Lento

Programming Language
Specification

Author | William Rågstad

Date | 2023-10-03

Abstract

This specification describes *the Lento programming language's* syntax, structure, and implementation design.

Lento is a general-purpose, **strong, statically typed**, and **functional** programming language designed to *purify* the object-oriented paradigm.
The Lento toolchain supports **interpreted** and **compiled** execution environments running on modern target platforms such as *Windows*, *Mac*, and *Linux*. The Lento project also aims to support compilation to WebAssembly.

Hygienic macros can **extend** the language in various aspects, such as syntax and semantics. However, Lento also supports third-party runtimes or VMs using custom compiler backends, offering a **rich set of interoperable libraries** and applications. Therefore, Lento is the optimal high-level language to learn, as it will allow you to extend your knowledge across many application areas in *just one language*.

For the moment, an official interpreter is developed using Rust.

Table of Contents

Introduction

This chapter delves into the foundational principles and design methodology behind Lento. It aims to provide insights into the language's purpose, ideals, approach to type systems, error handling, and the target audience and relationship to the developer community.

Our Values

Lento strives for elegance, conciseness, and a high level of expressivity, fighting against redundancy and boilerplate code. We also value simplicity and readability. For example, the ability to use compact mathematical notation via overloaded Unicode operators and macro systems should appeal to *the perfectionist programmer*. Who values craftsmanship and pragmatism in shaping code and reliable software architecture.

Methodology

The methodology behind Lento is a harmonious blend of theoretical rigor and practical utility. It adheres to a functional paradigm, enhancing machine-level efficiency and human-level comprehensibility.

A Good Type System™

The power of a language comes from its ability to support the developers in their work. This is why new programming languages should implement this capacity. Lento does this correctly: the type system is **strong** and **static**, but most importantly, it has a *good* type **inference** engine, directly reducing the amount of code required and the burden on developers.

A Pure Paradigm

Lento's design philosophy embodies a unique blend of paradigmatic purity and pragmatic flexibility. Its commitment to a functional paradigm, characterized by lazy evaluation and strong, static typing, serves as both an optimization strategy and a philosophical stance on computational epistemology. These features aim to enhance human cognition and machine computation, positioning Lento as a sharp programming tool and a medium for intellectual expression.

Exception Handling

It might be surprising that we bring error and exception handling up this early. However, this is a central concept and design principle in Lento.
Unlike some languages, it avoids traditional try-catch exception handling.

Lento instead favors type-based status indication. More *purely*, using a return type specifying the status of an unsafe operation. Setting the result status of a function can be done utilizing **enums** or **atoms**, for example. Take a look at the practical example below.

```
enum AddStatus = Success | AdditionFailed    // Enum
add : int, int -> AddStatus
type SubStatus = :Ok | :SubtractionFailed    // Atoms
sub : int, int -> SubStatus
```

(Function body is intentionally left out in this example.)
Consider adding a handler function for error cases for larger projects where an unsafe operation is often used.

Target Audience

Lento is designed to be used by a range of people, from beginners to professional software developers. Its strong type system and functional paradigm, readable syntax, and friendly compiler make it particularly pleasant for any programmer but also appealing to intellectually curious developers who value paradigmatic purity and pragmatic flexibility. At the same time, its pragmatic features and extensibility make it accessible for developers working on complex, real-world applications, thereby broadening its applicability across various domains.

Community

Inspired by TypeScript and Haskell.
Lento should have a close relation with its users being developed open-source on Github, enabling users to give immediate feedback on features and report bugs, as well as following the development of the language.

Language Design

The language design is inspired by *C#, F#, TypeScript, Haskell, Ada, OCaml, Elixir, and Rust*.

Variables

Variables are differentiated from functions and are immutable per default.
Variables are parameterless functions. Referencing a variable in code will be the same as invoking the variable as a function.

```
x = 10 // Constant (inferred integer)
y: string = "Foo"
mut z: char = 'A' // Mutable variable
z = 'B' // Re-assignable with value of the same type
```

Using the variables

```
println y + z + x // Prints FooB10 to the console
```

Function calls do not require surrounding parenthesis.

Identifier

A variable is an identifier of the following pattern: `[\w\_]+[\d\w\_]*`

```
snake_case
numbers123
PascalCase
UPPERCASE
camelCase
snake_case
_1
```

If a variable name begins with an underscore, it is interpreted as a “match everything”/**ignore** in assignments and pattern matching. Thus, it is **not** saved in the environment.

Immutability

Because a variable is not itself a data structure but more like a memory cell, we are only sure that one type of data structure fits in there, one at a time. Therefore, we can only read- and write to that variable if it is the same type. For a memory cell, that’s ones and zeros, but in Lento, we can write anything with the same type declared for the variable.
Any other data structures in Lento are fully immutable.

Mutability

In some cases, variables must be able to mutate to allow for different types of logic. That is why we decided to support a keyword called `mut`, which enables a variable's value to be reassigned in the same or a nested scope without creating a copy but performing the change *in place*. This is inspired by Rust.

Functions

Functions are located in a separate register in the environment. Functions are stored in a separate list in the application environment (Elixir?).
Invoked using trailing parenthesis and arguments. All function declarations support pattern matching in their parameter list.

