Lumi Language Guide

Release lumi-0.5

lumi

Nov 07, 2022

INSTALLATION AND USAGE

1 A Work in Progress… **3**

2 Contributing to Lumi **5**

2.1 Quick Start - Hello World Example 5
2.2 Installing and Building Lumi 6
2.3 Using the `lumi` Command 8
2.4 Using Lumi Compiler Directly 11
2.5 Lumi Language Goals and Features 13
2.6 Memory Management 14
2.7 Thread Safety 16
2.8 Integer Ranges and Overflow Prevention 16
2.9 Type System 19
2.10 General Syntax Highlights 23
2.11 Basic Syntax 23
2.12 Built-in Types 25
2.13 Variables and Constants 34
2.14 Functions 37
2.15 User Defined Types 40
2.16 Control Flow 48
2.17 Testing 50
2.18 Interacting with Other Languages 52
2.19 Standard Library 56
2.20 Code Examples 57
2.21 Serialization 59
2.22 Asynchronous IO 59

Index **61**

Welcome to **Lumi** programming language guide!

Lumi is a general purpose programming language that is good for all needs, specially for long-term projects, or where resource efficiency is needed.

Lumi aims to be safe, efficient, flexible, easy to write, and easy to maintain. See *Lumi Language Goals and Features* for more details.

Lumi files use `.lm` extension.

"Lumi" name was chosen because:

- "Lumi" is an abbreviation for "Illuminating", and Lumi language aims to cast new light to the programming world. (it's also the meaning of my first name)
- "Lumi" means "snow" in Finnish, which is bright, light, handy, flexible, strong and fun - such as the Lumi language. (and also - I like snow)
- "Lumi" is short, easy to pronounce, and fun to say.

The Lumi image is Designed by kjpargeter / Freepik

CHAPTER

ONE

A WORK IN PROGRESS…

Lumi is still under initial planning and building. Many features are already implemented and it is possible to write complex programs with Lumi, but some key elements are not, and many implemented feature may change dramatically.

The latest state of the language is named “*Temporary Lumi 5*”, or “TL5” in short, to emphasizing the current temporary state of the language.

CHAPTER

TWO

CONTRIBUTING TO LUMI

Any help, suggestion, comment or questions is welcome! See Lumi repository wiki for more details on how to contribute to Lumi.

2.1 Quick Start - Hello World Example

This is a quick guide to install, compile and run Lumi "hello-world" example in Linux.

2.1.1 Quick Installation

Clone Lumi repository (`git clone https://github.com/meircif/lumi-lang.git`).

Enter the repository root directory: `cd lumi-lang`.

Install it: `make install`.

2.1.2 Hello World Program

Listing 1: hello-world.5.lm

```
module hello-world

func ! show()
    sys.println(user "hello world")!

main!
    show()!
```

Compile it: `lumi docs/hello-world.5.lm`.

Run it:

```
>>> docs/hello-world
hello world
```

2.1.3 Hello World Test

Listing 2: hello-world-test.5.lm

```
module hello-world-test

var Bool println-raise
var String printed-text

mock ! sys.println(user Buffer text)
    if println-raise
        raise! "error in println"
    printed-text.concat(user text)!

test show-hello-world-test()
    println-raise := false
    printed-text.clear()
    hello-world.show()!
    assert! printed-text.equal(user "hello world")

test show-raise-test()
    println-raise := true
    assert-error! hello-world.show(), "error in println"
```

Compile it: `lumi -t hello-world docs/hello-world-test.5.lm docs/hello-world.5.lm`.

Run it:

```
>>> docs/hello-world-test
Running tests:
testing show-hello-world-test... OK
testing show-raise-test... OK
testing code coverage... 100%
Tests passed
```

2.2 Installing and Building Lumi

2.2.1 Syntax Highlighting

For Lumi syntax highlighting it's recommended to use one of:

- Visual Studio Code editor with Lumi Language extension installed
- Atom editor with language-lumi package installed

2.2.2 Installing Lumi

Note: In all shell examples below `$CC` assumed to hold the C compiler, and the current directory assumed to be on Lumi repository root directory

1. clone or download Lumi repository: `git clone https://github.com/meircif/lumi-lang.git`
2. run `make install`

or install manually:

3. *build the latest Lumi compiler* with a C compiler: `$CC TL5/tl5-compiler.c TL4/lumi.4.c -ITL4 -o tl5-compiler`
4. *build the "lumi" command* with a C compiler: `$CC lumi-command/lumi.c TL4/lumi.4.c -ITL4 -o lumi`
5. add the Lumi compiler and the `lumi` command to the system path (for example: `sudo install lumi tl5-compiler /usr/local/bin/`)

2.2.3 Building the Lumi Compiler

A Lumi compiler must first be built using a C compiler. That Lumi compiler can then be used to generate C code from Lumi code.

It is recommended to add the compiler executable to the system path, for example, in Linux move it to `/usr/local/bin/`.

Latest Version - TL5 Compiler

```
make tl5-compiler
# or
$CC TL5/tl5-compiler.c TL4/lumi.4.c -ITL4 -o tl5-compiler
```

Old Versions

```
## TL4 compiler
make tl4-compiler
# or
$CC TL4/tl4-compiler.c TL3/lumi.3.c -ITL3 -ITL4 -I. -o tl4-compiler

## TL3 compiler
make tl3-compiler
# or
$CC TL3/tl3-compiler.c TL2/lumi.2.c -ITL2 -o tl3-compiler

## TL2 compiler
make tl2-compiler
# or
$CC TL2/tl2-compiler.c TL1/lumi.1.c -ITL1 -o tl2-compiler
```

(continues on next page)

(continued from previous page)

```
## TL1 compiler
make tl1-compiler
# or
$CC TL1/tl1-compiler.c TL0/tl0-file.c TL0/tl0-string.c -ITL0 -o tl1-compiler

## TL0 compiler
make tl0-compiler
# or
$CC TL0/tl0-compiler.c TL0/tl0-file.c TL0/tl0-string.c -o tl0-compiler
```

2.2.4 Building the `lumi` Command

The `lumi` command must first be built using a C compiler. `lumi` command can then be *used* to compile an executable directly from Lumi code by running Lumi compiler and C compiler one after another.

```
make lumi
# or
$CC lumi-command/lumi.c TL4/lumi.4.c -ITL4 -o lumi
```

It is recommended to add the compiler executable to the system path, for example, in Linux move it to `/usr/local/bin/`.

2.3 Using the `lumi` Command

The `lumi` command is a tool that helps compile, test, and run Lumi code. For example, `lumi` command can be *used* to build an executable directly from Lumi code by simply running `lumi hello.5.lm`.

The `lumi` command:

- assumes the used Lumi compilers are already *built* and are in the system path
- uses the `CC` environment variable to determine the C compiler command, using `gcc` if not exists
- supports all Lumi version from TL0 to *TL5*

2.3.1 Command Help

Running `lumi -h` or `lumi --help` will print help:

```
>>> lumi -h
Usage: lumi [options] file...
Options:
  -h/--help        print this help
  --version        print lumi command version
  -o <file>        output file name
  -t <module>      compile test program for <module>
  -l <module>      compile shared library exporting <module>
  -c               only create C file(s)
  -TL<version>     only run C compiler for TL<version>
  -e <argument>    extra argument for C compilation
```

(continues on next page)

(continued from previous page)

```
  -p <lumipath>    path of lumi-lang repository
  -r               run the compiled program
  -ra <arguments>  run the compiled program with given arguments
  -v/--verbose     print executed commands
  -d/--debug       only print commands without execution
```

2.3.2 Usage

The basic usage of `lumi` command is to take one or more Lumi files and create a single executable from them. For example:

```
lumi hello.5.lm
```

will create a `hello` named executable compiled from `hello.5.lm`, generating a `hello.c` C file in the process. This is done by running Lumi compiler and C compiler one after another.

If multiple Lumi files are given, the generated name will be based on the first input file.

`lumi` command detects the TL version based on the input file extension `.[TL version].lm` and runs the respective Lumi compiler.

2.3.3 Specifying an Explicit Output File Name

The output file name can be explicitly defined with `-o <output file name>`. For example:

```
lumi hello.5.lm -o output
```

will generate `output` named executable, and `output.c` named C file in the process.

2.3.4 Compiling Tests

Lumi compiler allows generating *testing* code for a specific Lumi module. This feature can be used in `lumi` command with `-t <tested module name>`. For example:

```
lumi -t hello hello-tests.5.lm hello.5.lm
```

will generate `hello-tests` executable that tests the `hello` module.

Running a Lumi test executable with `-xml` argument will also generate a `cobertura.xml` named file with code coverage XML report in cobertura scheme.

2.3.5 Compiling a Shared Library

Lumi compiler allows generating code for a shared library *exporting C styled functions* from a specific Lumi module. This feature can be used in `lumi` command with `-l <exported module name>`, and only functions from the given module will be exported to the shared library. For example:

```
lumi -l hello hello.5.lm
```

will generate `libhello.so` shared library that exports functions from the `hello` module.

2.3.6 Only Running Lumi Compiler

To only run the Lumi compiler `-c` flag can be used. For example:

```
lumi -c hello.5.lm
```

will only generate `hello.c` C file.

2.3.7 Only Running C Compiler

To only run the C compiler `-TL<TL version>` flag can be used. The TL version number must be given as it cannot be detected from the input C file name. For example:

```
lumi -TL5 hello.c
```

will only generate `hello` executable, assuming `hello.c` was generated by TL5.

2.3.8 Extra C arguments

To add extra arguments to the C compilation `-e` can be used. For example:

```
lumi hello.5.lm -e external.c
```

will add `external.c` as an input to the C compiler, while ignoring it in the Lumi compilation. This is mainly needed when *external C code is called from Lumi*.

2.3.9 Running the Generated Executable

The generated executable can also be run using `-r`. For example:

```
lumi -r hello.5.lm
```

will generate `hello` executable and then run it.

It is possible to also send arguments to the executable using `-ra <arguments>`. For example:

```
lumi -r hello.5.lm -ra 'first-arg "second arg"'
```

Will run `hello first-arg "second arg"`.

2.3.10 Verbose and Debug

Adding `-v` or `--verbose` option will also print the executed commands.

Adding `-d` or `--debug` option will only print the commands without execution.

