The Lodge Programming Language (Hypothetically)

Version 0.0.1*

Dylan Scott Carroll

Contents

Preface

Goals

The Lodge Programming Language, as it is described in this text, *does not exist*. Ultimately, the goal of this document is to outline the syntax, and semantics of Lodge thoroughly enough to guide the implementation of an eventual compiler. The creation of this document itself is part of the language design process, acting as a assemblage of my current thoughts of what Lodge will be. Because of that, this document is fundamentally a work-in-progress and will be until the compiler is done. It has various mistakes, inconsistencies, and notes to the author.

Additionally, the deficiencies of the author reveal themselves in numerous places. Oftentimes, descriptions of Lodge's semantics and type system are lacking theoretical grounding and misuse vocabulary and notation; purists be warned.

Style

This document is written in a style as similar as possible to a guide for an existing programming language and acts, for the most part, as if an implementation of Lodge exists. It presents information in an order aiming to be useful to someone who would be learning the language and demonstrates syntax largely through example.

This document starts out by describing the basics of syntax before moving on to what is arguably more interesting: descriptions of Lodge's type system in a more theoretical way. If you don't care about syntax and want to learn about the type system, skip to part 2.

Plans

During what will likely end up being by very slow and sporadic compiler implementation, this document will be continuously updated to match the most up-to-date state of the language with the hope that, at such time as there is anything approaching a functional implementation, this document will act as a complete guide and specification for that language.

Basics

Statements

As lodge has no statement terminator, the end of statements is automatically inferred by newline characters in most cases. However, there are cases where multiple lines can be considered a single statement. This means that Lodge is not totally whitespace agnostic. It is still agnostic towards tab and space characters.

In Lodge, statement termination happens at newlines by default, but statements may span multiple lines ins some scenarios.

1. If a line opens a bracket (such as: `( [ <`), all lines until that bracket is closed will be treated as a single statement.
2. If a line ends with an operator that requires another following operand, the following line will be treated as part of the same statement.

```
!! Examples of rule 1
Int x := f(arg1,
           arg2,
           arg3)

 [Int] l := [val1, val2,
             val3, val4]

Int x := ( a
     + b
     + c
     + d )

!! Example of rule 2
Int y := 10 +
         11 +
         12
```

Comments

Comments are created with the `!!` token. Everything after the `!!` until the next newline will be considered a comment and completely ignored.

Multiline comments can be created with `!-` and `-!`. Everything between these two tokens will be ignored as comments. Such comments may also exist on only one line, and code before and after the comment will be treated normally.

Variables

Declaring and Assigning Variables

A variable declaration consists of a type expression, a variable name, and an assignment. Variables cannot be declared without being assigned a value. If a variable has no sensible default value, consider using a nonable type.

Assignment occurs using the `:=` operator.

```
Int value2 := 10    !! Declaring an int variable and assigning it a value
```

Assignment also does not need to be a standalone statement, but can be an expression that returns a value. The value returned by an assignment expression is the assigned value.

```
!! Print out the characters from a file until the character 'a' is reached
Str c := ""
loop while (c := file.getChar()) != 'a' {
  print(c)
}
```

Lodge is garbage-collected, so created values do not need to be manually freed.

Variable Types

In Lodge, variables types are not defined by the concrete type they are declared with. Instead, the type of a variable is the public interface of that type. This means that a variable is not restricted to just the type it was declared with. Instead, any value with a compatible interface can be stored in that variable. For example, a variable with the default `Int` type can store any integer value.

Likewise, the type the programmer variable declares a variable with does not have to be a type with a defined constructor. It could also be an interface or a type expression.

Var Keyword

When variables are declared with the `var` keyword instead of a type, the variable being declared takes the same type as the value on the right side of the assignment.

```
Int a = 10
var b = a !! b has the type of Int

!! Literals have types that var infers
var value4 := "hello"


!! Generic types are also inferred
var arr := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
!! arr has the type [Int]
```

Auto Keyword

This feature will likely need to be removed as it makes the type system much more complex and would result in very opaque error messages :(

The `auto` keyword allows the programmer define the type of a variable automatically based on what the variable is used to do. When `auto` is used to define a variable, every get, set, and method call are collected together. The broadest possible interface for those accesses is used to create the type of the given variable

Consider the following example:

```
fun function(auto a) {
  a.x = 10
  Float b = a.y
  Str c = a.function(1, 2, 3)
}
```

The type of `a` would be the following interface:

```
Interface {
  set Int x
  get Float y
  fun function Str(Int _1, Int _2, Int _3 |)
}
```

The full behaviors of Lodge's type checking system is found in The Type System

Union Types

Instead of creating a variable with just a single type, it is also possible in Lodge to define a variable with a composite type called a type union. This allows a variable to store any value that has a compatible interface with any type in the union.

```
(Str | Int) x

!! Allowed
x := 10
x := "Hello World"
```

Composite types can also be given names with the `as` keyword

```
(Str | Int) as StrInt
StrInt x = "10"
x = x->Int
```

Circular Type Definitions

There may be cases where a composite type declaration may want to refer to itself, which is allowed in Lodge.

Here's an example of how a tuple containing two values can be used to create something akin to a linked list

```
((int, listelement) | None) as listElement

listElement list = (0 (1, (2, (3, None))))
```

Built-In Types

Basic Types

Integer
value type

`Int`
- Defaults to 64 bit integer value, but other subtypes are available.
- Interface does not allow float values

Floating Point
value type

`Float`
- Defaults to 64 bit floating point value, but other subtypes are available.
- Interface also allows Int values

Numbers
Interface

`Num`
- An interface that can hold both Floats and Ints (also maybe complex numbers if they get introduced to the spec)
- No literals default to Num

Booleans
value type
`Bool`

Strings
reference type
`Str`
- A reference type object much like strings in other languages
- Basically just an immutable array of chars
  - Potentially just a chunk of data, allowing for various representations
  -
- String literals are created using text surrounded by double quotes like this:

```
Str aString := "A new string literal"
```

None
value type

The None type has the special property of being compatible with no interface besides None, having no other interface be compatible with it besides None.

Uses
- Indicate that something does not exist
- When a variable has a None in its union, the interface is empty and you would need to check that the value is not None in a type switch before you could do any operations with it.
- If variables don't have an initial value, setting the value to None is the only option, forcing a check.
- Used as a default value for function arguments when that argument doesn't always need to be present. This is useful for variadic functions and optional parameters.

Data Structure Types

All data structure types are reference types

Tuple
- literal: `(value1, value2, value3)`
  - `(value,)` for a single value
- type definition: `(type1, type2, type3)`
  - `(Type : size)` when it's all the same type
- You must declare the type for each variable in the tuple

```
(Int, Str) tuple1 := (10, "hello")
(Int,) tuple2 := (10,)
(Int: 5,) tuple3 := (1, 2, 3, 4, 5)
(Int: 2, Str) tuple3 := (1, 2, "a", b")
```

The interface for a particular tuple type contains fields for each of its elements:
- tuple.item0
- tuple.item1
- ...
- tuple.itemN

List
- literal: `[value1, value2, value3]`
- type definition: `[Type]`
  - The type of all elements of a list must match
- The type (union) must be the same for each element in the array

```
[Int] list1 := [0, 1, 2, 3, 4]
var list2 := [0, 1, 2, 3, 4]
```

Map
- literal: `{> key1 : value1, key2 : value2 }`
- type definition: `{> Type1 : Type2}`
- Mutable reference type
- Implemented as a hash table

```
{> Str : Int} map1 := {> "A" : 1, "B" : 20 }
```

Set
- literal: `{$ value1, value2, value3}`
- type definition: `{$ Type}`
- Implemented as a hash set

```
{Int} set1 = {$ ~2, 4, 6, 8 }
```

Generator
- Literal: No direct literal, but range expressions can be used
  - `<start:stop:end>`
- Type
  - <Type>

```
<Int> allSquares = <:>**2
```

Type Expressions

[This section needs to be expanded]: Create a more unified description of the type expression syntax

Type expressions are a kind of expression that exists when a type is being defined such as during variable declaration or on the right hand side of the special `->` operator.