Semantics

Function declarations can have parameters with surrounding parentheses, same with function invocation providing arguments. A function might have an explicit type signature unless it is inferred. Functions are denoted as a function name, parameter vector, equal sign, and function body. Functions may have multiple declarations for different parameter list matches. All of which must share the same return type. Therefore, using a processing type signature is good practice, which enforces the varying functions to return the same types. Unless a type signature is given, the first function declaration will control what will be the shared return type.

Function body

A function body is a **single expression** or a sequence of statements surrounded by curly brackets, this is a **code block**.

Declaration

Functions are overridable, meaning the most recent declaration of a function with a matching parameter list will be used. This is the same for variables, where any variable can be reassigned.

Type signature

A function may declare an explicit type signature to specify a formal function interface. This could also be inferred.

The signatures for parameterless functions are simply nothing. They are only generating output and are denoted like:

```
sayHi : () -> string
sayHi() = "Hi"
```

Named functions with single expression

```
myAdd(a: int, b: int) -> int = a + b
myAdd(a: int, b: int) = a + b         // Implicit (infer) return type
```

Function with type signature (similar to Haskell). The last `int` shows the function return type.

```
myAdd : int, int -> int
myAdd(a, b) = a + b
```

Named functions with body

A function body is a sequence of statements and expressions surrounded by curly brackets.

```
myAdd(a: int, b: int) -> int {
    c = a + b
    return c      // Explicit return value
}
```

Invocation/Application

Function applications are done by specifying a function identifier and any parameters separated by whitespace. This is called function application by adjacency, applying a function to one or more arguments.

```
myFunc 1 "hello"
```

Function application can also be formatted using a direct trailing tuple for each parameter.

```
myFunc(1, "hello")
```

The elements in the parameter vector tuple will be mapped to the function's arguments. Listen to why Anders Hejlsberg says this might be useful in his TSConf 2019 Keynote.

A space between the function identifier and the tuple indicates that the tuple is passed as a parameter to the function. This means the line below would throw an error for the same function used in the previous example.

```
myFunc (1, "hello") // Invalid Parameter Type
```

Empty parameter tuple

Because the function name itself acts as a reference and can be passed around, function variants with zero arguments must be passed the zero tuple `()`.

> **Functions with no arguments cannot be invoked without trailing parentheses. This is to avoid any ambiguities regarding function referencing and application.**

For all other function arities, the parameters may be passed without surrounding parentheses and separated by spaces instead of commas.

Function variations

If a function has multiple overloaded definitions. The first function variation that successfully pattern matches arguments to the parameter signature will be invoked. Otherwise, a new curried function will be returned, taking the rest of the arguments. The first least matching is the method to select which declaration to use to curry. This means that if a function declaration takes n parameters, and the function is given m arguments, where $m < n$. If those n number of arguments successfully match with n parameters of that declaration (from top to bottom), that function will be curried and returned.

> **A function cannot have two declarations with the same parameter vector! This breaks the matching property as the first occurrence will always match, and this is therefore omitted from the language as is seen as an error.**

Currying

Functions are using currying. If a function gets partially applied to a set of parameters, it returns a new function, taking the rest of the defined parameters.

```
myFunc2 : string -> string
myFunc2 = myFunc 1
myFunc2 "hello" // This is ok!
```

References

```
zero : int
zero() = 0 // same as zero = 0, aka a variable

add : int, int -> int
add a b = a + b // arity of 2

other(zero)
other(&add) // Because add is a function with arity > 0, we need to reference it using the & operator to show that we are not invoking the function
```

Code blocks

A code block is a sequence of statements surrounded by curly-braces `{}`, containing zero or more `return` calls to produce a final result value (unless it is `void`) and return to the default code branch. A code block will *always return the value of the last statement or expression* unless any `return` statements are present. The statements in the block may be inlined using a separating semi-colon '`;`' (like all C-like languages).

```
stms = { a = 21 * 2; sq(x) = x * x; sq a } // 1764
```

Return

The value returned by the code block is, per default, always the result of the last expression.

```
result = {
    a = cond {
        10 - x > 5 -> true
        2 * x < 100 -> false
        true -> false
    }
    !(false || a) // This expression is what is returned by the block
}
```

A return statement may be inserted before the expression on the last line to clarify that the value is indeed returned, but this is otherwise unnecessary. Return is a built-in macro that sets the return value of the context and immediately exits. The purpose of this *keyword* is mainly to give developers control over the program's flow of execution. Read more about return and other similar keywords like break and continue.

Comments

Single-line comments are denoted by a `//`. Or alternatively, a multi-line comment beginning with `/*` and ending with `*/`.

Documentation

Documentional comments describe code functionality on a more general level *(function or module)*. They are denoted by some lines starting with `///`.
An example doc-comment looks like the following:

```
/// Description: A function that adds two numbers together
/// Param(a: Num): First number
/// Param(b: Num): Second number
/// Returns: The sum of the two numbers
add : Num, Num -> Num
add a, b = a + b
```

Lambda Expressions

A lambda expression is an unnamed or “anonymous” function. They are **higher-order** functions **not** stored in the environment. The declaration for a lambda function follows the pattern: parameter vector followed by a thick arrow and, lastly, a single expression or code block. The parameter vector could be formatted just as a normal function. The only difference is that the lambda expression type is denoted using a surrounding ().

```
myHello : (string -> string), string -> string
myHello(nameGenerator: (string -> string), title: string) =
    "Hello " + nameGenerator(title)
greeting = myHello((title: string) => title + "William", "Mr. ")
greeting // Hello Mr. William
```

Lambda functions are invoked just like all other functions, unlike other languages such as Elixir. This is for the sake of continuity and cohesion.

