The syntax and constraints of the Onyx language;
- The semantic rules for interpreting Onyx programs;
- The restrictions and limits imposed by a conforming implementation of Onyx.

This specification does not describe:

- The mechanism by which Onyx programs are transformed for use by a data-processing system;
- The mechanism by which Onyx applications are invoked for use by a data-processing system;
- The mechanism by which input data are transformed for use by a Onyx application;
- The mechanism by which output data are transformed after being produced by a Onyx application;
- The size or complexity of a program and its data that will exceed the capacity of any specific data- processing system or the capacity of a particular processor;
- All minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

Chapter 2. Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO 9000:2015, Quality management systems — Fundamentals and vocabulary
- ISO/IEC 9899:2018, Information technology — Programming languages — C
- IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic
- ISO/IEC 10967 (all parts), Information technology — Language independent arithmetic
- ISO/IEC 10646, Information technology — Universal Coded Character Set (UCS)

Chapter 3. Terms and definitions

For the purposes of this specification, the following definitions apply. Other terms are defined where they appear in italic type or on the left side of a syntax rule. Terms explicitly defined in this specification are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this specification are to be interpreted according to ISO/IEC 2382.1. Mathematical symbols not defined in this specification are to be interpreted according to ISO 80000-2.

Chapter 4. Symbols and abbreviated terms

Chapter 5. Conventions

This section describes conventions used in this specification.

This specification may borrow terms and practices from different standards *on how to write standards*, such as *The ISO/IEC Directives, Part 2*. However, it does not aim to fully comply with either, unless explicitly stated another.

Specification language

The specification is written in English language with Oxford spelling.

5.1. Specification units

Specification unit

> A specification unit is a paragraph, an example block, a syntax definition listing, a table or a figure.

A specification unit other than an example block may be either normative or informative. An example block is always informative.

A specification unit other than an example block is normative by default. The following notation is used for informative units:

This is an informative paragraph.

A single paragraph contains a single statement or multiple closely related or logically bound statements. To help clarify the contents of a paragraph, it may contain a number of examples.

A paragraph may not contain the same statements worded differently. In other words, a paragraph shall be written in the most unambiguous and shortest form.

5.2. Examples

An example contained in a specification unit other than an example block is called an inline example and inherits the normative status of the unit.

This example is informative.

An example block must not be indexed. Instead, it is considered related to the previous specification unit, or the containing section.

Chapter 6. Terms

Contents of a specification unit exist in the context of parent sections, recursively. For example, the term "specification unit" is contained in the same section as the current paragraph, thus considered already known.

A term written in *italic* with an adjacent index has an explicit definition in either this or another document at the index location. For example, the term *grapheme* [T1] may be found at index T1 of this document. The term *specification* [ISO 9000:2015 § 3.8.7]) should be looked up in the ISO document. And this *term* [1] definition is to be found in the footnotes.

6.1. Syntax

Syntax definitions may contain semantic definitions in comments, but these semantic definitions are NOT normative, and are for convenience use only.

[1] An example definition.

Chapter 7. Design

This section describes design rationale of the Onyx language and accompanying standards.

Infer as much as possible, unless ambiguous.

Multiple syntaxes to achieve the same task is appropriate in some cases.

It is good to have full-character, short-character and symbolic representation of something so a developer could choose between readability and code size. For example, `equals?`, `eq` and `==`.

The language is built on a finite set of simple rule blocks. A lesser set is better. For example, function and generic arguments follow the same semantics and syntax.

7.1. Annotations VS keywords

The less keywords are in the language, the better.

Reuse keywords as much as possible.

If an annotation is very often used, consider replacing it with a keyword. However, low-level annotations should not be keywords. For example, `@[Align]` and `@[Volatile]` are annotations and not variable modifiers.

Chapter 8. Syntax

This specification uses EBNF for language and other syntax definition, unless mentioned another.

The entire compiled syntax can be found in Appendix A, *Compiled Syntax*.

The specification extends EBNF with variables: a nonterminal may have a list of arguments prefixed with `$` with optional default values, wrapped in parentheses. Once referenced within the nonterminal production rule, an argument is immediately replaced with its value. Recursion and nested variables are allowed. A nonterminal with arguments must always be used with parentheses, even if there are no actual arguments.

Example 1. Using variables in EBNF syntax

```
foo($arg = "none") = "bar" | $arg;

a = foo();      (* Expands to `"bar" | "none";`)
b = foo("qux"); (* Expands to `"bar" | "qux";` *)
c = foo(b);     (* Expands to `"bar" | "bar" | "qux";` *)
```

Listing 1. Fundamental syntax

```
binary = "0" | "1";
octal = binary | "2" | "3" | "4" | "5" | "6" | "7" | "8";
digit = octal | "8" | "9";

(* Design rationale: hexadecimals are in upper-case to clearly
  distingusigh them from exponents and prefixes (`Q` and `D`
  are exceptions). Multiplier prefixes can not be used in
  hexadecimal literals, so they don't mix. *)
hex = digit | "A" | "B" | "C" | "D" | "E" | "F";

(* Any Unicode abstract character, including
  those consisting of multiple codepoints. *)
unicode = (? A Unicode character ?);
```

Chapter 9. Variables

A variable is a pointer to a region of memory containg a value.

9.1. Declaration

TODO: Declaring means setting an identifier and a type. Defining is giving it a meaningful value. Defining is done using the `:=` operator.

An Onyx variable always has an either impilcitly or explicitly defined storage.

Accessing an uninitialized variable is undefined behavior, hence unsafe.

It is possible to explicitly assign a variable to an `uninitialized T` value, where `T` is the variable type; such an assignment is unsafe. Such an explicitly uninitialized variable would be considered safely initialized in the later code.

```
let x = unsafe! uninitialized SInt32 # let x : SInt32 = unsafe! uninitialized # Ditto, the type is infered

puts(x) # OK from the point of a compiler view.
```

Prior to an access from a specialization, a variable must be explicitly declared.

9.2. Variable storage

A variable storage defines its location and lifetime boundaries.

There are three variable storage options: static, instance or local.

A variable declaration within a primitive, trait or enum requires an explicit `static` storage modifier.

Variables declared within a struct have implicit `instance` storage by default. An explicit `instance` or `static` storage modifier can be specified for struct variables.

9.3. Static variables

A static variable lifetime spans to the lifetime of the containing process. Therefore, a static variable is accessible at any point of the program execution.

A final static variable must be both declared and assigned to in the same statement.

A final static variable containing a constant or scalar object may be placed into the read-only executable section.

9.4. Instance variables

An instance variable exists in the state of a particular object instance. All instance variables are finalized after the instance is finalized, in undefined order.

A final instance variable must be defined at the moment of the containing object instance initialization. Therefore, it must be defined either in the definition statement or in any initializer having access to the variable.

Only structs can contain instance variable declarations.

9.5. Local variables

Local variables exist in the scope of a single function call. Once the function returns, its local variables are finalized.

A final local variable can have its definition separate from declaration.

Prior to read, a local variable must be explicitly defined.

Variables declared within functions have local storage by default.

9.6. Final variables

A final variable is declared using the `final` keyword.

A final variable can not be re-assigned to another object, but it can still contain a mutable, i.e. a partially changeable, object.

xcite:final-static-var-inline-assignment[]

xcite:final-static-const-var-readonly[]

xcite:final-instance-var-initialization[]

xcite:final-local-var-separate-def[]

A definition of a non-final static variable within a function is guaranteed to happen at most once per this variable specialization.

9.7. Non-final variables

9.8. Lifetime

Primitives and enums don't have lifetime, but classes do.

In the end of their lifetime, a variable is finalized.

9.8.1. Moving variables

Only a **local** variable can be moved.

It is hardly possible to detect use-after-move of a static or an instance variable, hence the restriction.

Only an lvalue can be moved.

A moved instance is an rvalue.

Moving a variable does not preserve previous its lifetime status: if disabled (for example, with `@setalive(var, false)`), it would be enabled again.

A moved variable is treated in the same way as an undefined one. It is thus possible to define a previously moved variable again.

Example 2. Moving variables

```
final a := Std::Twine("foo")
let b := unsafe! uninitialized Std::Twine

# If not disabling the lifetime, then `b` would be
# finalized  upon moving into it, which is dangerous
unsafe! @setalive(b, false)

b <- a # Moving `a` into `b`; `b` could've been finalized here
# a # Panic! Use-after-move

unsafe! @setalive(b, true) # Make `b` alive again
assert(b.upcase == "FOO")

a := Std::Twine("bar") # Re-defining `a`

# This function accepts an instance
# rather than a pointer to one.
def foo(twine : Std::Twine);

foo(b)    # OK, would call a copy-initializer
foo(<- b) # OK, move `b` instead of copying it
# b # Panic! Use-after-move
```

It is not possible to move a varible into another local variable definition, as it is hardly practical. Thus, there is only a copy-define operator `:=`, but no move-define.

Example 3. No move-defining

```
final a = Std::Twine("foo")
final b := a # OK, copy-define
# final c <- a # Panic! `c` is not defined yet
```

9.9. Assignment

An assignment to a variable is re-writing the memory region it is pointing to.

An assignment of a new value to a variable may be performed by either copying the value or moving it, depending on the assignment operator. The return value of an assignment operation itself may be either a copy of the new variable value or a moved instance of the old variable value, depending on the assignment operator.

Operator	Example	`b` is...	`r` is...
`=`	`final r := a = b`	**Copied**	A **copy** of `a`
`<<=`	`final r := a <<= b`	**Copied**	**Moved** old instance of `a`
`<-`	`final r := a <- b`	**Moved**	A **copy** of `a`
`<<-`	`final r := a <<- b`	**Moved**	**Moved** old instance of `a`

For consistency reasons, `<=` would be a better choice for simple assignment instead of `=`. But it is already taken by the "less than or equal" comparison operator, and `=` is preferable due to familiarity.

`<<` may be read as “push the old value from the variable”.

- `sval` (storage value): variables, dereferenced pointers
- `mval` (moved value): return values, `(← x)` results
- `rval` (right value (historical)): everything else

Operation	Allowed?	Example
`sval = sval`	Yes, implies copying	`x = shared`
`sval = rval`	Yes, implies moving	`x = Std::Shared(y)`
`sval = mval`	Yes, implies moving	`x = copy(shared)`
`sval <- sval`	Yes, implies moving	`x <- shared`
`sval <- rval`	Yes, implies moving (TODO: Do not allow?)	`x ← 42`
`sval <- mval`	Yes, implies moving (TODO: Do not allow?)	`x ← (← 42)`

A pair of defined variables of any storage can be swapped using the swap operator `<->`. Swapping implies simulatenous moving of variable values into each other, thus it does neither copy nor finalize nor lead to use-after-move behavior. The return value of a swap operation itself is a copy of a new value of the left operand.

TODO: Taking an address of an undefined variable is unsafe. It returns a pointer with defined storage, though.

Example 4. Swapping variables

```
let a := 1
let b := 2

let new_a := a <-> b

assert(new_a == a == 2)
assert(b == 1)
```

```
(*
  A variable declaration. In different contexts, optional items
  may become mandatory; for example, a final static variable
  must have its definition in-line.
*)
var_decl =
  (* In most cases, it's implicitly `def` *)
  ["def" | "decl" | "impl"],

  (* A mandatory accessibility modifier *)
  ("final" | "let" | "get" | "set"),

  (* An optional storage modifier, otherwise infered *)
  ["static" | "instance" | "local" | "undefstor"],

  (* An optional `var` keyword *)
  ["var"],

  (* A mandatory variable identifier *)
  id,

  (* An optional type restriction, otherwise infered *)
  [":" | "~", type_expr],

  (* An optional inline definition of the variable *)
  ["=", expr];
```

TODO: `let x, y : T === let x : T, y : T; let x : ?, y : T === let x : Undef, y : T` VS. `let x, y : T === let x : Undef, y : T`.

TODO: final uninitialized variables MUST not be placed in read-only memory, thus unsafe initialization of them is possible.

Chapter 10. Functions

Can not user-declare non-operators ending with `=` (e.g. `name=` is invalid).

10.1. Declaration

```
closure_arg_decl = (* TODO: *);

(* A closure declaration may be explicitly empty (e.g. `[]`). *)
closure_decl =
  "[",
  [closure_arg_decl, {",", closure_arg_decl}, [","]],
  "]";

(*
  ```
  ~> || body # Any closure, types infered
  ~> |[]| body # Explicitly empty closure
  ~> |[C] A : R|
  ```
*)

block_restriction = "=>", [block_proto];

type_restriction = WS, (":" | "~"), WS, type_expr;

(*
  For declarations, at least either identifier or restriction
  must be specified, whereas aliasing is not possible.

  For implementations, at least either of three must be specified.
*)
arg_decl =
  (* Optional alias, unapplicable to function declarations *)
  [id, NB, ":"],

  (* An optional argument identifier *)
  [id],

  (* An optional argument restriction *)
  [type_restriction | block_restriction];

(*
  An named argument declaration with restriciton.
  If type of the restriction is omitted, it is assumed to be concrete.
  So that, `(arg: T)` == `(arg: : T)` != `(arg: ~ T)` == `(arg ~ T)`.
*)
arg_decl =
  [id, NB, ":"], (* Optional argument alias *)
```

```
  [var_decl], (* Optional argument variable declaration *)
  [":" | "~"], id;

(*
  An anonymous argument declaration
  with restriction, e.g. `(: T)` != `(~ T)`.
*)
arg_decl = (":" | "~"), id;

fun_decl = "decl", id, "(", {arg_decl}, ")";
```

10.2. Storage

Function storage may be `static`, `instance` or `undefstor`. The latter is only applicable to function declarations with traits.

10.3. Mutability

Function mutability may be `mut`, `const` or `undefmut`. The latter is only applicable to function declarations with traits.

10.4. Manipulating definitions

10.4.1. Unimplementation

An implementation may be fully or partially erased using an `unimpl` clause.

Only an existing implementation may be unimplemented.

Example 5. Unimplementing

```
def foo(x ~ Int);

# unimpl foo(x ~ FBin) # Panic! Can not find matching implementation

unimpl foo(x : SInt32) # OK, now we don't have an implementation
                       # for this particular type
# foo(42)              # Panic!
foo(42i16)             # OK

unimpl foo(x ~ SInt) # OK, now we only have implementation for `UInt`
# unimpl foo(x ~ SInt32) # Panic! Can not find matching implementation

foo(42) # OK
# foo(-1) # Panic! Can not find overload for `foo(x : SInt32)`
```

10.4.2. Reimplementation

10.5. Generators

Generator is like macro, but bound to specific instance.

A function accepting a block argument is called a generator function.

```
# `=>` is a special "block restriction"
decl foo(=>) # An anonymous block arg

# If block arity > 1, requires all block args to be named
decl bar(success: =>, failure: =>) # Named block args

# Having a label in argument declaration
# always expects it to be a block label.
impl foo(%block: =>) { %block(42) }
# foo(block: => { }) # Panic! Declaration prohibits named argument
foo(=> { }) # OK

# Aliasing
impl foo(block: %blk =>) { %blk(42) }
```

10.6. Returning

A returned value is always moved.

Variants are treated specially. If returned value *r* is rval or mval, then `return r` is a shortcut to `T?(r)`. If returned value *r* is sval, then `return r` is a shorcut to `T?(<- r)`.

```
@[Trivial]
class Klass
end

def foo : Klass
  final klass := Klass()
  return klass # Directly returned, thus implicitly moved
  # return Klass() # Ditto
end

def bar : Klass?
  if @rand?
    final klass := Klass()
    return klass # Shortcut to `.(<- klass)`
    # return Klass() # Ditto
  else
    return void # Shortcut to `.(void)`
  end

  # An absence of `return` is implicitly `return void`
end

def baz : (Klass, Klass)
  final klass := Klass()

  # Can not imply moving in a
  # non-directly returned expression,
  # thus by default `klass` is copied
  return (klass, <- klass)
end
```

TODO: Hidden (non-public) functions should be treated as public. For example, it may be better to mark them `unsafe` rather then wrap their bodies in `unsafe!`.

TODO: Can overload: visibility, storage, mutability. Can not: safety, throwing-ness, purity.

10.7. Pure functions

This section is to be discussed yet.

Only a function can be pure, neither generator nor proc.

A function can be marked as pure using the `pure` modifier. Without that modifier, a function is implicitly `impure`, even if a compiler can prove it is pure.

1. A pure function does not have side effects:
   a. It only writes to local storage;
   b. It does not call impure functions.

2. A pure function only reads from caller, local or temporal storages.

If any of the statements above is false for a function marked `pure`, a compiler panics.

A pure function can be throwing: modifying a backtrace is not considered a side-effect.

Example 6. Pure functions

```
pure def sum([]: a, []: b)
  return a + b
end

def global_add([]: a)
  static let global = 42
  return global + a # Reading from static storage, thus impure
end
```

Chapter 11. Safety

There are three levels of runtime safety in descending order, namely `threadsafe`, `fragile` and `unsafe`.

The following operations are `threadsafe`:

- Calling a method with at least `fragile` safety modifier on a value with local storage.

The following operations are `unsafe`:

- Coercion, unless special conditions are met;
- Some of pointer operations, depending on its storage.

All other operations are `fragile` by default.

11.1. Safety statements

A program shall not invoke a lesser-safe code from a higher safety environment. Instead, a lower safety statement shall be used to wrap the lesser-safe code.

Listing 2. Safety statements

```
fragile_statement =
  "fragile!",
  expr | block();

unsafe_statement =
  "unsafe!",
  expr | block();
```

A safety statement itself becomes of the specified safety level.

Safety statements transfer the safety responsibilities from the compiler to a developer. It is easy then to debug the program by grepping source files with `unsafe!` and `fragile!` patterns.

For example, calling an `unsafe` function from a `fragile` context would require wrapping the call into an `unsafe!` statement like so: `unsafe! foo()`. Here, a developer explicitly stated that they are fully responsible for consequences caused by calling `foo`.

11.2. Safety modifiers

There are `threadsafe`, `fragile`, `unsafe` and `undefsafe` safety modifiers. An `undefsafe` safety modifier implies that the modified entity has undefined safety. A type, function or macro declaration has a safety modifier.

Listing 3. Safety modifiers

```
safety_modifier =
  "threadsafe" |
  "fragile" |
  "unsafe" |
  "undefsafe";
```

The global namespace implicitly has the `fragile` safety modifier.

Declarations other than macro declarations implicitly inherit the containing declaration safety modifier unless overriden. A macro declaration always has an implicit `undefsafe` safety modifier unless overriden.

For example, a `struct Foo` declared in the top-level namespace would be implicitly `fragile struct Foo`.

Similarly, an instance method declared in a `threadsafe struct Bar` would be implicitly `def threadsafe foo`.

A trait function declaration with undefined storage has the `undefsafe` safety modifier unless explicitly overriden. A `derive` statement inherits the containing declaration safety modifier unless explicitly overriden.

```
trait Foo
  # Implicitly `undefstor undefsafe`.
  decl foo : Void
end

struct Bar
  derive Foo
    # Implicitly `instance fragile`.
    impl foo;
  end

  threadsafe derive Foo
    # Implicitly `instance threadsafe`.
    impl foo;
  end
end
```

Chapter 12. Types

A type is a fundamental qualia of a entity determining its characteristics.

The definition of *type* given by Wikipedia:

> […], a **data type** or simply **type** is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data.
>
> — Wikipedia contributors, Data Type at July 20, 2020

is different from that used in this document. Instead, this document has the aforementioned specialized *data type* definition, which aligns closely with the Wikipedia's.

A type is uniquely identified with a fully-qualified identifier.

Type kinds

A type is either a *namespace*, *trait*, *unit*, *enum*, *struct*, *class* or an *annotation*; this classification is known as the *type kind*. A type kind is a keyword.

Data types

A *data type* may have *instances* of that type known as *objects* which are deemed to physically exist in the space-time continuum, including stack memory, processing unit registers or sections within an executable file. info:[In practice, an object existence may be elided during translation for optimization purposes. Also, an object may be present in superposition, e.g. a quantum bit, which is not clearly a definition of existence. However, a human always considers an object existing somewhere and sometime.] Enum, struct and class types are considered data types.

TODO: A struct type may be named or anonymous. Anonymous struct also allow method declarations with reopening.

TODO: A class type has optional mutability. Read more in classes.

TODO: Entity declaration, implementation and definition.

A namespace, trait or unit type shall not be declared, but defined only. info:[An implementation statement requires a prior declaration statement, therefore these types can not be implemented either. The design decision is driven by the fact that these types do not have runtime state to be finalized. The annotation type kind is not included in that list because it may actually be defined with hooks.]

12.1. Members

A type contains zero or more *members*. A member may be a function, value, macro, or a type. This classification is known as *member kind*.

Every type is a name-space in sense of that it can have members declared.

Listing 4. Member reference syntax

```
value = "variable" | "constant"; (* TODO: Move away *)
member_storage = "instance" | "static";
member_kind = "function" | "value" | value | "macro" | "type";
member = [member_storage], member_kind, "member";
```

Any type may contain type and macro members.

Member storage

A function or value member has *storage*. The storage is either static or instance. A static member has lifetime aligned with the program execution. An instance member lifetime spans to the object's. In a data or trait type, the storage is instance by default, and can be altered with a storage modifier. Only struct and class types may contain instance value members. In a enum type, a value member declaration shall explicitly contain the static storage modifier. In a namespace, annotation, or a unit type, a member storage is always implicitly static and can not be altered.

12.2. Generic types

A *generic type* is a type containing at least one generic argument declaration.

A generic argument is accessible from within the generic type scope. info:[A generic argument is not accessible from type members nested in the generic type.]

```
struct Foo<T>
  struct Bar
    let x : T # Panic! `T` is not declared in `Bar`

  struct Bar<T>
    let x : T # OK, but distinct from `Foo`'s `T`
```

Upon identifier qualification, a generic argument has higher precedence over the outer scope.

By convention, generic argument identifiers consist of the smallest reasonable number of uppercase Latin letters, e.g. `Array<T>` or `Encoding<BitSize: BZ, CodeunitSize: CZ>`.

A generic argument may be queried from a generic type using the `t::<name>` accessor, where `name` is the name (or alias, with higher precedence) of the argument. info:[Because of that, `<>` is in the list restricted names for a user function declaration.]

Example 7. Querying a generic argument

```
decl struct Array<Type: T, Size: S>;
@assert(Array<SInt32>::<Type> == SInt32)

final ary = FBin64[3](1, 2, 3)
@assert(ary::<Size> == 3)

# Also works for variants!
final var = Std@rand(%i[1 2 3], %f[1 2 3 4])
# @assert(var::<Size> == 3) # Would work in 50% of cases
```

12.3. Type specialization

A *type specialization* is the process of establishing a distinct type with a unique generic arguments combination from a type definition for the first time when a compiler determines that there is a possibility for either an object or a type instance of the type to exist in runtime. info:[Simply referencing a type, for example from within a `reopen` statement or restriction, does not trigger type specialization. For the specialization to be triggered, there shall be a possibility of that the type would actually exist in runtime as an object or type instance.] A non-generic type is considered to have zero generic arity, which is a unique combination; therefore a non-generic type specializes at most once.

Example 8. Type specialization

```
struct Foo<T>
  val x : T
  \{% print "Spec'd with " .. nx.scope.T:dump() %}
end

let f1 = Foo<SInt32>() # => Spec'd with SInt32
let f2 = Foo<SInt32>() # => (already spec'd)
let f3 = Foo<FBin64>() # => Spec'd with FBin64
# let f4 = Foo<Real>() # Panic! Field `Foo:x : Real` has incomplete type

decl bar(foo: Foo<FBin32>) # Do not specialize, it's just a reference

let f5 = \Foo<FBin32> # => Spec'd with FBin32
```

Type specialization is recursive; that is, if a type `T` containing a field of type `U` specializes, `U` also specializes during `T`'s specialization.

A type specialization triggers specialization of each trait from its traits set, in the same order as in the set.

A specialization of a child type triggers specialization of its parent.

Delayed macros

ditto:[Delayed macro calls and blocks] written directly within a type declaration body are evaluated on every containing type specialization. ditto:[] within a trait type declaration are copied into every deriving non-trait type.