Type expressions differ from normal expressions in that
1. They can only exist in a few situations
2. The values of the expression must all be types
   - Non-type values will be coerced into types by resolving their interface
   - Explicitly defined type values mask non-type values
3. The operators must be the type set operators

To expand on point one, values inside of type expression will first

Operators

Mathematical Operators

op	action
+	addition
-	subtraction (and unary negation)
/	division
//	integer division
*	multiplication
%	modulus
**	power

Logical Operators

The operators here that are words are reserved keywords

op	action
`&`	Logical and
`|`	Logical or
`not`	unary logical negation operator
`^`	exclusive or
`==`	value equivalence
`is`	identity equivalence
`!=`	not value equivalent
`isnt`	not identity equivalent
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

The value equivalence operator `==` should not be confused with the assignment operator `:=`, or the function keyword token `=`.

Bitwise Operators

op	action
`&&`	bitwise and
`||`	bitwise or
`^^`	bitwise xor
`-|`	bitwise not (compliment)
`>>`	right shift
`<<`	left shift

Assignment Operators

op	action
`:=`	assignment
`++`	increment
`--`	decrement
`{op}=`	Any other infix operator followed by an '=' will be the same as that operator's assignment operator

Other Operators

op	action

`->`	type conversion
`()`	call as a function
`[]`	indexing and slicing
`!`	Error Resolution
`?`	None Resolution Operator
`{op}~`	broadcasting directive
`{op}$`	Reverse operator
`=`	hash
`@`	Unused

Broadcasting Operators

Using the symbol after an operator indicates that the operation should be broadcast. At least one of the operands of a broadcast operator should be a sequence of some kind. Broadcast operators have the same precedence of the non-broadcast version

This can be used to expand mathematical operations

```
var list = [0:10]
var powers_of_two = 2 **~ list
```

Function calls can also be broadcast

```
[Float] values = ( [:314] /~ 100 )
[Float] mapped_values = cos~(values) !!This applys the cosine function to every
value in values
```

You can also broadcast generator objects, which will be evaluated lazily

```
<Int> everySquare = <0:> **~ 2
```

Broadcasting multiple sequences together will pair them element-wise.

```
[Float] a = [1, 3, 5, 7, 9]
[Float] b = [2, 4, 6, 8, 10]

[Float] c = a + b
!! c == [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]

[Float] d = a +~ b
!! d == [3, 7, 11, 15, 19]
```

Reverse Operator

When any operator is followed by a dollar sign, it simply applies the reverse of the given operator

```
denominator /$ numerator
```

Type Conversion Operator

The type conversion operator `->` is the way to convert an object of one type into another.

```
Str myStrRepr := myObj->Str

Int myInt := myFloat->Int
```

The type conversion operator is special because the right hand side is parsed as a type expressions rather than a value expression. In that sense it is not a normal operator and more of a conversion directive. Like other operators, its behavior is determined by operator methods, but these methods have their own special semantics that differ from the other type conversion methods

Error Resolution Operator

Shorthand for resolving errors

Errable functions are functions that have the potential to return an error, meaning that the return type is the union of some type and `Err`. In order to use the return value of an errable function, you need to handle the possibility of an error.

Often times, handling an error just consists of passing it further up the call stack for the caller do handle. For this, the unary `!` operator is used. It will evaluate to a non-Err value or return Err from the function.

Here is a common pattern:

```
fun function() Int! {
  Int val := errableFunction()!
  !! ...
}
```

Which is equivalent to:

```
fun function() (Int | Err) {
  (Err | Int) val := errableFunction()
  swype val {
    Err : { return valOnc upon a time there was a litte bit too much going on
inside my head, but now I don't think of other things to be happening. }
  }
}
```

See Type Switching for more about this syntax.

Binary Error Resolution Operator

Sometimes, instead of passing up the error, it is better simply to have a default value in case of an error. For these cases, there is also a binary version of the error resolution operator, `!?`.

```
fun function() Err! {
  Int val := errableFunction() !? -1
}

!! Is equivalent to

fun function() Int! {
  !Int result = errableFunction()
  Int val := swype result {
    Err : { -1 }
    Str : { result }
  }
}
```

In Type Expressions

The `!` token is also used to make an Errable type:

```
fun errableFunction() (Err | Str) {
  !! ...
}

!! Is equivalent to

fun errableFunction() Str! {
  !! ...
}
```

None Resolution Operator

The None resolution operator applies to variables whose type is a union that contains None. It is essentially a shorthand for an otherwise common type switch.

The following two code snippets are equivalent:

```
Int? temp = nonableFunction()

Int variable = swype variable {
  None : { -1 }
  Int  : { temp }
}
```

```
Int variable = nonableFunction() ? -1
```

Unlike the error resolution operator, there is no unary version of the None resolution operator

Operator Precedence

This section is mostly for parser reference, you can mostly skip this.

Operators are listed in order of increasing precedence, categorized into groups, and given names for implementation purposes.

```
Lodge
!! Assignment : ass_expr
:=  +:=  -:=  *:=  /:=  //:=  ...

!! Special Infix operators : special_in_expr
!  ?


!! Special Postfix Operators : special_post_expr
!

!! Logical

!! or_expr
|  |!
!! xor_expr
^  ^!
!! and_expr
&  &!

!! com_expr
===  !=  is  isnt  <  >  >= <=

!! not_expr
not

!! Numerical

!! bor_expr
||
!! bxor_expr
^^
!! band_expr
&&
!! bshift_expr
>>  <<

!! a_expr
+ -

!! m_expr
-  /  //   %


!! Unary prefix : pre_expr
-  -|

!! Power Operator : pow_expr
-*

!! Increment operators (postfix) : inc_expr
++ --


!! Special Postfix Operators
```

```
!! call_expr
()

!! index_expr
[]

!! cast_expr
->

!! access_expr
.
```

Control Flow

Blocks

Blocks are opened and closed by curly brackets `{}`. Many statements such as if statements, loops, and class definitions require blocks.

Blocks in lodge create a new scope that can access variables from the outer scope, but whose variables only last during the inner scope.

```
Int x := 0
loop x <= 10 {
  if x%2 == 0
    { print(x) }
  x++
}
```

Naked blocks can also be created without an opening statement simply to create a new scope and group together a sequence of statements.

Block Expressions

If a block is placed in a location an expression is expected rather than a standalone statement, the entire block evaluates to the value of last expression in the block. For if statements, the value of the path taken is returned. For loops, the values from multiple iterations are accumulated into a list.

```
Str parity = if x%2 == 0 { "even" } else { "odd" }

(Int, Int) x = (0, 1)
[Int] fibs = loop {
  break if x > 100
  x := (x[1], x[0]+x[1])
}
```

Naked Blocks

This feature is neat, but may be removed in favor of the syntax referring to something else.