If an anonymous function is assigned to a named variable, the result is a regular function. It has the same type signature (without the extra '()')

```
myHello : () -> string
myHello() {
    return "Hello"
}
myHello   // Reference to function
myHello() // “Hello”
```

Or with no surrounding parentheses and single statement body.

```
myHello: string -> string = (name) => "Hello " + name
```

The code below is not a lambda function but a regular one.

```
myHello(name: string) -> string = "Hello" + name
```

Both functions are invoked as shown below.

```
myHello "Bob"   // “Hello Bob”
myHello("Bob") // “Hello Bob”
```

Zero-Cost Abstractions

We'll later talk about concepts mentioned in this chapter. Lento aims to be a pure functional language but still behave similarly to general-purpose mainstream languages. With this goal, some constructs must be implemented using immutable design hidden under a level of abstraction with an interface that looks and feels ergonomic.

Decorators

Function decorators change the behavior syntactically and/or logically.
Decorators are macros behind the scenes.
They work similar to decorators in Python.

```
decorator Infix<T1, T2> Function operator {
    register [T1, operator.Name, T2]
    transform AST expr {
        return [operator.Name, T1, T2]
    }
}
```

```
@Infix
+ : Num n => n, n -> n
Num + lhs rhs = Kernel.add(lhs, rhs)

Console.WriteLine 5 + 10 // Invokes +(5,10) -> Kernel.add(5,10) -> 15
```

Classes

Define a new object structure protocol.
A classification for object structures instantiated by the class. All objects are value-based, i.e., non-referenced. A class can implement a predefined protocol.
Classes may inherit from other classes.

```
class A : B {
   // A now contains the fields and methods in B
}
A == B // false
```

Protocols

A specification of data structure requirements. A protocol accepts any data structure or object instantiated by a class containing all required fields or properties.

Objects

Lento should support syntactic sugar for creating classes (structs) and objects.
Inspired by Object Initializer in C#.

```
class Car {
    // Fields
    Brand: string = :unknown,
    int Model, // Use null as default
    int Speed: 0
}
// Methods
Car Car.New string brand = Car{
    Brand: brand,
    Model: 1
    // Use default Speed value
}
Car Car.Drive Car car, int speed = modify car {
    Speed: speed
}
string Car.BrandModel1 Car car {
    Car { Brand: brand, Model: model } = car
    return brand + model
}
string Car.BrandModel2 Car{ Brand: brand, Model: model } = brand + model
// Pattern matching
string Car.BrandModel3 Car car = car.Brand + car.Model // Access class
map fields

Car car = Car.New("Volvo") // Hides class instantiation
car.Drive(50) // Syntactic sugar for the statement below
Car.Drive(car, 0)

car.Brand // "Volvo"
car.BrandModel1 // "Volvo"
car.BrandModel2 // "Volvo"

// Structs can also be immediately created from a class
Car car2 = Car {
    Brand: "Fiat",
    Model: 2,
    Speed: 5
}
```

In this code example, we specify a new class for objects/structs which provide a set of two public methods. There are no static keywords as in C# or Java due to all functions being pattern-matched with the passed parameters thus all functions are static by default. If there is a function/method in a class body that takes a reference to a class instance as its first argument, it will be callable using the `.` syntax. This means you can write an instance, followed by a dot, followed by the method that takes (a reference to) a struct/object. This is shown in the last two lines in the example.

Class

Defines a new object structure protocol.

Modify

Takes an instance and returns a modified copy with just the specified fields updated to new values.
Works like partial types in TypeScript.

Fields

All fields in a class are public, meaning they are accessible on objects using dot notation. The reason for this is that you don't have to write getters for each field, and because they are immutable there is no reason not to expose them.
See **Car.BrandModel3** for example.

Methods

Each method has an **inferred return type** of the parent class unless anything else is specified. All public methods must return a new or modified structure of the class described.

All methods act like extension methods to an object or protocol, similar to the C# implementation.

Static

Every method in a class is per default static. But if a method on a class takes an object of that class as the first argument, you can use dot notation to pass the object as the first argument to the method call and the rest of the arguments are written as parameters to that method call.
See **Car.Drive** for an example.

Unless a method on a class doesn't take an instantiated object as the first argument, then the method is only accessible from the class using dot notation as well.
See **Car.New**.

These types of methods are called static methods and specify that the function is only accessible on the class itself and not objects instantiating the class. These are most used when creating custom constructors. Static methods are functions that don't take a class object as the first argument.

Destructuring assignment

Classes are destructively pattern-matched, as shown in the example below.

```
Car car = Car.New("Volvo")
Car { Brand: brand } = car
brand // "Volvo"

// or using the get methods
car.GetBrand1() // "Volvo"
car.GetBrand2() // "Volvo"
```

Pattern Matching

A mechanism for checking a value against a number of patterns. It is a much more readable and expressive shorthand for conditional branching.