2.3.11 Old Version Limitations

- TL4 and below assumes *LUMIPATH* is correctly configured
- multiple input Lumi files are not supported in TL0 and TL1
- implicit output name is determined by the last file in TL2, and not the first
- TL2 and TL3 generate multiple C files - one C file for each input Lumi file, this also meas that an explicit output name for C files is not supported
- testing is only supported in TL4 and above

LUMIPATH

For C linking purposes in TL4 and below `lumi` command needs to know the local Lumi repository root directory path. This can be configured by one of:

1. running `lumi` command inside the Lumi repository root directory
2. setting the value of `LUMIPATH` environment variable to the path
3. running `lumi` with flag `-p <path>` (this will override `LUMIPATH` environment variable)

Path Separator

The default path separator in `lumi` command is `/`. In systems where the path separator is \ instead (such as Windows), the *LUMIPATH* must end with a \ character. Doing this will set the path separator in `lumi` command to \.

2.4 Using Lumi Compiler Directly

Using the lumi Command is recommended to compile Lumi code. Using the Lumi compiler directly is possible, but it is less convenient.

2.4.1 Latest Version - TL5 Compiler

The *TL5* compiler generates a single C file based on one or more Lumi files. For example

```
tl5-compiler hello.c hello.5.lm other.5.lm
```

will generate `hello.c` C file from `hello.5.lm` and `other.5.lm` Lumi files.

2.4.2 Compiling Testing Code

To generate C code that tests a specific Lumi module `-t <tested module name>` should be used. For example:

```
tl5-compiler -t hello hello-tests.c hello.5.lm hello-tests.5.lm
```

will generate `hello-tests.c` C file from `hello.5.lm` and `hello-tests.5.lm` Lumi files, with C code that tests the `hello` module.

2.4.3 Compiling Shared Library Code

To generate C code that can be compiled to a shared library *exporting C styled functions* from a specific Lumi module `-l <exported module name>` should be used. For example:

```
tl5-compiler -l hello hello-lib.c hello.5.lm other.5.lm
```

will generate `hello-lib.c` C file from `hello.5.lm` and `other.5.lm` Lumi files, with C code that can be compiled to a shared library exporting functions from the `hello` module.

2.4.4 Building an Executable

The generated C file can be compiled on its own to an executable using a C compiler. For example:

```
$CC hello.c -o hello
```

will generate `hello` executable.

2.4.5 Old Versions

```
## TL4
# compiling hello.4.lm to hello.c
tl4-compiler hello.c hello.4.lm other.4.lm
# compiling tests
tl4-compiler -t hello hello-tests.c hello.4.lm hello-tests.4.lm
# compiling the C code as an executable
$CC hello.c TL4/lumi.4.c -ITL4 -o hello

## TL3
# compiling hello.3.lm to hello.c
tl3-compiler hello.3.lm
# compiling the C code as an executable
$CC hello.c TL3/lumi.3.c -I. -ITL3 -o hello

## TL2
# compiling hello.2.lm to hello.c
tl2-compiler hello.2.lm
# compiling the C code as an executable
$CC hello.c TL2/lumi.2.c -ITL2 -o hello

## TL1
# compiling hello.1.lm to hello.c
tl1-compiler hello.1.lm hello.c
# compiling the C code as an executable
$CC hello.c TL1/lumi.1.c -ITL1 -o hello

## TL0
# compiling hello.0.lm to hello.c
tl0-compiler hello.0.lm hello.c
# compiling the C code as an executable
$CC hello.c TL0/tl0-file.c TL0/tl0-string.c -ITL0 -o hello
```

2.5 Lumi Language Goals and Features

2.5.1 Prioritized Goals

1. **Enforce safe and reliable code**, free from:
    - illegal memory access
    - memory corruption
    - memory leaks
    - integer overflow
    - unexpected crashes
    - etc…
2. **Allow writing of efficient code**, suitable for real-time embedded:

    > Modern programming languages solve goal #1 by making everything dynamically allocated and garbage collected, but this is inefficient and unpredictable. Lumi will allow as much efficiency and freedom as possible, as long as goal #1 is not harmed.

3. Be **flexible**, **easy to write**, and **easy to maintain**:

    This goal binds together 3 different goals:

    - **Flexible** - whenever possible Lumi should allow choosing from a variety of options
    - **Easy to write** - Lumi code should be easy to learn and writing code should be as efficient as possible
    - **Easy to maintain** - It should be easy to find and fix bugs and add new features, even in large and complex projects

    The above goals are bound together because no one is prioritized above the other and they sometimes contradict. Lumi will do its best to reach all of them with minimum harm to each one.

2.5.2 Features

- **Code generating**: Lumi compiler will initially generate **C code**. Other "modern machine language" codes may also be generated, such as: Java (for Android and other devices), Objective-C/Swift (for Apple devices), JavaScript/WebAssembly (for web application), and more…
- Safe *Memory Management* - allowing easy trade-off between flexibility and performance by the user
- *Thread Safety* - Lumi code is ensured to be thread safe
- *Integer Ranges and Overflow Prevention* - Lumi code is ensured to be free from integer overflow and underflow
- *Type System* - Lumi is **strongly typed**, and allows variety of typing styles with different trade-offs between simplicity, generality and performance
- Exception-like error handling implemented with return values
- Strict coding conventions - all Lumi code should look the same
- Built-in support for productivity features:
    - testing and mocking
    - documentation
    - profiling

2.6 Memory Management

Memory should be managed correctly to reach *Lumi goal* #1, to reach *Lumi goal* #2 - it should be done as efficiently as possible, and to reach *Lumi goal* #3 - it should be flexible.

This is achieved by combining 3 forms of managing - allowing easy trade-off between flexibility and performance by the user:

1. *No Performance Overhead - compile time only reference managing*
2. *More Flexible Reference Managing - with a small performance cost*
3. *Maximum Flexibility - but with performance issues*

Note: Below is a non-final suggestion to implement this, it will probably be developed further over time. See *Variables and Constants* section for the currently implemented syntax.

2.6.1 No Performance Overhead - compile time only reference managing

Lumi will allow performance free reference managing that will be done only in compile time.

Every objects has a single "owner" entity - which is another object or a stack block. When an owner is destroyed it automatically destroys the referenced object, unless the ownership was moved to another entity. Stack and global variables are treated as owners - but they cannot move their ownership.

This has some similarities to the memory management in Rust.

Owners can give the reference to multiple temporary "user" entities. Users are free to read and modify the referenced object - but cannot destroy it or modify its sub-owners. Users are temporary because they cannot be used after any object of their type is destroyed, as the compiler cannot guarantee they are pointing to a legal object any more.

Owners can "borrow" the reference to a single temporary owner that automatically returns the ownership back to the original owner at the end of its code block. While borrowing the original owner cannot be used. The temporary owner has full control over the reference, except the ability to destroy it.

This is *currently implemented*, but not fully optimized. In the future the syntax may be slightly different and look like this:

```
owner String some-string(String{16}())  ; new owner reference
user-func(user some-string)  ; give reference to a user
borrowing-func(temp some-string)  ; borrow ownership to a temporary owner
owning-func(owner some-string)  ; move ownership, cannot be used anymore
```

2.6.2 More Flexible Reference Managing - with a small performance cost

Lumi will allow more complex and flexible reference managing that come with a small and predictable performance cost.

Same as *No Performance Overhead - compile time only reference managing* with the addition of weak references. To allow this the owner should be declared as "strong". It will work the same way as a regular owner, plus that it can now give "weak" references to any other entity without limitations. Weak references will automatically test that the reference is still valid before accessing it.

There are several ways to implement this - but all need some extra memory to manage the weak references, and some extra processing time to check if the weak reference is valid. In all implementations the extra overhead is small and predictable.

This is *currently implemented*, but in the future the syntax may be slightly different and look like this:

```
strong String some-string(String{16}())  ; new strong owner reference
user-func(user some-string)  ; give reference to a user
borrowing-func(temp some-string)  ; borrow ownership to a temporary owner
weak-func(weak some-string)  ; give weak reference
owning-func(strong some-string)  ; move ownership
```

Note: Strong reference counting is currently not supported because it can cause memory leaks because of reference loops. It may be allowed in the future in cases where the compiler can ensure no reference loop is possible.

2.6.3 Maximum Flexibility - but with performance issues

Lumi will allow declaring a reference as garbage-collected, which will allow passing references freely without limitation. The memory will only be cleared when all "strong" references are destroyed. The garbage-collector must check and remove reference loops to avoid memory leaks.

To allow this a reference should be declared as "shared". This reference can then be passed to other "shared", "user" or "weak" references.

Implementing a garbage-collector has a significant and unpredictable performance cost, but some Lumi users may be willing to pay it in some sections of their project where performance is less important.

This is not implemented yet, but in the future the syntax may look like this:

```
shared String some-string(String{16}())  ; new shared reference
shared-func(shared some-string)  ; copy shared reference
user-func(user some-string)  ; give reference to a user
weak-func(weak some-string)  ; give weak reference
```

2.6.4 Conditional and Empty References

As default, (non-weak) references always point to a legal object. To allow empty references, the reference type must be declared as "conditional" using the ? sign. Empty value can be set using _ sign.

This is *currently implemented*, but not fully optimized. In the future the syntax may be slightly different and look like this:

```
user String? cond-str  ; initialized as empty
cond-str := some-string  ; now not empty
cond-str := _  ; now is empty again
if cond-str?  ; check if has value
    ; can be used safely here...
else
    ; here we know it's empty...
cond-str!.clear()  ; raise error if empty
func-with-cond(user _)  ; send empty to function
```

2.7 Thread Safety

Running threads should be safe to reach *Lumi goal* #1, therefore the compiler will enforce it.

This page explains a first suggestion to implement this.

2.7.1 The default approach - complete data isolation

The compiler takes care that each data is only accessible from a single thread - creating a complete isolation of data between threads. Global data will be duplicated for each thread (probably using "thread local" mechanism). This mean no data sharing is possible - ensuring thread safety.

This is the default approach on all data unless one of the thread data sharing mechanism described below is used.

2.7.2 Sharing data between threads

Constant and immutable data

Every compile-time constant or run-time immutable data can be safely shared between threads. The compiler will ensure no thread can modify these data.

Atomic operations

Based on the platform, atomic types will be provided, and all operations on an atomic item will be atomic.

Messaging

It will be possible to send messages between threads, the data on the message will be copied from one thread to another in a safe manner taken care by the compiler.

Built-in thread-safe data structures

Various built-in thread-safe data structures will be provided that can be used to share data between threads.

Automatic locks

It will be possible to mark data as thread-shared, and the compiler will automatically protect access to this data using various lock types.

2.8 Integer Ranges and Overflow Prevention

As part of *Lumi goal* #1 the compiler will enforce code that is without any integer overflow (or underflow).