Blocks without any keyword, also known as naked blocks, can simply be used as a way to cluster a sequence of statements into a single unit that returns a value. It has essentially no impact on the way the code executes, but may be used for conceptual groupings of actions.

```
Int x = {
  Int a = 200
  Int b = 10
    a / b
}

!! x == 20
```

Conditionals

Conditionals use the `if` keyword, a boolean expression, and a block to create an if-statement. If the given boolean expression evaluates to true, the block is entered. Otherwise, it is skipped.

If

```
if boolean_expression {
  !! conditional body
}
```

If / Else

You can also specify as block to enter if the boolean is false using the `else` keyword.

```
if boolean_expression {
  !! Enter if true
} else {
  !! Enter if false
}
```

If / Else If / Else

And you can also chain if statements using `else if`

```
if expr1 {
  !! exp1 is true
} else if expr2{
  !! exp1 is false and exp2 is true
} else {
  !! exp1 is false and exp2 is false
}
```

If Expressions

You can use if statements like an expression and the selected block will be evaluated

```
Int x := 10
Int result := if(x < 0) {
  -1
} else if (x == 0) {
  0
} else {
  1
}

!! result = 1
```

Switch Statements

Switch statements use basic pattern matching to select between a set of optional blocks. The simplest case of a switch statement is matching a single value.

```
Int x = 10
switch x {
  -1 : { !- x is  -1 -! }
   0 : { !- x is 0 -! }
   1 : { !- x is 1 -! }
   _ : { !- x is any other value -! }
}
```

A switch statement can also match multiple values at the same time.

```
Int x = 10
Int y = 0

switch x, y {
  10, 10 : { !- Both 10 -! }
  10, _  : { !- First 10, second not -! }
  _, 10  : { !- Second 10, First not -! }
  _, _  : { !- Neither 10 -! }

}
```

Additionally, sequences such as tuples and lists can be matched.

```
!! Tuple and sequence matching
```

Instead of using _ for the wildcard match, using a name will assign the matched value to that name. This works for sequence matchng as well.

```
x = 10
y = 15

switch x, y {
  10, b : { !- b == 15 -!}
  !! etc.
}
```

Loops

Loops

Plain Loop

A plain loop is a loop without any parameters. It is essentially the same as a `loop while true`, but it does not perform a comparison. It is usually used for replicating a do-while type pattern using a `break if` statement. Here is an example.

```
!- Read one character from the file onto the stack,
   stopping after a period is processed
   until a period is reached -!
loop {
  c := file.read()
  stack.push(c)

  break if c == '.'
}
```

While Loop

A while loop takes a boolean expression and will repeat its body code until the expression evaluates to false.

```
!! Print all the values on the stack
loop not stack.is_empty() {
  ints val = stack.pop()
  print(val)
}
```

For Loop

The for-loop works like for-each loops in other languages. It accepts any object that is compatible with the Iter interface and loops over each element. The type of the loop variable is inferred from the values in the container being iterated over.

```
!! Print all of the characters in the list
loop for c = char_list {
  print(c)
}
```

If you want to loop over multiple iterators of the same length at once, you can separate several associations with a comma as below.

```
loop for a, b = list1, list2 {
  list3.append(a + b)
}
```

Using an unbounded iterator, you can keep track of indices like so:

```
!! Print all of the characters in the list
loop for c, i = char_list, <0:> {
  print(c, "at", i)
}
```

Break and Continue

The keywords `break` and `continue` work as they do in similar languages. `break` will exit the nearest loop that encapsulates it. `continue` will skip to the beginning of the nearest loop that encapsulates it

Conditional break/continue

Lodge also has the `break if` and `continue if` statements which act as conditional statements.

```
loop for value = value_list {
  break if value == None

  print(value)
}

!! Equivalent to

loop for value = value_list {
  if value == None { break }

  print(value)
}
```

Deep break/continue

If the `break` or `continue` keywords is followed immediately by an integer literal, the statement will apply to that many nested loops. The value 1 is equivalent to an unspecified depth.

```
Float t = time()
loop for i := :100 {
  loop for j := :100 {
    print(i, j)
    break 2 if (time()-t > 10)
  }
}
```

This will result in an error if there are not that many nested loops in the current function.

Functions

Basic Syntax

Full function definitions are made of the following pieces: 1) The `fun` keyword, 2) the function name, 3) optional type parameters surrounded by angle brackets, 4) optionally,

the return type, 5) the argument description list surrounded by parentheses, and 6) the function body consisting of a sequence of statements surrounded with curly brackets. This syntax defines a variable with the function's name in the current scope and evaluates as an expression to that function.

```
!! Functions without return types may omit it,
fun functionName(Type1 arg1, Type2 arg2, Type3 arg3) {
  !! Function body
}

fun functionName ReturnType(Type1 arg1, Type2 arg2, Type3 arg3) {
  !! Function body
}
```

Alternatively, Lodge offers syntactic shorthand for defining anonymous functions. This form consists only of the function arguments surrounded with parentheses, the `=>` token, and the function body surrounded by curly brackets. This syntax does not declare a variable for the function.

```
(Type arg, Type arg, ...)=>{ !- Function body -! }
```

Return Keyword

To return a value from a function, use the `return` keyword

Absent the return keyword, a function with a return type will implicitly return the value of the final expression in the function body.

Argument Description Lists

Positional Arguments

Every function declaration has an argument description list surrounded by parentheses. At its most basic, this is simply a comma-separated list of type expression and identifier pairs, which each identify a positional argument. Each argument in that list describes a variable that will be defined within the scope of the function.

```
(Type1 arg1, Type2, arg2, Type3 arg3)
```

Optional Positional Arguments

In addition to a type and a name, an argument description can include a default value in the form of an expression following an equal sign `=` (note, not the assignment operator `:=`). The presence of this default value also marks the argument as optional.

Optional arguments may be omitted when calling a function, while non-optional arguments may not. When not provided at the time of function invocation, the default value expression is evaluated and assigned to that argument.

```
fun f returnType(Int a = 10, Int? b = None) {
  !! Function body
}
```

When calling a function with optional positional arguments, trailing optional arguments may be omitted entirely. If an optional argument is not one of the final arguments, it must be explicitly skipped by replacing its expression with an asterisk `*`.

```
f() !! All optionals omitted
f(1) !! Trailing optional omitted
f(*, 2) !! Positional optional skipped with *
f(1, 2) !! All arguments provided
```

Keyword Arguments

Positional arguments are identified purely by their order in the function signature and the name is purely for the variable defined in the scope of the function. This is important for the semantics of type checking function signatures. However, in some cases it might also be useful to identify arguments not by their positions but only by their names, particularly if there are a large number of

By replacing one of the commas in an argument list with the pipe character `|`, all following arguments are interpreted as keyword arguments. Keyword arguments may also be optional and non-optional in the same way as positional arguments.

```
fun functionName(Int x, Int y=10 | Int y, Int z=10){
  !! Function body
```

When calling a function with keyword arguments, the argument is given by providing the name of the keyword, the equal sign `=`, and the value of the argument. Keyword arguments must be provided after all positional arguments. Providing positional arguments after a keyword argument will result in a compiler error. Keyword arguments have no ordering and may not be provided positionally at function invocation. This is important for the semantics of type checking function signatures that use keyword arguments.