Example 9. Delayed macro evaluation during specialization

```
trait Container<Type: T>
  \{%
    -- This macro block would evaluate on every deriving type
    -- specialization, unique per deriving type.
    print "Spec'd"

    -- Would only declare if `T` is of `Int`
    if nx.scope.T.type < nx.lookup("Int") then
  %}
    # Return a sum of all of
    # the container's elements.
    decl sum() : T
  \{% end %}
end

decl struct Array<Type: T, Size: S ~ %z>
  derive Container<T>
end

decl struct List<Type: T>
  derive Container<T>
end

final ary = Array<SInt32, 3>(1, 2, 3) # => Spec'd
final list1 = List<SInt32>()          # => Spec'd
final list2 = List<SInt32>()          # => (already spec'ed)
```

12.4. Type completeness

A *complete type* is a data type specialization with defined size of occupied space, or a unit type specialization. Only a complete type can be used as a value type in runtime.

An *incomplete type* is a namespace type, annotation type, or a data type specialization with undefined size. A data type specialization is deemed to be incomplete if it contains a field of an incomplete type. info:[A complete type may have an incomplete generic argument, e.g. `Pointer<SInt>`. What matters is the runtime representation of a data type; if a data type does not contain fields of incomplete types, it is complete.] info:[For Core primitives, there are rules for every primitive regarding on how to infer its completeness.]

Incompleteness modifier

It may be possible to define structs, classes and enums with `incompl` modifier, which makes the type incomplete and disallows to use it in runtime unless reopened with `compl` modifier. The `compl` modifier is default for definitions and reopenings then.

12.5. Type inheritance

TODO: Classes can extend classes and structs; structs can extend structs only.

TODO: Shall only extend a defined type, so we know its layout.

12.6. Reopening

A type may be reopened using the `reopen` statement.

`reopen Foo<T> forall T` is similar to `decl Foo<T>`.

Can reopen Variants, Unions, anonymous structs, tuples, enums, primitives.

Listing 5. The `reopen` statement syntax

```
(* TODO: Move. *)
compl_mod =
  (* !keyword(mod) *) "compl" |
  (* !keyword(mod) *) "incompl";

reopen_statement =
  [compl_mod],

  (* !keyword(statement) *)
  "reopen",

  type_ref,
  ";" | block();
```

A `reopen` statement block semantics is similar to the declaration block semantics of the reopened type.

It is not possible to implement fields within a reopen statement.

It is a error to try reopening a non-existing type declaration.

Example 10. Reopening a type

```
decl struct Foo<T> forall T ~ Int
  let x : T
  \{% print "Specialized the " .. nx.scope.T:dump() %}
end

# reopen Foo<T: FBin64>; # Panic! `Foo<T>` istype
reopen Foo<SInt32>;
# => Specialized a SInt
# => Specialized the SInt32
```

Type expressions may also be reopened. info:[This gives a powerful and flexible mechanism for declaring members on trait sums, for example.]

Example 11. Reopening type expressions

Given these traits and struct:

```
trait Drawable
  decl draw(Canvas)
end

trait Colorized
  decl color : UInt8
  decl color=(UInt8)
end

struct Line
  <~ Drawable, Colorized

  let color : UInt8

  # impl ~Drawable.draw
  impl draw(canvas)
    # Draw without considering the color
  end
end
```

One option would be to reopen the traits sum with another-named method declaration.

```
reopen (Drawable && Colorized)
  decl draw_colorized(Canvas)
end

reopen struct Line
  # impl ~(Drawable && Colorized).draw_colorized(canvas)
  impl draw_colorized(canvas)
    # Draw with color!
  end
end

final line = Line()

# W/ color
line~(Drawable && Colorized).draw_colorized(canvas)
line.draw_colorized(canvas)

# W/o color
line~Drawable.draw(canvas)
line.draw(canvas)
```

Another option would be to reopen the sum with the same-named method declaration. But this

would require solving the arising collision problem in the struct.

```
reopen (Drawable && Colorized)
  decl draw(Canvas)
end

reopen struct Line
  impl ~(Drawable && Colorized).draw(canvas) as draw_colorized
    # Draw with color!
  end
end

final line = Line()

# W/ color
line~(Drawable && Colorized).draw(canvas)
line.draw_colorized(canvas)

# W/o color
line~(Drawable).draw(colorized)
line.draw(canvas)
```

Reopening may also happen right within a type; it shall be then referenced with `self`.

```
reopen Drawable
  reopen (self && Colorized)
    # Draw colorized.
    #
    # NOTE: It's not a part of `Drawable`,
    # but rather of `Drawable && Colorized`,
    # hence no collision if deriving `Drawable` only.
    decl draw(Canvas)
  end
end

# The semantics is similar to the
# second option mentioned above.
line~(Drawable && Colorized).draw(canvas)
```

The in-type reopening comes in particularly useful when dealing with generic arguments and namespaced types.

```
trait Container::Indexable<K, V>
  reopen (self && Enumerable<V>)
    decl each(-> |(index: K, value: V)|)
  end
end

# An alternative would be (quite wordy):
#

reopen (
  Container::Indexable<K, V> &&
  Container::Enumerable<V>
) forall K, V
  decl each(-> |(index: K, value: V)|)
end
```

12.7. Refining

TODO: `using refinement`.

12.8. Restriction

Behavioural bias is applied whenever a compiler can prove it. In every scope, a type restriction invests into the bias. The bias is reduced outside of the context. The bias does not exist in runtime.

Example 12. Behavioural bias

```
# In this example, `Drawable` is root trait for both `Drawable2D` and
# `Drawable3D`, causing name collision in a type deriving from both,
# because if we bias `point` to `Drawable` and call `draw` on it,
# which dimension-draw call would it be?
#
# A) Can not bias to `Drawable`, only `p~Drawable2D~Drawable`.

# Abstract traits, which are not deriveable directly?
indirect trait Drawable
  decl draw
end

trait Drawable2D <~ Drawable
end

trait Drawable3D <~ Drawable
end

struct Point
  derive Drawable2D
    impl ~draw as draw2d
  <~ Drawable3D

  impl ~Drawable2D.draw as draw2d;
  impl ~Drawable3D.draw as draw3d;

  def do_pointy_stuff;
end

# The argument is required to derive from `Drawable2D`.
# def draw2d(x : T) forall T ~ Drawable2D # Ditto (need e.g. for `sum`)
# def draw2d(x : T) forall T where T <~ Drawable2D # Ditto, allows more complex
expressions
# TODO: Have formal algorithm for that.
def draw2d(x ~ Drawable2D)
  \{%
    # `x` is biased to `Drawable2D` only
    nx.assert(nx.scope.x.type.bias ==
      nx.typexpr("Drawable2D"))
  %}

  x.draw() # OK, equivalent to `x~Drawable2D`
end

# An argument is required to derive from at
# least one trait deriving from `Drawable`.
def draw(x ~ Drawable)
  \{%
```

```
    # `x` is biased to `Drawable`
    nx.assert(nx.scope.x.type.bias ==
      nx.typexpr("Drawable"))
  %}

  # x.draw() # Panic! `~Drawable.draw()` is indirect declaration

  if x ~? Drawable2D
    # It may also derive from any other `Drawable`,
    # but here we're biased to `Drawable2D` only
    # an do not care about other possible traits.
    #

    \{%
      # `x` is biased to `Drawable2D`
      nx.assert(nx.scope.x.type.bias ==
        nx.typexpr("Drawable2D"))
    %}

    x.draw() # OK, eq. to `x~Drawable2D.draw()`
  end
end
```

12.9. Float

> In computing, floating-point arithmetic (FP) is arithmetic using formulaic representation of real numbers as an approximation to support a trade-off between range and precision.
>
> — Wikipedia

A particular built-in `Float` type specialization maps to a valid IEEE 754-2008 (further referenced as *IEEE 754*) floating-point interchange format.

A `Float` specialization representation in memory is defined in accordance to IEEE 754.

An implementation must implement five basic `Float` specializations in accordance to basic formats specified by IEEE 754, either natively or in software.

In addition to that, two non-basic specializations are declared, but are optional to implement.

Table 1. Basic `Float` specializations

IEEE 754 format	Bitsize	Alias	`Float` specialization	Is basic?
`binary16`	16	`FBin16`	`Float<2, 11, 15>`	No
`binary32`	32	`FBin32`	`Float<2, 24, 127>`	Yes

IEEE 754 format	Bitsize	Alias	`Float` specialization	Is basic?
`binary64`	64	`FBin64`	`Float<2, 53, 1023>`	Yes
`binary128`	128	`FBin128`	`Float<2, 113, 16383>`	Yes
`decimal32`	32	`FDec32`	`Float<10, 7, 96>`	No
`decimal64`	64	`FDec64`	`Float<10, 16, 384>`	Yes
`decimal128`	128	`FDec128`	`Float<10, 32, 6144>`	Yes

If a target natively supports an IEEE 754 format, then the type under the ISA namespace must alias to the according `Float` specialization, for example `alias SPARC::FQuad = FBin128`.

If a target natively supports a non-basic format (even with a limited operations set), an Onyx compiler implementation must implement that format. For example, the `binary16` format is not basic, but often implemented, including some ARM and NVPTX targets; the `FBin16` specialization must be implemented for such targets.

If a target natively supports an extended precision format, then it must be defined by the implementation. The naming convention for an extended precision format alias is `F{Base}E`, where `{Base}` is the extended basic format, for example `FBin64E`. For example, on x87 a target, the extended precision floating point format may be aliased as `alias X87::Float80 = FBin64E = Float<2, 64, 16383>`, and the `Float<2, 64, 16383>` specialization would map natively to that format.

The C18 § F.2 standard states the following:

> The `long double` type matches an IEC 60559[:1989] extended format, else a non-IEC 60559 extended format, else the IEC 60559 double format.
>
> [...]
>
> "Extended" is IEC 60559's double-extended data format. Extended refers to both the common 80-bit and quadruple 128-bit IEC 60559 formats.

Therefore, `long double` in C does not neccessarily maps to `FBin64E`. Instead, it may map to `FBin128`. An implementation (and a user) should keep that in mind upon providing a C-Onyx floating type mapping.

An implementation must provide a way to specify mapping from three C floating-point types `float`, `double` and `long double` to Onyx types, for particular target, prior to compilation. However, an explicit coercion or conversion is still required to interoperate with C types. The mapping defines according `as` methods, for example `float <=> FBin32` mapping defines `$float.as(FBin32)` and `FBin32.as($float)` methods.

An implementation should provide a pre-defined default mapping from C floating-point types to Onyx types for every target it supports.

If a target type has the same arithmetic format as an IEEE 754 format, then it must be aliased to an according `Float` specialization. Otherwise, it may be mapped to an extended-precision floating-point as dictated by IEEE 754, iff it matches the extended-precision format requirements; x87 conforms this requirement, for example `alias FBin64E = Float<2, 64, 15> = X87::Float80`. Implementations not conforming to IEEE 754 must not be mapped to a `Float` specialization; for example, PowerPC extended precision type does not conform to IEEE 754, as it's merely a software emulation. Thus, no mapping must exist, even for `FBin64E`.

A particular `Float` specialization implementation may be ensured during compilation using the `@@impl?` built-in compiler intrinsic and the `nx.Node:is_implemented` macro method, for example `if @@impl?(FBin16)` or `{% if nx.lkp("FBin16").is_implemented %}`.

12.10. Fixed-point numbers

> In computing, a fixed-point number representation is a real data type for a number that has a fixed number of digits after (and sometimes also before) the radix point (after the decimal point '.' in English decimal notation).

A fixed-point number is defined by its natural base `b` (either binary or decimal), natural integral part size `i` (in the base), and integer fractional part size `f` (in the base). For example, the number `-3.625` may be encoded exactly as `-0b11.101Q8f4 : XBin<Integral: 3, Fractional: 4>`.

`f <= 0 => i > 0`.

A fixed-point number is always signed.

A negative value of `f` implies absence of the fractional part in the fixed-point number, and a power in the number's base equal to the absolute value of `f` is applied to the fixed-point number value. A negative `f` does not contribute to the resulting bitsize of a fixed-point number. For example, `-25kD5f-3 : XDec<Integral: 2, Fractional: -3>` is read as `-25 xx 10^3`, but internally comprised of only two digits `25` and a sign bit, thus occupying exactly 8 bits.

An implementation is required to implement an arbitrary-sized fixed-point number in any of the defined bases with a reazonable maximum bitsize limit.

Listing 6. Fixed Core API

```
require "./fractional.nx"

# A fixed-point number.
decl primitive Fixed<
  Base: 2 || 10,

  # Size of the integral part in `Base`.
  Integral: ~%Natural = 0,

  # Size of the fractional part in `Base`.
  # May be negative for right-shift.
  Fractional: ~%Integer = 0
>
end

alias XBin<*> = Fixed<2, *>
alias XDec<*> = Fixed<10, *>
```

12.10.1. Binary fixed-point number encoding

In a binary fixed-point number, `f` defines the amount of bits the represented value is shifted **left** by. If `f < 0`, then the value is shifted **right** instead.

Example 13. Binary fixed-point numbers

```
@[Entry]
export void main() {
  let q = 96Q8f-4 : XBin<Integer: 7, Fraction: -4>

  assert(q == 0b00000110 << 4)
  assert(q == 0b01100000)
  assert(q == 96)
}
```

12.10.2. Decimal fixed-point number encoding

`{(f >= 0 => a = i + f), (f < 0 => a = i):}`, where `a` is the total amount of digits in a decimal fixed-point number.

Digits in a decimal fixed-point number are encoded using the Densely Packed Decimal encoding. An additional bit is occupied by the sign.

Listing 7. Decimal fixed-point number encoding scheme

```
Failed to generate image: Could not find the 'bytefield-svg' executable in PATH; add
it to the PATH or specify its location using the 'bytefield-svg' document attribute
(def boxes-per-row 8)
(def row-labels ["0", "1", "2", "3", "4", "5", "6", "7", "8"])
(draw-column-headers)
(draw-box "Sign" [:box-first])
(draw-gap "DPD" [:box-related])
(draw-bottom)
```

Given `17.42Q4f2 : XDec<2, 2>`, `0` is for sign, and `1742` is `1001 1000 0011 00` in DPD encoding.

```
Failed to generate image: no implicit conversion of nil into String
(def boxes-per-row 15)
(def left-margin 0)
(draw-column-headers {:labels (map str (range 15))})
(draw-box (text "0" :hex))
(draw-box (text "1" :hex) [:box-first])
(draw-box (text "0" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "0" :hex) [:box-related])
(draw-box (text "0" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "0" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "0" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "1" :hex) [:box-related])
(draw-box (text "0" :hex) [:box-last])
(draw-bottom)
```

The following table compiles sizes required for the first ten digits in a decimal fixed-point number.

# of digits	DPD bitsize	Resulting bitsize	Required bytes
1	4	5	1
2	7	8	1
3	10	11	2
4	14	15	2
5	17	18	3
6	20	21	3
7	24	25	4
8	27	28	4
9	30	31	4
10	34	35	5

12.11. Units

A unit type is an object, but it only exists in a single instance and does not occupy any memory in runtime.

Listing 8. Unit type declaration syntax

```
(* A unit type declaration. *)
unit =
  type_visibility_mod,

  (* !keyword(Unit type declaration) *)
  "unit",

  id,
  [generic_args_decl],

  ";" | block();
```

A unit type may derive from a trait.

A unit type shall not extend any other types.

Unit members do not have storage. Unit members may be accessed both in namespace- and object-access styleinfo:[, for example `Unit.foo()` is equivalent to `Unit::foo()`].

Akin to instance members in an object, unit functions are also referenced as *(unit) methods*, and unit values — as *(unit) fields*.

Within a unit method implementation, `this` is equivalent to `self`.

TODO: An instance storage bound to a unit always expands to the static storage.

A unit type is a functor with zero arity. A call to a unit type returns itself.

Example 14. Unit types

```
unit Foo
  let x = 42

  def get_x
    return x
  end
end

assert(Foo == Foo())
assert(Foo.x == Foo::x)
assert(Foo.get_x == Foo::get_x)
```

Chapter 13. Structs

Can only define `val` fields, which mutability depends on the containing struct. Struct have default initializer, can not have finalizers.

TODO: `: this` in function returns control exactly to `this`.

TODO: Returning or throwing an alive struct does not require moving it, as the act the passing an argument to the caller implicitly moves the lifetime responsibility to it. {#lifetime-return-move}

13.1. Members

A member is a struct instance variable.

13.2. Lifetime

13.2.1. Initialization

13.2.2. Finalization

13.3. Reopening

An already defined struct may be defined again. Instead of a complete redefining, the definition continues as if it was the original definition. Thus, instead of redefinition it is called *reopening*.

A struct reopening may be either complete or incomplete, set by `compl` or `incompl` definition modifiers accordingly. Following the usual definition rules, reopening a struct is implicitly complete by default.

Declaring new members is limited by the reopened struct's completeness.

13.4. Completeness

A complete struct has all its members declared, thus having a defined memory layout (unless reordered).

Only a complete struct may be initialized (see Section 13.2, "Lifetime").

Once complete, a struct may not be reopened as incomplete.

A complete reopening of an already complete struct prohibits declaring new members and extending from structs.

By default, a struct definition is implicitly complete.

13.5. Extending

A struct may extend at most one another struct.

An extending struct inherits all of the extended struct declarations, excluding type declarations, in the same order of declaration. Also see Section 13.6.1, “Ordering”.

Table 2. The possibility of an extension based on the completeness status of a struct

Extending struct	Extended struct	Extension possible?
Complete	Complete	Yes
Complete	Incomplete	No
Incomplete	Incomplete	Yes [1]

An `extend` statement shall be placed before any field implementations in a struct.

Regardless of where an `extend` statement is placed, the extending struct members follow the extended struct members in memory unless the extending struct is unordered.

```
struct Foo
  let x
end

# Memory layout of `Bar`
# would be `x, y, z`
struct Bar
  extend Foo;
  let y
  let z
end
```

13.6. Memory layout

13.6.1. Ordering

The order of a struct fields in memory is undefined.

A struct may be reordered. In that case, the order of its members is not defined.

A reordered struct memory layout is a subject to compiler optimizations.

To make a struct reordered, the built-in zero-arity `Reorder` annotation shall be applied to it.

A struct extending from a reordered struct has undefined order of extended struct members only, in the boundaries of memory occupied by extended struct members. Applying the `Reorder` annotation to an extending struct mixes the boundaries of memory occupied by all members, thus allowing for more optimal reordering.

Example 15. Deriving from a reordered struct

```
@[Reorder]
struct Foo
  let foo_a : SInt32
  let foo_b : FBin64
  let foo_c : SInt32
end

# The memory layout would be the following:
# `[foo_*]x3, bar_a, bar_b, bar_c`,
# where `foo_*` have undefined order.
struct Bar
  extend Foo;

  let bar_a : SInt32
  let bar_b : FBin64
  let bar_c : SInt32
end

# In this case, the order of members
# of `Baz` is totally undefined.
@[Reorder]
struct Baz
  extend Foo;

  let bar_a : SInt32
  let bar_b : FBin64
  let bar_c : SInt32
end
```

13.6.2. Packing

13.7. Anonymous structs

An anonymous struct may only declare indexed or named members.

Explicit indexing rule

Indexed member declarations following a named or an explicitly indexed member declaration must be explicitly indexed.

An anonymous struct member can not have a default value, thus each member must be restricted to a concrete type.

The memory layout of a non-reordered anonymous struct matches its members declaration order. Thus, the order of members (even named) in a non-reordered anonymous struct matters.

13.7.1. Anonymous struct literals

An anonymous struct literal consists of an arbitrary amount of implicitly or explicitly indexed or explicitly named values, wrapped in parentheses.

Values declaration follows the Explicit indexing rule.

Trailing commas are allowed in anonymous struct literals.

A single implicitly indexed value without a trailing commas is treated as an expression rather than an anonymous struct.

13.7.2. Anonymous struct destruction

An anonymous struct may be destructed using the splat operator.

A destructed anonymous struct may be multi-assigned to a set of variables, or become a varg restriction.

Upon multi-assignment, an anonymous struct member can be referenced by either its implicit or explicit index, or by its name. References follow the Explicit indexing rule.

Example 16. Multi-assignment of an anonymous struct

```
# Members `[0]` and `[1]` are referenced
# by their implicit indices 0 and 1.
# Members `foo` and `[2]` are referenced explicitly.
let a, b, foo: c, [2]: d = *('a', [1]: 'b', foo: 'c', [2]: 'd')
```

13.7.3. Anonymous struct member ordering

An anonymous struct may become reordered by setting its `Reordered` generic argument to `true` (`false` by default).

The memory layout of a reordered anonymous struct may not match its members declaration order due to possible reordering of its memebers.

Two reordered anonymous structs containing the same set of members but in different order of declarations are guaranteed to have the same memory layout.

An anonymous struct literal may become reordered if appended with `r`.

13.7.4. Packed anonymous structs

An anonymous struct may become packed by setting its `Packed` generic argument to `true` (`false` by default).

A packed struct and its members have alignment of 1.

An anonymous struct literal may become packed if appended with `p`.

Chapter 14. Traits

A trait is a composable unit of behaviour.

The traits implementation is inspired by *Traits: A mechanism for fine-grained reuse* by Ducasse et al., 2006.

TODO: Only units and objects may derive from a trait. Namespaces do not have behaviour.

14.1. Declaring a trait

Listing 9. Trait declaration syntax

```
trait_decl =
  trait_decl_mod,
  ["decl"],
  "trait",
  id,
  [nb, generic_arguments_decl],
  ";" | body();

trait_decl_mod = [type_visibility_mod];
```

A trait type can not be implemented; instead, it is continuously declared. A `trait T` statement is a shortcut to `decl trait T`. A trait declaration statement may include a type visibility modifier and a generic arguments declaration.

A trait type is always an incomplete type. A trait type declaration statement can not contain a completeness modifier.

A trait type function and value members have storage. The storage is implicitly instance.

A trait type allows method declaration and implementation. A trait method implementation is specialized for every deriving type separately. A trait method implementation may use `this` and `super` keywords, which evaluate in accordance to the context the method is specialzied within.

14.2. Trait arithmetic

An object or trait type conveys a set of traits it derives from, which is empty by default for object types and contains itself for trait types.

Deriving from a trait adds the trait term to the deriving type traits set.

Trait set composition is commutative: the ordering of adding traits to the set does not matter.

Nested traits are equivalent to flattened traits; the trait set composition hierarchy does not affect the traits behaviour.

The same trait may be derived multiple times by the same object; it would still count as a single traits set entry.

Recursive derivations are allowed.

14.3. Deriving from a trait

A type which has a `derive T` clause is said to *derive from* `T`.

Listing 10. The `derive` clause syntax

```
derive =
  ["undefstor" | "static" | "instance"],
  "derive",
  id,
  ";" | block();
```

Deriver, deriving (type), derivative

The type deriving from a trait.

Derivee, derived (type)

The trait a type derives from.

A deriving type may contain multiple `derive` statements.

Deriving from a trait includes all of its members with undefined storage into the deriving type as if they were defined within the deriving type itself, which may lead to name collisions. The storage of an included member depends on the storage modifier of the according `derive` clause.

Table 3. `derive` clause storage modifiers

Deriving type	Allowed modifiers	Default modifier
Namespace	`static`	`static`
Trait	`static`, `undefstor`	`undefstor`
Primitive, class, enum	`static`, `instance`	`instance`

Therefore, `static derive T` from type `U` is identical to reopening a `Type<U>` type and then instance-deriving it from `T`.

Example 17. Multiple ways to statically derive

Listing 11. Static derive in T

```
reopen SInt32
  static derive Debuggable
    impl debug(stream) { stream << "SInt32" }
  end
end
```

Listing 12. Instance derive in Type<T>

```
reopen Type<SInt32>
  derive Debuggable
    impl debug(stream) { stream << "SInt32" }
  end
end
```

Only statements allowed within a `derive` statement are `impl`, `reimpl`, `alias` and `moveimpl`.

A `derive` statement may be early-terminated with a semicolon, left without any inner statements.

Lookups within a `derive` statement check the trait's scope at first, and then go up to the deriving type.

A derived trait type may be referenced using the `derived` keyword. The deriving type may still be referenced using the `self` keyword.

Technically, it is allowed to reference other traits' members within a `derive` statement, e.g. `self~OtherTrait:foo` (see behavioral erasure below) or `~OtherTrait:foo` or even `foo` (unless there is an ambiguity).

A `derive` statement does not require all included members' implementations to be contained within it (the `derive` statement).