2.8.1 Integer Ranges

In Lumi all *integers* are declared with explicit range `Int{minimum:maximum}` and the compiler enforces that it will always contain values inside this legal range.

For example, the compiler will not allow these assignments:

```
func compute(Int{0:20} x)->(Int{-10:10} y)
   y := 20  ; error - clearly out of range
   y := x  ; error - may be out of range
```

Assigning a constant or an integer with overlapped range are naturally legal, for example:

```
func compute(Int{-10:10} x)->(Int{-20:20} y)
   y := 10  ; legal - inside legal range
   y := x  ; legal - all legal values of x are overlapped by y range
```

2.8.2 Integer Arithmetic

The result of any arithmetic operation is an integer with a new range based on the operator. For example:

```
func compute(Int{6:12} x, Int{0:100} y)
    x + y  ; return range is Int{6:112}
    x - y  ; return range is Int{-12:94}
    x * y  ; return range is Int{0:1200}
    x div y  ; return range is Int{0:16}
    x mod y  ; return range is Int{0:11}
```

2.8.3 Compile Time Overflow Prevention

The compiler will not allow operation that may result in an overflow (or and underflow). For example:

```
func compute(Uint64 a, Uint64 b)
    a + b  ; error - potential overflow
    -a  ; error - potential underflow
```

2.8.4 Run-Time Overflow Checking

It is possible to check for an overflow in run-time using `!` or `?` together with one of `+ - *` operators. For example:

```
~~~ raises an error when overflow detected ~~~
func ! raising-compute(Uint64 x, Uint64 y)->(Uint64 z)
    z := x +! y
    z := x -! y
    z := x *! y

func handling-compute(Uint64 a, Uint64 b)->(Uint64 z)
    if-error z := x +? y
        z := 0
    if-error z := x -? y
```

(continues on next page)

(continued from previous page)

```
        z := 0
    if-error z := x *? y
        z := 0
```

2.8.5 Efficient Native Wraparound

Utilizing native overhead-free wraparound is supported by using `wraparound` keyword. The result of native wraparound is naturally limited only for the ranges that native wraparound is guaranteed to be supported: `Uint8 Uint16 Uint32 Uint64` (`Int{0:255} Int{0:65535} Int{0:4294967295} Int{0:18446744073709551615}`). For other ranges wraparound must be done manually using `((value - min) mod (max - min + 1)) + min` for example.

`wraparound` can be used as a single unary operator before assignment or together with one of `+=` `-=` `*=` assignment operators:

```
func compute(Uint32 x)->(Uint32 y)
    y := wraparound x + 1
    y := compute(wraparound x - 1)
    y wraparound+= 1
    y wraparound-= 1
    y wraparound*= 1
```

The result range of using `wraparound` as above is the same as the target assignee range (`Uint32` in the examples above). This means that the assigned range must always be one of `Uint8 Uint16 Uint32 Uint64`.

`wraparound` can also be used together with `+` `-` `*` operators:

```
func compute(Int{0:1000000} x, Uint64 y)->(Uint32 z)
    z := x wraparound+ y
    z := x wraparound- y
    z := x wraparound* y
```

The result range of using `wraparound` as above is the minimal from `Uint8 Uint16 Uint32 Uint64` that overlaps the left operand (`Uint32` from `x` in the examples above). This means that the left operand can be any unsigned range.

2.8.6 Clamping

Clamping allows shrinking an integer range to a smaller range `Int{min:max}` by converting any value larger than `max` to `max` and smaller than `min` to `min`. This can be done automatically using `clamp` keyword. Clamping is not overhead-free because the checking and converting must be done at run-time.

`clamp` can be used as a single unary operator before assignment or together with one of `+=` `-=` `*=` assignment operators:

```
func compute(Uint32 x)->(Uint32 y)
    y := clamp x + 1
    y := compute(clamp x - 1)
    y clamp+= 1
    y clamp-= 1
    y clamp*= 1
```

Using `clamp` as above will clamp the result to the range of the target assignee (`Uint32` in the examples above).

`clamp` can also be used together with `+ - *` operators:

```
func compute(Uint32 x, Uint64 y)->(Uint32 z)
    z := x clamp+ y
    z := x clamp- y
    z := x clamp* y
```

Using `clamp` as above will clamp the result to the range of the left operand (`Uint32` from `x` in the examples above).

On assignment it is possible to raise an error instead of clamping using `!` or `?` together with `clamp`. Whenever a value is too small or big for the assignee target range - instead of setting `min` or `max` an error is raised. For example:

```
~~~ raises an error when clamping ~~~
func ! raising-compute(Uint32 x)->(Uint32 y)
    y := clamp! x + 1
    y := raising-compute(clamp! x - 1)

func handling-compute(Uint32 x)->(Uint32 y)
    if-error y := clamp? x + 1
        y := 0
```

2.8.7 Sequences Index Integer Range

planned - not supported in TL5

It is planned to support a special range that is bound to a sequence and can only hold values that are legal indices to the sequence.

It may look like this:

```
func example(Array{Char} array)->(Char result)
    var Int{array} index
    result := array[index]  ; no need to check index at run-time
```

2.9 Type System

Lumi is **strongly typed**, which means that:

1. Each symbol is statically bound to one specific type and can only hold data from this type.
2. Type correctness is enforced in compile time by the compiler.

Lumi has a variety of *built-in primitive and complex types*, and allows adding user defined types in a variety of typing styles. All these allow writing code with different trade-off between simplicity, generality and performance, and adapting different programming paradigms.

2.9.1 Typing Styles of User Defined Types

All user defined types in Lumi are built based on two basic typing styles:

1. *Static Structures*
2. *Dynamic Interfaces*

These can be combined to build more complex typing styles:

3. *Extending Types*
4. *Classes - Binding Dynamic Interfaces and Static Structures*
5. *Parameterized Types*
6. *Embedding a Dynamic Reference in a Static Structures*
7. *Automatic Dynamic Interfaces*

2.9.2 Static Structures

Typing style for the *static structure syntax*.

This is the most basic, simple and efficient typing style. A static structures (or “structure” in short) is simply a record that groups together multiple variables of any kind under one type.

Structures may contain methods. Methods are functions with an implicit first parameter named `self` that is a reference to an instance of the type.

It is possible to declare constants and global variables inside a structure. They are not part of the structure record, but are like normal constants and global variables that are named under the type name-space.

2.9.3 Dynamic Interfaces

Typing style for the *dynamic interface syntax*.

Dynamic interfaces are the basics of dynamic dispatch in Lumi. A dynamic interface (or “dynamic” in short) groups together multiple dynamic members that will be implemented differently for multiple objectives. Dynamics can be implemented for a specific *structure*, or purely without any binding.

Dynamic members are usually methods, but variables and references of any type can also be dynamic members - where each implementation initializes them with a different constant value.

Dynamics are always used as references and cannot be allocated because they have no structure. A dynamic references can be set with any type that implements the dynamic.

“Implementing” a dynamic means implementing each dynamic method and initializing each dynamic variable with a constant value.

Dynamics standard method dispatch is dynamic. This means that when an implementing type is passed to a dynamic reference, any method called on the dynamic reference will use the implementing type implementation.

Dynamics may contain constants and global variables, they are not dynamic members and behave exactly as global members in *structures* - they are just global elements under the type name-space.

Dynamics cannot have static fields, but may contain static methods. They are also not dynamic members and behave exactly as methods in *structures* - they must be implemented directly in the dynamic, and their dispatch is static.

2.9.4 Extending Types

Types can be extended by adding functionality to some base types.

Structures can extend other structures. An extending structure contains all members from all base structures, plus its own members. An extending structure can override methods of a base structure, other members may not be overridden.

Structures method dispatch is **static**. This means that when an extending structure is passed to a base structure reference, any method called on the base reference will use the base structure implementation even if the extending structure overrode that method.

Dynamics can extend other dynamics. An extending dynamic contains all members from all base dynamics, plus its own members.

An extending structure may override any dynamic implementation of its base structure. Nevertheless, if an extending structure reference is passed to a base structure reference, and then the base structure reference is passed to a dynamic reference, any method call on the dynamic reference will use the base structure implementation because structure dispatch is static.

2.9.5 Classes - Binding Dynamic Interfaces and Static Structures

Typing style for the *class syntax*.

Sometimes binding together *static structures* and *dynamic interfaces* under a single type is useful, mainly to adapt the OOP (object oriented programming) paradigm. A type with this kind of binding is also known as a "class".

Classes may be ad-hock binds between already declared structures and dynamics, or declared as classes up-front in a type definition. Types declared as classes may have both static and dynamic members, and the compiler creates an implicit static structure and an implicit dynamic interface - each with its respected members. The compiler then creates the class as a bind between these both implicit types.

Classes may extended any number of structures, dynamics, and other classes. The extending class implicit structure extends all base structures and the implicit structures of all base classes. Similarly, the extending class implicit dynamic extends all explicit and implicit base dynamics.

Classes may also implement dynamics. Any implementation method of the dynamic is also dynamic in the class. As opposed to structures, if an extending class reference is passed to a base class reference, and then the base class reference is passed to a dynamic reference, any method call on the dynamic reference will use the the extending class implementation because class dispatch is also dynamic.

2.9.6 Parameterized Types

Typing style for the *parameterized type syntax*.

It is possible to declare types with parameters to avoid code duplication of generic types. Each parameter can be either **static** or **dynamic**.

Static Parameters

Static parameters are like templates - for each different usage of any static parameter a new type will be automatically generated. Static parameters can be type names, or constant values of a specific type.

Dynamic Parameters

Dynamic parameters represent a generic type and only accept type names.

The main advantage of dynamic parameters is that - as oppose to static parameters - different usage of it will **not** generated a new type.

The disadvantage is that dynamic parameters can only be used as abstract references, as the same code handles references of different types with unknown structure.

2.9.7 Embedding a Dynamic Reference in a Static Structures

Typing style for the *embedded dynamic reference syntax*.

For some memory optimization scenarios, it is better if a dynamic reference of a class will be implemented only with one C pointer, and the dynamic structure reference will be embedded inside the type static structure (as done in C++).

Lumi will support this, but the exact implementation is still under planning.

2.9.8 Automatic Dynamic Interfaces

This is an experimental typing style idea that will allow automatic creation and implementation of dynamic interfaces based on the actual usage of a reference.