All non-optional keyword arguments must be provided exactly once. Failing to do so will result in a compiler error. This also means that non-optional arguments cannot be supplied via argument expansion.

```
!! Legal
functionName(1, 2, y=3, z=4)
functionName(1, y=2)
functionName(1, *, y=2)

!! Not legal
functionName(1, 2, 3, 4) !! Keyword argument 7 not provided, and two unexpected
positional arguments.
functionName(1, z=3) !! Keyword argument y not provided
functionName(x=2, y=2, z=3) !! Positional argument x not provided and Unexpected
keyword argument x
```

At invocation time, if there is a variable in scope of the caller whose identifier matches one of the keyword arguments, there is a shorthand for providing that variable as a keyword argument that involves omitting the keyword argument name.

```
fun foo(Int longVariableName = 10) { }
Int longVariableName = 20

!! The following are equivalent
foo(longVariableName = longVariableName)
foo(=longVariableName)
```

Positional Captures

Positional captures are special arguments that allow the function to accept an arbitrary number of extra positional arguments. When a positional capture argument is present, extra positional arguments at the function invocation will populate the positional capture as a list.

The positional capture argument must be placed as the last positional argument. It is indicated to be the positional capture by enclosing the entire argument description (type and identifier) in square brackets. Positional captures are always optional arguments and default values cannot be specified (the default value is simply the empty list when no extra positional arguments are given.)

```
fun f(int x, [Int vargs]) returnType {}
  print(length(args))
  print(args[x])
}

f(1, 2, 3, 4, 5) !! prints 4, 3
f(4, 0, 0, 0, 0, 100, 0) !! prints 6, 100
```

Keyword Captures

Like positional captures, keyword captures may be described by placing a keyword capture argument at the end of the keyword arguments. Keyword captures are indicated by enclosing the argument description in curly brackets.

```
fun functionName(|{Int kwargs}){
  print(kwarg["keyword"])
}

functionName(keyword="Hello World") !!Prints "Hello World"
```

Full Argument Description Syntax

Here, the ellipses ... descrive repetition rather than indicating language syntax.

```
(Type pos_arg, ..., [Type PosCapture] | Type kw_arg, ..., {Type KeywordCapture} )
```

Argument Expansion

List Expansion

You can pass sequences of values into the arguments of a function with the ... suffix. This only works if the types of the sequence match with the types of the corresponding arguments of the function and if the lengths can be verified to match at compile time.

```
fun func1(Int a, Int b, Int c) {}
  !! body
}

fun func2(Int a, Str b, Int? c=None, [Int d]) {
  !! body
}

!! Expanding arrays
[int] list := [1, 2, 3]
func1(list...) !! Type error because the number of arguments may not match if list
had a different value.
func2(1, "2", list...) !! Success as the excess length of the list can be caputed
by the sequence capturing arugment, d.
func2(*list) !! type error because the second argument type does not match

!! Expanding a tuple with heterogenous types
(Int, Str, Int) tuple := (1, "2", 3)
func2(tuple...)

!! Passing a subset
[int] tuple2 := [1, 2]
func1(arr2..., 3)
```

Type Expression for a Function Type

The type expression for a function mirrors the anonymous function syntax with a return type instead of a body, and with type names instead of identifiers for non-keyword arguments.

```
!! Full syntax
(Type, ... | Type kw_name, Type keyword=, ..., [Type], {Type})=>ReturnType

!! Only positional arguments
(Type, Type2, ...)=>ReturnType

!! Only keyword arguments
(| Type1 keyword1, Type2 keyword2=, ..., TypeN keywordN)=>ReturnType

!! Only Captures
([Type])=>ReturnType
({>Type})=>ReturnType
```

Map Expansion

It is also possible to do perform similar action with keyword arguments using the `...` suffix. Arguments provided this way much be optional as the contents of the map cannot be verified at compile time. With this, it is possible to supply the same argument multiple times. When that is the case, the latest provided value will be the final value of the argument.

```
fun func1(Int? a=None, Int? b=None, Int? c=None){
  !! ...
}

fun func1(Int? a=None, {Int kwargs}){
  !! ...
}

{> Str:Int} dict = {> "a" : 1, "b": 2, "c":3}

func1(dict...) !! a=1, b=2, c=3
```

Generator Functions

generator function implementation

There is no special syntax for generator functions besides the yield keyword. Any function with the `yield` keyword in the function body can be a generator function.

When the yield keyword is encountered, that function instead returns a generator. This generator has a new function closure that represents the state of the function

When defining a generator function, the return type is expressed as a generator.

```
<Type>
```

Here's an example of a generator

```
fun FibonacciGenerator <Int>() {
  a, b := (0, 1)
  loop {
    a, b := (b, a+b)
    yield b
  }
}


<Int> gen = FibonacciGenerator()

print(gen.next()) !! 1
print(gen.next()) !! 1
print(gen.next()) !! 2
print(gen.next()) !! 3
print(gen.next()) !! 5
```

Generic Functions

Functions may also be declared with type parameters that allow multiple instances of the function that vary only be their argument and return types.

```
fun funName<T> T(T arg){
  !! ...
}
```

Take for example this simplistic implementation of a map function without generics.

```
fun map [Obj]([Obj] seq, (Obj)=>{Obj} f) {
    return loop for val = seq { f(val) }
}
```

Since the type checker has no assurance that anything besides objects are being passed around here, the programmer is forced to perform numerous cumbersome swype operations to invoke any particular functionality.

```
fun square Obj(Obj x) {
  return swype x {Int: {x**2}, *: { panic() }}
}

[Obj] objs = map([1, 2, 3, 4, 5], square)
[Int] result = swype result objs { [Int]: { objs }, *: { panic() }}
```

Instead, functions declared with type parameters tell the type checker that the input is the same as the output.

```
fun map<S, T> T([S] seq, (S)=>{T} f){
  return loop for val = seq { f(val) }
}

fun square Int(Int x){ return x**2 }
[Int] result = map<Int, Int>([1, 2, 3, 4, 5], square)
```

Additionally, explicit type parameters can be omitted if the type parameters can all be inferred from argument types.

In the following example, the type `T` is inferred from the value of the first argument and the type `S` is inferred from the return type of the second argument.

```
fun square Int(Int x){ return x**2 }
result = map[1, 2, 3, 4, 5], square)
```

Constructors

Constructors

New types in lodge are defined using special functions called constructors (struct for short). When a constructor executes, it returns a new object of the type it defines. The members of this new object are variables that existed in the scope of the constructor. The definition of a constructor also implicitly defines an interface with the same name.

Multiple constructors can produce values with the same public interface and they will be treated equivalently. In that way, types in lodge are defined entirely by form and function rather than a notion of identity.

There are several special pieces of syntax that differentiate constructors from normal functions including getters, setters, and operator functions.

Constructor Syntax

Basic Syntax

The basic syntax of a constructor mirrors the syntax of a function definition as constructors are essentially functions. Constructors do not specify a return type as the return type is automatic.

```
struct ConstructorName(Type1 arg1, Type2 arg2, ...) {
  Int field = arg

  fun method(args...) ReturnType {
    !! Method Body
  }
}
```

The objects produced by a constructor have an interface consisting of the variable names and types in the scope of the constructor. This example constructor produces the following interface for its values.

```
interface ConstructorName {
  Int field

  fun method(args...) ReturnType
}
```

The constructor itself has the type of a function that takes the constructor arguments and returns the type described by the constructor.

Anonymous Constructors

Constructors can also be created without names in order to allow the creation of an object without needing to write a dedicated named constructor for it.

The anonymous values will participate in the typing system as normal, and can even be placed in other variables using the `var` keyword.

```
var anonValue := struct (){
  !! Value body
}()
```

Even though the constructor does not have a defined name, it will still have an internal name and interface like any other class.

Public/Private Methods and Fields

Any variable whose name begins with an underscore will not be included in the public interface of objects produced by the constructor, which means that they cannot be accessed from outside the class and will not be considered for any type set operations.

```
struct A {
  Int _a := 0 !! Private field
  Int a := 10 !! Public field

  fun F Int(Int arg) {
    !! External function
  }

  fun _F Int(Int) {
    !! Private functions
  }

}
```

Properties

A property is a type of struct member that resembles a field from outside the struct, but behaves like a function internally. A property is made out of a getter/setter pair (or just one for a read/write-only property).