As any other, derived implementations are lazily checked upon specialization. If a declaration is not implemented at all, the implementation responsibility is passed to the next deriving type, if any.

14.3.1. Behavioral erasure

A derived trait's members are always accessible via their original identifiers with behavioral erasure achieved with fuzzy restriction.

Example 18. Using fuzzy restriction to access original trait identifiers

```
# In this example, both `Drawable2D` and `Drawable3D`
# have `draw` method declaration. A deriving type has
# to deal with the collision. One way would be to
# alias (both) methods under different names.
trait Drawable2D
  decl draw
end

trait Drawable3D
  decl draw
end

@[Trivial]
class Point
  derive Drawable2D
    impl draw;

    # Must use `self` here, otherwise
    # would attempt to declare a new
    # identifier within a `derive` clause,
    # which is prohibited.
    alias self:draw2d to draw
  end

  derive Drawable3D
    impl draw;
    alias self:draw3d to draw
  end
end

final p = Point()

p.draw2d # OK
p.draw3d # OK

# p.draw # Panic! Ambiguous choice between
         # `Point~Drawable2D.draw` and
         # `Point~Drawable3D.draw`

p~Drawable2D.draw     # OK
(p ~ Drawable3D).draw # OK

def draw2d(x ~ Drawable2D)
  x.draw # OK. Thanks to behavior erasure, exactly
         # `Drawable2D~draw` is always referenced
end

draw2d(p) # OK
```

```
def draw_or(x ~ Drawable2D || Drawable3D)
  # x.draw # Panic! `x` may be BOTH `Drawable2D` and `Drawable3D`,
           # hence unable to choose right implementation

  if not x ~? Drawable2D
    x.draw # OK. The compiler can prove that it can only
           # be `Drawable3D`. Even if we add `Drawable4D`
           # with the same `draw` declaration,
           # the behavior erasure feature would always
           # reference `~Drawable3D.draw` exactly.
  end
end

draw_or(p) # OK

def draw_and(x ~ Drawable2D ^ Drawable3D)
  x.draw # OK. There will never be an ambiguity
end

# draw_and(p) # Panic! Point contains BOTH traits, but they're XORed
```

14.4. Abstract traits

A trait may be declared as `abstract`. An abstract trait shall not be derived by a non-trait type. An abstract trait may be extended by a trait type. info:[Effectively, an abstract trait is merged into the extending trait instead of being treated as a separate term participating in the traits sum. All instance members of an abstract trait are treated as if they were declared in the deriving trait itself.] Biasing to an abstract trait does not treat its methods as ever implemented.

Example 19. Abstract traits

```
abstract trait Drawable
  decl draw()

  def double_draw
    draw()
    draw()
  end
end

trait Drawable2D < Drawable;
trait Drawable3D < Drawable;

struct Point
  derive Drawable2D
    impl draw() as self~draw2d;
  end

  derive Drawable3D
    impl draw() as self~draw3d;
  end
end

struct Line
  derive Drawable2D
    impl draw()
  end
end

def double_draw2d(x ~ Drawable)
  # x.draw() # Panic! Can not call an abstract trait method declaration

  if x ~? Drawable2D
    x.double_draw() # Eq. to `x~Drawable2D.double_draw()`
  else
    throw "Not a 2D, rejecting"
  end
end

double_draw2d(Point()) # Calls `Point~Drawable2D:double_draw`
double_draw2d(Line())  # Calls `Line:double_draw`
```

If `Drawable` was not an abstract trait, but rather a simple trait, it would participate in the trait sum, leading to unwanted collisions.

```
trait A
  decl foo;

  def double_foo
    foo()
    foo()
  end
end

# Effectively, `B` = 0 + `A`.
trait B <~ A;

# Effectively, `C` = 0 + `A`.
trait C <~ A;

struct Foo
  # `B + C = 0 + A + 0 + A = A`
  <~ B, C

  # Either one of these would
  # suffice, but exactly one.
  impl A~foo;
  impl B~foo;
  impl C~foo;

  # An attempt to implement
  # either again would panic.
  impl A~foo; # Panic!
  impl B~foo; # Panic!
  impl C~foo; # Panic!
end

struct Bar
  <~ B
  impl A~foo;
end

Foo~A:foo # OK
Foo~B:foo # Nope.

reopen (A && B)
  decl foo
end
```

TODO: `decl let color : UInt8` vs. `decl color : UInt8` && `decl color=(UInt8) : Void`. `line..color=(42).color` is acceptable?

Chapter 15. Core API

The Onyx Core API is the `Core::` namespace including types, functions and macros built into the language.

The `Core::` namespace is available from **every** Onyx source code file.

Criteria for being included in the Core API are:

- Have a great chance of being implemented on hardware level. For example, integers are ubiquitous, floating points are common, and tensors are typical for GPUs. For the same reason most basic math operations are in the Core API;
- Have literals built into the language. For example, ranges;
- Have traits otherwise impossible to express using the language. For example, blocks, lambdas, functions, unions, variants, void etc.

Core API types' memory layout is usually undefined.

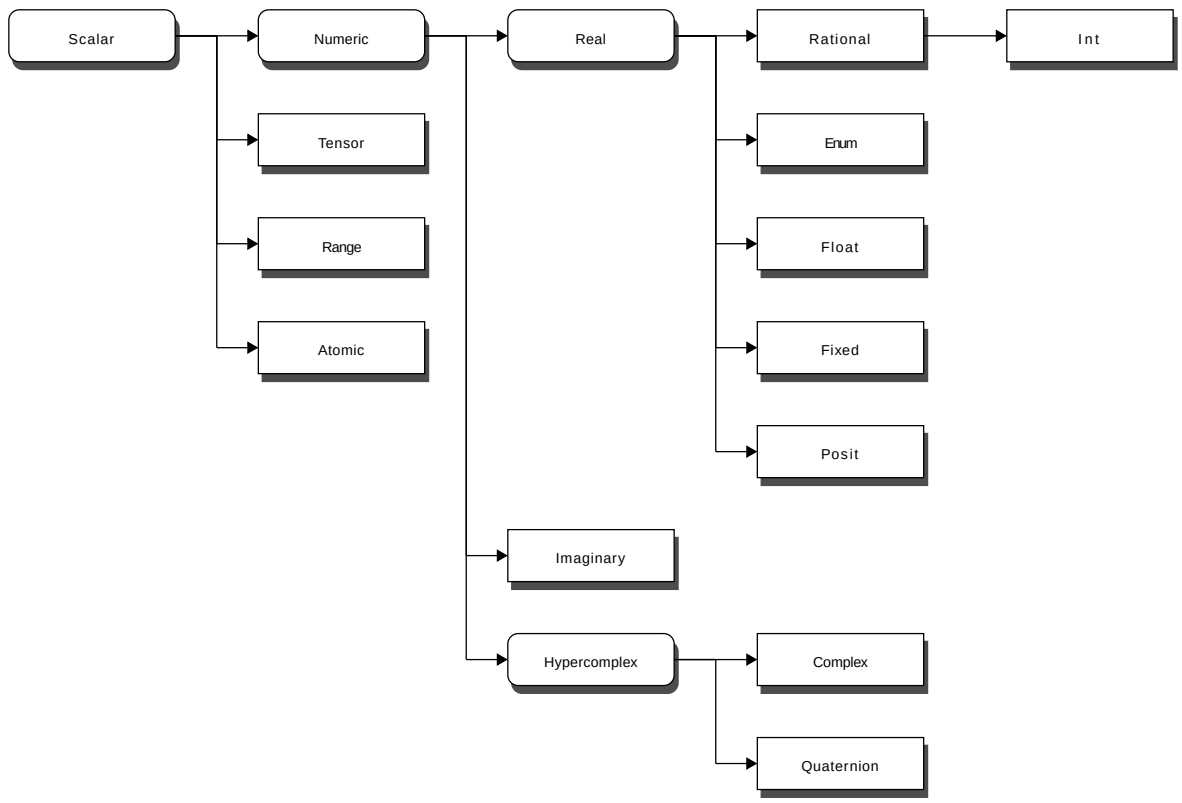

Figure 1. Core API scalar types diagram

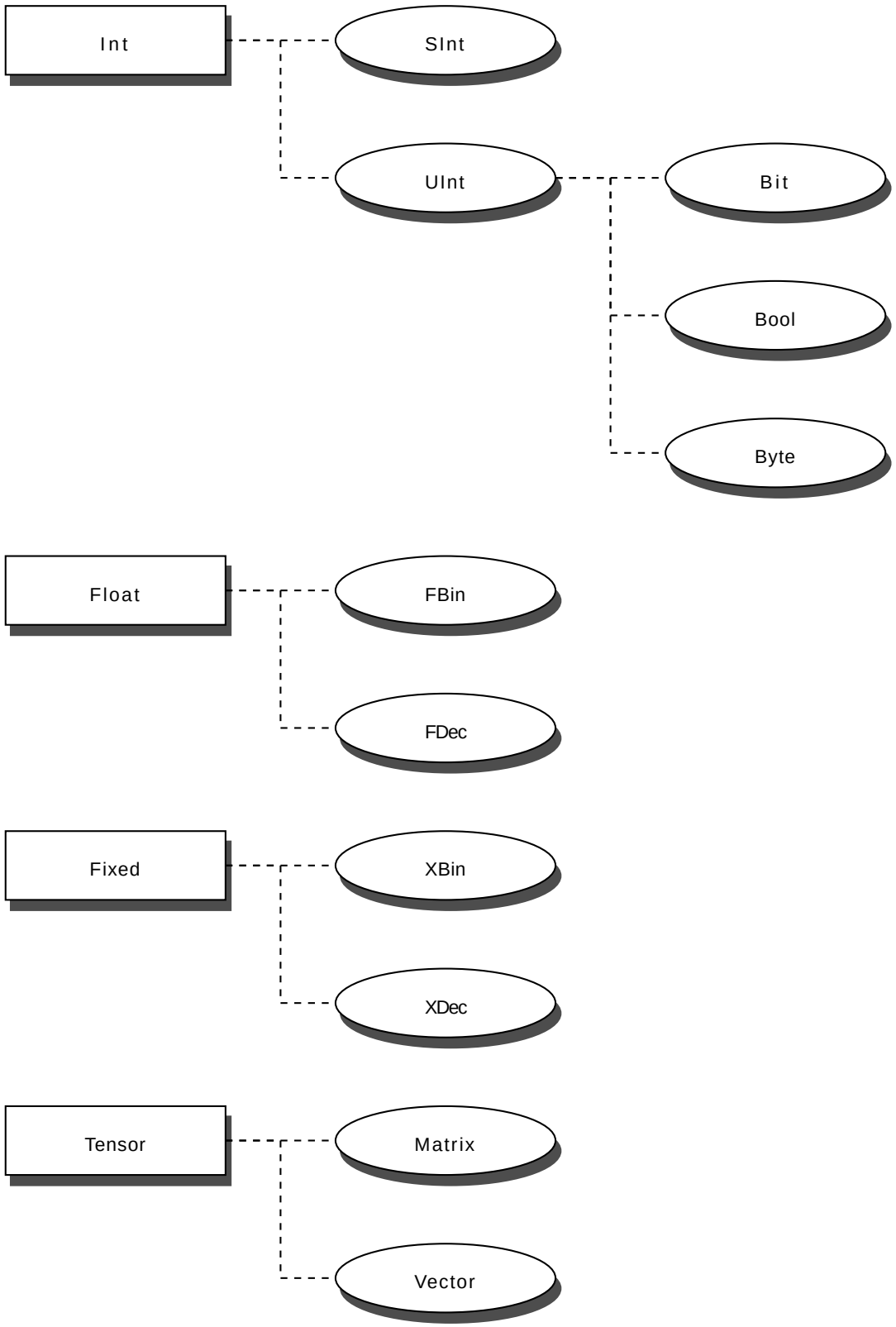

Figure 2. Core API aliases diagram

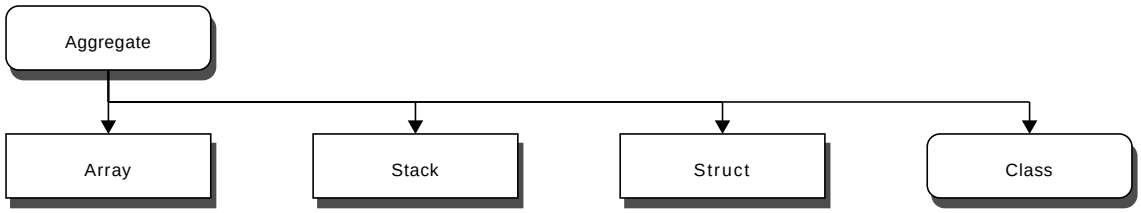

Figure 3. Core API aggregate types diagram

Other than types deriving from `Scalar` and `Aggregate`, the Core API also includes freestanding `Void`, `Union`, `Vector` and `Quire` types.

Chapter 16. Directives

A *directive* is an instruction to the compiler.

16.1. File dependency directives

File dependency directives are `require` and `import` directives.

A file dependency directive makes the program depend on a source file at the provided pathname. The same file may be referenced from multiple file dependency directives, but it is guaranteed to be processed exactly once first time it is referenced. If the same file is referenced both from a `require` and `import` directive, then it is processed exactly twice.

The order in which files are referenced both from inside a single file dependency directive and the global order matter. A user should keep in mind the actual entities structure so they do not reference an undeclared yet entity.

A single file dependency directive may contain multiple pathnames, as well as patterns in pathnames.

Filename and pathname resolution rules in a file dependency directive shall conform to sections 4.7, 4.8 and 4.13 of IEEE Std 1003.1-2017. Pathname pattern matching notation shall conform to section 2.13 of IEEE Std 1003.1-2017.

The IEEE Std 1003.1-2017 is known as the POSIX standard. Pathnames in file dependency directives shall conform to the POSIX standard so they stay cross-platform.

An optional `at` clause in a file dependency directive substitutes the provided pathname with every pathname listed in the `from` clause in a file dependency directive. If an `at` clause is present, then `from` clause shall be present with the explicit `from` keyword. For example, `require "bar/foo.nx"` is equivalent to `require from "foo.nx" at "bar"`.

An implementation shall provide a way to add to the list of base lookup paths for every file dependency directive class in the manner similar to `-I` and `-L` flags ubiquitous in C compilers.

The Macro API provides ways to both invoke a file dependency directive as well as alter the base lookup paths.

16.1.1. `require` directive

A `require` directive makes the program to include an Onyx source file into the program source.

Recursive `require` directives are allowed.

```
require_directive =
  "require",

  @unord(
    (* Path(s) of the required file *)
    (["from"], path, {",", path}),

    (* The base path *)
    ["at", path]
  );
```

16.1.2. `import` directive

An `import` directive makes the program aware of entities contained in the imported file such that those entities are not a part of the program itself, but rather can be referenced from the program.

An imported file may contain source code in a language other than Onyx.

The Standard defines interoperability specifications for the following languages: Onyx, C, Lua. However, an implementation is not limited to those languages defined in the Standard.

Read more about interoperability in Interoperability.

```
(*
  Import a file written possibly
  in a language other than Onyx.

  ```
  import from "stdio.h" in "C"
  import in "Rust" from "main.rs", "aux.rs" at "/mypath/"
  ```
*)
import_directive =
  "import",

  @unord(
    (* The source language of an imported file *)
    ("in", string),

    (* Paths to the imported files *)
    ("from", string, {",", string}),

    (* The base path for the imported files *)
    ["at", string]
  );
```

16.2. `using` directive

The `using` directive either injects a namespace `N` into the local scope, so that members of `N` are directly accessible from the local scope; or applies a refinement `R` in the local scope; or defines an alias `A` to `B` effective in the local scope.

```
using =
  using_namespace |
  using_refinement |
  using_alias;

using_namespace =
  "using",
  ["namespace"],
  type_ref;

using_refinement =
  "using",
  ["refinement"],
  type_ref;

using_alias =
  "using",
  ["alias"],
  decl,
  ("=", "to"),
  ref;
```

If a `using` directive contains `=` or `to`, it is then implicitly treated as `using alias`. Otherwise, the kind of `using` is inferred from the referenced identifier. An explicit `namespace`, `reference` or `alias` clause may be present to enforce the kind of `using`.

For a `using` directive, the scope is limited to the containing source file.

Listing 13. Using a namespace

```
namespace Foo
  let bar = 42
end

using Foo # `using namespace` is inferred
bar = 43  # OK
```

Listing 14. Explicit `using` kind

```
namespace Foo
end

using namespace Foo # OK
# using refinement Foo # Panic!
```

Listing 15. Using an alias

```
namespace Foo
  let bar = 42
end

using Baz = Foo # `using alias` inferred

Baz.bar = 43 # OK
Foo.bar = 44 # Still OK
# bar = 45   # Panic! `bar` is not declared in current scope
```

Chapter 17. Literals

A literal is a constant value known at compilation time, as opposed to a non-literal value computed at runtime.

Despite of the knowledge of existence of such a literal value, a compiler may decide to change its form or completely remove from a binary for optimization purposes. In other words, for some kinds of literals, there is no guarantee that a literal would be stored anywhere in a resulting binary; but it must be transparent to a user of the program.

A literal is an rvalue.

17.1. Literal constrainment

Each literal conveys a certain amount of information, which determines its level of constrainment.

A literal constrainment determines the set of restrictions it can match.

A literal constrainment may be altered with literal prefixes and suffixes.

A literal without any prefixes or suffixes is said to be *unconstrained*. Some literal kinds have default types defined for their unconstrained literals, e.g. an unconstrained integral literal is `SInt32` by default.

Even an unconstrained literal still have some constraints. For example, a numeric literal can never become a string. Or a negative whole number can never become an unsigned integer.

A literal can be instance-restricted (both with concrete and fuzzy restrictions) to a type, e.g. `42 : SInt32`. Applying an instance restriction to a literal takes into consideration and contributes into its constrainment level. For example, `42f : SInt32` would be invalid, as the literal is already constrained to a binary floating-point value. Or, `42 ~ FBin : SInt32` would also fail on the second restriction.

A literal can be type-restricted to either an exact literal value (e.g. `42 :\: @{42}`) or a built-in type having literals (e.g. `42 ~~ @{Size}`). Such a restriction, where the right operand is wrapped in `@{}`, is called a *literal restriction*.

Literal restrictions are ubiquitous in generic type arguments, e.g. `primitive Array<Type: T, Size: S ~ @{Size}>`.

TODO: Refine exact and fuzzy restrictions, standardize scalar and aggregate literals.

Only a type restriction may be a literal restriction. Only a literal may be the left operand of a literal restriction. A literal restriction returns its left operand, the literal.

An exact literal restriction must match a complete type literal. For example, `@{42i}` and `@{FBin}` are invalid restrictions, because neither literal represents a complete type (despite of the fact that `i` defaults to `SInt32`; in a literal restriction it expands to simply `SInt`). A valid example would be

`@{42i32}`.

A fuzzy literal restriction may match an incomplete literal kind, but it erasures the left operand behavior, which affects the literal constainment.

Example 20. Literal constrainment

```
# `1` is unconstrained. An unconstrained
# integral literal defaults to `SInt`.
# However, it may be further restricted to other types.
1 ~ Integer
1 ~ FloatingPoint
1 ~ Integer<Signed: false> : UInt8

# Adding `-` to the literal reduces the
# set of restrictions it can match
-1 ~ Integer<Signed: true>
-1 !~ Integer<Signed: false>
-1 ~ FloatingPoint

# Adding a fractional part turns
# a literal into a non-integer
1.0 ~ FloatingPoint
1.0 ~ Fixed
1.0 !~ Integer

1.0f ~ FBin
1.0f64 : FBin64
```

Listing 16. TODO:

```
def foo(x: L) forall L : %i
  \{{ nx.scope.L.value }} # `foo(42i) -> 42i`
  \{{ nx.scope.L.number }} # `foo(42i) -> 42`
end

def foo(x : L) forall L : %q
  \{{ nx.scope.L.value }} # `foo("x"utf8) -> "x"utf8`
  \{{ nx.scope.L.string }} # `foo("x"utf8) -> "x"`
end

%i, %si  # Integer
%ui      # Unsigned integer (natural)
%i32, %z # Limit values

%r, %ri32 # Any-end range

# For clarity with `()`, this is disallowed
# %ri[) # Specific-end range

%q # Any-encoding string, value is
%qucs # Accept `"foo"utf8` and `"foo"ucs2`, reject `"foo"ascii`

%ucs, %c, %cucs # A char

# Array of literals is prohibited?
# Because array becomes a runtime value?
# %c[] # An array of chars

%c(2) # Exactly 2 chars
%c(2..) # At least 2 chars
%c(1..3) # 1 to 3 chars

def foo(*x : L) forall L : %c(2)
foo(('a', 'b'), ('a', 'b'))

def foo(x : L) forall L : %c(2..)
foo('a', 'b', 'c')

def foo(*x : L) forall L : %c
foo('a', 'a', 'a') # Useless

def foo(*x : L) forall L in ('a', 'b', 'c')
foo('a', 'c') # OK
```

Example 21. Literal restrictions

```
42 ~ @{42} : @{42f} : FBin64
42 ~ @{42i} !: @{42f} # Can not further restrict `i` to `f`

struct Foo<T : @{Size}>
end

Foo<42> # OK
# Foo<-42> # Panic!

struct Bar<T : (@{"foo"} || @{"bar"}) ~ @{String}>
end

Bar<"foo"> # OK
# Bar<"baz"> # Panic!
```

17.2. Underscores

Underscores can be used in a literal to either visually divide its parts without contributing into its value, or as a part of its value (true for text literals), or both.

Example 22. Application of underscores in literals

```
1_000_i32     # Underscores do not contribute into the value
"Hello_world" # Underscore is a part of the value
"a_b"_u16     # Both
```

Literal suffixes and prefixes (see [literal-multiplier-prefix], [literal-encoding-prefix]) are treated as a whole and therefore can not contain underscores.

A preceding underscore is treated as (a part of) an identifier and therefore prohibited in literals, e.g. `1 is an identifier, and 'a'` is a error.

Sequential underscores are allowed, e.g. `1__000`.

Trailing underscores are allowed in numeric and text literals, but only after a prefix.

Due to the fact that suffixes are read from the end, trailing underscores are required in cases when it is desired to emphasize the absence of a suffix, for example:

Example 23. Using trailing underscores to emphasize the absence of suffix in literals

```
0z : Size     # "z" is "size" suffix
0z_ : FBin64  # "z" is "Zepto" multiplier prefix
0z_f : FBin64 # Ditto

# 0fz # Panic! Can not apply "Zepto" prefix to a `Size` literal
0fz_ : FBin64 # Prefixes: "f" for "Femto", "z" for "Zepto"

# # A compiler sees it as ``'a'asci_i`
# 'a'ascii # Panic! Can not find encoding `ASCI`

'a'ascii_ : CPoint<UCS2>
'a'asciii : SInt32
```

17.3. Literal suffixes

A literal suffix is a combination of graphemes adjacent to a literal contributing into the literal's constrainment.

Literal suffixes are only applicable to either scalar or magic literals.

TODO: A scalar literal is an nvalue. A scalar literal wrapped in parentheses is still a literal instance. Therefore, literal suffixes are applicable to scalar literals even wrapped in parentheses, e.g. `(42)i32`, `("foo")utf16be`.

In a non-magic literal, literal suffixes follow the literal value, e.g. `1i`. In a magic literal, literal suffixes precede the literal value(s), e.g. `%i[1]`

The first literal suffix in a literal, if any, is *major*. The following suffixes are *minor*.

In a literal, literal suffixes are parsed from right to left. A literal suffix has higher priority than other parts of a literal.

A literal suffix can not contain underscores.

Underscores may be used to visually separate suffixes from other parts of a literal. Moreover, the fact that a trailing underscore is not allowed after a suffix makes it possible to distinguish between suffixes and, say, prefixes, for example `5f == 5.0f64` but `5f_ ~= 0.5e-15f64`.

A literal suffix may be defined for a type using the `LiteralSuffix` annotation. A type with a literal suffix defined must be a functor accepting a literal instance. Whichever literal kinds are accepted

by the functor defines which literal kinds the suffix may be applied to.

A custom literal suffix has higher precedence than a built-in literal suffix. For example, `ascii` would be parsed as `ascii` if defined, and not `asci + i`.