For each reference typed as `Auto` the compiler will automatically create a dynamic interface based on the methods called on this reference. Any type that implements the same methods used by the reference can be assigned to it, and an implementation of the dynamic interface will be automatically created by the compiler. For example:

```
; a dynamic interface with "example" method will be created buy the compiler
; and used as the parameter actual type
func auto-example(user Auto automatically-typed-dynamic-reference)
    automatically-typed-dynamic-reference.example()

struct SomeStruct
    func some-method()

var SomeStruct some-item
; implementation to the automatically created dynamic interface will be
; created by the compiler that uses "SomeStruct.example" method as the
; implementation to the "example" dynamic method
auto-example(user some-item)
```

This feature is an experimental idea because it’s unclear whether it is a good idea, and there may be some edge cases that will make it hard to implement.

2.10 General Syntax Highlights

- blocks are declared with indentation (as in Python)
- strict white-spacing:
  - tabs are syntax errors
  - indentation is exactly 4 spaces
  - no spaces in line end
  - extra indentation for line breaking is exactly 8 spaces
  - exactly one space around operators
  - file must end with a single newline
  - in general, any whitespace in the syntax must be used exactly
- strict naming conventions:
  - the default for everything is `lowercase-only-with-hyphens` (a.k.a "kebab-case")
  - types are `FirstLetterUppercase` (a.k.a "CamelCase)
  - compile time constants are `UPPERCASE-ONLY-WITH-HYPHENS` (a.k.a "FAT-KEBAB-CASE")

2.10.1 TL[number] - Temporary Lumi Language

Lumi language development is done in an iterative style, where in each step a compiler is written to a temporary Lumi language - "TL" in short - which is a partial (or different) syntax of the final Lumi language.

These temporary Lumi languages are marked as "TL[number]" where "number is the iteration step number. "TL0" is the initial compiler temporary Lumi language, the next iteration is TL1 and so on…

2.10.2 Latest Version - TL5

The Lumi language is still a work in progress and the final syntax is not decided yet. The latest working compiler is for Temporary Lumi 5 (TL5) syntax, and this guide will describe it, and the differences between it and the planned final Lumi syntax.

The final Lumi syntax is still under planning, so this guide refers only to the current planning state of the final syntax. Changes will happily be made based on coding experience and suggestions.

2.11 Basic Syntax

2.11.1 Comments

In *TL5* Single line comments start with `;`, multi-line comments start with `[;`, and end with `;]`. Comments that are not in line start are not supported yet - but will be supported in the final syntax.

```
; single line comment
[; <-- multi-line comment start
multi
line
```

(continues on next page)

(continued from previous page)

```
comment
multi-line comment end --> ;]
var Uint32 x  ; not supported yet :(
```

Some suggest to change this in the final syntax to be as in C with `//`, `/*` and `*/`.

2.11.2 Documentation

Documentation have their own dedicated syntax: they start and end with `~~~`. Documentation must be placed at line start and may be single or multi-line.

In *TL5* documentation are treated as comments. In the final syntax they must come before the element they are documenting, they could be used dynamically in the code, and would be used to automatically generate external documentation.

```
~~~ single line documentation ~~~
func documented-function()
    ; do stuff

~~~  <-- multi-line documentation start
multi
line
documentation
multi-line documentation end --> ~~~
func another-documented-function()
    ; do stuff
```

2.11.3 Operators

- assignment: `:=`
- arithmetic: `+`, `-`, `*`, `div`, `mod`, `clamp`, `wraparound`
- assignment and arithmetic: `+=`, `-=`, `*=`
- bitwise: `bnot`, `bor`, `band`, `xor`, `shr`, `shl`
- relational (arithmetic): `=`, `<>`, `<`, `>`, `<=`, `>=`
- relational (referential): `is`, `is-not`, `?`
- logical: `not`, `or`, `and`
- miscellaneous: `.`, `[]`, `[:]`, `()`, `:=:`

Any operator may be followed by a line brake with additional indentation of exactly 8 spaces:

```
x := 3 +
        4
y :=
        3 + 4
z :=
        3 +
        4
```

Operator Precedence

1. `. [] () ?`, left-to-right
2. `bnot`
3. `- +`, `* div mod`, `bor band xor shr shl`, left-to-right *[1]*
4. `= != > < >= <= is is-not`, left-to-right *[2]*
5. `not`
6. `or`, `and`, left-to-right *[1]*
7. `clamp wraparound`, only one allowed
8. `:= += -= *= :=:`, only one allowed

[1] cannot combine operators from different sub-groups of this group, they must be separated using `()`, for example `a + b * c` is not legal and should be changed to `a + (b * c)`

[2] multiple operators from this group combined will be separated with `and` operator, for example, `a < b < c < d` is treated as `a < b and b < c and c < d`

2.11.4 Modules

In *TL5* each Lumi file is declared under a single module, multiple files may be declared under the same module.

The first line of each file must declare its module using the `module` keyword:

```
module my-module-name
```

Only a single documentation block can come before it.

Using any item of another module must come after the other module prefix:

```
var other-module.SomeType variable
other-moudle.function(user variable)
```

In the final syntax modules and libraries support will be greatly extended - the exact syntax is still under planning.

2.12 Built-in Types

2.12.1 Integer

`class Int`(*minimum-value*, *maximum-value*)

Parameters

- **`minimum-value`** – (optional) the minimum legal value (inclusive), default is `0`
- **`maximum-value`** – the maximum legal value (inclusive)

An integer that can only hold a range of values between its minimum and maximum (inclusive), enforced by the compiler. Obviously "minimum-value" should be equal or smaller than "maximum-value".

For example: `Int{10:20}` can only hold numbers between `10` and `20`.

If only one parameter is given it is treated as the maximum and the minimum is automatically `0`.

For example: `Int{100}` can only hold numbers between `0` and `100`.

If the minimum is negative the integer is consider “signed”, else it is “unsigned”.

The actual representation of an integer in the memory is the minimal possible from 8,16,32,64 bits based on the limits. This mean that only ranges that fit in `uint64_t` or `int64_t` are supported: `-9223372036854775808` to `9223372036854775807` for signed integer, and `0` to `18446744073709551615` for unsigned.

The default value of an uninitialized integer is `0` if it is within its legal range, otherwise the integer must be manually initialized.

Aliases for common ranges:

- `Uint8` for `Int{0:255}`
- `Sint8` for `Int{-127:127}`
- `Uint16` for `Int{0:65535}`
- `Sint16` for `Int{-32767:32767}`
- `Uint32` for `Int{0:4294967295}`
- `Sint32` for `Int{-2147483647:2147483647}`
- `Uint64` for `Int{0:18446744073709551615}`
- `Sint64` for `Int{-9223372036854775807:9223372036854775807}`

Supported integer literals:

- decimal: `70695`
- binary: `0b100101`
- octal: `0175`
- hexadecimal: `0xa30eb9f6`
- negative integer: `-` before any positive integer literal: `-23`

A complete guide to integer range management is in *Integer Ranges and Overflow Prevention*.

`str`(*user String text*)

write into `text` the integer converted to string in decimal

Raises

if `text` is too short to store the number

2.12.2 Infinitely Long Integer

planned - not supported yet in TL5

`class Long`

A signed integer that can be infinitely long (practically limited by the system memory). Automatically allocates heap memory to store the number. Is dramatically less efficient that a normal *integer*.

2.12.3 Boolean

`class Bool`

A boolean value that can only be a `true` or a `false` constant.

Constants:

`Bool true`

`Bool false`

2.12.4 Character

`class Char`

A single Unicode code-point, which is the same as `Int{1114111}`.

Character literal are surrounded with `'` characters: `'a'`. Special characters can be written with \ escape character as in C: `\'` `\"` `\?` `\a` `\b` `\f` `\n` `\r` `\t` `\v` `\\`.

2.12.5 Byte

`class Byte`

A single memory byte value.

Byte is treated as `Int{255}`.

2.12.6 Real Number

planned - not supported yet in TL5

`class Real`

Floating point real number, same as `float` in C.

Real number literal is a decimal number with `.` character in the middle, with optional exponential suffix: `2.4`, `-0.3`, `4.0`, `2.34e2`, `-5.678e-12`.

2.12.7 Function

`class Func`(*arguments*)

Holds (pointer to) a function.

Parameters

`arguments` – the function input and output *arguments*

For example: `Func{()}`, `Func{(copy Uint32 in)}`, `Func{()->(var Uint32 out)}` , `Func{(copy Uint32 in)->(var Uint32 out)}`.

2.12.8 Array

class `Array`(*length*, *subtype*)

Sequence of any typed item with static length.

Parameters

- **`length`** – array static length and the actual allocation size
- **`subtype`** – the type of each item in the array

For example: `Array{12:Uint32}`, `Array{6:String{16}}`.

Array references should be declared without the `length` parameter: just `Array{Uint32}` or `Array{String}` for example.

Accessing a single item can be done using `array[index]`.

Note: If the index can be out of range it is checked at run-time and an error may raise. In such case the `!` warning sign must be used if error is to be propagated: `array[index]!`.

It is possible to extract a sub-array from an array by slicing: `array[start-index:sub-array-length]`. This will not copy the array but return an array reference that points to the original array.

`length`()*->(var Uint32 length)*

return (static) length of the array

2.12.9 Buffer

class `Buffer`(*length*)

Alias for an *`Array`* with *`Byte`* items.

Parameters

`length` – length of the buffer and the actual allocation size

For example: `Buffer{5}`, `Buffer{256}`.

Buffer literals are hexadecimal strings surrounded by ` characters: `` `4a0069ff3487beef2649` ``.

2.12.10 String

class `String`

Holds a legal UTF-8 string with dynamic length. The compiler ensures that the last character is a null-terminator (`'\0'`).

String literals are strings surrounded by " characters: `"I am a string literal"`. Escape *characters* can be used.

String literals may contain line breaks, with additional indentation of exactly 8 spaces. It is treated as `\n`, or ignored if `\` is used before it:

```
; the same as "line\nbrake"
s := "line
        break"

```

(continues on next page)

(continued from previous page)

```
; the same as "linebrake"
s := "line\
        break"
```

String is currently not implicitly converted to `Array{Byte}` when needed.