Match statement

Pattern matching is mainly facilitated through the match construct, which takes an expression and compares it against multiple patterns until it finds a matching one defined in its body. The code follows each pattern clause to execute if the pattern matches. Patterns can also have guards specified by the when keyword, which must also be true for the match to be successful. Keep in mind errors in guards do not leak but simply make the guard fail.

```
match x {
    1 => "One",
    2 => "Two",
    _ => "Other"
}
```

In the example above, the variable x is matched against the patterns 1, 2, and _ (wildcard). If x is 1, it returns "One"; if x is 2, it returns "Two"; for any other value, it returns "Other".
Or using larger records/objects.

```
match {1, 2, 3} {
      {4, 5, 6} => "This clause won't match",
      {1, x, 3} when x > 10 => "This clause won't match, x = 2 < 10",
      {1, x, 3} => "This clause will match and bind x to 2",
      _ => "This clause would match any value"
}
// -> "This clause will match and bind x to 2"
```

https://elixir-lang.org/getting-started/case-cond-and-if.html

Cond statement

The condition statement is a subset and alias for the case statement. The logic is limited to the true boolean instead of matching any value. Every clause will be evaluated; if the result matches with `true`, that clause is evaluated, and the condition returns.

```
cond {
    1+1 == 3 => "We live in a simulation",
    true => {
         // This will always match, like ‘else’
         return 42
    }
}

cond !false == true => "Always true" // One-liner
```

A condition with only one clause can be inlined as seen above. This will return `null` if the expression does not match or is not evaluated to `true`.

The is Operator

The is operator in Lento serves a dual purpose: it destructures a value and performs a conditional check. This is *similar to the if let construct in Rust* but is more flexible as it allows for multiple instances in a single if statement. If all instances match, the if block is executed.

```
x: Option<int | bool> = // Code ...
if x is Some(y) and y is int and y > 10 {
    // This block will execute if x is Some(y) and y > 10
    println y
}
```

In the example above, x is destructured to y, and a conditional check y > 10 is performed. If both conditions are met, the code inside the if block will execute.

Other Keywords

With keywords, we do not mean the hardcoded overloadable keywords you see in other languages. Instead, keywords are prepackaged functions in Lento but may be overloaded with a simple function or macro declaration in your code.

Return

The return keyword does whatever you expect it to do. It returns from the current context with no or a single value. The function sets the return value “register” and exits out of the code block. Read more about the return.

Type

Custom types are created using the type keyword.

```
type Numeric = int | float | decimal
add : Numeric, Numeric -> Numeric
add a b = a + b
```

Enums

An enum is a type specification that limits the value to a set of given unique values.

Iterators

For an object to be an iterator, it must implement the standard `Iterator` protocol. This ensures that you can call a `.next()` method on it, returning a value and an indicator for whether the loop shall continue after or if the method simply returns `null`.
There will be an operator '`in`' that gives iterators a more concise syntax. It may look something like this

```
list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for value in list {
   println value
}
```

More operators

http://hackage.haskell.org/package/cond-0.1/docs/Control-Cond.html
https://elixir-lang.org/getting-started/case-cond-and-if.html

Concurrent Model

Message passing between processes, should support network abstraction.
Inspired by Elixir and C#.
Modified **Asynchronous Communicating Sequential Processes** (ACSP), meaning messages are sent through channels, and a process can choose to read a message from one or more channels.

Synchronous Channel

This will be the "main" channel for the process.

```
println "Receiving..."

match receive self {
    {:msg, message} -> println message
    _ -> println "No message received!"
}
println "Sending"
send self {:msg, "My message"}
println "Done"

# Output:
# Receiving...
# -> Stuck in infinite loop
```

Asynchronous Channels

```
println "Receiving..."
```

```
match receive :channel1 {
    {:msg, message} -> println message
    _ -> println "No message received!"
}
println "Sending"
send :channel1 {:msg, "My message"}
println "Done"

# Output:
# Receiving...
# Sending
# My message
# Done
```

Type System

Inspired by Haskell, C#, and TypeScript.
Nominal type system.
Allows for specifying new type aliases and custom type formats and how they are stored in binary, classes, generics, structs, tuples (varying length), and maps.
It follows a parametric polymorphism.

Dependent types

Hierarchy

1. **U** - The universal type of all types.
2. **any** - The *any*-type.

Primitive Types

Type	**Description**	**Syntax**	**Identifier**
Atom	An atom	:\w[\w\d]*	-
Binary	Bit sequence	[01]+b	-
Int	32-bit integer	\d+	int
~~Long~~	64-bit integer	\d+	int64
~~long int~~	32 + 64		int96
~~long long~~	64 + 64		int128
Float	IEEE Floating point number. Single precision.	\d*.\d+	float
~~Double~~	Double precision.		float64
~~Decimal~~	128-bit floating point.		float128

Collection Types

Type	Description	Syntax	Read complexity	Space complexity
Vector/ Tuple	An array of elements with a fixed length and fixed positions. It is possible to index into.	(a, b, c, ...)	**1** using pattern matching and indexing	**N**
List	Linked list or varying length but fixed positions using tuples. It is possible to index into.	[a, b, c]	**1** using pattern matching. **N** when indexing.	**N**
Map	Hashmap of varying length and floating positions. See Associative array and Elixir map.	(a: *A*, b: *B*)	**log N**	**N**

Container objects are created dynamically and are considered value-based data types/structures.

https://www.jeremyshanks.com/c-variables-primitive-nonprimitive-types/

Tuples and Vectors

The difference between tuple and vector definitions is the context. It is often called a tuple when dealing with set theory and a vector in mathematics. But in theory, they are representing the same core concept. *“Mathematicians usually write tuples by listing the elements within parentheses "( )" and separated by commas”* and similarly *“a vector in* $\mathbb{R}^n$ *can be specified using an ordered set of components, enclosed in either parentheses or angle brackets.”*