Table 4. Defaults for unsuffixed literals

Examples	Restriction	Default type
`42`, `-42`, `2.5k`	`Numeric`	`SInt32`
`0.5`, `-1f_`	`Fractional`	`FBin64`
`'f'`	`Char`	`Char<UCS>`
`"foo"`	`String`	`String<UTF8>`

Table 5. Built-in literal suffixes

Regex	Applicable to	Constraint	Default type
`/s?i(?<Bitsize>\d+)/`	`Integer`, `Char`	`SInt<Bitsize>`	`SInt<Bitsize>`
`/s?i/`	`Integer`, `Char`	`SInt`	`SInt32`
`/ui?(?<Bitsize>\d+)/`	`Integer`, `Char`	`UInt<Bitsize>`	`UInt<Bitsize>`
`/ui?/`	`Integer`, `Char`	`UInt`	`UInt32`
`/c/`	`Char`	`Char`	`Char<UCS>`
`/s/`	`String`	`String`	`String<UTF8>`
`/ucs/`	`Char`	`Char<UCS>`	`Char<UCS>`
`/utf8/`, `/utf(16|32)[lb]e/`, `/ucs(2|4)/`	`String`	`String<Encoding>`	`String<Encoding>`
`/fb?(?<Bitsize>\d+)/`	`Numeric`	`FBin<Bitsize>`	`FBin<Bitsize>`
`/fb?/`	`Numeric`	`FBin`	`FBin64`
`/f?d(?<Bitsize>\d+)/`	`Numeric`	`FDec<Bitsize>`	`FDec<Bitsize>`
`/f?d/`	`Numeric`	`FDec`	`FDec64`
`/Q(?<Bitsize>\d+)?(e(?<Exponent>[-]?\d))?/`	`XBin<Bitsize, Exponent>`	`\(Real)`	`Either Bitsize or Exponent is required`
`/D(?<Total>\d+)?(f(?<Fractional>\d+))?/`	`XDec<Total, Fractional>`	`\(Real)`	`Either Total or Fractional is required`
`/j/`	`Imaginary`	`\(Real)`	`Turns a literal constrainment from T into Imaginary<T>`
`/p(?<Bitsize>\d+)?/`	`Posit<Bitsize>`	`\(Real)`	`Bitsize defaults to 32`
`/bf(?<Bitsize>\d+)?/`	`BFloat<Bitsize>`	`\(Real)`	`Bitsize defaults to 16`

Table 6. Built-in literal suffixes

Regex	Type	Accepted literals	Notes
Integer literal suffixes			

Regex	Type	Accepted literals	Notes
/s?i(?<Bitsize>\d+)?/	SInt<Bitsize>	\(Int), \(Char) (xnum:character-literal-integral-suffix[]), \(String) (xnum:string-literal-integral-suffix[])	Bitsize defaults to 32. Default constraint for a signed
/ui?(?<Bitsize>\d+)?	UInt<Bitsize>	\(Int), \(Char) (xnum:character-literal-integral-suffix[]), \(String) (xnum:string-literal-integral-suffix[])	Bitsize defaults to 32
/c/	Char<UCS>	\(Int), \(Char)	Turns a literal constrainment into Char with default charset of UCS
/ucs/	Char<UCS>	\(Char)	Constrains a character literal charset to UCS
Real literal suffixes			
/fb?(?<Bitsize>\d+)?/	FBin<Bitsize>	\(Real)	Bitsize defaults to 64
/f?d(?<Bitsize>\d+)?/	FDec<Bitsize>	\(Real)	Bitsize defaults to 64
/Q(?<Bitsize>\d+)?(e(?<Exponent>[-]?\d))?/	XBin<Bitsize, Exponent>	\(Real)	Either Bitsize or Exponent is required
/D(?<Total>\d+)?(f(?<Fractional>\d+))?/	XDec<Total, Fractional>	\(Real)	Either Total or Fractional is required
/j/	Imaginary	\(Real)	Turns a literal constrainment from T into Imaginary<T>
/p(?<Bitsize>\d+)?/	Posit<Bitsize>	\(Real)	Bitsize defaults to 32
/bf(?<Bitsize>\d+)?/	BFloat<Bitsize>	\(Real)	Bitsize defaults to 16
String literal suffixes			
/utf8/, /utf16[bl]e/, /utf32[bl]e/	String<UTF8>, String<UTF16BE>, String<UTF16LE>, String<UTF32BE>, String<UTF32LE>	\(String)	Constrains a string literal encoding
/ucs2/, /ucs4/	String<UCS2>, String<UCS4>	\(String)	Constrains a string literal encoding
C integer literal suffixes			
/\$[su]?h/	$short, $`unsigned short`	\(Int)	
/\$[su]?i/	$int, $`unsigned int`	\(Int)	

Regex	Type	Accepted literals	Notes
`/\$[su]?l/`	`$long`, `$\`unsigned long\``	`\(Int)`	
`/\$[su]?ll/`	`$\`long long\``, `$\`unsigned long long\``	`\(Int)`	
`/\$[su]?z/`	`$size_t`, `$ssize_t`	`\(Int)`	
C real literal suffixes			
`/\$f/`	`$float`	`\(Real)`	
`/\$d/`	`$double`	`\(Real)`	
`/\$ld/`	`$\`long double\``	`\(Real)`	

`to_*` *and* `as_*` *lookup algorithm*

If a receiver `R` does not have a `to_*` or `as_*` method defined explicitly, a compiler then looks up for a literal suffix identical to the `*` part. If such a suffix exists for a type `T`, and `R` has method `to(:: T)` defined, it is called then. Otherwise, a compiler panics because of missing method. For example, if `FBin64` had `to(:: SInt32)` defined, then all `fbin.to(SInt32)`, `fbin.to_si32` and even `fbin.to_i` would be valid. Presence of the algoritm effectively allows hyphens and dollar sign in exclusively in function names beginning with `to_` and `as_`, for example `to_Qe-7` and `as_$f`.

17.4. Scalar literals

17.4.1. Numeric literals

TODO: Literal structure: base prefix, value: (numeric representation (including multipliers)), suffix.

Literals are designed to not scream until needed: the syntax is almost invariant in regards to casing, and most things are in lower case. Exceptions are the Q number that is widely known by its upper-case name, hexadecimals to distinguish them from other parts, and big multiplier prefixes.

Listing 17. Numeric literals syntax

```
multiplier_prefix_iec =
  "Yi" | "Zi" | "Ei" | "Pi" |
  "Ti" | "Gi" | "Mi" | "Ki";

multiplier_prefix_si =
  "Y" | "Z" | "E" | "P" | "T" | "G" | "M" | "k" | "h" | "da" |
  "d" | "c" | "m" | "u" | "μ" | "n" | "p" | "f" | "a" | "z" | "y";

sint_suffix = ["s"], "i", {digit}; (* 32 bits by default *)
uint_suffix = "u", ["i"], {digit}; (* 32 bits by default *)
size_suffix = ["s" | "u"], "z"; (* Unsigned by default *)

fbin_suffix = "f", ["b"], {digit}; (* 64 bits by default *)
fdec_suffix = ["f"], "d", {digit}; (* 64 bits by default *)
```

```
(*
  Design rationale: despite of the fact that `p` can only be in
  lowercase in Onyx, it'd be confusing to see `42P8` and treat it
  other than a "42 * 2 ** 8". Had to pick another symbol therefore.
*)
posit_suffix = "t", {digit};
posit_suffix = "p", {digit};

(*
  Design rationale: would not use `B`, because
  brain float is "smaller" than usual float.
*)
bfloat_suffix = "bf", {digit};

literal_suffix =
  sint_suffix |
  uint_suffix |
  size_suffix |
  fbin_suffix |
  fdec_suffix |
  xbin_suffix |
  xdec_suffix |
  posit_suffix |
  bfloat_suffix;

sign = "-" | "+";

non_decimal_value ($prefix, $base) =
  $prefix, {$base | "_"}, $base,
  [".", $base, {$base | "_"}],
  ["p", [sign], {digit}-]

numeric_literal =
  [sign],
  (
    non_decimal_value("0b", binary) |
    non_decimal_value("0o", octal) |
    non_decimal_value("0x", hex) |
    (
      digit, {
        (*
          Multiplier prefixes allow to
          have multiple radix points.
        *)
        ["."], digit,

        (* The prefix families must not be mixed. *)
        {digit | "_" | multiplier_prefix_iec | multiplier_prefix_si}
      },
      ["e", [sign], {digit}-]
    )
```

```
  ),
  {"_"},
  [literal_suffix];
```

Underscores may be used to:

- Split literal numeric representation into chunks (`1_000_000`);
- Split literal value and suffix parts (`1_f`);
- Split literal numeric representation and exponent prefix parts (`1_M`);
- Split literal exponent prefix and suffix parts (`1M_i`);
- Designate an absence of suffix (`1z_`).

A recommended chunk size should be preserved when applying underscores (e.g. `10_000_00` is confusing).

Numeric literals always have big-endian representation, i.e. most significant bits of data come first. For example, `0x2a == 0x002a == 0b101010 == 0b0010_1010 == 42`.

TODO: Applying an exponent (`e` for decimal, `p` for other bases) makes a literal fractional. However, it still can be forced to be integral, if is allowed to.

Regardless of the literal representation endianess, the actual memory layout of a numeric value in runtime may be defined elsewhere.

Only the decimal base may be used for floating and fixed decimal literals.

TODO: Which bases and literals allow signs?

Table 7. Numeric literal bases summary

Base prefix	Base	Recommended chunk size	Example
`0b`	Binary	4 (nibble)	`0b1100_0001`
`0o`	Octal	3	`0o777_001`
`0`	Decimal	3 (a thousand)	`42_000`
`0x`	Hexadecimal	2	`0xab_CD`

In decimal literals, `e` is used to denote a decimal **e**xponent (`* 10 ** e`). In literals with other bases, `p` is used to denote a binary exponent, i.e. **p**ower of two (`* 2 ** p`). An omitted exponent defaults to 0 (`* 2 ** 0` or `* 10 ** 0`, which is always `* 1`).

TODO: Non-decimal literals can be used for any number. Rules are the same:

```
0x10.1 == 16 + (1 * 16 ** -1) == 16.0625
0b10000.0001 = 16 + (1 * 2 ** -4) = 16.0625
0o20.04 = 16 + (4 * 8 ** -2) = 16.0625

0b101010f == 0b101010.0f
# , thus non-decimal literals are sometimes not exact representations.
```

An *integral literal* can be restricted to an integer type.

A *likely-fractional literal* still can be restricted to an integer type, but defaults to a non-integer type, e.g. `2e2 : SInt32` would be `FBin64` by default. A likely-fractional literal can be turned into integral by applying an integral suffix, e.g. `2e2i` would be `SInt32` by default.

A *fractional literal* can not be restricted to an integer type, e.g. `2e-2 : FBin64`.

A literal with big multiplier prefixes (including multiplier radix points) and/or positive exponents is likely-fractional.

A literal with at least one small multipler prefix, or negative exponent, or fractional radix point; is fractional.

Multiplier prefixes

Numeric literals support both binary (IEC) and decimal (SI) multiplier prefixes.

***big* multiplier prefix**

One having a positive exponent.

***small* multiplier prefix**

One having a negative exponent.

Table 8. Binary multiplier prefixes (IEC)

Prefix	Name	Multiplier
Big prefixes		
`Yi`	Yobi	1024^{8}
`Zi`	Zebi	1024^{7}
`Ei`	Exi	1024^{6}
`Pi`	Pebi	1024^{5}
`Ti`	Tebi	1024^{4}
`Gi`	Gibi	1024^{3}
`Mi`	Mebi	1024^{2}
`Ki`	Kibi	1024^{1}

Table 9. Decimal multiplier prefixes (SI)

Prefix	Name	Multiplier
Big prefixes		
`Y`	Yotta	10^{24}
`Z`	Zetta	10^{21}
`E`	Exa	10^{18}
`P`	Peta	10^{15}
`T`	Tera	10^{12}
`G`	Giga	10^{9}
`M`	Mega	10^{6}
`k`	Kilo	10^{3}
`h`	Hecto	10^{2}
`da`	Deca	10^{1}
Small prefixes		
`d`	Deci	10^{-1}
`c`	Centi	10^{-2}
`m`	Milli	10^{-3}
`u, µ`	Micro	10^{-6}
`n`	Nano	10^{-9}
`p`	Pico	10^{-12}
`f`	Femto	10^{-15}
`a`	Atto	10^{-18}
`z`	Zepto	10^{-21}
`y`	Yocto	10^{-24}

Multiplier prefixes are case-sensitive.

Multiplier prefixes can only be used in decimal-based literals.

Multiplier prefixes from different standards must not be mixed, i.e. no binary and decimal prefixes in a single literal.

Multiplier prefixes from the same standard may be mixed in a single literal.

Multiple multiplier prefixes in a single literal must be in the same order as defined in the tables below. For example, `10M300k == 10300000` is valid, but `300k10M` is not, because `M` precedes `k` in Table 9, “Decimal multiplier prefixes (SI)”.

Due to the suffix priority in literals, it may be desirable to split value and kind with an underscore for readability, e.g. `1Mi_i` and `1M_i`.

Example 24. Readability of suffixes in literals

```
0xff64 == 0xf_f64 ~= 15.0
0xff64f64 == 0xff64_f64 ~= 65380.0

42d == 42_d32   # Simply "FDec"
42dd == 4.2_d32 # "Deci" + "FDec"

1Mii == 1_048_576_i32 # "Mebi" + "SInt"
1Mi == 1_000_000_i32  # "Mega" + "SInt"
```

Integer literals

An unconstrained integral literal default type is `SInt32`.

Floating-point literals

An unconstrained likely-fractional and fractional literal default type is `FBin64`.

Binary fixed-point literals

Listing 18. Q number literal suffix syntax

```
(* For consistency, it is still
  referenced as "`XBin` suffix". *)
xbin_suffix =
  (* An optional signedness. *)
  ["s" | "u"],

  (* The binary fixed-point (a.k.a.
    Q number) literal symbol. *)
  "Q",

  (* An optional bitsize. *)
  {digit},

  (* An optional Fractional part size in bits. Can also
    be read as an amount of bits to left-shiFt by. *)
  ["f", [sign], {digit}-];
```

xcite::term-q-number[]

A Q number literal is signed by default.

A Q number literal value may be in any base, contain optional multiplier prefixes, optional

exponent and optional fractional radix point.

Either bitsize or _f_ractional part size is required in a Q number literal. If bitsize is omitted, the nearest greater or equal byte size is inferred, e.g. `0Qe10 == 0Q16e10`, `0Qe-5 == 0Q8e-5`, `0Qe8 == 0Q16e8`. If fractional part size is omitted, it equals to the bitsize minus one bit if signed, e.g. `0Q8 == 0Q8e7`, `0uQ8 == 0uQ8e8`.

Example 25. Q number literals

```
1Qf-8 == 2 ** 8 == 256

0.625uQ8 + 0.0078125uQ8 ==
  0b0.101uQ8 + 0b0.0000001uQ8 =
  0b0.10100001uQ8 == 0.6328125d

#   1010 0000
# + 0000 0010
# = 1010 0010

# In signed Q numbers, the most
# significant bit is the sign bit,
# if the implementation is 2's complement.
#
# TODO: Come up with a valid example.
#

-0.625Q8 + 0.0078125Q8 ==
  -0b0.101Q8e7 + 0b0.0000001Q8e7 =
  -0b0.1011001Q8 == 0.60546875d

#   1011 0000
# + 0000 0010
# = 1011 0010
```

Decimal fixed-point literals

Listing 19. Decimal fixed-point literal syntax

```
xdec_suffix =
  (* An optional signedness. *)
  ["s" | "u"],

  (* The decimal fixed-point literal symbol. *)
  "D",

  (* An optional total amount of digits. *)
  {digit},

  (* An optional amount of digits which are fractional. *)
  ["f", {digit}-];
```

A decimal fixed-point literal can only be in decimal base.

A decimal fixed-point literal is signed by default.

A decimal fixed-point literal requires either total amount of digits or amount of digits which are fractional, or both, set. In case if only the total amount is set, the fractional amount is deemed to be zero. In case if only the fractional amount is set, the total amount is deemed to be equal to the fractional.

xcite::dpd-bitsize-omitted[]

17.4.2. Text literals

Abstract character

A character from the Unicode character set, which may consist of one or many codepoints.

Source-encoded abstract character

An abstract character encoded in the source file encoding, for example □a□ encoded as a (U+0061); □å□ encoded as å (U+61, U+030A) or å (U+00E5).

A text literal is a sequence of abstract characters which are either source-encoded or made up from Section 17.4.2.1, “Numerical codepoints”.

An abstract character is normalized upon parsing. Then, the normalized abstract character is encoded into a single codepoint or a sequence of codepoints in the encoding the literal is constrained to, i.e. the target encoding, with preference given to a variant comprised of a lesser amount of codepoints. When the target encoding supports multiple codepoint sequences for a single abstract character representation, and a specific sequence variant is desired, numerical codepoints shall be used instead of source-encoded abstract characters. If the target encoding does not support the abstract character at all, a compiler panics.

The standard defined a number of escaped abstract character sequences consisting of a backslash (\, U+005C) followed by a single Latin letter, which expand to a predefined abstract character.

Table 10. Escaped abstract character sequences

Sequence	Character	Description
`\a`	`U+0007`	Alert (Beep, Bell)
`\b`	`U+0008`	Backspace
`\e`	`U+001B`	Escape character
`\f`	`U+000C`	Formfeed Page Break
`\n`	`U+000A`	Line Feed
`\r`	`U+000D`	Carriage Return
`\t`	`U+0009`	Horizontal Tab
`\v`	`U+000B`	Vertical Tab
`\s`	`U+0020`	Whitespace
`\` (escaped newline in source file)	N/A	Ignores the newline

The following normalization scenarios are possible, yet to discuss.

- Forced normalization.

  A combined grapheme is always attempted to be normalized, even if the target encoding supports the combination.

  Pros:

  - Abstract characters are always encoded in the most compact way.
  - If a editor displays a combined grapheme as a single glyph, then it would be natural to think of it as of a single codepoint in the target encoding.

  Cons:

  - A editor may display a combined grapheme as two distinct glyphs, and it would be confusing to think of it as of a single codepoint in the target encoding.
  - A developer must always keep Unicode normalization rules in mind.

- Normalization on-demand.

  If a combined grapheme is directly supported by the target encoding, it is used. Othwerise, if the target encoding supports a normalized grapheme variant only, it is used instead.

  Pros:

  - The mapping stays as close to the source as possible, making use of suitable alternative grapheme representations only when required.

  Cons:

  - Even if a target encoding supports the normalized variant, and a editor displays it as a single glyph; the resulting grapheme would still be combined.

- No normalization at all. If target encoding does not support this exact combination, it would panic.

Numerical codepoints

A numerical codepoint is a codepoint encoded as an explicit numerical value, e.g. □f□ encoded as `\102`.

A numerical codepoint may be in either octal (`\o141`), decimal (`\96`) or hexadecimal (`\x61`) base. In either base, regardless of the literal's encoding, the numerical value has big endianess, for example `ϕ` is encoded as `\x84d1`.

A numerical codepoint may map to an invalid (i.e. not defined in) abstract character in the literal's

charset, but its bitsize must not exceed the maximum allowed by the charset. For example, `'\xffffffffffffffff'` would trigger panic for the most of the possible charsets, because no charset defines 64 bits as its codepoint's size.

A numerical codepoint value may be wrapped in curly brackets to clearly define its boundaries. For example, `"{97}1" == "a1"`. An unmatched bracket would trigger panic.

Without brackets, a numerical codepoint value is terminated either with a grapheme outside of the base's allowed graphemes range, or at the end of the containing literal. For example, in `"\97f" == "af"`, the numerical codepoint in decimal base is terminated with a non-decimal grapheme `f`.

Numerical codepoint values do not allow neither literal prefixes nor literal suffixes, for example `{1ki8}` would trigger panic.

Numerical codepoint values allow non-trailing underscores.

Listing 20. Numerical codepoint syntax

```
numerical_codepoint =
  ("\o", {octal, "_"}-, "_"-) |
  ("\o", "{", {octal, "_"}-, "_"-, "}") |

  ("\", {digit, "_"}-, "_"-) |
  ("\", "{", {digit, "_"}-, "_"-, "}") |

  ("\x", {hex, "_"}-, "_"-) |
  ("\x", "{", {hex, "_"}-, "_"-, "}");
```

Character literals

A character literal is a text literal comprised of a single abstract character comprised of a single codepoint wrapped in single quotes (`'`), for example `'a' == '\97'`.

A character literal is considered a scalar literal.

An unconstrained character literal has type `Char<UCS>`.

`UCS` is the language-defined default charset for a character literal.

A charset literal suffix is defined by applying the `@[LiteralSuffix]` annotation to a static functor constrained to a single `@(Char)` argument. A charset suffix may be applied to a character literal, which alters the literal's charset, for example `'a␣ucs`. Depending on the functor implementation, if the target charset does not contain the literal's abstract character, a compiler should panic.

Example 26. Applying a charset suffix to a character literal

```
'a' == 'a'ucs : Char<UCS> # Default

@[LiteralSuffix(/(us)?ascii/, /iso646us/)]
namespace ISO646::US <~ Charset<7>
  def (~@(Char)) ~@(Char<self>)
    # Basically, return the codepoint if it's less than 128.
    # Otherwise, panic!
  end
end

'a'ascii : Char<ISO646::US> # Another charset applied
# 'ϕ'ascii # Panic! `ISO646::US` does not contain this abstract character
```

A character literal may have an integral literal suffix, which would constrain it to an integral literal of kind defined by the suffix equal to the codepoint's numerical value. For example, `'a'u8 == 97u8`, `''i16 == (0xF900)i16 == -1792i16`.

Listing 21. Character literal syntax

```
character_literal = "'", unicode | {numerical_codepoint}, "'";
```

String literals

A string literal is a text literal comprised of an arbitrary amount (including zero) of abstract characters wrapped in double quotes (") or slashes (/), for example `"f"`, `"a'\0"`, `/bar/`.

A string literal is considered a scalar literal.

The NUL character is NOT appended implicitly to a string literal. Instead, it must be done manually.

A string literal has type `String<E, Z>`, where `Z` is the exact amount of codeunits in the literal, for example `"fa'\0" : String<UTF8, 4>`, `"foo"ucs2 : String<UCS2, 6>`.

`UTF8` is the language-defined default encoding for a string literal.

An encoding literal suffix is defined by applying the `@[LiteralSuffix]` annotation to a static functor constrained to a single `@(String)` argument. An encoding suffix may be applied to a string literal, which alters the literal's encoding, for example `'a'utf8`. Depending on the functor implementation, if the charset defined by the target encoding does not contain at least one abstract character from contained in the literal, a compiler should panic.

Example 27. Applying an encoding suffix to a string literal

```
"a" == "a"utf8 : String<UTF8, 1> # Default values

@[LiteralSuffix(/(us)?ascii/, /iso646us/)]
namespace ISO646::US <~ Encoding<self, 7>
  def (~@(String)) ~@(String<self>)
    # Basically, build a string containing the same
    # codepoints while their values are less than 128.
    # Otherwise, panic!
  end
end

"a"ascii : String<ISO646::US, 1> # Another encoding applied
# "ϕ"ascii # Panic! `ISO646::US` does not contain this abstract character
```

A string literal may have an integral suffix, which would constrain it to an array of integral literals of kind defined by the suffix equal to the numerical values of codeunits' making up the literal's abstract characters. If the type defined by the literal can not contain a codeunit value in the literal's encoding, a compiler panics. For example, `"aϕ"u8 == [0x61u8, 0xd1u8, 0x84u8] : UInt8[3]`.

Listing 22. String literal syntax

```
string_literal = "\"", {unicode | numerical_codepoint}, "\""
string_literal = "/", {unicode}, "/";
```

Regex literals

A string literal wrapped in `//` is called a regex literal.

Neither of the escape sequences work in a regex literal, but `\/`. Therefore, numerical codepoints are not usable in regex literals.

Heredocs

A heredoc is a string literal spanning through multiple lines. It begins and terminates with the same sequence of ASCII characters.

A heredoc literal value (the text) must be wrapped in newlines.

A heredoc opening sequence may be wrapped in parentheses and optionally applied literal suffixes to. Suffixes affect a heredoc literal in the same way as they would affect a simple string literal.