`length`()->(*var Uint32 length*)

returns current (dynamic) string length, not counting the null-terminator

`new`(*user Buffer value*)

initialize this string with a copy of `value`

Raises
if not enough memory

`clear`()

make this string empty

`equal`(*user Buffer other)->(var Bool is-equal*)

return whether this string is exactly equal to `other`

`get`(*copy Uint32 index)->(var Char value*)

return character at place `index`

Raises
if `index` is out of range

`set`(*copy Uint32 index*, *copy Char value*)

set character at place `index` to `value`

Raises
if `index` is out of range

`append`(*copy Char character*)

append `character` to this string end

Raises
if has no room for another character

`concat`(*user Buffer other*)

concatenate `other` to this string end

Raises
if has no room for `other`

`concat-int(copy Sint64 number)`

covert `number` to string and concatenate it to this string end

Raises
if has no room for `number` string

`find`(*user Buffer pattern)->(copy Uint32 index*)

return index of first occurrence of `pattern` in this string, return this string *`length`* if `pattern` not found

`has`(*copy Char character)->(var Bool has*)

return whether this string contains `character`

2.12.11 Files

class File

Basic type for managing files, is extended by these types:

- `FileReadText`
- `FileReadBinary`
- `FileWriteText`
- `FileWriteBinary`
- `FileReadWriteText`
- `FileReadWriteBinary`

`FileReadText`(*user String file-name*)

open `file-name` for read only in textual mode

Raises
if file opening failed

`FileReadBinary`(*user String file-name*)

open `file-name` for read only in binary mode

Raises
if file opening failed

`FileWriteText`(*user String file-name*, *copy Bool append*)

open `file-name` for write only in textual mode

file is created if it does not exist

if `append` is true all writes will be appended to the file end

Raises
if file opening failed

`FileWriteBinary`(*user String file-name*, *copy Bool append*)

open `file-name` for write only in binary mode

file is created if it does not exist

if `append` is true all writes will be appended to the file end

Raises
if file opening failed

`FileReadWriteText`(*user String file-name*, *copy Bool append*, *copy Bool create*)

open `file-name` for read and write in textual mode

if `append` is true:

file is created if it does not exist

all writes will be appended to the file end

`create` is ignored

else, if `create` is true file is cleared of data if exists, or created if it does not exist

Raises
if file opening failed

`FileReadWriteBinary`(*user String file-name*, *copy Bool append*, *copy Bool exist*)

open `file-name` for read and write in binary mode

if `append` is true:

file is created if it does not exist

all writes will be appended to the file end

`create` is ignored

else, if `create` is true file is cleared of data if exists, or created if it does not exist

Raises
if file opening failed

`close`()

close this file, does nothing if this file is already closed

Raises
if closing failed

`tell`(*)->(var Sint64 offset*)

return current position of the file

Raises
if getting offset failed

`seek-set(var Sint64 offset)`

set file position to `offset` relative to file start

Raises
if setting offset failed

`seek-cur(var Sint64 offset)`

set file position to `offset` relative to the current position

Raises
if setting offset failed

`seek-end(var Sint64 offset)`

set file position to `offset` relative to file end

Raises
if setting offset failed

`flush`()

flush any buffered written data to the file

Raises
if flush failed

`get`(*)->(var Char value*, *var Bool is-eof*)

only available in `FileReadText` *and* `FileReadWriteText`

read one character from this file into `value` or set `is-eof` to `true` if end-of-file reached

Raises
if read failed

`get`*()->(var Byte value*, *var Bool is-eof*)

only available in `FileReadBinary` *and* `FileReadWriteBinary`

read one byte from this file into `value` or set `is-eof` to `true` if end-of-file reached

Raises
if read failed

`getline`*()->(user String line*, *var Bool is-eof*)

only available in `FileReadText` *and* `FileReadWriteText`

read one line from this file into `line` or set `is-eof` to `true` if end-of-file reached

new-line character is not added to `line` end

Raises
if read failed or `line` is too short to store the line

`read`(*user Array{Byte} data)->(var Uint32 bytes-read*)

only available in `FileReadBinary` *and* `FileReadWriteBinary`

read bytes from file to `data` up to the its length, set in `bytes-read` the number of actual read bytes

Raises
if read failed

`put`(*copy Char value*)

only available in `FileWriteText` *and* `FileReadWriteText`

write `value` character to this file

Raises
if writing failed

`put`(*copy Byte value*)

only available in `FileWriteBinary` *and* `FileReadWriteBinary`

write `value` byte to this file

Raises
if writing failed

`write`(*user Array{Char} data)->(var Uint32 written*)

only available in `FileWriteText` *and* `FileReadWriteText`

try write all `data` characters to this file, set in `written` the number of actual written characters

Raises
if writing failed

`write`(*user Array{Byte} data)->(var Uint32 written*)

only available in `FileWriteBinary` *and* `FileReadWriteBinary`

try write all `data` bytes to this file, set in `written` the number of actual written bytes

Raises
if writing failed

2.12.12 `sys` Module

`Array{String} sys.argv`

holds program arguments

`FileReadText sys.stdin`

can be used to **read** from the standard input stream

`FileWriteText sys.stdout`

can be used to **write** to the standard output stream

`FileWriteText sys.stderr`

can be used to **write** to the standard error stream

`sys.`**`print`**(*user String text*)

print `text` to the standard output stream, same as calling `sys.stdout.write`

Raises
if writing failed

`sys.`**`println`**(*user String text*)

print `text` appended with new-line character to the standard output stream

Raises
if writing failed

`sys.`**`getchar`**(*)->(var Char character*, *var Bool is-eof*)

read one character from the standard input stream into `value` or set `is-eof` to `true` if end-of-file reached, same as calling `sys.stdin.get`

Raises
if read failed

`sys.`**`getline`**(*user String line)->(var Bool is-eof*)

read one line from the standard input stream into `line` or set `is-eof` to `true` if end-of-file reached, same as calling `sys.stdin.getline`

new-line character is not added to `line` end

Raises
if read failed or `line` is too short to store the line

`sys.`**`exit`**(*copy Sint32 status*)

terminates execution of the program immediately with `status` as the exit status value

calls C `exit` function

Raises
if exiting failed

`sys.`**`system`**(*user String command)->(var Sint32 status*)

execute `command` by the host command processor and return the return status of the command

calls C `system` function

Raises
if command fails to execute

`sys.`**`getenv`**(*user String name*, *user String value)->(var Bool exists*)

set into `value` the value of the environment variable `name`, or set `exists` to `false` if it does not exist

2.13 Variables and Constants

In *TL5* the *Lumi memory management* is mostly implemented, excluding the *third management form* from the 3 planned.

2.13.1 Compile-Time Constants

In *TL5* only global integer compile-time constants are supported. The final Lumi syntax is planned to support constants from all types, and allow definition of constants inside the name-space of a specific type.

Integer compile-time constants are declared in *TL5* by:

```
const Int CONSTANT-NAME 12
```

Here `12` is an example constant value with range `Int{12:12}`. The constant value can be any constant expression, which may include:

1. integer numbers
2. other integer constants
3. enumerators
4. integer operators, where the operands are constant expressions

2.13.2 Enumerators

Enumerators are a set of constant symbols that are treated as integer compile-time constants. The first symbol is allocated a value of `0`, the second is `1` and so on...

In *TL5* enumerators can only be declared in the global scope. The final Lumi syntax is planned to support enumerators under a specific type, will allow definition of specific values for the enumerator symbols, and will generate automatic conversion functions between symbol names and values.

Enumerators are declared in *TL5* by:

```
enum EnumeratorName
    FIRST-SYMBOL-NAME
    SECOND-SYMBOL-NAME
    THIRD-SYMBOL-NAME
```

Using an enumerator is done by `EnumeratorName.SYMBOL-NAME`.

The amount of values is defined by a special `length` value, for example `EnumeratorName.length` is `3`.

2.13.3 Primitive Variables

Primitive variables are declared using `var` keyword:

```
var Int{100} integer-variable
var Int{10:20} with-initialization(copy 12)
```

If no explicit initialization value given - primitive values are initialized with each type's default initialization value:

- *`Int`* : `0` (if in range)
- *`Bool`* : `false`

- *Char* : `\0`
- *Byte* : `0x00`
- *Real* : `0.0`
- *Func* : empty (_)

2.13.4 References

References are declared using the wanted memory access keywords:

- `owner`: simple owner reference
- `user`: simple user reference
- `temp`: simple temporary owner reference
- `strong`: reference counted strong reference
- `weak`: reference counted weak reference

```
owner String string-owner-reference
user Array{Uint32} user-reference-with-initialization(user some-int-array)
temp String temporary-owner-reference
strong String string-strong-reference
weak Array{Uint32} weak-reference-with-initialization(weak some-int-array)
```

References must be assigned with a value before used.

For primitive type references the pointed value can be accessed using a special `value` named field:

```
int-reference.value := 4
```

Conditionals

Conditional references are declared by appending ? character in type end:

```
owner String? conditional-owner-reference
user Array?{Uint32} conditional-array-with-initialization(user some-int-array)
```

The _ sign can be used to represent an empty reference:

```
conditional-reference := _   ; setting the reference to be empty
func-with-conditional-argument(user _)   ; passing empty to a function
```

If no explicit initialization value given - conditional references are by default initialized as empty (_).

When a conditional reference is used it is checked at run-time to not be empty. If it is - an error is raised.

Note: In such case the `!` warning sign must be used if error is to be propagated: `conditional-reference!.field`

Weak References

Weak references may point to outdated data that was removed in the past. Therefor, when a weak reference is used it is checked at run-time to not be outdated. If it is - an error is raised.

Note: In such case the `!` warning sign must be used if error is to be propagated: `weak-reference!.field`

Comparisons

Comparing references by-reference is done using the `is` and `is-not` operators.

the `?` operator can be used to check if a reference is usable: not empty and not an outdated weak reference. To explicitly check for emptiness and not for being outdated of a reference that is both conditional and weak `is` and `is-not` operators can be used with `_` as operand.

```
if first-reference is second-reference
    ; both references reference to the same object, or both are empty
if first-reference is-not second-reference
    ; both references do not reference to the same object
if reference?
    ; reference is usable - not empty and not outdated
if not reference?
    ; reference is not usable - it is empty or outdated
if reference is _
    ; reference is empty
if reference is-not _
    ; reference is not empty, but may or may not be outdated
```

String and Buffer Literals

String and *buffer* literals are allocated in the global data section. In the scope they are used they are treated as `user` access references to the global data.

```
user String string(user "a string literal")
user Buffer buffer(user `baffdaca`)
```

2.13.5 Static Allocation

Static allocation is done using `var` or `s-var` keywords:

```
var String{256} string-static-allocation
s-var Array{34:Uint32} static-strong-int-array!
```

Note: `s-var` initialization may fail - the `!` warning sign must be used if error is to be propagated

Doing this in the global scope will allocate the data in the process global data section. Doing this in a function scope will allocate the data in the stack.

Statically allocated variables cannot pass their ownership to `owner` references.