In **Lento**, a tuple or vector is a fixed-size data structure passed by value and located on the stack, while a vector in Lento is a cons-list (linked list) on the heap.
An interesting use case for tuples may be in functions returning multiple values; see the example below.

```
enum Status = Succeeded | Failed
myFunc : int, int -> (int, Status)
myFunc  a  b  =  (a  +  b,  Status.Succeeded)  //This  may  be  an  error-prone
operation
```

Here, the type of each element in the return tuple is directly inferred from the function type signature. This could be inferred as well for variables, for example.

```
myVar = (10, Status.Succeeded) // Inferred type: (int, Status)
```

Or explicitly.

```
(int, Status) myVar = (10, Status.Succeeded) // Explicit
```

Lists

In **Lento**, a list is a variable-length data structure that can hold any number of elements of *possibly* different types. It is implemented as a linked list (or cons-list), and elements are typically allocated on the heap.

```
myList : [string] = ["Hello", "world"]
myList.push "!"
println myList.join(" ") // Hello world !
```

Maps

In theory, a map is a tuple with named elements.
A map could be created in many ways. Using implicit type inference for each element, or explicit, a type signature or not.
All keys in a map must be unique, and value types may vary.

```
myVar = (value: 10, status: Status.Succeeded)
// Inferred type (implicit): (a: int, b: Status)
```

Or explicitly.

```
myVar : (status: Status, value: int) // Explicit
myVar = (value: 10, status: Status.Succeeded)
```

User-defined types

Lento, like many other strongly typed languages, allows users to declare their types using rich, expressive notation.

```
type Day = Number in 1..31;
enum  Weekday  =  Monday  |  Tuesday  |  Wednesday  |  Thursday  |  Friday  |
Saturday | Sunday;
type Hours = Number in 0..24;
```

Type Inference

In Lento, type inference analyzes the types of expressions and variables to deduce the types of other expressions and variables automatically. This eliminates the need for explicit type annotations in many cases, making the code more concise while maintaining strong type safety.

Basic Inference

```
x = 42  // x is inferred to be of type Int
y = x + 1  // y is also inferred to be of type Int
```

Implicit Function Return Types

```
add(a, b) = a + b  // Return type is inferred based on a and b
```

Conditional Statements

```
z = if x > 0 then x else -x  // z is inferred to be of type Int
```

Collections

```
arr = [1, 2, 3]  // arr is inferred to be of type List[Int]
```

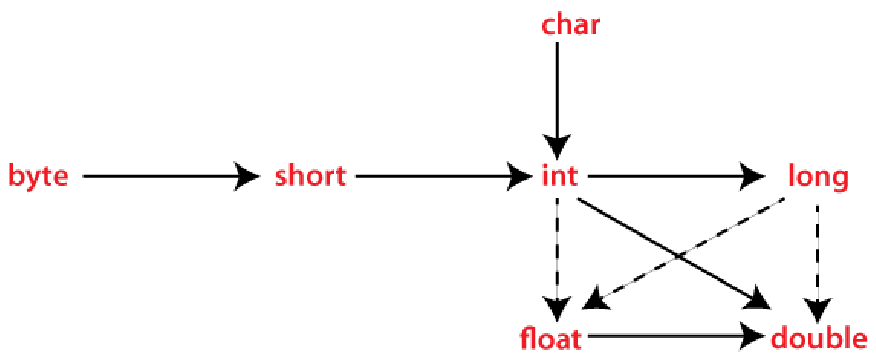
Coercion with Explicit Types

```
a: Float = x
// a is explicitly typed as Float, x (Int) is coerced to Float
```

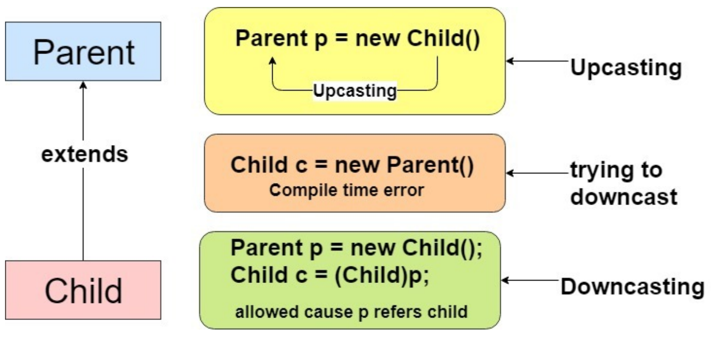
In cases where type inference encounters ambiguity or conflicting types, it typically raises a compile-time **type error**, prompting the developer to provide explicit type annotations to resolve the conflict. By leveraging type inference, Lento aims to balance simplicity, conciseness, and type safety, reducing boilerplate while ensuring robustness.

Type Conversion

Other names include type coercion, casting, conversion, or mapping. These serve as the canonical methods for *directionally* converting between types.


$$\begin{tikzcd}
 & & \text{char} \arrow[d] & \\
\text{byte} \arrow[r] & \text{short} \arrow[r] & \text{int} \arrow[r] \arrow[d, dashed] \arrow[dr] & \text{long} \arrow[d, dashed] \arrow[dl, dashed] \\
 & & \text{float} \arrow[r] & \text{double}
\end{tikzcd}$$

There are failable downcasting in Lento, these do not return a value directly, but an option with some value if it is successful. These concepts also exist in other languages like Java:



In Lento, this is denoted by wrapping the value in a call to the destination type. All values cannot be converted between any types. Thus, these coercions are statically checked. There is built-in intrinsic support for basic primitive types.

```
x: int = 42
y: float = x  // Implicit conversion from int to float
z = float(x)  // Explicit conversion from int to float
```

Type Safety

Examples

1. Type Conversion: Provide examples of implicit and explicit type conversions, if applicable.
2. Polymorphism: If Lento supports any form of polymorphism, describe how it works and provide examples.
3. Type Matching: Since Lento has pattern matching, show how types can be used in this context.
4. Error Handling: Demonstrate how the type system interacts with your error-handling mechanisms.