The first token of a heredoc terminating sequence must be the first non-control character in a source line. Once the terminating sequence is matched, a compiler treates the heredoc literal as complete; it allows, for example, to call a method immediately after the terminating sequence, e.g. `SQL.upcase`.

A heredoc is aligned at the least indentation. Escaping a newline (\␤) may affect the least indentation. The first line of a heredoc does not have a newline character prepended. The last line of a heredoc does not have a newline character appended.

A good formatting practice is to always make the least indentation to align with the line containing the beginning heredoc sequence. The terminating sequence should be placed on a new line. Sequences should be in uppercase.

Example 28. Heredocs

```
assert(<<-SQL
SELECT foo
  FROM bar
SQL == "SELECT foo\n  FROM bar")

(
  # This is ill-formatted: the line containing `SELECT`
  # should be indented by four spaces instead of two.
  # Note that the indentation of the
  # terminating sequence does not matter.
  assert(<<-(sql)ascii
  SELECT foo
FROM bar
sql == "  SELECT foo\nFROM bar")
)

assert(<<-SQL

  SELECT
  FROM bar\
    WHERE SQL

SQL == "\nSELECT\nFROM bar  WHERE SQL\n"
```

Listing 23. Heredoc syntax

```
(* NOTE: Opening and closing sequences must be equal! *)
heredoc =
  "<<-",
  (
    "(", seq(en | underscore), ")",
    {literal_suffix}
  ) | seq(en | underscore),
  nl,
  seq(unicode),
  nl,
  seq(en | underscore);
```

17.4.3. Symbol literals

For a type restriction `R`, a symbol literal may unambiguously match a type `T` satisfying the restriction `T ~ R`, whereas the symbol value matches the last element of full path identifier of `T` in accordance to the Symbol literal matching algorithm.

Symbol literal matching algorithm

The algorithm is defined as follows:

1. The exact value of a symbol is compared directly with the last part of every type path matching the expression, until equal. If matched, success.
2. If the symbol value is not explicit:
    a. The symbol value is lower-cased, any non-letter and non-digit abstract character is removed; this is called a normalized value.
    b. Last parts of every type path to be compared with which is not an explicit identifier, is also normalized.
    c. The normalized symbol value is compared with each normalized last part, until equal. If matched, success.
3. No match, thus failure.

If a symbol literal expands to a compile-time constant value, it may be then suffix-casted to another type, e.g. `(:max)u8 == 255u8`.

Symbols can only be used as literals and expand during compilation. There is no built-in runtime `Symbol` type.

Example 29. Using symbols

The following example demonstrates suffix-casting of a symbol literal.

```
assert((:max)u8 == UInt8::Max)
assert((:Min)i == SInt32::Min)
```

The following example demonstrates enum values lookup.

```
enum Foo
  val Bar
  alias `Бар to Bar

  val Baz
end

def is_baz?(foo : Foo)
  # `Foo#==` expects `Foo`,
  # thus `:bar` properly
  # expands to `Foo::Bar`
  foo == :bar
end

# Using an explicit symbol value here
assert(not is_baz?(:`Бар))
```

The following example demonstrates symbol expansion in case branches.

```
struct A::B::C;

final var = Std@rand(42, C())

case var
when :int # Properly expands to `Int`, which is legal
  Std.out << "Is int"
when :c # Expands to `A::B::C`
  Std.out << "Is A::B::C"
end
```

The following example demonstrates symbol ambiguity.

```
enum Foo
  val Bar
  val Baz
end

enum Qux
  val Bar
  val Quux
end

def is_bar?(e)
  e == :bar
end

# # It would go through every type matching
# # the restriction  (which is none), and panic
# # upon `Foo::Bar` vs. `Qux::Bar` ambiguity.
# assert(is_bar?(:bar)) # Panic!

# This works, because the specialization
# would have `e : Foo` restriction.
assert(is_bar?(Foo::Bar))

# # This would panic upon ambiguity between
# # `e == Foo::Bar` vs. `e == Qux::Bar`.
# assert(is_bar?(Std@rand(Foo::Bar, Qux::Bar))) # Panic!
```

17.5. Aggregate literals

TODO: Restrictions with aggregate types? `~ \(Array)`, `~ \(Array<Size>)`.

The generic argument determining the element type of the type an aggregate literal is constrained to is inferred to a variant of the literal's values' types. If an aggregate literal consists of likely-fractional and fractional literals, then the element type is inferred to be a fractional type.

For a tuple literal, the type is inferred for each element separately.

```
[1, 1] : SInt32[2]
[1i16, 1] : SInt16[2]
[1u, 1] : UInt32[2]
[1u, 1i] : ?<UInt32, SInt32>[2]
[1, 0.5] : FBin64[2]
[1, 0.5d32] : FDec32[2]
[1u, 0.5] : ?<UInt32, FBin64>[2]
[1u, 1f] : ?<UInt8, FBin64>[2]

[1, 1p32] : Posit32[2]
[1p, 1p32] : Posit32[2]

# [1, 1p32, 1u8] # Panic! Can not unambiguosly infer type
                 # of `1` -- must constrain explicitly
[1u, 1p32, 1u8] : ?<Posit32, UInt8>[2] # OK
```

17.5.1. Array literals

An array literal is constrained to the `Array` type.

17.5.2. Tensor literals

```
# r, l0 -- row
# c, l1 -- column

# It could work, because `(\SInt32)x2` is
# not allowed (must be a runtime type).
(SInt32)x4 : Vector<SInt32, 4> # (\i)x4
(FBin64)x4x4c : Matrix<FBin64, 4, 4, 0> # (\f)x4x4c
($double)x2x3x4l2 : Tensor<$double, 2, 3, 4, 2> # (\$d)x2x2x4l2
(XDec<2, 2>)x2 # (\Df2)x2
(XBin8)x3      # (\Q8)x3
```

17.5.3. Tuple literals

17.5.4. Range literals

17.6. Magic literals

TODO:

```
# Quoted string
#

%q(), %qucs2[], %qutf16le<>
%utf32be[] # Works if `utf32be` is unambiguously `Charcoding`

%qutf16-EOF
Hello
EOF

# Range
#

%r[1 2], %ri(min 0), %fr[2 5), %ru8(1 max]

# Containers
#

%i[1 2] # Array
%i<1 2> # Vector
%i(1 2) # Tuple, but why?
%i(foo: 1, bar: 2) # That's why
%i|[1 2]|r # Tensor (row-major); `|m0` -- ditto
%c[hi] # Array of chars
%casciiu8<hi> # Vector of chars
%ucs[hello] # Works if `ucs` is unambiguous `Charset`
%w[hello world], %wutf16le(a b) # OK
```

Magic literals is a unified syntax for:

1. Array, tuple and vector container literals, allowing:
    a. To apply literal suffixes on the container itself instead of on each of its elements;
    b. To avoid repeated typing of separating commas.
2. Section 17.6.2, “Quoted string literals”.

A magic literal requires at least one literal suffix to be specified.

A magic literal which is a character container literal or a quoted string literal is a *magic text literal*.

Elements within a magic literal are separated by spaces, unless it is a magic text literal. In a magic text literal, every character is meaningful, including spaces.

Outside of a magic text literal, a sequence of non-digit characters is considered a symbol literal (assuming that `:` is omitted).

Listing 24. Magic literal syntax

```
magic_literal_values =
  [
    literal_value,
    {" ", literal_value}
  ];

magic_tuple_literal =
  "(", magic_literal_values, ")";

magic_array_literal =
  "[", magic_literal_values, "]";

magic_vector_literal =
  "<", magic_literal_values, ">";

magic_tensor_literal =
  "|", magic_literal_values, "|",
  [tensor_literal_dimension_appendix];

(* If literal suffixes are missing, the container
  type is inferred from its values. *)
magic_literal =
  "%",
  {literal_suffix}-,
  (
    magic_tuple_literal |
    magic_array_literal |
    magic_vector_literal |
    magic_tensor_literal
  );
```

17.6.1. Word literals

A magic literal allows special `w` and `*w` major suffixes. Either turns the literal into a container of space-separated strings, i.e. **w**ords. The `*w` variant expands to a container of hard string copies instead of pointers to strings.

Example 30. Word literals

```
assert(%wutf8(foo barbaz) ==
  ("foo", "barbaz") :
  Tuple<
    String<UTF8, 3>*s,
    String<UTF8, 6>*s
  >)

assert(%*w[foo barbaz] ==
  [*"foo", *"barbaz"] :
  Variant<
    String<UTF8, 3>,
    String<UTF8, 6>
  >[2])
```

17.6.2. Quoted string literals

A **q**oted string literal begins with the `q` suffix. In a quoted string literal brackets act as the string quotes.

Example 31. Quoted string literals

```
assert(%qutf8["foo"] == %q("foo") == "\"foo\"")
```

Chapter 18. Interoperability

Onyx provides native means of interoperability with the C programming language.

TODO: An alternative would be to process C macro output, and if it's a constant, update it accordingly. So it's like `foo($SOME_MACRO)` ⇒ `foo($('a'))`. Panic if got a non-constant C expression.

In this section, C terms may use *smaller italic script*; C keywords and types may use `smaller monoscript`. For example, Onyx struct is different from C *struct*, and Onyx `const` is different from C `const`.

18.1. C standard

Onyx supports Section 18.2, "Importing" and Section 18.3, "Exporting" a subset of C entities in accordance to ISO/IEC 9899:2018 a.k.a. C18; herein *the C standard*.

An Onyx compiler MUST be able to compile imported and exported C code.

C features other than included in the C standard (e.g. a specific C compiler extension) are NOT required by the Onyx standard.

18.2. Importing

A C file may be imported using an `import` statement.

Listing 25. The import statement syntax

```
(*
  The `import` statement shares syntax and semantics
  with the `require statement`.

  The following statements are equivalent:

  ```
  # NOTE: `.h` is implicitly appended only if
  # a file missing its extension is not found
  import "foo", "bar" from "baz"

  import "foo.h", "bar.h" from "baz"
  import "baz/foo.h", "baz/bar.h"

  import "baz/foo.h"
  import "baz/bar.h"
  ```

  As with `require`, the order of importing matters.
*)
import = "import", path, {",", path}, ["from", path];
```

The `import` statement shares syntax and semantics with the `require` statement with the following

differences:

- Instead of compilation-defined *require* paths it relies on *import* paths, possibly specified by passing an `-I` flag.
- The default imported file extension is `.h` if missing.

Akin to `require`, re-importing the same file at the same path multiple times is allowed, leading to an actual importing happening exactly once at the moment when the first `import` statement with the path is met. Files with the same contents, but at different paths, are treated as distinct imports.

All variable declarations and definitions, function declarations and definitions, *enum* definitions, *union* declarations and definitions, *struct* declarations and definitions, *typedef* declarations with *external linkage* and *preprocessor text macros* are imported from a C header into the Onyx scope.

A C entity declared outside of an imported C header, but availaible from within it (e.g. via a nested `#include` directive), is also imported.

An imported entity exists in the top-level Onyx namespace.

An imported entity can be referenced from the Onyx scope by prepending `$` to its identifier, e.g. `$int`. Akin to Onyx identifiers, wrapping backticks may be used to reference multi-word entities, e.g. `` $`unsigned long` ``.

The Macro API has means to access an imported entity information.

18.2.1. Type mapping

All C *fundamental types* are interchangeable with Onyx types.

Some C *fundamental types* may be target-dependent. For example, `int` could map at least to `SInt16` or `SInt32` depending on the target data layout model.

The following table defines mapping rules between C *fundamental types* and Onyx types.

Table 11. Mapping between C and Onyx fundamental types

C type	Onyx type
`void`	`void : Void`
`char`	TODO: Same as either `signed char` or `unsigned char`, depending on what?
`_Bool`	TODO: `Bool : Bit` or target-dependent? How to determine exactly?
`signed char`	`SInt8`
`unsigned char`	`UInt8 : Byte`
`(signed) short (int)`	At least 16 bits, usually `SInt16`
`unsigned short (int)`	At least 16 bits, , usually `UInt16`
`(signed) (int)`	At least 16 bits, usually `SInt16` or `SInt32`

C type	Onyx type
`unsigned (int)`	At least 16 bits, usually `SInt16` or `SInt32`
`(signed) long (int)`	At least 32 bits, usually `SInt32` or `SInt64`
`unsigned long (int)`	At least 32 bits, usually `UInt32` or `UInt64`
`(signed) long long (int)`	At least 64 bits, usually `SInt64`
`unsigned long long (int)`	At least 64 bits, usually `UInt64`
`float`	`FBin32`
`double`	`FBin64`
`long double`	`FBin128` or target-dependent types, e.g. `X86::FBin80`
`float _Complex` etc.	`Complex<$float>` etc.
`float _Imaginary` etc.	`Imaginary<$float>` etc.
`_Atomic int` etc.	`Atomic<$int>` etc.
`int b : N`	`SInt<N>` or `UInt<N>` depending on implementation. TODO: How to make it consistent?
`signed (int) b : N`	`SInt<N>`
`unsigned (int) b : N`	`UInt<N>`
`_Bool b : 1`	`UInt1 : Bool : Bit`
`type*`	`$type*`

18.2.2. Importing variables

All imported variables are available in the Onyx scope as static variables mapped in accordance to the Section 18.2.1, “Type mapping” scheme.

An imported variable with *const type qualifier* is a final and immutable (if applicable) variable in Onyx.

An imported variable with *volatile type qualifier* is a `volatile` variable in Onyx.

The *restrict type qualifier* is ignored by Onyx.

Example 32. Importing a variable

Listing 26. variables.h

```
int i;
volatile const float j = 42;

struct point_t {
  int x, y;
};

struct point_t point;
const struct point_t cpoint = { 10, 20 };

char sa[] = "foo"; // The type is completed as `char[4]`
char* sb = "bar";

const char sc[] = "qux"; // The type is completed as `char[4]`
const char* sd = "kax";
```

Listing 27. main.nx

```
import "./variables.h"

@[Entry]
export void main () {
  assert($i is? $int)
  assert(&$i is? $int*srw)
  assert($i == 0)

  assert($j is? $float)
  assert(&$j is? $float*sr)
  assert($j == 42)
  assert({{ nx.c.j.is_volatile }})

  assert($point is? <mut $point_t>)
  assert(&$point is? <mut $point_t>*srw)
  assert($point.x += 1 == 1)

  assert($cpoint is? <const $point_t>)
  assert(&$cpoint is? $point_t*sr)
  assert($cpoint.y == 20)

  assert($sa is? <mut $char[4]>)
  assert(&$sa is? <mut $char[4]>*srw)
  assert($sa == %c[foo], $sa[2] == 'o')

  assert($sb is? $char*srw)
  # assert($sb == "bar") # There is no such a guarantee
  assert((unsafe! $sb as String*sr)->eq?("bar"))

  assert($sc is? <const $char[4]>)
  assert(&$sc is? $char[4]*sr)

  assert($sd is? $char*sr)
}
```

18.2.3. Importing functions

A function definition is compiled upon importing.

Calling an imported function is always unsafe.

An imported function argument can not be referenced by its name upon call.

Example 33. Importing a function

Listing 28. functions.h

```
// A function declaration.
int foo();

// A function definition.
int bar(int a, int b) {
  return a + b;
}
```

Listing 29. main.nx

```
import "./functions.h"

@[Entry]
export void main () {
  # It's a linker's responsiblity to
  # ensure the symbol is actually defined
  unsafe! foo()

  # Calling an actual definition
  assert((unsafe! $bar(1, 2)) == 3)
}
```

18.2.4. Importing enums

Importing a *enum* imports its values as *macros*.

Imported *enums* can not be reopened.

Example 34. Importing a enum

Listing 30. enum.h

```
enum color_t { RED, GREEN = 2, BLUE };
```

Listing 31. main.nx

```
import "./enum.h"

@[Entry]
export void main () {
  final color = $GREEN # Would evaluate to literal `2`, hence `$int`
}
```

18.2.5. Importing unions

An imported *union* may be initialized using the C *struct initializer*.

Accessing an imported *union's* member is always unsafe.

Imported *unions* can not be reopened.

Example 35. Importing a union

Listing 32. union.h

```
union union_t {
  int a;
  double b;
};
```

Listing 33. main.nx

```
import "./union.h"

@[Entry]
export void main () {
  final union = $union_t{ .a = 42 }
  assert(unsafe! union.a == 42) catch return 1
}
```

18.2.6. Importing structs

An imported *struct* may be initialized using the C *struct initializer*.

Accessing a *non-atomic* member of a *non-atomic* imported *struct* is fragile. Accessing an *atomic* member of a *non-atomic* imported *struct* is threadsafe. Accessing a member of an *atomic struct* is unsafe. Accessing a member of a *volatile struct* is volatile.

Mutability modifers are applicable to imported *structs*. By default, an imported *struct* type is implicitly `const`.

Imported *structs* can not be reopened.

Example 36. Importing a struct

Listing 34. struct.h

```
struct struct_t {
  int a;
  double b;
};
```

Listing 35. main.nx

```
import "./struct.h"

@[Entry]
export void main () {
  final strukt = mut $struct_t{ .a = 42, .b = 0.5 }
  assert((strukt.a += 1) == 43) catch return 1
}
```

18.2.7. Importing typedefs

Referencing an imported *typedef* is the same as referencing the type it *aliases*.

Example 37. Importing typedefs

Listing 36. typedef.h

```
typedef struct { double hi, lo; } range;
```

Listing 37. main.nx

```
import "./typedef.h"

@[Entry]
export void main () {
  final range = $range {
    .hi = 0, .lo = 1 }
}
```

18.2.8. Importing preprocessor macros

An imported *preprocessor text macro* (hereby *macro*) may be referenced from the Onyx scope as a regular C entity by prepending the `$` symbol to its identifier.

An imported *macro* reference allows arguments to be passed to it. It is a error to use parentheses on an *object-like macro*.

Once referenced, a *macro* is immediately *evaluated* in accordance to the Section 18.1, “C standard”. The evaluation result is then embedded directly into the source code.

Therefore, the concept of safety is not applicable to a *macro* reference itself. Instead, code generated by its evaluation is a subject to safety judgement.

Imported *macros* are also available from the Macro API.

Example 38. Importing macros

Listing 38. macros.h

```
#define FOO "foo"
#define DOUBLE(arg) #arg * 2
```

Listing 39. main.nx

```
import "./macros"

@[Entry]
export void main () {
  assert($FOO == "foo")
  assert($DOUBLE(2) == 4)
}
```

18.3. Exporting

A C declaration, definition, *preprocessor directive* or a block of C code may be exported using an `export` statement.

An exported entity is available in the assembly code iff it has *external linkage*.

After a entity is exported, it is then treated in the same way as if it was imported from a header.

Exported functions are compiled as soon as they’re met by an Onyx compiler.

Exported entities have their identifiers unmangled.

Exporting from within an Onyx namespace does not alter a entity’s identifier in any way.

An `export` statement contents may be preprocessed with Onyx macros.

Onyx annotations are applicable to `export` statements.

A non-block version of the `export` statement is terminated in accordance to the rules defined by the Section 18.1, “C standard”. For instance, a *struct* definition must be terminated with a semicolon, but a function definition does not have to. A *preprocessor directive* is terminated with a newline unless it (the newline) is preceded by a backslash.

An exported block of code is enclosed in curly brackets and does not require a semicolon.

Listing 40. The `export` statement syntax

```
c = (? Raw C code ?)

c_var_decl = (? A C variable declaration ?);
c_var_def = (? A C variable definition ?);
c_struct_decl = (? A C struct declaration ?);
c_struct_def = (? A C struct definition ?);
c_union_decl = (? A C union declaration ?);
c_union_def = (? A C union definition ?);
c_enum = (? A C enumeration definition ?);
c_typedef = (? A C typedef ?);
c_directive = (? A C preprocessor directive ?);

export = "export",
  (* An export statement may accept a raw block of C code... *)
  ("{", c, "}"),

  (* or a single C declaration or definition... *)
  c_var_decl |
  c_var_def |
  c_union_def |
  c_struct_def |
  c_union_decl |
  c_struct_decl |
  c_enum |
  c_typedef |
  c_directive |

  (* or a function definition with C prototype, but Onyx body *)
  nxc_function_def;
```

An Onyx compiler SHOULD provide a way to generate a C header file from source Onyx files, which MUST contain:

- Declarations of exported function definitions.
- Raw contents of any other `export` statement.
- Onyx comments related to `export` statements, as inline C comments.

18.3.1. Exporting functions

Exported C function

A function definition contained in a block of C code, i.e. in the block version of the `export` statement.

An exported C function is written in C and compiled by an Onyx compiler.

Exported Onyx function

A function definition immediately following the `export` keyword.

An exported Onyx function begins with a prototype written in C. The prototype ends with an opening curly bracket.

An exported Onyx function body is written in Onyx.

An exported Onyx function is terminated by a closing curly bracket matching the one the prototype ended with.

An exported Onyx function body has fragile environment.

Calling an exported Onyx function is always unsafe for consistency reasons.

If an exported Onyx function is within an Onyx namespace, Onyx lookup rules are still applicable within its body.

An exported Onyx function arguments are accessible within its body in accordance to the following table.

Table 12. Exported Onyx function argument mapping

C argument declaration	Onyx argument declaration
`T` is scalar (i.e. *number* or *union*)	
`T arg`	`let arg : T`
`const T arg`	`final arg : T`
`T* arg`	`let arg : T*urw`
`const T* arg`	`final arg : T*ur`
`T` is aggregate (i.e. *array* or *struct*)	
`T arg`	`let arg : mut T`
`const T arg`	`final arg : const T`
`T* arg`	`let arg : mut T*urw`
`const T* arg`	`final arg : const T*ur`
Other	
… (*variadic function arguments*)	Accessible via `@varg` macro
`volatile T arg`	`volatile let arg : T`
`restrict T arg`	`let arg : T` (no effect)

Listing 41. The exported Onyx function syntax

```
c_proto = (? A C function prototype, e.g. `void main(void)` ?);
nxc_function_def = c_proto, "{", {expr}, "}";
```

Example 39. Exporting

Listing 42. main.nx

```
# This is an exported Onyx function.
@[Entry] # An Onyx annotation!
export void main() {
  # Onyx body
}

# Has a non-external linkage
export static _Alignas(8) const int i = 42;

# This comment is not going to be exported
# because it is not related to the statement

export struct struct_t {
  int a, b;
};

export #ifndef FOO
export #define FOO 42, \
                   43

export #endif

export {
  // All C comments would be exported
  // # Onyx comments aren't allowed here

  _Atomic _Alignas(double) int j;

  int k;

#define OUTPUT(arg) puts ( #arg );

  // This is an exported C function.
  static void foo () {
    // C code
  }
}
```

A possible variant of a generated C header:

Listing 43. main.h

```
// This is an exported Onyx function.
void main();
static _Alignas(8) const int i = 42;
struct struct_t {
  int a, b;
};
#ifndef FOO
#define FOO 42, \
            43
#endif
  // All C comments would be exported
  // # Onyx comments aren't allowed here
  _Atomic _Alignas(double) int j;

  int k;

  #define OUTPUT(arg) puts ( #arg );

  // This is an exported C function.
  static void foo () {
    // C code
  }
```

Appendix A: Compiled Syntax

The following is the entire compiled Onyx language syntax.

Listing 44. Onyx language syntax

```
binary = "0" | "1";
octal = binary | "2" | "3" | "4" | "5" | "6" | "7" | "8";
digit = octal | "8" | "9";

(* Design rationale: hexadecimals are in upper-case to clearly
  distingusigh them from exponents and prefixes (`Q` and `D`
  are exceptions). Multiplier prefixes can not be used in
  hexadecimal literals, so they don't mix. *)
hex = digit | "A" | "B" | "C" | "D" | "E" | "F";

(* Any Unicode abstract character, including
  those consisting of multiple codepoints. *)
unicode = (? A Unicode character ?);

fragile_statement =
  "fragile!",
  expr | block();

unsafe_statement =
  "unsafe!",
  expr | block();

safety_modifier =
  "threadsafe" |
  "fragile" |
  "unsafe" |
  "undefsafe";

(* TODO: Move. *)
compl_mod =
  (* !keyword(mod) *) "compl" |
  (* !keyword(mod) *) "incompl";

reopen_statement =
  [compl_mod],

  (* !keyword(statement) *)
  "reopen",

  type_ref,
  ";" | block();

(* A unit type declaration. *)
unit =
```

```
  type_visibility_mod,

  (* !keyword(Unit type declaration) *)
  "unit",

  id,
  [generic_args_decl],

  ";" | block();

trait_decl =
  trait_decl_mod,
  ["decl"],
  "trait",
  id,
  [nb, generic_arguments_decl],
  ";" | body();

trait_decl_mod = [type_visibility_mod];

derive =
  ["undefstor" | "static" | "instance"],
  "derive",
  id,
  ";" | block();

require_directive =
  "require",

  @unord(
    (* Path(s) of the required file *)
    (["from"], path, {",", path}),

    (* The base path *)
    ["at", path]
  );

(*
  Import a file written possibly
  in a language other than Onyx.