2.13.6 Dynamic Allocation

Dynamic allocation is done by using the type as a function:

```
string-owner-reference := String{256}()!
array-strong-reference := Array{34:Uint32}()!
```

Note: dynamic allocation may fail - the `!` warning sign must be used if error is to be propagated

It's probably a good idea to store the returned object in an `owner` reference, otherwise it will be deleted in the end of the block.

2.14 Functions

2.14.1 Summary

Function are declared using the `func` keyword

```
func func-name(copy Uint32 input-argument)->(var Uint32 output-argument)
    ; function implementation...
```

Functions are called using their name:

```
func-name(copy 4)->(var some-int)
```

2.14.2 Arguments

First input arguments (a.k.a "parameters") are written inside `(...)`. If the function has no input arguments an empty `()` should be used.

If the function has output arguments ("outputs" in short) they are written second inside a different `(...)` after a `->` symbol with no whitespace. Multiple outputs are supported.

In function deceleration each argument must be declared with `access type name` trio. In function calling only `access name` duo is needed. See *Access* explanation below.

A new line can be placed before any argument access, with additional indentation of exactly 4 spaces:

```
func split-arguments0(copy Uint32 x, copy Uint32 y)->(
    var Uint32 z, var Uint32 w)

func split-arguments1(
    copy Uint32 x, copy Uint32 y)->(var Uint32 z, var Uint32 w)

func split-arguments2(
    copy Uint32 x, copy Uint32 y)->(
    var Uint32 z, var Uint32 w)

func split-arguments3(
    copy Uint32 x,
    copy Uint32 y)->(
```

(continues on next page)

(continued from previous page)

```
    var Uint32 z,
    var Uint32 w)
```

2.14.3 Access

An "access" defines the memory access of the argument. It can be one of:

`copy` parameter:

- the parameter is a new memory copy of the called argument
- changes to the parameter will **not** change any caller variable
- any expression can be given

`var` output:

- the parameter is a reflection of an actual variable
- changes to the parameter will **also** change the caller variable
- only a writable value can be given

`user`, `owner`, `temp`, `strong` or `weak` parameter:

- the argument is a reference to an object
- changes to the reference itself will **not** change the called reference
- changes to the object will change the called object
- any expression can be given
- `user` or `weak` means the parameter is a simple reference
- `owner` or `strong` means the caller has passed the ownership of the referenced object to the function, and the object will be deleted in the function end if the ownership is not passed in the function body
- `temp` means the caller has temporally passed the ownership of the referenced object to the function, and the ownership will be returned after function end and cannot be deleted or permanently passed forward by the function

`user`, `owner`, `strong` or `weak` output:

- the parameter is a reflection of an actual reference
- changes to the reference itself will **also** change the called reference
- only a writable value can be given

In *TL5* `copy` and `var` are not yet supported for complex types.

In the planned final syntax this will be extended, and the access may be omitted in a default usage.

2.14.4 Return and Output

In *TL5* output is written by setting a value to an output argument:

```
func example()->(var Uint32 first-output, user String second-output)
    first-output := 4
    second-output := String{16}()
```

A `return` statement can be used to stop the function in the middle:

```
func example(copy int x)
    if x < 0
        return
    ; do something
```

In the final syntax this may be possible:

```
func example()->(var Uint32 first-output, owner String second-output)
  return 4, String{16}()
```

2.14.5 Error Handling

Raising an error can be done using the `raise` statement. Functions that may raise an error must be marked with `!`:

```
func ! example()
    raise!
```

In *TL5* an optional string expression can be raised:

```
func ! example()
    raise! "error message"
```

Error Propagation

Unless caught, raised error will propagate to the caller function, up until the main function - where uncaught errors will stop the execution of the program, print the raised error message if given, and print a call traceback.

In the function code whenever an error may be raised and propagated to the caller - the `!` warning sign must be added. A functions that may raise an error must also add the `!` warning sign to its deceleration.

Error Catching

A local error can be handled using `if-ok` or `if-error`:

```
if-ok x := array[3]
    ; no error raised
else
    ; index out of bound handling

if-error x := array[4]
    ; index out of bound handling
```

(continues on next page)

(continued from previous page)

```
else-if-ok x := array[6]
    ; no error raised
else-if-error x := array[5]
    ; index out of bound handling
else
    ; no error raised
```

Note: `if-ok` must be followed by `else` to ensure error is not ignored

A `try` statement will catch an error raised inside it and break the execution of the rest of the block. The error will be ignored unless `try` is followed by a `catch` statement. The code under the `catch` statement will only run if the above `try` statement caught an error.

```
try
    ; do something that may raise errors
catch
    ; do some error handling
```

2.14.6 Calling a Function

When calling a function the access of each argument must be written:

```
example(copy primitive-input, user reference-input)->(
        var primitive-output, owner owner-output)
```

If the function may raise an error and the caller propagates the error - `!` warning sign must be used:

```
raising-example(copy input)->(var output)!
```

2.15 User Defined Types

As explained in *Type System*, Lumi allows variety of typing styles for creating user defined types.

In *TL5* only basic *structures* and *classes* are supported.

User defined types behave like built-in complex types with *references*, *static allocation* and *dynamic allocation*.

2.15.1 Static Structures

Syntax for the *static structure typing style*.

In *TL5*, structures may not contain string or array variables, only their respected references. It will be supported in the final Lumi syntax.

Structures are declared using the `struct` keyword:

```
struct ExampleStruct
    var Uint32 integer-variable
    user String string-reference
```

Members of struct can be accessed using . operator:

```
var ExampleStruct struct-variable
struct-variable.integer-variable := 3
struct-variable.string-reference := "some string"
```

Structures are implemented in C as simple C structures. Structure references are implemented as pointers to the C structure.

Global Members

This is not supported yet in *TL5*.

Global members are declared under the type scope:

```
struct ExampleStruct
    const Uint32 GLOBAL-CONSTANT(12)
    global var Uint32 global-variable
```

Outside the type definitions they can only be accessed with the type name as prefix:

```
some-integer := ExampleStruct.GLOBAL-CONSTANT
ExampleStruct.global-variable := 5
```

Methods

Methods are declared as normal *functions*, except they are declared inside the type scope, and the `self` parameter should not be declared, instead, its access is declared before the function name.

Inside the method implementation `self` keyword can be used to access the implicit self parameter. Constants and global variables of the type can be accessed using `global` keyword.

```
struct ExampleStruct
    var Uint32 integer-variable

    const Uint32 GLOBAL-CONSTANT(12)
    global var Uint32 global-variable

  ~~~ "self" access is "user" ~~~
  func user method(copy Uint32 num)
      self.integer-variable := num + global.GLOBAL-CONSTANT
      global.global-variable := num
```

It possible to split the function deceleration from its implementation. In this case the function deceleration should be followed by `_`.

```
struct ExampleStruct
    func user method(copy Uint32 num) _

func user ExampleStruct.method(copy Uint32 num)
    ; implementation...
```

There are two ways to call a method:

```
instance.method(copy 4)  ; OOP style
ExampleStruct.method(var instance, copy 4)  ; functional style
```

Constructor Method

If possible, structure members are automatically initialized to their default value on construction. This can be extended by defining a “constructor” method for the structure. This method will be called on every instance construction after the default initialization. A constructor is declared with a dedicated name `new`.

```
struct ExampleStruct
    new() _

func ExampleStruct.new()
    ; custom initialization
```

A constructor cannot have outputs, and if it has parameters - they must be given on every object creation:

```
struct ExampleStruct
   var Uint32 integer-variable
   owner String string-reference

   new(copy Uint32 x, owner String s)
       self.integer-variable := x
       self.string-reference := s

func usage()
    var ExampleStruct variable(copy 4, owner String{12}(user "some string"))
    owner ExampleStruct reference := ExampleStruct(copy 4,
           owner String{12}(user "some string"))
```

Structures that have members without a defined default value must implement a constructor. The constructor must also directly initialize these fields. Members without a defined default value are:

- non-conditional references
- integers that `0` is not a legal value of their range
- variables of types with a constructor

Destructor Method

A “destructor” method can also be defined for a structure. This method will be called just before any object destruction. A destructor is declared as a normal method with a dedicated name `cleanup`.

```
struct ExampleStruct
    cleanup() _

func ExampleStruct.cleanup()
    ; destruction code
```

A destructor cannot have any kind of arguments.

In *TL5* destructors cannot raise errors - but it may be supported in the future.

Note: Lumi Automatically deletes any memory allocated in the structure and calls the cleanup function of all members and base classes - there is no need to do it manually

Extending Structures

In *TL5* a structure may only extend one other structure.

```
struct ExtendingStruct(BaseStruct, OtherBaseStruct)
    var Uint32 additional-field
```

The extending structure may be used in any place one of its base structures is expected:

```
owner BaseStruct base-struct := ExtendingStruct()
```

The extending structure may overwrite a base method, the overwriting method arguments access and type must be identical to the base overridden method.

```
struct BaseStruct
    func method(copy Uint32 num)
        ; implementation...

struct ExtendingStruct(BaseStruct)
    func method(copy Uint32 num)
        ; other implementation...
```

An overwriting function can call the overwritten function using `base` keyword. Other overwritten methods can be called using `base.other-method`.