Macro System

Lento's powerful hygienic macro system allows for advanced code transformation rules, enabling users to write more expressive and concise programs. It is a feature for vastly extending the language's capabilities. (DSLs, Infix operators, keywords, etc...)

Hygiene

This ensures the prevention of variable capture and naming conflicts in macros behave consistently and predictably.

Phases

Lento's macro system operates during the post-process phase. The entire file is parsed into an Abstract Syntax Tree (AST) before any macro transformations occur. This approach offers a unified view of the code, aiding in more accurate transformations.

Syntax

Macros in Lento have a specific syntax for both definition and invocation. This syntax is designed to be expressive and unambiguous, facilitating ease of use and minimizing potential errors.

Extensibility

One of the most powerful features of Lento's macro system is its extensibility. Users can define new infix operators, keywords, and other syntactic constructs, tailoring the language to specific needs or domains.

Examples

To aid comprehension and practical usage, the specification will include concrete examples demonstrating the versatility and power of macros in Lento. These examples will cover common use cases and advanced scenarios.

Limitations

While macros offer extensive capabilities, they have limitations, such as potential complexity and debugging challenges. Users should be aware of these caveats to employ macros judiciously.

Best Practices

The specification will provide guidelines on best practices for macro usage. These recommendations aim to help users utilize macros in a manner that enhances code quality and maintainability.

Modules

Namespaces

Everything is located in the root namespace

Modules

Module packages must have unique identifying module names. There could be multiple module declarations in the same file.

```
mod my_first_module {
    // Declare the following code as a module
    a = 1
}

mod my_second_module {
    // Declare the following code as a module
    b = 2
}
```

Two packages can share the same namespace but with unique module names. I.e., `import Core.Generics.List` and `import Core.Generics.HashMap`. These would share the `Core.Generics` namespace but import different classes from the `Generics` module.

Garbage Collector

Because Lento aims to be as pure functional as possible, meaning that we don't need references. Objects are sent as values, and methods are static functions taking this struct, and returning a new modification.

The purpose of a garbage collector in most other object-oriented languages is mainly to clean up any memory with objects without any references to it. This is not the case for Lento; on the other hand, we might want to add any behavior to Lento that moves away from the fully pure functional paradigm a bit, meaning this might be a feature added anyways to support a certain syntax or control flow in the future.

Supervisor aware

This means that each data structure stored in memory occupies a section starting from a certain address. This system uses an 8-bit header that specifies how many references that particular object has. When a supervisor removes its reference to that part of memory, the value is decreased and a small GC adds that part to the free-memory block list or gives it back to the operating system.
For supervisors to know when they should stop referencing other parts of memory, we use the Dispose functions. This enables us to tell the runtime/system when we are done with different parts of the memory. These dispose function calls are abstracted away from the regular user and are handled by the compiler to generate code that frees the memory location of a, e.g. re-assigned variable value.
This method is used instead of the mark stage in a regular mark-and-sweep garbage collector.

Sweep stage

Because we use a top-down approach instead of a backtrack reference lookup approach. The sweep stage becomes a lot simpler, only requiring us to add parts of the memory that have no current supervisors to the free memory list, meaning nothing in the program is referencing that object anymore and has been disposed of. This can, therefore run concurrently with the main thread and never interrupt the execution to stop for garbage collection suddenly. Instead, the main thread marks the parts in memory when a supervisor frees its reference. If the total number of supervisors for a memory block is zero, the garbage collector could either directly add it to the free memory list or make a single pass over the whole memory later, collecting every free memory section.

When to dispose?

There are several suitable occasions to dispose of memory references, most listed below.

- Variable reassignment
  When a variable is reassigned, the previous value is no longer used.

- Function return
  After a function returns, the local scoped variables will no longer be used.

Program exit

After the main thread of the program process finishes, all references to any memory are released.

The standard library

This section describes the standard library, which is a collection of pre-defined functions and modules that are available to all programs written in the language. Several different modules provide different kinds of features, such as system, std, core, and ffi-logic.

System

Contains modules with functions for interacting with the local operating system. Example modules would be:

- Memory
- Processes
- Threads
- File system I/O
- Network I/O, Sockets
- Debugging
- Environment variables

Standard Library

The standard library serves as the cornerstone of cross-platform Lento applications, offering a minimal yet robust set of shared abstractions for the wider Lento ecosystem. It provides core types, along with library-defined operations on language primitives, standard macros, I/O handling, concurrent processing, and multithreading capabilities, among additional features. The standard library is available to all Lento programs by default.

- String formatting
- Serialization/Deserialization
    - JSON, XML, HTML
- Command-line arguments parsing
- SSL/TLS/HTTPS protocols
- Console

Core Library

The Lento Core Library is the foundational layer of the Lento Standard Library, free from external dependencies. It serves as the essential bridge between the Lento language and its libraries, defining the basic building blocks of all Lento code. This library is self-contained, requiring no external, system-specific, or C-standard libraries.