Getters and setters can be defined by the `get` and `set` keywords, which are only allowed in struct bodies. What follows the keyword is a syntax that mirrors function definitions without argument lists. Inside the body of the setter, the name of the setter takes the value being provided to the setter.

```
struct A {
  Int _val;
  get Int val {
    return _val
  }
  set Str val {
    _val = Val->Int
  }
}

A a := A()
a.propertyName := "20"
Int val := a.properyName
!! val == 20
```

The get and set type for a given property do not have to be the same. In fact, the get and set properties act essentially independently from one another despite the fact that they have thee same name.

This

Anywhere inside the body of a struct, the identifier `this` is in scope, referring to the value being created by the innermost enclosing struct.

It is considered (by me) bad practice not to use `this` when referring to a struct member from inside a method.

```
struct A(){
  Int x = 0

  fun method(Int x){
    !! The argument to this method masks the member x in the scope of this method,
so we can use this to disambiguate.
    this.x = x
  }
}
```

Operator Methods

In order to implement an operator for a given struct, an operator method can be implemented for the given struct with a special function syntax where the identifier has been replaced with an operator.

```
struct A {
  fun + Int(auto other) {
    !! Logic for adding objects
  }
}

A var1 := A()
A var2 := A()
A var3 := var1 + var2
```

Functions with operator names are only legal within constructors.

There is also special syntax for the index/slice operator `[]` to distinguish between getting and setting values.

```
struct A {
  fun [get] Int(Int index, Int? End=None, Int? Step=None){
    !! Value retrieving logic
  }

fun [set] Int value, Int index, Int? End=None, Int? Step=None){
    !! Value assigning logic
  }
}
```

Type switching inside operator methods

It is best to use the broadest possible type for the operator method. These methods are a particularly good use case of the auto keyword as the type of the argument grows automatically as the ways that the variable is used grows.

Reverse operator methods

When an operator acts on two values, the compiler will look in the interface of the first object that can accept the interface of the second object. If it does not, it will look in the second interface for the reverse operator method (the operator followed by the ~ symbol) that can accept the first interface.

For example, take the expression `a / b` for objects of classes `A` and `B`.

If `A` implements `/` for the type of `B`, then that method will be called on `a` with `b` as an argument.

If `A` does not implement `/` for the type `B`, then the compiler will check for the `/~` method in `B` on the type of `A`.

The intention being that even if `A` was implemented with no knowledge of `B`, `B` can be implemented such that it is still compatible with `A`, by implementing the reciprocal of the operator.

Type Conversion Methods

The type conversion operator must always be implemented using a function generic as the return type must mach the constructor argument given. In fact, the implementation of the type conversion operator must be a function with a generic return type which takes a function that returns that same type.

I'm realizing here that there isn't a good way to represent all possible functions generically; the union between all functions fails...

```
fun -> <T> ( (*)=>T con)
```

Then, the body of the type conversion operator would rely on type switching on struct types to

```
fun -> <T> T ( (*)=>T ) {
  swype

}
```

Promising

When a constructor promises an interface, all public fields and methods are required to be explicitly defined in the promising constructor. Because of automatic interfacing, satisfied promises only makes a difference from the perspective the programmer and doesn't result in code that compiles any differently. The advantage to promising is that it becomes impossible to fail to implement any of the promised methods or fields.

If a constructor promises multiple interfaces containing fields with matching names, the type of that field must be the interface union (type intersection) of the value from all interfaces.

Even if a class/interface is not promised, it will be able to participate in polymorphism via automatic interfacing as long as it matches the criteria to do so. Promising simply ensures that the programmer is required to satisfy those criteria as well as providing more self-documenting code.

```
struct ConstructorName() {
  promises Interface1, Constructor2

  !! Constructor Body
}
```

Wrapping

Note: this replaces the concept of one type “using” another, so any straggling references to using ought to refer to this instead.

The concept of one struct wrapping another in Lodge is very similar to subclassing in other languages, but explicitly uses composition instead of inheritance. Specifically, the wrapping struct holds onto an instance of the wrapped struct as a field.

A wrap statement looks very similar to a variable declaration. However, in addition to creating a field, the wrap statement also creates a method in the wrapping struct for each of the public methods in the wrapped struct. These methods that simply call into the corresponding method in the wrapped instance. Likewise, a getter/setter is created for every public field and property. In that way, A struct that wraps another also inherently satisfies that struct’s interface (and is required to).

Take for example the following struct.

```
struct A() {
    Num val := 10
    Str _privateField := "string"

    fun doThing Int(Str arg) {
      !! Perform some action
    }
}
```

We can write a struct `B` that wraps an instance of `A` with syntax demonstrated by the following example:

```
struct B() {
  wraps _a := A()
  Float z = 1.5
}
```

This is roughly equivalent to the following code where `_a` is declared as a member with the type `A` and all of the public methods, getters, and setters are conveyed.

```
struct B() {
    A _a = A()
    get Int val { _a.val }
    set Int val { _a.val := val}
    fun doThing Int(Str arg) {
      _a.doThing(arg)
    }

    Float z = 1.4
}
```

The difference between wrapping a class and manually defining the members as in the last example is that members created by wrapping are specially marked to allow being overwritten without causing an compiler error so long as the struct still satisfies the required interface.

Multiple Wrapping

It is possible for one struct to wrap multiple others. In the case that public properties or methods would result in a name collision, members from later wrap declarations take priority as they are defined later. However, the struct as a whole must still satisfy the interfaces of both wrapped structs. If wrapping multiple structs would violate this, the code will not compile.

Aware Wrappers

One difference between wrapping and types of superclass extension present in other languages is that a wrapped class will always call its own implementation of a function and never the overwritten version. Consider the following example

```
struct Shape(Int size) {

  fun printArea() {
    print(this.area)
  }
}

struct B(Int age){
  wraps _a = A(age)


}
```

Generics

Struct declarations can also take additional type parameters before the argument list inside angle brackets. In the body of the struct, these parameters can be used as normal types.

```
struct A<T>(){
  T field := ....
}

A<Num> a := new A<Num>()
int val := a.field
```

When the `<T>` is specified, all instances of the type name `T` found in the class definition will be replaced with whatever the type is at the object creation. `T` could be any identifier, but a single uppercase letter is standard.

There is only one implementation for each generic, which simply uses the narrows type union that encapsulates all possible values of T. The difference occurs with the type checker; the type of T is known at compile time for a given object, so the return types of methods can be checked statically and type errors are avoided.

You can also specify restrictions on the generic types with interfaces or type unions.

```
struct A<Iterable T> {
  ...
}

struct B<(int | str | list) T> {
  ...
}
```

Generics vs Unions

There is a small, yet important difference between creating non-generic class whose member types are a type union vs creating a generic class whose generic type is that same type union. The difference arises from the binding time of the type. With the generic, the single allowed type is specified at value creation, whereas the type union allows all the types in the union forever. This difference is demonstrated by the following example.

```
struct A {
  (Int|Str) field
}

struct B<(Int|Str) T> {
  T field
}

A a := A()
B<int> bInt:= B<Int>()
B<string> bStr := B<Str>()

a.field := 10
a.field := "string"

b1.field := 10
b1.field := "string"    !!Compile error

b2.field := 10          !!Compile error
b2.field := "string"
```

For constructor A, both strings and integers can be placed into the field regardless, but with class B, only the generic type specified at the creation of the object can be placed in the field.