  ```
  import from "stdio.h" in "C"
  import in "Rust" from "main.rs", "aux.rs" at "/mypath/"
  ```
*)
import_directive =
  "import",

  @unord(
    (* The source language of an imported file *)
```

```
    ("in", string),

    (* Paths to the imported files *)
    ("from", string, {",", string}),

    (* The base path for the imported files *)
    ["at", string]
  );

using =
  using_namespace |
  using_refinement |
  using_alias;

using_namespace =
  "using",
  ["namespace"],
  type_ref;

using_refinement =
  "using",
  ["refinement"],
  type_ref;

using_alias =
  "using",
  ["alias"],
  decl,
  ("=", "to"),
  ref;

multiplier_prefix_iec =
  "Yi" | "Zi" | "Ei" | "Pi" |
  "Ti" | "Gi" | "Mi" | "Ki";

multiplier_prefix_si =
  "Y" | "Z" | "E" | "P" | "T" | "G" | "M" | "k" | "h" | "da" |
  "d" | "c" | "m" | "u" | "µ" | "n" | "p" | "f" | "a" | "z" | "y";

sint_suffix = ["s"], "i", {digit}; (* 32 bits by default *)
uint_suffix = "u", ["i"], {digit}; (* 32 bits by default *)
size_suffix = ["s" | "u"], "z"; (* Unsigned by default *)

fbin_suffix = "f", ["b"], {digit}; (* 64 bits by default *)
fdec_suffix = ["f"], "d", {digit}; (* 64 bits by default *)

(*
  Design rationale: despite of the fact that `p` can only be in
  lowercase in Onyx, it'd be confusing to see `42P8` and treat it
  other than a "42 * 2 ** 8". Had to pick another symbol therefore.
*)
```

```
posit_suffix = "t", {digit};
posit_suffix = "p", {digit};

(*
  Design rationale: would not use `B`, because
  brain float is "smaller" than usual float.
*)
bfloat_suffix = "bf", {digit};

literal_suffix =
  sint_suffix |
  uint_suffix |
  size_suffix |
  fbin_suffix |
  fdec_suffix |
  xbin_suffix |
  xdec_suffix |
  posit_suffix |
  bfloat_suffix;

sign = "-" | "+";

non_decimal_value ($prefix, $base) =
  $prefix, {$base | "_"}, $base,
  [".", $base, {$base | "_"}],
  ["p", [sign], {digit}-]

numeric_literal =
  [sign],
  (
    non_decimal_value("0b", binary) |
    non_decimal_value("0o", octal) |
    non_decimal_value("0x", hex) |
    (
      digit, {
        (*
          Multiplier prefixes allow to
          have multiple radix points.
        *)
        ["."], digit,

        (* The prefix families must not be mixed. *)
        {digit | "_" | multiplier_prefix_iec | multiplier_prefix_si}
      },
      ["e", [sign], {digit}-]
    )
  ),
  {"_"},
  [literal_suffix];

(* For consistency, it is still
```

```
  referenced as "`XBin` suffix". *)
xbin_suffix =
  (* An optional signedness. *)
  ["s" | "u"],

  (* The binary fixed-point (a.k.a.
    Q number) literal symbol. *)
  "Q",

  (* An optional bitsize. *)
  {digit},

  (* An optional Fractional part size in bits. Can also
    be read as an amount of bits to left-shiFt by. *)
  ["f", [sign], {digit}-];

xdec_suffix =
  (* An optional signedness. *)
  ["s" | "u"],

  (* The decimal fixed-point literal symbol. *)
  "D",

  (* An optional total amount of digits. *)
  {digit},

  (* An optional amount of digits which are fractional. *)
  ["f", {digit}-];

numerical_codepoint =
  ("\o", {octal, "_"}-, "_"-) |
  ("\o", "{", {octal, "_"}-, "_"-, "}") |

  ("\", {digit, "_"}-, "_"-) |
  ("\", "{", {digit, "_"}-, "_"-, "}") |

  ("\x", {hex, "_"}-, "_"-) |
  ("\x", "{", {hex, "_"}-, "_"-, "}");

character_literal = "'", unicode | {numerical_codepoint}, "'";

string_literal = "\"", {unicode | numerical_codepoint}, "\""
string_literal = "/", {unicode}, "/";

(* NOTE: Opening and closing sequences must be equal! *)
heredoc =
  "<<-",
  (
    "(", seq(en | underscore), ")",
    {literal_suffix}
  ) | seq(en | underscore),
```

```
  nl,
  seq(unicode),
  nl,
  seq(en | underscore);

magic_literal_values =
  [
    literal_value,
    {" ", literal_value}
  ];

magic_tuple_literal =
  "(", magic_literal_values, ")";

magic_array_literal =
  "[", magic_literal_values, "]";

magic_vector_literal =
  "<", magic_literal_values, ">";

magic_tensor_literal =
  "|", magic_literal_values, "|",
  [tensor_literal_dimension_appendix];

(* If literal suffixes are missing, the container
  type is inferred from its values. *)
magic_literal =
  "%",
  {literal_suffix}-,
  (
    magic_tuple_literal |
    magic_array_literal |
    magic_vector_literal |
    magic_tensor_literal
  );
```

Appendix B: Optimization

Informative.

This section acts as a single source of expectations on when it is possible to rely on a compiler for certain optimizations rather than write optimized code manually. This allows a user to write simpler code but be sure on that this code would be optimized.

Appendix C: NXAPI

An Onyx compiler is able to generate documentation for exported functions only in form of C headers. C standard for header files is applied in such cases.

For Onyx language APIs, the current standard is applied, named NXAPI.

The NXAPI standard defines a schema to document entities declared in an Onyx program. The standard also provides standards for common interchange formats, including binary, JSON, DSON, YAML, XML and TOML.

Herein "API" stays for Onyx language API if not mentioned another.

API documentation is intended to provide the exact information as in the source file. Due to the inferring freedom of the Onyx language, it is often impossible to predetermine the conditions an API is going to be used in.

For example, a `Foo` type mentioned from within an API function arguments list could possibly mean another type if the namespace containing the functions gets an type also named `Foo` in a program-consumer of the API. However, full path like `::Foo` is unambiguous and is considered a good practice.

Similarly, while it is possible to infer a function return type in the scope of the API, it (the return type) may turn out to be different when the API is used in a consumer program. Therefore, library API authors are encouraged to provide concrete, unambiguous return types to their functions whenever possible.

In fact, Onyx compilers are expected to have some kind of a strict compilation mode, which would require concrete typing everywhere. {paper-noindex}

NXAPI schema includes documentation for namespaces, types, functions, annotations and macros.

C.1. Schema

NXAPI schema definition describes a set of available NXAPI nodes. {paper-excerpt=25,-1}

<div class="toc" paper-noindex>

```
{{ paper-toc }}
```

C.2. Structure

Each node in the schema definition has a `kind` value. This value is used to determine the kind of a node in an NXAPI document.

In the schema definition, a node also has a set of `fields`, which defines the set of properties for the

node.

The schema definition uses the DSON format for simplicity.

The implicit root object of a NXAPI document is the anonymous top-level namespace, therefore an NXAPI document is an array of the top-level namespace declarations.

Consider the following example: {paper-noindex}

```
# That is an actual NXAPI document
[
  {
    kind = namespace,    # Kind of this node is a namespace
    path = "::Foo::Bar", # A `path` field with some value
    declarations = []    # A `declaration` field with some values
  }
]
```

C.3. Nodes

List of NXAPI nodes. {paper-excerpt=0,-1 paper-noindex}

C.3.1. Special nodes

There are some special nodes which can not be reasonably defined in the schema. {paper-excerpt=16,65 paper-noindex}

A `decl` node is one of:

- [`namespace_decl`](#nxapi-namespace_decl){paper-link}; or
- [`struct_decl`](#nxapi-struct_decl){paper-link}; or
- [`var_decl`](#nxapi-var_decl){paper-link}; or
- [`function_decl`](#nxapi-function_decl){paper-link}; or
- [`annotation_decl`](#nxapi-annotation_decl){paper-link}; or
- [`macro_decl`](#nxapi-macro_decl){paper-link}.

An `expr` node…

C.3.2. `delayed_macro_expr`

A delayed macro expression node could be used instead of any other node. {paper-excerpt=31}

```
{
  kind = delayed_macro_expr,
  fields = [{
    name = expr,
    type = string,
    required = true
  }]
}
```

C.3.3. namespace_decl

A namespace declaration node contains its name and declarations. {paper-excerpt=28}

A namespace declaration can not be virtual.

Namespace and type declarations must have fully resolved paths, e.g. `::Foo::Bar`. {#nxapi-namespace-full-path}

```
{
  kind = namespace_decl,
  fields = [{
    name = doc,
    type = string,
    required = false
  }, {
    name = path,
    type = string,
    required = true
  },{
    name = declarations,
    type = array<decl>,
    required = true # May be empty
  }]
}
```

C.3.4. var_decl

A variable declaration node may be virtual. {paper-excerpt=28}

```
{
  kind = var_decl,
  fields = [{
    name = doc,
    type = string,
    required = false
  }, {
    name = virtual,
    type = bool,
    required = true
  }, {
    name = accessibility,
    type = enum<let, get, set, final>,
    required = true
  }, {
    name = storage,
    type = enum<static, instance>,
    required = undefined # Only applicable to struct variables
  }, {
    name = visibility,
    type = enum<public, protected, private>,
    required = false # Undefined if missing
  }, {
    name = name,
    type = string,
    required = true
  }, {
    name = restriction,
    type = type_ref,
    required = false
  }, {
    name = value,
    type = expr,
    required = false
  }, {
    name = annotations,
    type = array<annotation_call>,
    required = true # May be empty
  }]
}
```

C.3.5. function_decl

A function declaration node may be virtual. {paper-excerpt=27}

```
{
  kind = function_decl,
  fields = [{
    name = doc,
```

```
    type = string,
    required = false,
  }, {
    name = virtual,
    type = bool,
    required = true,
  }, {
    name = implementation,
    type = enum<decl, impl, def, redef>,
    required = true,
  }, {
    name = storage,
    type = enum<static, instance>,
    required = undefined # Only applicable to trait,
                         # struct and enum functions
  }, {
    name = mutability,
    type = enum<mut, const>,
    required = false # Undefined if missing
  }, {
    name = visibility,
    type = enum<public, protected, private>,
    required = false # Undefined if missing
  }, {
    name = name,
    type = string,
    required = true
  }, {
    name = arguments,
    type = array<arg_decl>,
    required = true # May be empty
  }, {
    name = return,
    type = type_ref,
    required = false # Undefined if missing
  }, {
    name = annotations,
    type = array<annotation_call>,
    required = true # May be empty
  }, {
    name = foralls,

    # A function declaration may include multiple foralls, e.g.
    # `def foo(arg : T) forall T : List<U> forall U : Numeric`
    type = array<array<forall_element>>,

    required = true # May be empty
  }, {
    # An optional `where` macro expression,
    # e.g. `def foo() where \{{ expr }}`
    #
```

```
    name = where,
    type = delayed_macro_expr,
    required = false
  }]
}
```

C.3.6. `forall_element`

A `forall` element declaration, e.g. `forall T : U`. {paper-noindex}

```
{
  kind = forall_element,
  fields = [{
    name = name,
    type = string,
    required = true
  }, {
    name = restriction,
    type = type_ref,
    required = false
  }]
}
```

C.3.7. `arg_decl`

An argument declaration node. {paper-excerpt=0,-1 paper-noindex}

The semantics is shared for both function and generic type argument declarations, hence a single node. {#arg-shared-semantics}

TODO: Move these to main spec, and just reference from here.

Generic type arguments are always type-restricted, unlike function arguments which may be either instance- or type-restricted:

```
struct Int<Bitsize: S :: %unum>
  # `another` is instance-restricted.
  def add(another : T) forall T :: Int;

  # `target` is type-restricted.
  def to(target: :: T) forall T :: Int;
end
```

```
{
  kind = arg_decl,
  fields = [{
    name = doc,
    type = string,
    required = false
  }, {
    name = annotations,
    type = array<annotation_call>,
    required = true # May be empty
  }, {
    name = name,
    type = string,
    required = undefined # Must be absent if it is a
                         # `type_restricted` function argument
  }, {
    name = alias,
    type = string,
    required = false
  }, {
    # Determine whether is it a type-restricted argument,
    # e.g. `foo :: T`. A type-restricted argument must not have its
    # `restriction` field left empty and if it is a function argument,
    # it also can not have `name` field set.
    name = type_restricted,
    type = bool,
    required = true,
  }, {
    name = restriction,
    type = type_ref,
    required = undefined # Required only if it is
                         # a type-restricted argument
  }, {
    name = default_value,
    type = expr,
    required = false
  }]
}
```

C.3.8. arg_pass

An argument node passed to a callee. {paper-excerpt=0,-1 paper-noindex}

Due to [the shared argument semantics](#arg-shared-semantics){paper-link} argument passing to a callee is defined as a single node.

```
{
  kind = arg_pass,
  fields = [{
    # An explicit name of an argument, e.g. `call(foo: bar)`
    name = explicit_name,
    type = string,
    required = undefined # Incompatible with `explicit_order`
  }, {
    # An explicit order of an argument, e.g. `call([0]: bar)`
    name = explicit_order,
    type = number,
    required = undefined # Incompatible with `explicit_name`
  }, {
    name = value,
    type = expr,
    required = true
  }]
}
```

C.3.9. type_expr

A valid type expression node. {paper-excerpt paper-noindex}

```
{
  kind = type_expr,
  fields = [{
    name = expr,
    type = array<variant<type_expr_op, type_ref>>,
    required = true
  }]
}
```

C.3.10. type_expr_op

A type expression operator node. {paper-excerpt paper-noindex}

```
{
  kind = type_expr_op,
  fields = [{
    name = op,

    # One of `(`, `)`, `&`, `|`, `^` or `!`
    type = enum<paren_open, paren_close, and, or, xor, not>,

    required = true
  }]
}
```

C.3.11. `type_ref`

A reference to a type node. {paper-excerpt paper-noindex}

A type reference node `path` field may be unresolved.

```
{
  kind = type_ref,
  fields = [{
    name = path,
    type = string,
    required = true
  }, {
    name = generic_arguments,
    type = array<arg_pass>,
    required = true # May be empty
  }]
}
```

C.3.12. `trait_decl`

C.3.13. `struct_decl`

A struct declaration node `name` field [must be a fully resolved path](#nxapi-namespace-full-path){paper-link}. {paper-excerpt="0,26,."}

```
{
  kind = struct_decl,
  fields = [{
    name = doc,
    type = string,
    required = false
  }, {
    name = virtual,
    type = bool,
    required = true
  }, {
    name = path,
    type = string,
    required = true
  }, {
    name = mutability,
    type = enum<mut, const>,
    required = false # Undefined if missing
  }, {
    name = visibility,
    type = enum<public, private>,
    required = false # Undefined if missing
  }, {
    name = annotations,
    type = array<annotation_call>,
    required = true # May be empty
  }, {
    name = declarations,
    type = array<decl>,
    required = true # May be empty
  }]
}
```

C.3.14. enum_decl

A enum declaration node. {paper-excerpt paper-noindex}

```
{
  kind = enum_decl,
  fields = [{
    name = statement,
    type = enum<virtual, reopen, declare>,
    required = true
  }, {
    name = is_flag,
    type = bool,
    required = true
  }, {
    name = doc,
    type = string,
    required = false
  }, {
    name = type,
    type = type_ref,
    required = undefined # Required if `is_flag`,
                         # otherwise `SInt32` if missing
  }, {
    name = values,
    type = array<enum_val_decl>,
    required = true # May be empty for virtual enums
  }, {
    name = declarations,
    type = array<decl>,
    required = true # May be empty
  }]
}
```

C.3.15. enum_val_decl

A enum value declaration node. {paper-excerpt paper-noindex}

```
{
  kind = enum_val_decl,
  fields = [{
    name = doc,
    type = string,
    required = false
  }, {
    name = name,
    type = string,
    required = true
  }, {
    name = value,
    type = expr,
    required = false
  }]
}
```

C.3.16. annotation_decl

An annotation declaration node. {paper-noindex}

```
{
  kind = annotation_decl,
  fields = [{
    name = is_virtual,
    type = bool,
    required = true
  }, {
    name = doc,
    type = string,
    required = false
  }, {
    name = path, # A fully resolved path
    type = string,
    required = true
  }, {
    name = arguments,
    type = array<arg_decl>,
    required = true # May be empty
  }]
}
```

C.3.17. annotation_call

An annotation call can not have any documentation.

```
{
  kind = annotation_call,
  fields = [{
    name = annotation,
    type = type_ref,
    required = true
  }, {
    name = arguments,
    type = array<arg_pass>,
    required = true # May be empty
  }]
}
```

C.3.18. `intrinsic_decl`

An intrinsic (i.e. `macro`) declaration node. {paper-excerpt paper-noindex}

```
{
  kind = intrinsic_decl,
  fields = [{
    name = doc,
    type = string,
    required = false
  }, {
    name = annotations,
    type = array<annotation_call>,
    required = true # May be empty
  }, {
    name = is_virtual,
    type = bool,
    required = true
  }, {
    name = visibility,
    type = enum<public, private>,
    required = true
  }, {
    name = name,
    type = string, # Without `@`
    required = true
  }, {
    name = arguments,
    type = array<argument_decl>,
    required = true # May be empty
  }]
}
```

C.3.19. `delayed_intrinsic_call`

A delayed intrinsic call node may replace any non-declaration node.

```
{
  kind = delayed_intrinsic_call,
  fields = [{
    name = content,
    type = string,
    required = true
  }]
}
```

C.4. Examples

```
require "io" from "std"

# This is the main function.
@[Entry]
def main
  @cout << "Hello, world!"
end
```

```
[{
  kind = function,
  implementation = def,
  name = main,
  documentation = "This is the main function."
  annotations = [{
    kind = annotation,
    type = Entry
  }]
}]
```