The Core Library includes kernel functions that underlie standard built-in operations. For example, the addition operator "+" is implemented as a function that calls the core "add" function. The library is also commonly used when building macros. Despite its minimal scope and *lack of features* like heap allocation, concurrency, and I/O, it remains **platform-agnostic**.

- Iterators
- Generators
- Streams
- Regular expressions

- Math
- Cryptography
- Date and Time
- Compression
- Reflection and meta-programming

FFI

This module exposes low-level C-ABI function interoperability and adds support for loading dynamically linked libraries. This enables the use of libraries compiled and built using other programming languages and extends the domain of available libraries for Lento developers. The FFI module provides:

- **Dynamic Library Loading**: Support for dynamically loading shared libraries (.dll, .so, .dylib) at runtime.
- **C-ABI Compatibility**: Ensure compatibility with the C Application Binary Interface (ABI) for seamless integration with C libraries and other languages that can interface with C.
- **Function Binding**: Facilities for binding to external functions, specifying their signatures and calling conventions.
- **Type Mapping**: Automatic and custom mapping between Lento types and foreign function types, including primitives and complex structures.
- **Memory Management**: Utilities for allocating and deallocating memory in the foreign language, with hooks for Lento's garbage collector.
- **Error Handling**: Robust error handling mechanisms for dealing with failures in foreign code, including exceptions and return codes.
- **Thread Safety**: Features to ensure thread-safe calls to foreign functions, aligned with Lento's concurrency model.
- **Data Marshalling**: Utilities for marshaling complex data types like arrays, structs, and unions between Lento and foreign code.
- **Inline Assembly**: Support for embedding inline assembly code for performance-critical sections, if applicable.
- **Safety Checks**: Compile-time and runtime checks to ensure type safety and adherence to calling conventions.
- **Performance Tuning**: Tools for profiling and optimizing the performance of FFI calls, including caching and lazy loading.

Configuration

All projects may contain a single config file specifying information, scripts, pre-process and post-process stage execution.

Execution

Programs are executed by either being interpreted or compiled. If a compiler should compiler a program, it must first look for an entry point. Unless this is found, the whole program is executed from top to bottom like a script.

Entry point

The entry point to a program is a function with the `@EntryPoint` decorator. There may only be one entry point to a program. Otherwise, it will be executed as a script running from the top of the source file to the end. All other functions are seen as subroutines.
Read more about entry points.

Entry points are used in this way by other languages like F#.

https://dotnet.microsoft.com/learn/languages/fsharp-hello-world-tutorial/create

File extension

Use `.LT`, or `.lt`, denotes a Lento source file and executes everything line by line unless an entry point has been explicitly specified in the project configuration file.

The Interpreter

The Lento interpreter will be an official, open-source standard interpreter and interactive REPL environment for the language.

The Compiler

The Lento compiler will be an official, open-source standard compiler for the language. It shall be available under a subcommand of the regular `lt` CLI tool.
An excellent addition would be to support the compilation of DLL files, and exporting functions for use in other languages or by the system.

Backends

Platform	**Target**	**Method**	**Technology**
Cross Platform	-	Interpreter	Rust
Native Code	PE *(.exe)*, ELF, DLL	Compiler	Rust, Cranelift, LLVM?
Web *(browser)*	JS, Wasm	Compiler	Rust, Cranelift
Third Party *(unofficial)*	.NET Runtime, JVM	Compiler	Rust, Backend Plugin

The Package Manager

Lento package manager. Provides the service to download and publish code as packages that can be reused. All code should be published on a Github, Gitlab, or self-hosted repository, meaning you could upload a package to your website. Packages must be zipped or single Lento files (that may include other files in the same package or depend on other packages).
Packages will not be controlled by the manager but only redistributing existing ones hosted by other services.
The package manager holds a register of known packages. New packages are added with just a link, title, and description, which adds a new row to the registry.
Users should be able to upvote, downvote, and comment on the package manager website, and warnings should be shown when installing new packages.

Inspired by the Go, Nim, and Cargo package manager.

A nice touch would be to add support for executing external files from the internet just by providing a URI link instead of a file location. The same goes for compiling files, this would make building *installers* redundant.

Tooling

The Lento language ecosystem is designed to provide a comprehensive suite of tools and services to facilitate the development process. This includes various features ranging from package management platforms to online playgrounds.

Type	Description	Priority
Package management platform, LPM	Like NPM, Hex, NuGet	3
LSP, IntelliSense, completions	Language Server	5
Linter	Per-editor extension	4
Syntax Highlighting	Per-editor extension	4
Code completion and snippets	Per-editor extension	7
Online playground	Inspired by TypeScript	2

Ecosystem

The official LPM package manager platform aims to centralize shared modules and features in the community. This could, for example, be:

- Domain Specific Languages embedded with a runtime and compile-time macros. Piggybacking on the capabilities of Lento as a toolchain and platform.

Main Points

There is a unique power of utilizing a high-level language combined with a robust package manager. This combination results in a highly flexible and efficient development environment that can significantly streamline the software development process.