```
struct ExtendingStruct(BaseStruct)
    func method(copy Uint32 num)
        base(copy num)
        base.other-method()
```

Example for the **static** dispatch of structures:

```
var ExtendingStruct extending-struct
user BaseStruct base-struct(user extending-struct)
extending-struct.method(copy 4)  ; will call ExtendingStruct.method
base-struct.method(copy 4)  ; will call BaseStruct.method
BaseStruct.method(var extending-struct, copy 4)  ; will call BaseStruct.method
```

2.15.2 Dynamic Interfaces

Syntax for the *dynamic interface typing style*.

Dynamic interfaces (or “dynamics” in short) are declared using the `dynamic` keyword:

```
dynamic ExampleDynamic
    func dynamic-method(copy Uint32 num)
    func another-method()->(var Uint32 result)
    var Uint32 dynamic-variable
```

Dynamic variables are not supported in *TL5*.

Dynamics are always used as references and cannot be allocated because they have no structure:

```
func use-dynamic(user ExampleDynamic example)
   example.dynamic-method(copy 3)
```

Now `use-dynamic` function can be called with any item that implements `ExampleDynamic`.

Dynamics are implemented in C as a C structure containing all the dynamic members, where dynamic methods are implemented as pointer to functions. Each implementation of the dynamic is a global instance of this structure. Dynamic references are implemented as 2 references: one reference to the dynamic structure and another reference to the implementing type instance.

Non-Dynamic Members

This is not supported in *TL5*.

Constants and global variables are declared and used exactly as *global members in static structures*.

Static methods must be declared using `static` prefix:

```
dynamic ExampleDynamic
    func dynamic-method(copy Uint32 num)
    static func static-method(copy Uint32 num)
        ; implementation
```

Extending Dynamics

This is not supported in *TL5*.

Same syntax as structures:

```
dynamic ExtendingDynamic(BaseDynamic, OtherBaseDynamic)
    func additional-method(copy Uint32 num)
```

Support Dynamics in Structures

A *structure* can support a dynamic by implementing all its dynamic members and explicitly declare it using the `support` keyword. some implemented members may be added under the `support` line:

```
struct ExampleStructure
    func method(copy Uint32 num)
        ; implementation...

support ExampleDynamic in ExampleStructure
    func another-method()->(var Uint32 result)
        ; another implementation...
```

When a *structure* supports a dynamic, every structure that extends it also supports the dynamic using the base structure implementation. The extending structure may override some members of the dynamic, but to use these overrides as the implementation of the dynamic another `support` statement for the extended structure must be added:

```
struct BaseStruct
    func method(copy Uint32 num)
        ; base implementation...

support ExampleDynamic in BaseStruct

struct ExtendingStruct(BaseStruct)
    func method(copy Uint32 num)
        ; overriding implementation...

support ExampleDynamic in ExtendingStruct
```

Example for the **dynamic** dispatch of dynamics:

```
var ExampleStructure example-struct
var ExtendingStruct extending-struct
user BaseStruct base-struct(user extending-struct)
user ExampleDynamic example-dynamic

example-dynamic := example-struct
example-dynamic.method(copy 4)  ; will call ExampleStructure.method

example-dynamic := extending-struct
example-dynamic.method(copy 4)  ; will call ExtendingStruct.method

example-dynamic := base-struct
example-dynamic.method(copy 4)  ; will call BaseStruct.method
; will not call ExtendingStruct.method becasue structure dispach is static
```

Default Dynamic Member Implementation

This is not supported in *TL5*.

A dynamic may give a default implementation to some or all of its members and its base dynamics members. Method implementations can use `self` and `global` keywords to access its own members.

```
dynamic ExampleDynamic
    func implemented-method(copy Uint32 num) _
    func unimplemented-method()->(var Uint32 result)
    var Uint32 implemented-variable(copy 5)
    var Uint32 unimplemented-variable

func ExampleDynamic.implemented-method(copy Uint32 num) _
    ; implementation...
```

2.15.3 Classes and Binds

Syntax for the *class typing style*.

In *TL5* this only partially implemented:

- Only `class` type definition is supported, `Bind` is not
- All restrictions on structures also apply to classes
- Only methods can be dynamic
- Variables don't need to start with `static` keyword - as they cannot be dynamic or global

A straightforward way to use classes is using the built-in `Bind` typed references. References of this type only accept types that extend all bound structures and implement all bound dynamics.

```
user Bind{ExampleStruct:ExampleDynamic} class-reference
```

Another way to use classes is to declare a type as a class in its definition using the `class` keyword. Each non-global member of the class must come after a `static` or a `dynamic` keyword to declare witch implicit type this member belongs to: the structure or the dynamic. Global members are only defined under the name-space of the class.

```
class ExampleClass
    static var Uint32 static-field  ; part of the implicit structure
    dynamic func dynamic-method(copy Uint32 num)  ; part of the implicit dynamic
    global var Uint32 global-variable  ; defined under the class name-space
```

Classes can implement dynamics using the *same syntax as structures*.

Class references are implemented using two C pointers: one for the structure, and one for the dynamic.

Extending Classes

As all types:

```
class ExtendingClass(BaseStruct, BaseDynamic, BaseClass)
    static var Uint32 addition-static-field
    dynamic func addition-dynamic-method(copy Uint32 num)
```

In *TL5* a class may only extend one other class or structure.

Example for the **dynamic** dispatch of classes:

```
var ExtendingClass extending-class
user BaseClass base-class(user extending-class)
user ExampleDynamic example-dynamic

extending-class.method()  ; will call ExtendingClass.method
base-class.method()  ; will call ExtendingClass.method

example-dynamic := extending-struct
example-dynamic.method()  ; will call ExtendingStruct.method

example-dynamic := base-struct
example-dynamic.method()  ; will call ExtendingStruct.method
```

Using the Implicit Structure or Dynamic of a Class

This is not supported in *TL5*.

The implicit structure of a class can be used using the built-in `Struct` type, and the implicit dynamic can be used using the built-in `Dynamic` type. This is not supported in *TL5*.

```
var Struct{ExampleClass} static-structure-only
user Dynamic{ExampleClass} dynamic-interface-only
```

2.15.4 Parameterized Types

Syntax for the *parameterized type typing style*.

Each type parameter must have a type and a name. For static type names `Type` should be used as the parameter type, and for dynamic parameters `Generic` should be used as the parameter type. The parameter name must conform the naming standard of types if one of these is used, else it must conform naming standard of constants.

```
struct ParametrizedType{Uint32 CONSTANT-PARAMETER:Type TypeParameter:Generic␣
↪GenericParameter}
    var String{CONSTANT-PARAMETER} parametrized-sized-string
    var TypeParameter static-parametrized-typed-variable
    user GenericParameter dynamic-parametrized-typed-reference
```

Whenever a parameterized type is used it must be set with appropriate values for each parameter

```
var ParametrizedType{8:Uint32:File} specific-variable
```

This is partially supported in *TL5*:

- only dynamic parameters are supported (`Generic` type)
- no need to add `Generic` - only the parameter name is needed
- Arrays are not supported as parameter values

2.15.5 Embedded Dynamic Reference

Syntax for the *embedded dynamic reference typing style*.

This is not supported in *TL5*.

Embedded classes can be declared using the built-in `Embed` type:

```
; "ExampleStruct" structure with "ExampleDynamic" reference embedded
; inside it
var Embed{ExampleStruct:ExampleDynamic} explicit-embedded-variable

; "ExampleClass" static structure with a reference to its dynamic structure
; embedded inside it
var Embed{ExampleClass} implicit-embedded-variable
```

The syntax may change as this typing style is still under planning.

2.16 Control Flow

2.16.1 If-Else Condition

If-else condition is declared using `if`, `else`, and `else-if` keywords. The condition expression must be boolean typed.

```
if x > 4
    ; do something
else-if x < 2
    ; do something
else
    ; do something
```

2.16.2 Switch-Case Condition

This is not supported in *TL5*.

Switch-case condition is declared using the `switch` keyword, contains multiple `case` blocks, and optionally one last `default` block. A `fallthrough` statement must be used to fall-through the next case - the default is not to fall-through.

```
switch number
case 34
    ; do something
case 23
    ; do something
    fallthrough
case 45, 67, 26, 56, 67, 89, 56, 87
    ; do something
default
    ; do something
```

2.16.3 Simple Loop

Simple loop is declared using the `loop` keyword and contains one or more `while` statements inside it. The loop continues while every `while` statement inside it is true, and stops immediately when the first `while` statement inside it is false.

```
loop
    ; do something
    while number < 6
    ; do something
    while not boolean-variable
    ; do something
```

Loops can be broken immediately using a `break` statement:

```
loop
    ; do something
    if number = 0
        break
    ; do something
```

That makes `while condition` the same as `if not condition break`.

A `continue` statement can be used to only stop the current iteration and start over from loop beginning:

```
loop
    ; do something
    if num = 3
        continue
    ; do something
```

It is possible to limit the number of loop iterations, when the limit is reached an error is raised:

```
loop! number
    ; do something
    while condition
```

Note: The `!` warning sign must be used if error is to be propagated.

Loops must contain at least one `while`, `break` or `return` statement - otherwise the compiler will complain about an infinite loop. If an infinite loop is intentional `loop-infinite` must be used:

```
loop-infinite
   ; do something forever
```

2.16.4 Repeat Loop

A simple loop that just repeats itself a specific number of times:

```
repeat number
    ; do this "number" times
```

2.16.5 For Loop

For loop iterates over a specific set of values, and is declared using the `for` keyword.

Iterating numbers incrementally, limits can be any integer expression:

```
for number in 3:7
    ; "number" will iterate 3,4,5,6
```

Number iteration with explicit step amount, this is not supported in *TL5*:

```
for number in 9:1:-2
    ; "number" will iterate 9,7,5,3
```

Array iteration:

```
for item in array
    ; "item" will iterate each item of "array"
```

String iteration:

```
for character in "Example"
    ; "character" will iterate E,x,a,m,p,l,e
```

Buffer iteration:

```
for byte in `baffdaca`
    ; "character" will iterate ba,ff,da,ca
```

In all for loops it is possible to ignore the iteration item by replacing it with `_`:

```
for _ in 3:7
    ; will iterate 4 times
```

User Defined Iterators

A type can be made into an iterator in *TL5* by implementing a `step` named method that has the following deceleration:

`step`()->(*user SomeType? value*, *var Bool has-another-item*)

Is called once before any iteration. Iteration continue only if `has-another-item` is `true`. In such case `value` returns the next iteration value, and the iteration should advance one step. `SomeType` declared in this method is used as the iterator value type.

An instance of such iterator type can be used in for loops:

```
for item in iterator-instance
    ; "item" will iterate as implemented by "iterator-instance" type
```

This interface may change in the final syntax - the exact syntax is still under planning.

2.17 Testing

Lumi has built-in testing and mocking capabilities.

Testing code should be written under a different module than the tested code. In the testing code there should be *test cases* that test the tested module using *assertions*. *Mocking* should be used to simulate external interfaces.

Lumi can then run all test cases automatically, while also checking code coverage for the tested module.

2.17.1 Test Cases

Can be written using the `test` keyword:

```
test test-case-name()
    ; test code
```

The test case is consider a success if the code runs without any assertion or runtime errors.

2.17.2 Assertions

Assertions test that a certain condition is true.

The basic assertion is the `assert` statement that checks whether a given boolean expression is true. If not, this statement will raise an assertion error and the test will fail.

```
assert! number = 4
```

Another assertion is the `assert-error` statement that checks whether a given expression raises an error. If the execution of the statement didn't raise an error this statement will raise an assertion error and the test will fail.

```
assert-error! raising-function()
```

In *TL5* `assert-error` supports an optional additional string literal that is the expected error message. If the raised error message is not exactly the same as this literal, `assert-error` will raise an assertion error and the test will fail.