Appendix D: Core API

```
trait Core::Aggregate;

require "./aggregate.nx"

decl primitive Core::Array<Type: T, Size ~ %z>
  derive Core::Aggregate;

  # Return a reference to an element at *index*.
  decl def [](index: Size) : T&ir(w: \mut?) throws Core::SizeError
end

require "./scalar.nx"

# Operations on atomic types may be synchronized, depending on
# provided memory ordering constraints and fences.
# Atomic operations are still `fragile` and require extra attention
# from the developer to become truly `threadsafe`.
#
# The memory layout of an atomic type is undefined.
#
# Theoretically, any `Numeric` type can be wrapped in `Atomic`.
# In practice, only `Int`, `Float`, `Fixed` and their
# `Imaginary` counterparts are expected to so.
#
# `Int` and `Float` variants are expected to be interchangeable
# with their `_Atomic` counterparts, where appropriate
# (see `as` method declarations in according reopenings).
decl primitive Core::Atomic<Type: T ~ (Numeric)>
  derive Scalar;

  # Non-atomically retreive the underlying value.
  # This method is allowed to be called on a `final` variable.
  decl raw_get : T

  # Non-atomically update the underlying value.
  # Does not return anything to avoid confusion with `swap`.
  decl raw_set(val : T) : Void

  # Atomically fetch the underlying value.
  # This method is allowed to be called on a `final` variable.
  #
  # ```
  # final a = Core::Atomic(42)
  # Core.assert(a == a.fetch == 42)
  # ```
  decl fetch(
    ordering: Memory::Ordering = :seqcst,
```

```
  syncscope: SS = ""
) : T forall SS : @~string
alias == to fetch

# Atomically store the underlying value.
# Does not return anything.
# To store and also get the old value, use `swap` instead.
#
# ```
# let a = Core::Atomic(42)
# a.store(43)
# Core.assert(a == 42)
# ```
#
# TODO: Do we really need this method?
decl store(
  val: T,
  ordering: Memory::Ordering = :seqcst,
  syncscope: SS = ""
) : Void forall SS : @~string

# Atomically swap the underlying value.
# Returns the old value.
#
# ```
# let a = Core::Atomic(42)
# Core.assert(a = 43 == 42)
# ```
decl swap(
  val: T,
  ordering: Memory::Ordering = :seqcst,
  syncscope: SS = ""
) : T forall SS : @~string
alias = to swap

# TODO:
decl compare_and_exchange(
  old: T,
  new: T,
  success_ordering: Memory::Ordering = :seqcst,
  failure_ordering: Memory::Ordering = :seqcst,
  syncscope: SS = ""
) : Bool forall SS : @~string
alias cmpxchg to compare_and_exchange

# Atomically add to the underlying value.
# Returns the old value.
#
# Wraps if called on a `Rational`.
#
# ```
```

```
  # let a = Core::Atomic(127i8)
  # Core.assert(a += 1 == 127)
  # Core.assert(a == -128)
  # ```
  decl add(
    val: T,
    ordering: Memory::Ordering = :seqcst,
    syncscope: SS = ""
  ) : T forall SS : @~string
  alias += to add

  # Atomically subtract from the underlying value.
  # Returns the old value.
  #
  # Wraps if called on a `Rational`.
  #
  # ```
  # let a = Core::Atomic(-128i8)
  # Core.assert(a -= 1 == -128)
  # Core.assert(a == 127)
  # ```
  decl subtract(
    val: T,
    ordering: Memory::Ordering = :seqcst,
    syncscope: SS = ""
  ) : T forall SS : @~string
  alias sub, -= to subtract
end

reopen Core::Atomic<Int>
  # Atomically apply AND operation to the underlying value.
  # Returns the old value.
  #
  # ```
  # let a = Core::Atomic(0b1001_0110)
  # Core.assert(a &= 0b0011_0011 == 0b1001_0110)
  # Core.assert(a == 0b0001_0010)
  # ```
  decl and(
    val: T,
    ordering: Memory::Ordering = :seqcst,
    syncscope: SS = ""
  ) : T forall SS : @~string
  alias &= to and

  # Atomically apply NAND operation to the underlying value.
  # Returns the old value.
  #
  # ```
  # let a = Core::Atomic(0b1001_0110)
  # Core.assert(a.nand(0b0011_0011) == 0b1001_0110)
```

```
  # Core.assert(a ==   0b1110_1101)
  # ```
  decl nand(
    val: T,
    ordering: Memory::Ordering = :seqcst,
    syncscope: SS = ""
  ) : T forall SS : @~string

  # Atomically apply OR operation to the underlying value.
  # Returns the old value.
  #
  # ```
  # let a = Core::Atomic(0b1001_0110)
  # Core.assert(a |= 0b0011_0011 == 0b1001_0110)
  # Core.assert(a == 0b1011_0111)
  # ```
  decl or(
    val: T,
    ordering: Memory::Ordering = :seqcst,
    syncscope: SS = ""
  ) : T forall SS : @~string
  alias |= to or

  # Atomically apply XOR operation to the underlying value.
  # Returns the old value.
  #
  # ```
  # let a = Core::Atomic(0b1001_0110)
  # Core.assert(a ^= 0b0011_0011 == 0b1001_0110)
  # Core.assert(a == 0b0100_1010)
  # ```
  decl xor(
    val: T,
    ordering: Memory::Ordering = :seqcst,
    syncscope: SS = ""
  ) : T forall SS : @~string
  alias ^= to xor

  # Convert to an `_Atomic` integer counterpart.
  #
  # MUST be implemented if the target supports `_Atomic` types,
  # i.e. `$__STDC_NO_ATOMICS__` is not defined.
  #
  # Non-fixed width integer types (e.g. `$int`) MUST be supported.
  # Fixed (from `stdint.h`) width integer types SHOULD be supported.
  #
  # Signednesses of the types must match. A `final` atomic type
  # can not be converted to a non-const type.
  #
  # It is not possible to directly convert to a lesser-bitsize type.
  # Target information contained in the Macro API may be used
```

```
  # to aid in implementing proper conversion logic.
  #
  # ```
  # final a = Core::Atomic(42) : Core::Atomic<$int>
  # a as $`const _Atomic int` # OK
  # # a as $`_Atomic int` # Panic! Constantness mismatch
  # # a as $`const _Atomic unsigned int` # Panic! Signedness mismatch
  # a as $`const _Atomic int32_t` # May work, depending on target
  #
  # let b = Core::Atomic(42u16)
  # a as $`const _Atomic unsigned short` # OK, turns into const
  # ```
  @[MayPanic("If target bitsize is less than the type's")]
  @[MayPanic("On signednesses mismatch")]
  @[MayPanic("On constantnesses mismatch")]
  virtual decl as(:: T) forall T
end

reopen Core::Atomic<Float>
  # Convert to an `_Atomic` C floating-point type.
  #
  # MUST be implemented if the target supports `_Atomic` types,
  # i.e. `$__STDC_NO_ATOMICS__` is not defined.
  #
  # The target memory layout is defined by the
  # `nx.c.const.[type].layout` macro constraint.
  #
  # Direct conversion is only possible to the same
  # or greater bitsize floating point type, defined by the
  # `nx.c.const.[type].bitsize` constraint.
  # Otherwise, the `as!` method should be used.
  #
  # ```
  # let a = Core::Atomic(42f64) : Core::Atomic<FBin64>
  #
  # # Is likely to succeed, unless `double`
  # # has bitsize greater than 64
  # a as $double # OK
  #
  # # as as $float # Most likely to fail
  # ```
  @[MayPanic("On a possibility of precision loss")]
  virtual decl as(:: T) forall T

  # Convert to an `_Atomic` C floatin-point type,
  # allowing possible precision loss.
  #
  # ```
  # let a = Core::Atomic(42f64) : Core::Atomic<FBin64>
  # a as! $float # OK, but lesser precision
  # ```
```

```
  virtual decl as!(:: T) forall T
end

require "./numeric.nx"
require "./hypercomplex.nx"
require "./imaginary.nx"

# The memory layout of a complex number matches such of an array
# of two numbers of the type `T`, whereas the first element
# is the real part, and the second element is the imaginary part.
decl primitive Core::Complex<Type: T ~ Real>
  # A complex number is a specialization of a hypercomplex number.
  derive Hypercomplex<T>;

  # A built-in constructor for a complex number.
  #
  # ```
  # using Core
  #
  # assert(Complex(1, 2j) : Complex<$int> == Complex(1, .(2)))
  # assert(Complex<$double>() == Complex(0.0, .(0))
  # ```
  static nothrow decl ()(
    real : T = 0,
    imaginary : Imaginary<T> = .(0))

  # Return a reference to the real part of the complex number.
  #
  # NOTE: Would return a read-only reference if the number is final.
  decl nothrow real() : T&irw

  # Return a reference to the imaginary part of the complex number.
  #
  # NOTE: Would return a read-only reference if the number is final.
  decl nothrow imaginary() : Imaginary<T>&irw

  # Convert the complex number to the same
  # or higher-bits representation.
  # Would call `as!` on its parts recusively.
  @[MayPanic("If data loss is possible")]
  @[MayPanic("In case of underlying type mismatch")]
  decl nothrow as(~ Complex)

  # Convert the complex number to another (including lower-)
  # bits representaion, allowing data loss.
  # Would call `as!` on its parts recusively.
  @[MayPanic("In case of underlying type mismatch")]
  decl nothrow as!(~ Complex)

  {% for t, _ in ipairs{"float", "double", "long double"} do %}
    # Convert to the same or higher-bits complex floating C type.
```

```
    #
    # NOTE: This method is only declared if `T ~ Float`.
    @[MayPanic("If data loss is possible")]
    virtual decl nothrow as(: $`{{ t }} _Complex`)

    # Convert to another (including lower-) bits
    # complex floating C type, allowing data loss.
    #
    # NOTE: This method is only declared if `T ~ Float`.
    virtual decl nothrow as!(: $`{{ t }} _Complex`)
  {% end %}
end

struct Core::SizeError;

require "./fractional.nx"

# A fixed-point number.
decl primitive Fixed<
  Base: 2 || 10,

  # Size of the integral part in `Base`.
  Integral: ~%Natural = 0,

  # Size of the fractional part in `Base`.
  # May be negative for right-shift.
  Fractional: ~%Integer = 0
>
end

alias XBin<*> = Fixed<2, *>
alias XDec<*> = Fixed<10, *>

require "./fractional.nx"

# An IEEE-754 floating-point number.
decl primitive Float<
  Base: ~%Natural,
  Precision: ~%Natural,
  EMax: ~%Natural>
  derive Fractional;
end

alias FBin<
  SignificandBits: ~%Natural,
  ExponentBits: ~%Natural
> = Float<
  2,
  Precision: SignificandBits,
  EMax: \{{ nx.util.natural(2 ^ (nx.ctx.ExponentBits - 1) - 1) }}
>
```

```
alias FBin16 = FBin<11, 5> # Not basic, but often implemented
alias FBin32 = FBin<24, 8>
alias FBin64 = FBin<53, 11>
alias FBin128 = FBin<113, 15>

alias FDec<
  Digits: ~%Natural,
  ExponentBits: ~%Natural
> = Float<
  10,
  Precision: Digits,
  EMax: \{{ nx.util.natural(3 * 2 ^ (nx.ctx.ExponentBits - 2) / 2) }}
>

alias FDec32 = FDec<7, 8> # Not basic, but often implemented
alias FDec64 = FDec<16, 10>
alias FDec128 = FDec<34, 14>

# It is only implemented for targets supporting the extended
# double-precision floating-point number format natively.
decl type FBin64E;

# The [Brain floating-point number format][1]
# is ubiquitous in AI computing.
#
# [1]: https://en.wikipedia.org/wiki/Bfloat16_floating-point_format.
alias BFloat16 = FBin<8, 8>

require "./real.nx"

# A fractional, i.e. non-whole, rational number.
trait Fractional
  derive Rational;
end

require "./numeric.nx"

trait Core::Hypercomplex<Type: T ~ Real>
  derive Numeric;
end

require "./real.nx"

# ```
# final i = 5j
# assert(i :? Imaginary<SInt32>)
# assert(i ** 2 == -25)
# ```
@[LiteralSuffix(/j/)]
decl primitive Imaginary<Type: T ~ Real>
```

```
  derive Numeric;

  decl initialize(~ T)

  # # Cast a `Real` literal into an `Imaginary` literal.
  # @[LiteralSuffix(/j/)]
  # static decl (~ %(Real)) : %(Imaginary)

  # Coerce to the same or higher bits.
  # Would call `as` on the underlying value.
  #
  # ```
  # 1uj.as_u64j : Imaginary<UInt64>
  # # 1uj.as_u16j # Panic!
  #
  # 1ij.as_i32 : SInt32
  # # 1ij.as_i16 # Panic!
  # ```
  @[MayPanic("If data loss is possible")]
  decl nothrow as(~ (\Imaginary | \Real))

  # Coerce to other (including lower) bits, allowing data loss.
  # Would call `as!` on the underlying value.
  #
  # ```
  # 1uj.as_u16j! : Imaginary<UInt16>
  # 1j.as_i16! : SInt16
  # ```
  decl nothrow as!(~ (\Imaginary | \Real))

  {% for t, _ in ipairs{"float", "double", "long double"} do %}
    # Convert to the same or higher-bits complex floating C type.
    @[MayPanic("If data loss is possible")]
    decl nothrow as(: $`{{ t }} _Complex`)

    # Convert to another (including lower-) bits
    # complex floating C type, allowing data loss.
    decl nothrow as!(: $`{{ t }} _Complex`)
  {% end %}
end

require "./integer.nx"

# You can go from `Natural` to `Integer`, but not vice versa
%!Rational, %!□ # -0.5, 1/3, □, □
%!Integer,  %!□ # -1, 0, 1
%!Natural,  %!□ # 0, 1
%!Boolean,  %!□ # true, false

%!Range<T>  # 0..0.5
```

```
%!Tensor<T> # |[0, 0.5]|
%!Vector<T> # <0, 0.5> # Vector can be treated as `Tensor`

%!Character # 'f'
%!String    # "f"
%!Symbol    # :f

# A binary 2's complement integer.
decl primitive Int<Signed ~ %Bool, Bitsize ~ %Size>
  derive Integer;
  decl initialize(value : self)
end

alias SInt<*> = Int<true, *>
alias UInt<*> = Int<false, *>

alias SInt8 = SInt⑧
alias SInt16 = SInt⑯
alias SInt32 = SInt
alias SInt64 = SInt
alias SInt128 = SInt

alias UInt8 = UInt⑧
alias UInt16 = UInt⑯
alias UInt32 = UInt
alias UInt64 = UInt
alias UInt128 = UInt

alias Bit, Bool = UInt①
alias Byte = UInt8

# # The following aliases are implementation-defined.
# # They must be interchangeable with according C types.
# #

# # Every valid character is valid integer, but not vice versa.
# # Therefore, character types are distinct aliases.
# #

# decl distinct alias UChar # Usually `UInt8`
# decl distinct alias SChar # Usually `SInt8`
# decl distinct alias Char  # `$char` (`SChar` or `UChar`)
# decl distinct alias WChar # `$wchar_t` (C17§7.19.2)

# virtual alias SShort    # Usually `SInt16`
# virtual alias SInt      # Usually `SInt16` or `SInt32`
# virtual alias SLong     # Usually `SInt32`
# virtual alias SLongLong # Usually `SInt32` or `SInt64`

# virtual alias UShort    # Usually `UInt16`
# virtual alias UInt      # Usually `UInt16` or `UInt32`
```

```
# virtual alias ULong     # Usually `UInt32` or `UInt64`
# virtual alias ULongLong # Usually `UInt64`

# virtual alias Size, USize # `$size_t` (C17§7.19.2)
# virtual alias SSize       # Signed version of `Size`

# # The `Fast` family aliases to the implementation-defined
# # fastest types with at least provided bitsize (C17§7.20.1.3).
# # They must be interchangeable with according C types,
# # for example `UFast16` and `$uint_fast16_t`.

# virtual alias SFast8
# virtual alias SFast16
# virtual alias SFast32
# virtual alias SFast64

# virtual alias UFast8
# virtual alias UFast16
# virtual alias UFast32
# virtual alias UFast64

# # The `Least` family aliases to the implementation-defined
# # smallest types with at least provided bitsize (C17§7.20.1.2).
# # They must be interchangeable with according C types,
# # for example `ULeast16` and `$uint_least16_t`.

# virtual alias SLeast8
# virtual alias SLeast16
# virtual alias SLeast32
# virtual alias SLeast64

# virtual alias ULeast8
# virtual alias ULeast16
# virtual alias ULeast32
# virtual alias ULeast64

# # Fixed-width character types
# # must be interchangeable with
# # according C types (C17§7.28).
# #

# decl distinct alias Char16 to ULeast16; # `$char16_t`
# decl distinct alias Char32 to ULeast32; # `$char32_t`

require "./real.nx"

# A whole number.
trait Integer
  derive Real;
end
```

```
require "./aggregate.nx"

decl primitive Core::Array<Type: T, Size ~ %z>
  derive Core::Aggregate;

  # Return a reference to an element at *index*.
  decl def [](index: Size) : T&ir(w: \mut?) throws Core::SizeError
end

require "./tensor.nx"

# A matrix is a 2-dimensional specialization of `Tensor`.
#
# Matrices have literals:
#
# ```
# let m1 = |[1, 2],
#           [3, 4]|r ~ (\%n)x2x2r : (SInt32)x2x2r
#
# let m2 = %i|[1 2]
#             [3 4]|l1 : (SInt32)x2x2c
# ```
alias Core::Matrix<
  Type: T,
  Rows: R,
  Columns: C,
  Leading: L
> to Tensor<T, R, C, L>

require "./scalar.nx"

trait Core::Numeric
  derive Scalar;

  decl equals?(~ Numeric) : Bool
  alias eq?, == to equals?

  decl add(~ Numeric)
  alias + to add

  decl subtract(~ Numeric)
  alias sub, - to subtract

  decl multiply(~ Numeric)
  alias mul, * to subtract

  decl divide(~ Numeric)
  alias div, / to divide

  # Return the modulo operation result, where the result
  # is either zero or has the same sign as the argument.
```

```
  decl modulo(~ Numeric)
  alias mod, % to modulo

  # Return the remainder from the division, where the result
  # is either zero or has the same sign as the callee.
  decl remainder(~ Numeric)
  alias rem to remainder

  decl power(~ Numeric)
  alias pow, ** to power
end

# > The quotient of two directed lines in a three-dimensional space.
# > -- <cite>William Rowan Hamilton</cite>
decl primitive Core::Quaternion<Type: T ~ Real>
  derive Hypercomplex<T>;
end

namespace Quantum
  primitive Bit
    # Set the qubit to a superposition,
    # i.e. apply the Hadamard gate.
    #
    # ```
    # let q = QBit()
    # q.unset()
    # q.get() # Returns either 0 or 1
    # ```
    decl mut unset()

    # Flip the qubit state,
    # i.e. apply the Pauli-X gate.
    #
    # ```
    # let q = QBit()
    # q = 1
    # assert(~q == 0)
    # ```
    decl mut flip()
    alias ~ to flip

    # Set the qubit state.
    #
    # ```
    # let q = QBit()
    # q = 1
    # assert(q == 1)
    # ```
    #
    # NOTE: Change the underlying variable value with `q := QBit()`.
    decl mut set(state : ::Bit)
```

```
    alias = to set

    # Measure the qubit value.
    # Returns a binary bit: either 0 or 1.
    #
    # #!see(set)
    decl get() : ::Bit

    # Measure the qubit value and compare it with another qubit's.
    decl equals?(qubit : self) : Bool

    # Measure the qubit value and compare it with another value.
    decl equals?(value : ::Bit) : Bool

    alias eq?, == to equals?
  end
end

alias QBit, Qubit to Quantum::Bit

require "./scalar.nx"

# TODO: IEEE-1788.

# 0..3.each(1).to_a == [0, 1, 2, 3] # %R[0, 3], %Rf[0 3]
# 0...3.each(1).to_a == [0, 1, 2]   # %R[0, 3), %Ri[0 3)
# 0....3.each(1).to_a == [1, 2]     # %R(0, 3), %Ri32j(0 3)
# Range<SInt32, false, true>(0, 3)  # %R(0, 3], %RQ8e-6(0 3]

decl primitive Range<
  Type: T ~ Numeric,
  BoundLower: ~ %Boolean = true,
  BoundUpper: ~ %Boolean = true
>
  derive Scalar;

  decl ()(lower: T,upper: T) : self

  decl .lower() : T
  decl .upper() : T

  decl lower=(self&w, T) : T
  decl upper=(self&w, T) : T

  decl .includes?(T) : Bool
  alias ∋ to includes?

  # Is self subset of another?
  decl .sub?(self) : Bool
  alias ⊂ to sub?
```

```
  # Is self superset of another?
  decl .super?(self) : Bool
  alias ⊃ to super?

  # Is self subset and not equals to another?
  decl .strictsub?(self) : Bool
  alias ⊂ to strictsub?

  # Is self superset and not equals to another?
  decl .strictsuper?(self) : Bool
  alias ⊃ to strictsuper?
end

# let range = %R[1, 10]
# range.lower = 0

# trait Foo
#   decl .foo()
# end

# struct Point
#   let (.x, .y, .z) : FBin64
#   let stat : SInt32 = 42

#   def .double: self
#     self(x * 2, y * 2)
#   end
# end

# TODO: A reference (T&) does not neccessarily occupies any memory!

require "./fractional.nx"

# A rational number.
decl primitive Rational<Type: T ~ Integer>
  derive Fractional;

  # An arithmetic overflow.
  struct Overflow;
end

alias Rat<Bitsize: B>, SRat<Bitsize: B> = Rational<SInt<B>>
alias URat<Bitsize: B> = Rational<UInt<B>>

alias Rat8, SRat8 = SRat⑧
alias Rat16, SRat16 = SRat⑯
alias Rat32, SRat32 = SRat
alias Rat64, SRat64 = SRat

alias URat8 = URat⑧
alias URat16 = URat⑯
```

```
alias URat32 = URat
alias URat64 = URat

alias Rat, SRat = Rational<SInt>
alias URat = Rational<UInt>

alias ShortRat, SShortRat = Rational<SShort>
alias IntRat, SIntRat = Rational<SInt>
alias LongRat, SLongRat = Rational<SLong>
alias LongLongRat, SLongLongRat = Rational<SLongLong>

alias UShortRat = Rational<UShort>
alias UIntRat = Rational<UInt>
alias ULongRat = Rational<ULong>
alias ULongLongRat = Rational<ULongLong>

alias FastRat<Bitsize: B>, SFastRat<Bitsize: B> = Rational<SFast<B>>
alias UFastRat<Bitsize: B> = Rational<UFast<B>>

alias FastRat8, SFastRat8 = FastRat⑧
alias FastRat16, SFastRat16 = FastRat⑯
alias FastRat32, SFastRat32 = FastRat
alias FastRat64, SFastRat64 = FastRat

alias UFastRat8 = UFastRat⑧
alias UFastRat16 = UFastRat⑯
alias UFastRat32 = UFastRat
alias UFastRat64 = UFastRat

require "./numeric.nx"

# A real number.
trait Real
  derive Numeric;
end

trait Core::Scalar;

# Slice maintains a bit-list of defined elements.
#
# ```
# let s = mut Slice<SInt8, 5>()
#
# assert(@bitsizeof(s) == 45) # 5 * 8 + 5
# assert(@sizeof(s) == 6) # (45f / 8).ceil.to_z
# assert(s.size != s<Size>)
#
# assert(s.size == 0)
# # s[0] # Would throw, because no element is defined yet
# assert(s[0] = 42 == 42)
# assert(s.size == 1)
```

```
# assert(s.remove(0) == 42)
# # s[0] # Would throw again
# ```
decl primitive Slice<Type: T, Size: S : @{Size}>
  # Would throw if element is not defined.
  decl mut? get(index: Size): T&i(w: \mut?) throws SizeError
  alias [] to get

  # Define or rewrite an element at *index*.
  decl mut set(index: Size, value: T): T&iw throws SizeError
  alias []= to set

  # Un-define an element at *index*.
  # Further `get` calls would throw.
  decl mut remove(index: Size): T throws SizeError

  # Return an actual size.
  decl size: Size
end

require "./aggregate.nx"

# A growing stack of values.
decl mut primitive Core::Stack<Type: T, Size: S : @~size>
  derive Aggregate;

  # NOTE: The returned reference is temporal because the stack
  # may be shrinked, so the memory region becomes undefined.
  decl push(value: T) : T&tw throws SizeError

  decl pop() : T throws SizeError
  decl pop?() : T?
  decl const size : $size

  # NOTE: The returned reference is temporal because the stack
  # may be shrinked, so the memory region becomes undefined.
  decl mut? [](index: Size) : T&t(w = \mut?) throws SizeError
end

decl primitive String<Encoding: E, Size: S : @{Size}>
  # Access character at *index*. Depending on the encoding,
  # the operation has different complexity.
  # For example, `UTF8` has O(N), while `UCS2` has O(1).
  decl [](index: Size) : Char<E>
end

trait Char::Encoding<
  # A size in bits of a single code unit.
  UnitSize: UZ : @{Size},

  # How many code units are at most
```

```
  # required to encode a code point.
  PointSize: PZ : @{Size}
>
  enum Error
    val Encoding
  end

  # Re-encode *char* in this encoding.
  #
  # ```
  # UCS2.encode('a'ascii_) == 'a'ucs2
  # ```
  decl encode(char: Char<E>): Char<this> throws Error forall E

  # A literal initialize macro for this encoding.
  #
  # The macro is called implicitly for each char or string literal.
  #
  # The restriction argument is set in accordance to
  # the autocasting rules. A macro implementation is expected
  # to accept at least `Char<this>` and `Int` restrictions.
  #
  # By default, char and string literals have `UTF16` encoding.
  # Otherwise, the encoding type is calculated by upcasing
  # the literal prefix. A prefix can not contain underscores,
  # thus an encoding type name should not as well.
  #
  # This macro is likely to be implemented natively for built-in
  # encodings. Custom encodings, however, are expected to implement
  # it in user-code if literal suffix availability is desired.
  #

  decl macro @(literal ~ @{Char}, restriction :: Char<this> || Int)
end

# Represents a single codepoint in given `Encoding`.
# A char's size in bits equals to the encoding's
# codeunit size times codepoint size.
#
# ```
# assert(@sizeof('f'ascii_) == 1)
# assert(@sizeof('f'utf8) == 4)
# assert(@sizeof('f') == 2) # UCS2 by default
# ```
primitive Char<Encoding: E ~ Encoding>
  # Initialize from an explicit
  # integral codepoint value.
  #
  # ```
  # 'f'   # Calls `.initialize(literal)`
  # '\66' # Calls THIS initializer
```

```
  # ```
  decl initialize(codepoint~ Int);

  # Initialize from a character literal.
  #
  # ```
  # 'f'   # Calls THIS initializer
  # '\66' # Calls `.initialize(codepoint)`
  # ```
  #
  # An example implementation:
  #
  # ```
  # reopen Char<Windows::CP1251>
  #   @[Inline]
  #   impl initialize(literal ~~ L) forall L ~ @{Char}
  #     # Within macros, text values are normalized to Unicode.
  #     \{% if L.value == 'ϕ' then %}
  #       return self(0xF4)
  # ```
  decl initialize(literal~~ @{Char});

  # ```
  # final a = 'a'ascii_
  #
  # assert(a.to_utf16 == a to Char<UTF16> ==
  #   a.to(Char<UTF16>) == 'a'utf16)
  # ```
  decl to(:: C) : C forall C ~ Char

  # Compare to a character with the same encoding.
  decl equals?(: self<E>): Bool
  alias eq?, == to equals?
end

%utf8["Привет!\0"] : Char<UTF8>[10] # 40 bytes = 320 bits
%<"Ditto"> : Char<UCS2>[5] # 10 bytes = 80 bits
"Привет"utf8u8 # ?

# String and char literals can have encoding prefix and kind suffixes.
"foo" : Array<Char<UTF16>, 3>
"foo"u8 : Array<UInt8, 3> # Only if allowed to do so
# "ϕy"u8 # Panic!
"ϕy"utf8u8 : Array<UInt8, 4> # UTF-8 code units take one byte
"ϕy"utf8 : Array<Char<UTF8>, 4>

"Hello world" : Array<Char<UTF16>, 11>
# Char<ASCII>[12]* can be cast to $char*
# Char<UTF8>[]* as well.
# Any char with encoding unit size == 1 can be?
unsafe! $puts(&"Hello world\0"ascii_) # Address of a passed rvalue
```

```
"Привет!\0"utf8
"Привет!\0" # 16 bytes (UCS-2)

module ASCII   < Char::Encoding<8, 1>;
module UCS2    < Char::Encoding<16, 1>;
module UCS4    < Char::Encoding<32, 1>;
module UTF8    < Char::Encoding<8, 4>;
module UTF16   < Char::Encoding<16, 2>;
module UTF32   < Char::Encoding<32, 1>;
module UTF16LE < Char::Encoding<16, 2>;
module UTF32LE < Char::Encoding<32, 1>;
module UTF16BE < Char::Encoding<16, 2>;
module UTF32BE < Char::Encoding<32, 1>;

decl primitive Codeunit<Encoding>;
decl primitive Codepoint<Encoding>;

'f' : CPoint<UCS2> # 2 bytes
"f" : Array<CUnit<UCS2>, 1> # 2 bytes
'f'utf8 : CPoint<UTF8> # 4 bytes
"f"utf8 : Array<CUnit<UTF8>, 1> # 1 byte

'ϕ'.cunits : Stack<CPoint<UCS2>, 1>
'ϕ'.cunits.each => Std.print(&.to_u16) # 1 time
"ϕ".each => Std.print(&) # 1 time
"ϕ"utf8.each => Std.print(&) # 2 times

require "./aggregate.nx"

# A *D*-dimensional tensor of type *T*
# with *L* being the leading dimension.
#
# As an aggregate type, a tensor has optional mutability.
#
# Tensor types can be shortcut in the following form:
#
# ```
# let t = (SInt32)x2x3x4r
# ```
#
# Where `SInt32` is the type, `2`, `3` and `4` are dimensions,
# and `a` is from "aisle", which is shortcut for setting
# the third dimension leading. Explicit `lN` form can be used
# to specify leading dimension, beginning from 0.
# There are also `r` (`l0`) for "row" and `c` (`l1`) for "column".
decl primitive Tensor<
  Type: T,
  Dimensions: *D ~ %z,
  Leading: L ~ %z
>
```

```
  derive Aggregate;

  # TODO: It's not a product, but something else.
  # Nonetheless, it demonstrates passing index pairs:
  # `t1.product(t2, 1 => 2, 2 => 4)`.
  decl product(another: self, *index_pairs[D::Size]: %z => %z)
end

require "./fractional.nx"

# (Type III universal numbers)[1], also known as posits with quires.
#
# Currently, posits are rarely implemented in hardware, therefore
# they shall be used in cases when increased precision is preferable
# over performance loss.
#
# [1]: https://en.wikipedia.org/wiki/Universal_numbers_(data_format).
#

# Type III unums part, a posit.
# Used for floating-point numbers representation.
decl primitive Posit<Bitsize: Z ~ %Size, Exponent: E ~ %Size>
  derive Fractional;

  decl initialize(value : self)
  decl initialize(literal ~ %!Rational)

  # For floating-point numbers, precision loss is considered a norm.
  # Therefore, there is no need in distinct `to_*!` methods family.
  decl to([0]: \T): T forall T ~ Real
end

# The recommended posit sizes and corresponding
# exponent bits and quire sizes:
#
# Posit size (bits) | Number of exponent bits    | Quire size (bits)
# ---               | ---                        | ---
# 8                 | 0                          | 32
# 16                | 1                          | 128
# 32                | 2                          | 512
# 64                | 3                          | 2048
#
# NOTE: 32-bit posit is expected to be sufficient to solve
# almost all classes of applications[citation needed].

alias Posit8<Exponent = 0> = Posit<8, Exponent>
alias Posit16<Exponent = 1> = Posit<16, Exponent>
alias Posit32<Exponent = 2> = Posit<32, Exponent>
alias Posit64<Exponent = 3> = Posit<64, Exponent>

# Type III unums part, a quire.
```

```
# Used for floating-point numbers computation.
#
# ```
# let q = Quire<128>()
# assert(q.add(10p, 20p).to_p == 30)
# ```
decl primitive Quire<Bitsize: B ~ %Size>
  # Fused-multiply□add, i.e. `a * b + c`.
  #
  # ```
  # let q = Quire<128>()
  # assert(q.fmadd(10p, 2p, 5p).to_p == 25)
  # ```
  decl fmadd(multiplied: P1, factor: P2, added: P3): self
  forall P1 ~ Posit, P2 ~ Posit, P3 ~ Posit

  # Fused-dot-product add *a* to *b*.
  #
  # ```
  # let q = Quire<128>()
  # assert(q.add(10p, 20p).to_p == 30)
  # ```
  decl add([0]: P1, [0]: P2): self
  forall P1 ~ Posit, P2 ~ Posit

  # Fused-dot-product subtract *b* from *a*.
  #
  # ```
  # let q = Quire<128>()
  # assert(q.sub(30p, 20p).to_p == 10)
  # ```
  decl subtract(minued: P1, subtrahend: P2): self
  forall P1 ~ Posit, P2 ~ Posit
  alias sub = subtract

  decl to([0]: \T): T forall T ~ Real
end

# TODO: Do we need variadic arguments in the core?
#
# ```
# export void simple_printf(const char* fmt, ...) {
#   final varg = CVArg.init
#
#   # We assume that the pointer is always valid
#   fmt = unsafe! fmt as! $char*cr
#
#   while (*fmt != '\0')
#     if (*fmt == 'd')
#       final i = unsafe! vargs.fetch($int)
#       unsafe! $printf("%d\n", i)
```

```
#     else if (*fmt == 'c')
#       final c = unsafe! vargs.fetch($int)
#       unsafe! printf("%c\n", c)
#     else if (*fmt == 'f')
#       final d = unsafe! vargs.fetch($double)
#       printf("%f\n", d)
#     end
#
#     fmt += 1
#   end
# }
#
# export int main() {
#   unsafe! $simple_printf("dcff", 3, 'a', 1.999, 42.5)
# }
# ```
virtual type CVArg
  # Initialize a `CVArg` instance.
  # Must panic if called from not within an exported function.
  # Can be called at most once from a function; panic othwerwise.
  virtual static def init : CVArg

  # Unsafely interpret the next variadic argument as *type*.
  virtual def unsafe fetch(type: :: T) forall T

  # Shallow-copy a `CVArg` instance.
  virtual def copy : CVArg
end

# A variant is considered a scalar object rather than a container,
# thus it does not have mutability. "Mutating" methods such as `set`
# are only callable on non-final variant instances. Check methods
# such as `is?` are subject to standard local behavioural erasure technics.
decl primitive Variant<Types: *T>
  # ```
  # # Would panic if `var` is not local,
  # # otherwise behavioural erasure is applied.
  # (var : Variant<SInt32, FBin64> = SInt32(2)) += 3
  # ```
  decl .set(U) : this forall U in T
  alias .= to .set

  # Check if the actual variant value is of exact *type*.
  #
  # ```
  # final v = @rand(42, "foo")
  # if v.is?(SInt32) then (v : SInt32) += 1
  # assert(v == 43 || v == "foo")
  # ```
  virtual def nothrow is?(type :: U) : Bool forall U
```

```
  # Check if the actual variant value is of *type*,
  # i.e. perform a fuzzy match.
  #
  # ```
  # final v = @rand(42, "foo")
  # if v.of?(Int) then (v : SInt32) += 1
  # assert(v == 43 || v == "foo")
  # ```
  virtual def nothrow of?(type :: U) : Bool forall U

  # Try interpreting the actual variant value as *type* exactly.
  # If the value is not the given *type*, returns `Void`.
  #
  # ```
  # final v = @rand(42, "foo")
  # final i = v.as?(SInt32)
  # if i then (i : SInt32) += 1
  # assert(v == 43 || v == "foo")
  # ```
  virtual def nothrow as?(type :: U) : U? forall U

  # Try interpreting the actual variant value as *type* exactly,
  # throwing `AssertionError` in case of mismatch.
  #
  # ```
  # final v = @rand > 0 ? 42 : "foo"
  # v.as!(SInt32) += 1
  # assert(v == 42)
  # ```
  virtual def as!(type :: U) : U throws AssertionError forall U

  # Unsafely intepret actual variant value as *type*.
  #
  # PERFORMANCE: It is faster due to not checking the switch value.
  #
  # ```
  # final v = @rand > 0 ? 42 : "foo"
  # unsafe! v.as!!(SInt32) += 1
  # assert(v == 42)
  # ```
  virtual unsafe def nothrow as!!(type :: U) : U

  # Check if the actual variant value is `Void`.
  #
  # ```
  # final v = @rand(42, Void) : Variant<SInt32, Void> : SInt32?
  # if not v.void? then (v : SInt32) += 1
  # assert(v == 43 || v.void?)
  # ```
  virtual def nothrow void? : Bool
```

```
  # Ensure that the actual variant value
  # is not `Void`, throwing otherwise.
  #
  # ```
  # final v = @rand > 0 ? 42 : Void
  # v.novoid! += 1
  # assert(v == 43)
  # ```
  virtual def novoid! : Undef throws AssertionError

  # Check if the actual variant value equals to *value*.
  # It delegates `==` call to underlying types, which may throw.
  #
  # ```
  # final v = @rand(42, "foo")
  # assert(v == 42 or v == "foo")
  # if v == 42 then assert(v :? SInt32)
  # ```
  virtual def maythrow equals?(value : U) : Bool forall U
  alias eq?, == to equals?

  # Set the actual variant value to a new *value*.
  # The old value is finalized.
  # Copy-returns the **new** value if the call has a receiver.
  # Acts similarly to (simple) assignment (`=` or `<-`).
  # Setting is not applicable to final variant instances.
  #
  # ```
  # let v := @rand(42, "foo")
  # assert((v = 43) == v.set(43))
  # ```
  virtual def set(value : U) : (\recv? ? U : Void)

  # Replace the actual variant value with a new *value*.
  # Move-returns the **old** value if the call has a receiver.
  # Otherwise, finalizes the old value.
  # Acts similarly to replace-assignment (`<<=` or `<<-`).
  # Replacing is not applicable to final variant instances.
  #
  # ```
  # let v := @rand(42, "foo")
  # final old := v <<= 43 # Or `v.replace(43)`, `v.replace(<- 43)`
  # assert(old == 42 || old == "foo")
  # ```
  virtual def replace(value : U) : (\recv? ? (<-U) : Void)

  # Swap the actual variant value with target's,
  # updating the switches accordingly.
  # Acts similarly to swap-assignment (`<=>`).
  # Swapping is not applicable to final variant instances.
  #
```

```
  # ```
  # let v1 := @rand(42, "foo")
  # let v2 := @rand(43, "bar")
  # final new := v1 <=> v2 # Or `v1.swap(v2)`, `v1.swap(<- v2)`
  # assert(new == 43 || new == "bar")
  # ```
  virtual def swap(target : self*crw) : (\recv? ? U : Void)
end

require "./tensor.nx"

# A vector is a 1-D specialization of `Tensor`.
#
# ```
# let v : (SInt32)x4 = <1, 2, 3, 4>
# let v = %i<1 2 3 4>
# ```
alias Core::Vector<Type: T, Size: S> to Tensor<T, S, 0>
end
```