1. The strength of a high-level language is its ability to abstract complex low-level operations, allowing developers to focus more on the business logic and functionality of their applications, rather than getting bogged down in the details of memory management and hardware interactions. High-level languages are also typically easier to read and write, which can lead to increased productivity and fewer errors.
2. Package managers automate installing, upgrading, configuring, and removing software packages. This can save developers a significant amount of time and effort and can also help to ensure that the software environment is consistent and reliable.
3. The Lento language ecosystem takes this a step further by enabling cross-platform development when only depending on *libc (Linux)* or *native dlls (Windows)* from the operating systems. All other libraries are native Lento and combined pre-compile-time in one static bundle. This eliminates the need for linking, which can be complex and

error-prone. Instead, all dependencies are resolved at compile time, which can lead to more reliable and efficient code. Applications can be easily moved between different operating systems without extensive modifications, saving developers significant time and effort and making it easier to distribute software to a broader audience, not ending up in DLL Hell.

In conclusion, combining a high-level language like Lento with a robust package manager provides a powerful toolset for software development. It allows for greater **productivity**, **reliability**, and **portability** and can significantly **streamline** development.

Embedded Runtime

The Lento toolchain also offers a unique feature in the form of easy and accessible SDKs and API bindings to the Lento **core module** runtime and interpreter, which can be easily **embedded and integrated** into other applications. This feature provides a seamless interface for developers to use Lento as a scripting language tailored to their specific application domain *(similar to how Lua is used in various applications, from game development to embedded systems)*.

All the necessary Lento dependencies, libraries, and modules can be statically embedded with the runtime and thus included in the application **at compile time**. This can lead to more reliable and efficient applications, as there is no need to worry about missing or incompatible libraries at runtime.

Compiler Plug-ins

- Parser plug-ins? Or macros instead?
- Optimization passes plug-ins
- Backend plug-ins

Language Comparison

Lento offers meticulous design, expressiveness, and robust features that set it apart from other languages. Its unique benefits leap towards a more intuitive, robust, and empowering coding experience. By choosing Lento, developers become part of a vision for *a refined programming paradigm*. Below is a comparative table featuring Lento alongside other notable programming languages based on various aspects.

Aspect	**Lento**	**Python**	**Java**	**C#**	**Haskell**	**Rust**
Type System[1]	Static	Dynamic	Static	Static	Static	Static
	Strong	Moderate	Moderate	Strong	Strong	Strong
	Inferred	Inferred	Manifest	Manifest	Inferred	Inferred
Type Inference[2]	Yes	N/A	Limited	Yes	Yes	Yes
Cross-Platform	Yes	Yes	Yes	Yes	Yes	Yes
Multi-Paradigm[3]	Moderate	Moderate	Moderate	Yes	Yes	Moderate
Operator Overloading	Yes	Yes	No	Yes	Yes	Yes
Lazy Evaluation	Yes	No	No	No	Yes	No
Macro System	Yes	No	No	No	No[4]	Yes
Error Handling	Explicit Types	Exception	Exception	Exception	Explicit Types	Explicit Types
Zero-Cost Abstractions	Yes	No	No	No	Yes	Yes
Generic Data Types[5]	Yes	Yes	Yes	No	Yes	Yes
Performance[6]	Moderate	Moderate	Moderate	High	High	High

This table highlights the unique features of Lento in comparison to other popular programming languages. *Each language has its strengths*, and the choice among them would depend on the specific requirements of your project.

[1] https://en.wikipedia.org/wiki/Strong_and_weak_typing
[2] https://en.wikipedia.org/wiki/Type_inference
[3] https://en.wikipedia.org/wiki/Programming_paradigm and https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages
[4] https://chrisdone.com/posts/haskell-doesnt-have-macros/
[5] https://en.wikipedia.org/wiki/Algebraic_data_type
[6] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html and https://niklas-heer.github.io/speed-comparison/

Naming Conventions

A well-defined set of naming conventions can significantly improve the readability and maintainability of code written in a language. Here's a table outlining naming conventions for different aspects of Lento:

Type of Value	**Naming Convention**	**Example**
Classes	PascalCase	MyClass
Functions	camelCase	myFunction
Variables	camelCase	myVariable
Constants	UPPER_SNAKE_CASE	MY_CONSTANT
Enums	PascalCase	TestSuccess, Some
Protocols	PascalCase	MyProtocol
Types *(non-primitive)*	PascalCase	MyType
Modules	PascalCase	MyModule
Package Names	kebab-case	my-package
Private Fields	_camelCase	_myField

Read more about conventions:
https://en.wikipedia.org/wiki/Naming_convention_(programming)

Branding

The current chapter focuses on product branding, including aspects such as logo design, color schemes, and packaging.

Background

When picking a logo, we thought about the underlying meaning of the word *"Lento."* It translates to *slow*, *careful*, and *gentle*. It is often used as **an expression in music**. Hence, we picked the **majestic Barn owl**, as it reflects on the **delicateness** of birds, not to mention **wise**, **skillful**, and **careful**. Being one of the most silent-flying birds.

Logotypes

Font: Numans
Primary colors: #2F292C and #FFFFFF

Main	Italic text	Only owl
Lento	*Lento*	

Mockup Code

Hello, World!

```
println "Hello, world!"
```

Or the slightly longer version using a main function as an entry point.

```
main : [string] -> ()
main (args) {
    println "Hello, world!"
}
```

Execute through the command line:

```
> lt hello_world.lt
Hello, world!
```

Pies and Books

```
type Pie = float in [0, 1]
type Book = int in [0, 2000]
Pie p = 56,7
Book b = 800
2.5 p + 5 b // Error: Cannot add ‘Pie’ and ‘Book’
```

License

The official implementation is open source and licensed under the MIT software license.