Early Returns within Structs

The `return` keyword is not allowed in structs whatsoever, as returning early from a struct could lead to variables defined after the `return` not being in scope yet. However, this would result in the struct having multiple possible interfaces.

It could be allowed iion scope? If they're just in scope inside functions,f every member defined after the return were made private, but this is quite opaque.

Interfaces

Interfaces

An Interfaces is a set of getters, setters, and methods. All values have an interface, which is made up of the public members of that value where fields are represented by pairs of getters and setters. The interface for any particular value is defined in code by the constructor that created it. Interfaces exist separate from the implementation behind those members.

A lodge program consists of many concrete types and many interfaces. Interfaces may be created manually, implicitly by defining a constructor, or implicitly when creating a type union.

When a variable or container is declared, it automatically has some interface. Any concrete type that has a compatible interface can be placed in that container. See for more detail.

Interface Creation

There are several ways that interfaces are created in Lodge

1. Explicitly with an interface definition
   - Using the interface definition syntax
2. Implicitly by creating a constructor
   - Every constructor definition will also create an interface of the same name out of that constructor's public members.
3. Using a type union to combine existing interfaces into a interface set.
4. Using a type intersection to combine existing interfaces interface set.

Behind the scenes, the members that can make up an interface are getters, setters, and methods. This is distinct from how a programmer defines an interface or class, but the separation between getters and setters fields is important for interface intersection, type unions, and properties.

When a constructor or interface definition specifies a public field, both a getter and setter will be created of that type in the interface. When either a getter or a setter is specified, only that getter or setter is created in the interface. Getters and setters of a given name can have the same or different types. That means that, in an interface definition, getters and setters of the same type will have the same result as a field definition. Fields and properties are identical for both external access and interface satisfaction (assuming the getter and setters have the same type).

Interface Name collision

It is possible to create an interface in a given scope even if another (non-interface) name already exists in that scope. This happens every time a struct is defined, as the struct definition creates the callable constructor itself as well as an interface of the same name.

Interface values are only referenceable inside of a type expression, and other values are typically only referencable inside normal expressions.

Interface Definition Syntax

The syntax for defining an interface is similar to the constructor definition syntax, but without any values, method/property implementations, or private members.

Here is an example:

```
interface InterfaceName {
  Int fieldName

  fun functionName Int(Int arg1, Str arg2)

  get Int propertyName
  set Int propertyName
}
```

Generic Interfaces

Like generic structs, generic interfaces can also be created using the same syntax.

```
interface A<T> {
  T fieldName
}
```

Built-in Interfaces

Iterator

```
interface Iterator<T> {
  fun [] T()
  get length
}
```

Generator

```
interface Generator<T> {
  fun next T()
}
```

Iterable

```
interface Iterable<T> {
  fun iterator (Iterator<T>|Generator<T>)

}
```

Err Interface

```
interface Err {
  get Str Type
  get Str Message


  !! There are probable more fields useful in an error, but I can't think of it
}
```

Object Interface

```
interface Object {
  !! probably stuff like string casting and such? Will add as I find it useful
}
```

None Interface

The None interface is a special case. It is defined as the interface that is compatible with no other interface and with which no other interface is compatible.

For the purpose of interface intersection, None is the empty interface, but for interface compatibility, has special properties. This would normally mean that every interface would compatible with the interface. However, since None is specifically defined as the interface that no other interfaces are compatible with, only interfaces compatible with the other interfaces participating in the union would be compatible. This means that nonable types (like `Int?`) have the same set of compatible types (plus specifically None), but the fields are inaccessible until the None case is handled.

```
(Int & None) val !! Failed set operation, and compile error if in a type
expression.
(Int | None) val !! Allowed,
```

ContextManager Interface

This is the interface required to be used with a `while` block

```
interface ContextManager {
  fun context_enter Object([Object args])

  fun context_exit ()
}
```

The Type System

Interface Compatibility

Interface compatibility in Lodge is the process by which the type checker determines if the one value can be substituted for another without violating the expected interface. For one interface to be compatible with another, the interface must contain a superset of the other interface's members.

Checking Compatibility

Consider interfaces A and B. Checking if B is compatible with A consists of evaluating all of the members of A and checking for suitable matching members of B. Recall that interfaces don't really have fields, as fields are the same as getter/setter pairs.

Firstly, for each member in A, there must be a member of B with the same name. Secondly, that member must be compatible with its corresponding member of A.

Getter compatibility
Getters are covariant: for B to be compatible with A, the type of each getter in B must be compatible with the corresponding getter in A.

Setter compatibility
Setters are contravariant: For B to be compatible with A, the type of each setter in A must be compatible with the corresponding getter in B.

For every setter in A
- B must have a setter with the same name and a type that the setter in A is compatible with

Method compatibility
For methods, both covariance and contravariance apply; the return types are covariant, and the argument types are contravariant.

This is complicated by the nuances of variadic method and keyword arguments.

Let's consider a method called B.f checking compatibility with method A.f

Return Type
The return type of B.f must be compatible with the return type of A.f.

Positional Arguments
B.f must have at least as many positional arguments as A.f. If B.f has more positional arguments than A.f, the extra must be optional.

Contravariance: Positional arguments in A.f must be compatible with corresponding (same position) positional arguments in B.f.

Keyword Arguments
B.f's set of keyword arguments must be the superset of A.f's. B.f's keyword arguments that aren't in A.f must be optional.

Contravariance: Keyword arguments in A.f must be compatible with corresponding (same name) arguments in B.f.

Argument Captures
If A.f has a positional capture, B.f must have a positional capture with a compatible type. If A.f has a keyword capture, B.f must have a keyword capture with a compatible type

The Lodge Type Checker

The goal of Lodge's type system is to give the flexibility and feel of a dynamic type system while giving type annotations and full static type checking. To that end, every operation needs to be checked for type correctness at compile time.

These operations include:
- *Variable Types*: Resolving the type and interface of every variable, return, and argument.
- *Expression types*: Resolving the type resulting from an expression
- Object field access: does a given object have the field being accessed as part of its public interface
- *Assignment*: Are all of the values compatible with the type on right hand side of an assignment also compatible with the type on the left hand side
- *Function invocation*: Is a function invocation valid for all of the combinations of concrete types compatible with the variables and expressions used to invoke the function as arguments.

[This section needs to be expanded]

Type Set Operations

Type Unions and Type Intersections

Type unions and type intersections are two ways of creating composite types out of multiple other types.

Type unions are the most common and allow the programmer to create a flexible variable that hold multiple different concrete types, while still being fully type checked.

Type intersections allow the programmer to create a variable that can only contain values that satisfy multiple interfaces. This use case is less common, but allows variables to be entirely

This behavior is represented in the syntax of type unions and intersections:
- `(Type1 | Type2)` is reminiscent of the logical or, indicating that a type compatible with Type1 or Type2 would also be compatible with the type union.
- `(type1 & type2)` is likewise reminiscent of logical and, indicating that a type must be compatible wit both Type1 and Type2.