Appendix E: Macros API

```
-- This file defines the Onyx Macro API.
--
-- The following identifiers: `string`, `nil`, `boolean`, `number` and
-- `table`; represent fundamental Lua types. Additional identifiers
-- `int` and `uint` are used to represent whole numbers. Logical
-- operators may be used to express variativity of possible values.
-- Concrete literals may be used to represent exact possible values.
-- Arrays are represented using the `{ type }` notation,
-- e.g. `{ string }`.
--
-- For example, `{foo = { string } or 42 or nil}` means that the `foo`
-- entry may contain either an array of strings, exact number 42, or nil.
--
-- Sum of tables represents an extensions. For example,
-- `foo = bar + { baz = 42 }` means that the `foo` table
-- includes the contents of the `bar` table.
--
-- Special strings beginning from `@` are local to this documentation
-- and should not be implemented. Purposes of each special string are
-- documentented individually.

-- The global Onyx table accessible from any Onyx source code
-- file without any thread-safety considerations.
nx = {
  -- Access current compilation context; the contents may be
  -- sensitive to the position within a source file.
  ctx = {
    -- Access current function implementation, if any.
    impl = nx.Node.FunctionDef or nil,

    -- Access current type (which may
    -- also be the top-level namespace).
    type = nx.Node['@Type'],

    -- Access information about the source code
    -- file currently being compiled.
    file = {
      path = string
    },

    -- A function performing Onyx path lookup in accordance to
    -- the regular lookup rules, from the current context.
    lookup = function (path)
      -- Returns `nil` if lookup failed
      return nx.Node or nil
    end,
  },
```

```
-- Access current compilation target information.
target = {
  -- The target Instruction Set Architecture.
  -- It may contain additional fields
  -- defined in the Targets Appendix.
  isa = {
    id = string,
  },

  -- A list of deviant target Instruction Set Extensions,
  -- which may be empty. The list of possible ISEs
  -- is defined in the Targets Appendix.
  ise = { string },

  -- An exact target Processing Unit, which may be unknown.
  pu = string or nil,

  -- The target Operating System information,
  -- which may be empty. It may contain additional
  -- fields defined in the Targets Appendix.
  os = {
    id = { string },
  },

  -- A list of target Application Binary Interfaces,
  -- which may be empty. An ABI may have additional
  -- fields defined in the Targets Appendix.
  abi = { { id = string } }
},

-- C interoperability constraints and utilities. Note that these
-- do not neccessarily align with the target's capatibilites.
--
-- The table defines some C constraints which are barely express-able
-- by the C standard, e.g. signedness representation of integers.
-- Additionally, this information is useful in cases when a program
-- does not have access to target-specific _hosted_ C headers (§4.6).
--
-- Note that it should always be assumed that a program has access
-- to the target-specific _freestanding_ C headers (§4.6):
-- float.h, iso646.h, limits.h, stdalign.h, stdarg.h, stdbool.h,
-- stddef.h, stdint.h and stdnoreturn.h.
c = {
  -- A function performing C path lookup.
  lookup = function (path)
    -- Returns `nil` if lookup failed
    return nx.c.Node or nil
  end,

  -- Access C integer environment.
```

```
    integer = {
      -- "Sign and magnitude", two's, or one's complement.
      signedness = 'SM' or '2C' or '1C',

      -- > ... is whether the value with sign bit 1 and all value bits
      -- zero (for the first two), or with sign bit and all value bits 1
      -- (for ones' complement), is a trap representation or a normal
      -- value. In the case of sign and magnitude and ones' complement,
      -- if this representation is a normal value it is called a
      -- negative zero.
      --
      -- Therefore, `{signedness = 'SM', can_trap = false}` means that
      -- negative zero is supported.
      can_trap = bool
    },

    -- TODO:
    -- -- Access C floating-point environment.
    -- floating_point = {
    --   -- Is `INFINITY` supported?
    --   infinity = boolean,

    --   -- Is negative zero supported?
    --   negative_zero = boolean,

    --   -- Are NaNs supported? Note that C treats all NaNs as qNaNs.
    --   nan = boolean,
    -- },

    type = {
      float = {
        format = 'IEEE-754'
      },
      double = {
        format = 'IEEE-754'
      },
      'long double' = {
        format = 'IEEE-754' or 'X87' or 'PPC'
      },
      char = {
        signed = bool -- A char's sign is not defined by the standard
      },
      short = {
        bitsize = int
      },
      int = {
        bitsize = int
      },
      long = {
        bitsize = int
      },
```

```
    'long long' = {
      bitsize = int
    }
  },

  Node = {
    Macro = {
      id = value,
      arity = int,
      call = function (args)
        -- Return the result of macro evaluation
        return string
      end
    },
    Constant = {
      Integer = {
        -- The type may be unknown for freestanding constants
        type = nx.c.Node.Type.Int or nil,

        base = 'decimal' or 'octal' or 'hex',
        value = int,
        suffixes = {'unsigned' or ('long' or 'long long')}
      },

      Char = {
        -- The type may be unknown for freestanding constants
        type = nx.c.Node.Type.Int or nil,

        value = string,

        -- TODO: > (byte) If c-char is not representable as a single
        -- byte in the execution character set, the value is
        -- implementation-defined.
        -- TODO: > (16bit, 32bit, wide) If c-char is not
        -- or maps to more than one 16-bit character, the behavior is
        -- implementation-defined.
        -- TODO: > (multichar) has [...] implementation-defined value
        kind = 'byte' or -- E.g. ``'a'`` or ``'\n'`` or ``'\13'``
          'utf8' or      -- E.g. `u8'a'`
          '16bit' or     -- E.g. `u'□'`, but not `u'□'`
          '32bit' or     -- E.g. `U'□'`, but not `U'□'`
          'wide' or      -- E.g. `L'β'` or `L'□'`
          'multichar'    -- E.g. ``'AB'``
      },

      Floating = {
        -- The type may be unknown for freestanding constants
        type = nx.c.Node.Type.Int or nil,

        base = 'decimal' or 'hex',
```

```
          significand = {
            whole = int,
            fraction = uint
          },

          exponent = int or nil,
          suffix = 'f' or 'l' or nil
        },

        String = {
          -- The type may be unknown for freestanding constants,
          -- e.g. `char* = "foo"` has type, but `"foo"` does not.
          type = nx.c.Node.Type.Pointer or nx.c.Node.Type.Array or nil,

          value = string,

          -- TODO: > (byte) If c-char is not representable as a single
          -- byte in the execution character set, the value is
          -- implementation-defined.
          -- TODO: > (16bit, 32bit, wide) If c-char is not
          -- or maps to more than one 16-bit character, the behavior is
          -- implementation-defined.
          -- TODO: > (multichar) has [...] implementation-defined value
          kind = 'byte' or -- E.g. ``'a'` or ``'\n'` or ``'\13'`
            'utf8' or      -- E.g. `u8'a'`
            '16bit' or     -- E.g. `u'□'`, but not `u'🐱'`
            '32bit' or     -- E.g. `U'🐱'`, but not `U'□'`
            'wide' or      -- E.g. `L'β'` or `L'🐱'`
            'multichar'    -- E.g. ``'AB'`
        }
      },
      Type = {
        Int = {
          kind = 'short' or 'int' or 'long' or 'long long',
          signed = boolean,
          atomic = boolean,
          constant = boolean,
          volatile = boolean,
          alignment = int or nil
        },
        Float = {
          kind = 'float' or 'double' or 'long double',
          complex = boolean,
          imaginary = boolean,
          atomic = boolean,
          alignment = int or nil
        },
        Pointer = {
          to = nx.c.Node.Type,
          alignment = int
        },
```

```
      Reference = {
        to = nx.c.Node.Type,
        alignment = int
      },
      Struct = {

      },
      Enum = {

      },
      Union = {

      },
      Function = {

      },
    },
    TypeDef = {
      id = string,
      target = nx.c.Node.Type or nil
    },
    FunctionDecl = {

    }
  }
},

Node = {
  -- Represent any referencable type.
  ['@Type'] = nx.Node.Namespace or
    nx.Node.Trait or
    nx.Node.Struct or
    nx.Node.Enum or
    nx.Node.Primitive or
    nx.Node.TypeAlias

  -- Represent a declaration.
  ['@Decl'] = nx.Node['@Type'] or
    nx.Node.FunctionDecl or
    nx.Node.FunctionDef or
    nx.Node.VarDecl or
    nx.Node.AnnotationDef or
    nx.Node.MacroDef

  -- A namespace node. Any other type declaration
  -- is also considered a name space.
  Namespace = {
    name = string, -- E.g. `''Bar'` for `::Foo::Bar`

    -- Would be nil for the top-level namespace.
    parent = nx.Node['@Type'] or nil,
```

```
  declarations = { nx.Node['@Decl'] },

  -- Return a fully qualified path to the namespace in form of
  -- an array of strings, e.g. `{'Foo', 'Bar'}` for `::Foo::Bar`.
  path = function ()
    return {string}
  end
},

-- A trait node
Trait = nx.Node.Namespace + {
  annotations = { nx.Node.AnnotationCall },

  -- A trait may derive from other traits
  derivees = {
    [1] = {
      trait = nx.Node.TypeRef,
      declarations = { nx.Node['@Decl'] }
    }
  }
},

-- A struct node derives from the Trait node
Struct = nx.Node.Trait,

-- A enum node, which may also be a flag
Enum = nx.Node.Namespace + {
  is_flag = true or false,

  -- The enum type, which is `$int` by default
  type = nx.Node.TypeRef,

  elements = { nx.Node.Enum.Element },
  annotations = { nx.Node.TypeRef  },

  -- A enum element
  Element = {
    name = string,
    value = nx.Node.Literal or nil
  }
},

Primitive = nx.Node.Namespace,

FunctionDecl = {
  parent = nx.Node['@Type'],

  visibility = 'public' or 'protected' or 'private' or nil,
  storage = 'static' or 'instance' or nil,
  safety = 'threadsafe' or 'fragile' or 'unsafe' or nil,
```

```
    mutability = 'mut' or 'const' or nil,

    name = string,
    arguments = { nx.Node.ArgDecl },

    annotations = { nx.Node.AnnotationCall }
  },

  FunctionDef = {
    parent = nx.Node['@Type'],

    proto = nx.Node.FunctionDecl, -- A prototype per specialization
    body = { nx.Node.Expr },

    annotations = { nx.Node.AnnotationCall }
  },

  VarDecl = {
    parent = nx.Node['@Type'],

    -- Would be nil if undefined
    is_static = true or false or nil,

    -- Would be nil if undefined
    visibility = 'public' or 'protected' or 'private' or nil,

    -- Always defined
    is_final = true or false,

    -- Would be false for `set foo`, always defined
    has_getter = true or false,

    -- Would be false for `get foo` or `final foo`, always defined
    has_setter = true or false,

    name = 'MyVar',
    type = nx.Node.TypeRef or nil,
    annotations = { nx.Node.AnnotationCall }
  },

  -- An annotation definition node
  AnnotationDef = {
    parent = nx.Node['@Type'],
    name = string,

    -- An annotation consists of macro code (may be empty)
    macros = { nx.Node.Macro },

    -- Resolve a full path to the annotation
    path = function,
```

```
    -- Annotations may be annotated themselves!
    annotations = { nx.Node.AnnotationCall }
  },

  -- A macro definition node
  MacroDef = {
    parent = nx.Node['@Type'],
    is_builtin = boolean,
    visibility = 'public' or 'protected' or 'private' or nil,
    name = string,
    arguments = { nx.Node.ArgDecl },
    body = { nx.Node.Macro }
  }

  -- An annotation call (i.e. the act of annotating) node
  AnnotationCall = {
    annotation = nx.Node.AnnotationDef,
    arguments = { nx.Node.ArgCall }
  },

  TypeAlias = {
    parent = nx.Node['@Type'],

    name = string,
    arguments = { nx.Node.ArgDecl },

    target_type = nx.Node['@Type'],
    target_arguments = { nx.Node.ArgCall }
  },

  TypeRef = {
    type = nx.Node['@Type'],
    arguments = { nx.Node.ArgCall }
  },

  ArgDecl = {
    name = string or nil,
    alias = string or nil,
    is_static = boolean,
    type = nx.Node.TypeRef or nil,
    annotations = { nx.Node.AnnotationCall },
    has_default_value = boolean
  },

  ArgCall = {
    parent = { nx.Node.TypeRef or nx.Node.FunctionCall },

    -- A link to the target argument declaration, if resolved
    declaration = nx.Node.ArgDecl or nil,

    -- How the argument is referenced.
```

```
      -- Would be a number for `[0]:` reference,
      -- a string for named reference,
      -- and nil for a simple sequence
      ref = number or string or nil,

      value = nx.Node.Expr
    }
  }
}
```

Glossary

Standard

A widely accepted specification.

Specification

> A document that states requirements.
>
> — ISO 9000:2015 § 3.8.7

Abstract character

A entry in a *character set* representing a *grapheme* or a *sememe*, usually mapped to a single *codepoint*.

A grapheme may be represented in multiple ways in a single character set, and also as a sequence of multiple abstract characters. For example "å" in Unicode is a◌̊ (U+61 followed by U+030A) and also å (U+00E5).

Allograph

One of many *glyphs* representing the same *grapheme*.

Character set

A mapping of *abstract characters* to *codepoints*. Examples of character set are ISO/IEC 10646, ISO/IEC 646:US.

Codepoint

A numerical value representing an *abstract character* from a *character set*.

A codepoint may be *encoded* in one or multiple *codeunits*.

Codeunit

A sequence of bits in a certain *encoding*.

Depending on the encoding, a single or multiple codeunits make up a *codepoint*.

Combined grapheme

A *grapheme* containing at least one *combining abstract character*.

Combining abstract character

An *abstract character* usually representing a *sememe* and intended to modify other abstract characters when in a sequence, e.g. ◌̊ (U+030A).

Character encoding

A way to encode a *codepoint* into a sequence of bits.

Character encoding is not to be confused with *character set*. However, sometimes a character encoding has the name of the character set, for example ISO/IEC 646:US (a.k.a. US-ASCII).

Other examples of character encoding are UTF-8, UCS-2.

Glyph

An elemental printable symbol intended to represent a *grapheme*.

Grapheme

In the scope of this standard, a grapheme is what Unicode defines as a printable character

> A unit of writing which is (1) lexically distinctive, (2) has linguistic value (mostly by referring to phonemes, syllables, morphemes, etc.), and is (3) minimal.
>
> — Meletis, D., Writing Systems Research (2019)

Graphemes include not only letters and digits, but also other printable symbols, such as arrows. A grapheme is notated with angle brackets, e.g. ⟨a⟩, ⟨A⟩, ⟨å⟩, ⟨1⟩, ⟨→⟩.

In a *character set*, a grapheme is represented by one or multiple *abstract characters*.

Sememe

An abstract semantic unit of meaning.

In a character set, a sememe is an *abstract character*.