Type Unions

Type unions are an important component of Lodge's typing system. A type union consists of one or more types that are combined into a single type that allows for the

inclusion of any type specified in the union. The interface for a given type union must be represented as an interface intersection and interface set of the participating interfaces. Any value compatible with every interface in the set may be placed in any variable or container with that union as its type

Defining type unions

A type union can be defined by combining several types with the `|` character in a type expression.

```
(Int | Str) variable := ...    !! Declaring a variable with an unnamed union
```

An unnamed type union refers to all cases where the type union is not bound to a name such as the case below where the variable is being declared to have a type which is the union if int and string

It is also possible in lodge to bind a type union to a name using the `as` keyword. After defining a name for a type union, that name can be used as shorthand to refer to the union after the name definition statement and in all enclosed scopes. The name can be used in any case where the type union could have been used including variable declaration, type switching, and even the definition of further type unions.

```
(Int | Str) as Intstr

Intstr variable := ...
```

The Interface of a Type Union

The interface of a type union is represented by the interface intersection of all participating interfaces.

Adding more types to a union decreases the number of members in that union, the set of types compatible with that interface may grow beyond just the set of participating types. However, compatibility with a type union requires compatibility with at least one of the participating interfaces. Thus, for the sake of determining compatibility, type unions must also maintain the set of participating interfaces.

Type Intersections

A type intersection is the opposite of a type union. It is important for type checking relating to multiple interface promising, as the required public interface of a class that promises multiple interfaces is equivalent to the type intersection of those interface types.

Syntax

`(Type1 & Type2)`

This form is most useful for combining component interfaces and requiring objects to promise several interfaces for compatibility with type intersections. For example, it

might be the case that you want to require that a class promises an Iterable interface and also serializable with something like this: `(Iterable & Serializable)`

Thus, Lodge code is able to be fully agnostic to the concrete type of an object while still mandating the behavior it needs from the object.

The Interface of a Type Intersections

The interface of a type union is represented by the interface union of all participating interfaces.

Type Intersections do not need to maintain an interface set as compatibility with the combined interface is enough to determine compatibility with the type intersection.

Interface Set Operations

Interface Intersection and Interface Union

These operations differ somewhat from strict definitions of mathematical set operations as the equivalence of interface members is depended on both the name and types of the members. Additionally, the merging of two members is more complex than simply checking for their presence. See the sections on interface intersection and functions for more information on this behavior.

The naming of type type unions and type intersections might seem counter-intuitive as the result of a type union is an interface intersection and vice versa. However, this naming is sensible when you consider the set of compatible types for a give type union/intersection:

- When you create a type union out of several types, the set of interfaces that are compatible with that type union is the union of the sets of types that are compatible with the types that participate in the union.
- When you create a type intersection out of several types, the set of interfaces that are compatible with that type intersection is the intersection of the sets of types that are compatible with the types that participate in the intersection.

Interface Intersection

Interface intersection is the process by which multiple interfaces are combined into one for the purposes of type unioning.

When several types exist in a union together, that type union will have its own interface that is the intersection of the public interfaces of all participating types where compatibility is determined by compatibility with all interfaces in a interface set.

It is worth noting that not every intersection is possible. Since any setters or method arguments require their types to be intersected, there is the chance that a namespace collision results in that type intersection being impossible.

The rules for determining an interface intersection are complex however, in most realistic situations, compatibility will be determined mostly by the presence or absence of fields.

Process of interface intersection

Each interface contains a set of members that are either getters, setters, or methods.

During interface intersection, each member that has a corresponding member in every interface is a candidate for being included in the intersection. Candidate members do not need to have the same type, but they do need to be the same sort of member (getter / setter / method).

Notably, any member that doesn't have a matching name in all interfaces will not be included in the intersection at all.

Then, each candidate is evaluated to see how it can be merged together to form a member of the final intersected interface.

How this merging works depends on what kind of field, but the result of this merging is either an update of the member's set of types or a removal of the member from candidacy entirely.

Merging Candidate Getters

Any code dealing with the intersection of several getters must be able to handle the types from any of the getters matching the candidate. Thus, the type of the member in the final intersection is simply the type union made up of all of the members matching this candidate.

For example, when merging the two following interfaces:

```
interface A {
  get Int val
}

interface B {
  get Str val
}
```

The result of `(A | B) as C` would be the following:

```
interface C {
  get val (Int | Str)
}
```

In this case, when accessing the `val` getter from a variable of type C, the object stored behind the interface C could be either a B or an A, and `val` could either be an int or a string. This means that code that deals with C has to be able to handle the type of `val` regardless of what it is.

In the case where the type union of the getter types fails or results in an empty interface, that getter will be removed from interface of the union.

Merging Candidate Setters

The type of a setter in the intersected interface must only be able to accept types that all setters matching the candidate can accept. Thus, the new type of the setter will be the type intersection (not interface intersection).

For example, when merging the two following interfaces:

```
interface A {
  set val Int
}

interface B {
  set val Str
}
```

The result of `(A | B) as C` would be the following:

```
interface C {
  get val (Int & Str)
}
```

In the case where the type intersection of the setter types fails, that setter will be removed from the interface of the union.

Merging Candidate Methods
See the section on function interfaces.

Interface Union

A union between multiple interfaces is the result of a .

Process of interface union

To start with, an interface is formed by placing the members of all participating interfaces together. However, there may be some elements with shared names, that require special consideration for their types. This is also the way in which some interface unions may fail.

Getters with shared names
An object that is compatible with the union of multiple interfaces must be able to properly behave like any of the of the types in the interface union. For example, consider the type intersection `(A & B)`. Any class that would be compatible with this interface union must be compatible with both A and B in every case. Thus, if both A and B have a getter of the same name, those two getters must be mutually compatible (A is compatible with B and B is compatible with A). Mutual compatibility requires that the two interfaces are identical.

If there are getters with shared names and all getters do not have the same exact interface, the interface union will fail with a compiler error. Is this true???

For example, consider merging the two following interfaces:

```
interface A {
  get val Int
}

interface B {
  get val Str
}
```

The line `(A & B)` would result in a compiler error.

The failure of union is a non-ideal outcome, thus it is best practice in lodge to avoid constructing classes that are likely to result in failures when unioned with interfaces that they are likely to be unioned with.
- Using distinctive names for members to reduce collisions
- Reducing the public interfaces of classes as much as possible

Setters with shared names
Setters with shared names are somewhat simpler as the types of each setter is simply the type union of all the candidate setters.

Much like in interface intersection, when the interface of a member becomes empty as the result of type unioning, that member is removed from the resulting interface.

For example, when merging the two following interfaces:

```
interface A {
  get val Int
}

interface B {
  get val Str
}
```

The result of `(A & B) as C` would be the following:

```
interface C {
  get val (Int | Str)
}
```

Methods with shared names
See the section on function interfaces.

Errable and Nonable Types

Errable and Nonable types simply refer to any type union that contains `Err` or `None` respectively. These use cases are common enough that they have their own special syntax for declaration.

Errable

Types with the `!` token are unioned with `Err`

```
(Err | Str)

!! Is the same as

Str!
```

Nonable

Types with the `?` token are unioned with `None`

```
(None | Str)

!! Is the same as

Str?
```

Type Switching

It may sometimes be useful to differentiate between types in the set of types compatible with a particular interface. To achieve this while maintaining type safety, type switches can be used. The difference between a type switch and a value switch is that, within a type switch, the variable will be treated as matched type for the purposes of member visibility and type checking. Example of a basic type switch.

Inside each of those blocks, the variable will have the interface of the type it was matched to. This allows members to be accessed inside the scope that would not be accessible outside of the scope.

```
(List | Int | String) variable := !! Something
swype variable {
  List : { print(length(variable)) }
  Int  : { print(variable/2) }
  *    : { Catches the remaining cases }
}
```

If a type is captured by the `*` case, will have the interface that the variable had in the enclosing scope.

Swype statements can also be place in-line using commas to separate the cases instead of newlines.

```
(Int | String) var = "11"

Int var2 = swype var {Int: { var/10 }, Str: { 0->Int / 10 } }
```

Switching multiple types at once

```
(Int | Str) a := 10
(Int | Str) b := "20"

swype a, b {
  Int, Int : { !- Code -! }
  Str, Str : { !- Code -! }
  Str, *   : { !- Code -! }
  *, Int   : { !- Code -! }
}
```

Switching the Type Created by a Constructor

[This section needs to be expanded]

I haven't settled on the syntax for this yet, but there should be some way to specify that an argument in a type switch should be treated as a constructor and the swype should look at the type produced by that constructor. Due to the fact that structs are a special kind of function, this would also let the programmer switch over the type returned by a function.

```
swype <<Con>> {
  Int : { !- The struct produces Ints -! }
  Str : { !- The struct produces Strs -! }
  *   : { !- The struct produces something else-! }
}
```

Formalization

This section is an attempt at constructing a formalization of the type system that can be used to prove correctness and decidability of the type checker.

We make two simplifying assumptiojs about the problem:

1. Method-Only members
In order to reduce the number of cases that need to be considered, we reduce all type members to methods. Fields can be described as a getter and a setter, while getters and setters can both be describeds as methods. A getter can be described by a method that takes no arguments and returns a single value, and a setter can be described by a method that takes a single value

2. Member Sets
We assume that a type can be fully described by a set containing all of its member methods. Additionally, we make no distinction between concrete, and interface types as compatibility checking operations do not depend on a concrete implementation to exist.

Members
A member consists of a name, a return type, and a sequence of argument types. A member then is, formally, a

$M = (N, R, A)$

1. N is a string representing the name of the member
2. R is a member set
3. A is an ordered sequence of member sets $(a_1, a_2, a_3, ..., a_n)$

Member Set
A type, then, is a set of members with distinct names.
$T = \{M_1, M_2, M_3, ...\}$ s.t. $M_i \neq M_j$ for all $i, j \in \mathbb{N}$ where $i \neq j$
When we refer to a type we refer to a particular set of members.

Compatibility Relations

To describe the relationships between types and their memebers, we define two relations, member compatibility, ${}_m\supseteq$, and type compatibility ${}_t\supseteq$. Both of these relations are defined in terms of the other.

Member Compatibility
For x, y, which are members $x \ {}_m\supseteq \ y \Leftrightarrow x.N = y.N \wedge x.R \ {}_t\supseteq \ y.R \wedge |x.A| = |y.A| \wedge y.A_i \ {}_t\supseteq \ x.A_i \forall i \in [1, |x.A|]$
In other words, a member x is compatible with another member y if and only if they have the same, the return type of x is compatible with the return type of y, they have the same number of arguments, and each argument type of y is compatible with the corresponding argument type of x.

Type Compatibility (Member Set Compatibility)
For A, B, which are types, $A \ {}_t\supseteq \ B \Leftrightarrow \forall a \in A \exists b \in B$ s.t. $a \ {}_m\supseteq \ b$
In other words, a type A is compatible with another type B if and only if all of the members in A have a member in B that they are compatible with.

Intermediate

Scoping Rules

Lodge is a lexically-scoped language; blocks create new scopes, a variable defined in a block will go out of scope at the end of the scope, and each block's scope will contain the variables from the enclosing scope. Variables aren't in scope until after their declaration.

Scope Lifetimes

There are cases where a particular instance of a scope can continue to exist after program execution leaves that block. Namely, functions defined inside of a scope that have references to values in that scope can cause a scope to persist if that function is accessed elsewhere as. Likewise,

Function Hoisting

[This section needs to be expanded]

We really want to be able to call functions before they're defined in the global scope as this would get really annoying to have to worry about in structs.

If functions exist in the whole scope they're defined in, what value do other variables in the same scope get inside the function scope?

I think I may make a special exception for the highest-level scope in a file where functions are in-scope the entire time. I would then just have to enforce the rule that the highest-level scope could only have statically determinable declarations, which would be quite unfortunate. I could also execute lodge code in the highest scope at compile time, but that is also quite odd.

```
struct ImmutableStruct(...){
  Int _x = 10;
  get x { return _x }
  set x { _x = x }
}
```

Enums

Enums are tricky to fit into the type system, but are super useful...

[This section needs to be expanded]

Regular enum

```
enum EnumName {
  OPTION1,
  OPTION2,
  OPTION3,
}
```

Enums with values

```
enum EnumName {
  OPTION1(Type, Type),
  OPTION2(Type, Type),
  OPTION3(Type, Type),
}
```

Getting the values from an enum value (calling?)

```
EnumName variable := EnumName.OPTION1
val1, val1 := variable()
```

With Statements

With statements let the programmer automatically manage resources that need to be explicitly opened and closed using a block in which the resource

Any object that satisfies the ContextManager interface can be placed in argument of a with statement.

```
with mutext_lock {

}
```

You can also place an assignment in the statement.

```
with file_handle = open("filname".txt) {

}
```

Broadcasting

[This section needs to be expanded]

Closures

[This section needs to be expanded]

Error Handling

Lodge's error system is based on errable types.

[This section needs to be expanded]

Imports

```
import packageA
import packageA as name
import packageA.subpackage
import packageB include name1, name2, ...
import packageC include *


import packageD
packageD include *
```

In this case, `package_name` could be a lodge package, a file, or a directory.

Include Statements

Maybe in a future version:

Brings all of the values from a specified namespace into a given scope

```
fun Myfunc (SomeType value) {
  value include *
  !! All of the accessible public members of value are now local variables in this
function
}
```

Memory Managed Types

Constructors automatically allocate just the amount of memory needed for the fields they contain. However, there are some cases in which the programmer might want more fine-grained control over the memory.

For such cases, lodge provides the `Data` and `Byte` types. Data looks like other types syntactically but specifically circumvents the type systems to allow things that wouldn't otherwise be able to be implemented with as much memory efficiency.

When the present scope is a struct, it is legal to declare a Data field. Data fields use a special syntax only valid inside of a struct. The given name, which is a function that returns the type produced by the can be referenced inside the struct's scope, but may not be placed in another variable or returned, and may not be public.

```
struct A {
  Data _name = size
}
```

The primary way to interact with a Data field is to access it at a byte level using slice/index notation, which returns a Bytes object, which can be assigned and passed around. Unlike array bounds access, this does not return an errable value. If a Data field has an out-of-bounds access, lodge will simply crash (maybe not?)

Additionally, the Bytes object defines a type conversion operator `->` capable of handling . If the Bytes object contains a number of bytes equal to the size of objects

defined by the struct, it will produce. Likewise, it defines the reverse type conversion operator that any struct can be defined.

Both of these operator applications are a special case that only The Bytes implementation is capable of achieving.

```
struct A {
  Data _data = 50

  !! Interpretin a segment of data as an integer
  Bytes four_bytes := _data[0:4]!
  Int x := four_bytes->Int
  x *= 2
  four_bytes := x->Bytes
  _data[0:4] := four_bytes

  !! Or equivalentlyrse conversion is also a special case as it fills in t
  _data[0:4] := (_data[0:4]->Int! * 2)->Bytes!
}
```

This can be especially useful for making more compact array-like objects that don't have the overhead of type tags and padding.

```
struct IntArray(Int length) {
  Data data = length*4

  fun [get] Int!(Int index) {
    if (index < 0 | index >= length) { return IndexError() }
    return data[index*4:index*4+4]->Int
  }

  fun [set] Int!(Int index, Int val) {
    if (index < 0 | index >= length) { return IndexError() }

    data[index*4:index*4+4] = val->Bytes
  }
}
```