```
assert-error! raising-function(), "expected error message"
```

Note: The `!` warning sign must be used if error is to be propagated, which is recommended for the test to fail…

2.17.3 Mocking

Mocking can replace an external interface with simulated behavior.

In *TL5*, only functions and methods can be mocked.

Mocking a function or method is done using the `mock` keyword. The function name and its arguments access and type must much the mocked function exactly. Whenever the mocked function is used the mocking function is called instead.

```
mock mocked-module.mocked-function(copy Uint32 input)->(var Uint32 output)
    ; mocking body

mock mocked-module.MockedType.mocked-method()
    ; mocking body
```

Built-in functions and methods can also be mocked:

```
mock Sys.print(user String text)
    ; mocking body

mock FileReadText.new(user String filename)
    ; mocking body
```

The mocked function can still be called using `mocked` member:

```
mocked-function-name.mocked()
```

To disable mocking of a function `active` member can be set to `false`. To re-enable mocking it can be set back to `true`. Mocks are active by default.

```
mocked-function-name.active := false
mocked-function-name()
mocked-function-name.active := true
```

2.18 Interacting with Other Languages

Lumi allows calling C code directly. This is useful for using an external C library, or using already written C code within Lumi code.

Warning: Calling C code cannot be guaranteed to be safe as the C code can mangle with the Lumi memory management.

It is also possible to call Lumi functions from other languages by compiling Lumi to a shared library that *exports C style functions*.

2.18.1 The `cdef` Module

The builtin `cdef` module contains various C declarations that help interacting with C code.

C Primitives

`cdef` contains many C primitive types:

- `cdef.Char`
- `cdef.Uchar`
- `cdef.Short`
- `cdef.Ushort`
- `cdef.Int`
- `cdef.Uint`
- `cdef.Long`
- `cdef.Ulong`
- `cdef.Size`
- `cdef.Float`
- `cdef.Double`
- `cdef.LongDouble`

C Pointers

`cdef` contains `cdef.Pointer` generic type to declare C pointers:

```
var cdef.Pointer{cdef.Int} pointer-to-int
var cdef.Pointer{cdef.Char} pointer-to-char
; "cdef.CharP" is an alias to "cdef.Pointer{cdef.Char}"
var cdef.CharP also-pointer-to-char
var cdef.Pointer{cdef.Uchar} pointer-to-uchar
var cdef.Pointer{cdef.Pointer{cdef.Int}} pointer-to-pointer-to-int
var cdef.Pointer void-pointer
```

Set and get pointed value:

```
var cdef.Int value
pointer-to-int.set-point-to(var value)
pointer-to-pointer-to-int.set-point-to(var pointer-to-int)
pointer-to-int := pointer-to-pointer-to-int.get-pointed-at(copy 0)
value := pointer-to-int.get-pointed-at(copy 0)
```

Get and set pointer from array, string and buffer:

```
user Array{cdef.Int} int-array
user String string
user Buffer buffer
pointer-to-int := int-array
pointer-to-char := string.cdef-pointer()
pointer-to-uchar := buffer
value := pointer-to-int.get-pointed-at(copy 3)
cdef.copy-to-string(copy pointer-to-char, user string)!
cdef.copy-to-buffer(copy pointer-to-uchar, copy 4, user buffer)!
```

Pointer to complex types:

```
user SomeStruct reference
var cdef.Pointer{SomeStruct} pointer-to-struct
pointer-to-struct := reference
reference := pointer-to-struct.get-ref-at(copy 0)
```

2.18.2 Calling C Functions

To call a C function from Lumi it must first be declared using the `native func` statement:

```
; allow calling a C function with header:
; "int some_c_function(int number)"
native func cdef.Int some-c-function(copy cdef.Int number)
; "_" characters in the C function name are replaced with "-" in Lumi

; allow calling a C function with header:
; "void CfunctionName(int number, char* text)"
native func name-used-in-lumi(
        copy cdef.Int number,
        copy cdef.Pointer{cdef.Char} text) "CfunctionName"
; Lumi name can be different than the C name
```

These functions can now be called from Lumi code using their Lumi name.

Native functions Have some limitation compared to normal Lumi functions:

- have no output arguments, only input parameters and an optional single return type
- all parameters must be primitives, with `copy` access
- cannot be called with output arguments: `native-function()->(copy output-argument)` is illegal, may use `output-argument := native-function()` instead

2.18.3 Accessing C Global Variables

In order to access a C global variable it must be declared using the `native var` statement:

```
; allow accessing "int some_c_variable" global variable
native var cdef.Int some-c-variable

; allow accessing "char* CvariableName" global variable
native var cdef.Pointer{cdef.Char} name-used-in-lumi "CvariableName"
```

These variables can now be accessed from Lumi code using their Lumi name.

Only primitive types can be declared as native variables.

2.18.4 Accessing C Global Constants or Defines

In order to access a C global constant or a `#define` value it must be declared using the `native const` statement:

```
; allow accessing "SOME_C_CONSTANT" global constant
native const cdef.Int SOME-C-CONSTANT

; allow accessing "c_constant_name" global constant
native const cdef.Int NAME-USED-IN-LUMI "c_constant_name"
```

These constant can now be accessed from Lumi code using their Lumi name.

Only primitive types can be declared as native constants. Currently in *TL5* only integer types are supported.

2.18.5 Accessing C Structures

It is possible to access custom C structures and their internal fields using the `native struct` statement with `var` lines for each needed field:

```
; allow using "SomeCStruct" structure that have fields:
;   int some_filed;
;   char* other_field;
native struct SomeCStruct
    var cdef.Int some-filed
    var cdef.Pointer{cdef.Char} other-field

; allow using "struct c_struct_name" structure that have fields:
;   int CfieldName;
;   char* CanotherName;
native struct NameUsedInLumi "struct c_struct_name"
    var cdef.Int field-name-used-in-lumi "CfieldName"
    var cdef.Pointer{cdef.Char} another-lumi-field "CanotherName"
```

Not all of the original fields must be declared - only the ones that are needed to be used in Lumi. It is also legal to not declare any fields at all:

```
native struct SomeCStruct
```

These structures can now be accessed from Lumi code using their Lumi name.

Native structures are treated as values and not as references like Lumi structures. A pointer to the native structures can be used instead:

```
var cdef.Pointer{SomeCStruct} pointer-to-native-struct
```

Native structures fields are accessed as in Lumi structures: `native-struct.some-filed`. This also works with pointers to native structures: `pointer-to-native-struct.some-filed`.

Native structures can be used in other native functions, variables, constants, and structures:

```
native func SomeCStruct c-func-name(copy SomeCStruct input)
native func cdef.Pointer{SomeCStruct} other-func(
    copy cdef.Pointer{SomeCStruct} input)
native var SomeCStruct c-var-name
native var cdef.Pointer{SomeCStruct} other-var
native struct CstructName
   var SomeCStruct struct-field
   var cdef.Pointer{SomeCStruct} pointer-field
   var cdef.Pointer{OtherStruct} self-pointer
```

2.18.6 Accessing Custom C Types

It is possible to handle values for custom C types that may be of any kind: integers, structures, pointers, etc. These types are treated as “abstract” values in Lumi, meaning that their exact structure is unknown and cannot be accessed.

C types can be declared using the `native type` statement:

```
; allow using "SomeCtype" type:
native type SomeCtype

; allow using "c_type_name" type:
native type NameUsedInLumi "c_type_name"
```

These types can now be accessed from Lumi code using their Lumi name.

Native types are treated as abstract unknown values, the only way to use their content is by other C functions.

2.18.7 Writing C Code Directly

It is possible to write C code directly using `native code` in global scope, or just `native` inside a function

```
native code "#define SOME_NEEDED_DEFINE 1"

func is-unix()->(var Bool result)
    native "#ifdef __UNIX__"
    result := true
    native "#else"
    result := false
    native "#endif"
```

This may be used in some special cases where the other methods above are not sufficient, or to write some special glue code between Lumi and C.

2.18.8 C Wrapper Code

It's recommended to wrap native C declarations with pure Lumi declarations that takes care for correct usage of the C declarations, and to present a simple and safe pure Lumi interface.

2.18.9 Exporting C style Functions in a Shared Library

Functions that are meant to be exported when compiling Lumi code to a shared library must be declared using `native export` statement:

```
; exporting a function with C header:
; "int some_exported_function(int number)"
native export cdef.Int some-exported-function(copy cdef.Int number)
    ; function body...
    return 0
; "-" characters in the Lumi function name are replaced with "_" in C

; exporting a function with C header:
; "void CfunctionName(int number, char* text)"
native export name-used-in-lumi(
        copy cdef.Int number,
        copy cdef.Pointer{cdef.Char} text) "CfunctionName"
    ; function body...
; Lumi name can be different than the C name
```

These export functions have the same rules and limitation as *native functions*.

To return a value from the body of a native export functions that has a return type `return <value>` statement can be used.

These functions are exported as C style functions and can be used from any program using the same mechanic C functions are called from a shared library. Only functions declared with `native export` and that are inside the exported module will be accessible in the compiled shared library.

2.19 Standard Library

Currently implemented modules in *TL5* standard library:

- data structures (basic)
- time
- math (basic)
- os (basic)
- zlib (basic)

Planned future modules in the standard library:

- more data structures
- math (full)
- os (full)
- zlib (full)

- system
- IO
- multiprocessing
- paths and files
- more compressions
- web-server (DB, HTTP, ext…)
- GUI

2.20 Code Examples

2.20.1 Hello World Program

A simple "Hello World" program written in *TL5*:

```
module hello-world

func ! show()
    sys.println(user "hello world")!

main!
    show()!
```

2.20.2 Testing the Hello World Program

Testing code for the "Hello World" program:

```
module hello-world-test

var Bool println-raise
var String printed-text

mock ! sys.println(user Buffer text)
    if println-raise
        raise! "error in println"
    printed-text.concat(user text)!

test show-hello-world-test()
    println-raise := false
    printed-text.clear()
    hello-world.show()!
    assert! printed-text.equal(user "hello world")

test show-raise-test()
    println-raise := true
    assert-error! hello-world.show(), "error in println"
```

2.20.3 Fibonacci Function

```
func fibonacci(copy Uint64 n)->(var Uint64 res)
    var Uint64 prev(copy 1)
    res := 0
    repeat n
        var Uint64 sum(copy res clamp+ prev)
        prev := res
        res := sum
```

2.20.4 Complex number Type

```
struct Complex
    var Sint64 real
    var Sint64 imaginary

    new(copy Sint64 real, copy Sint64 imaginary)
        self.real := real
        self.imaginary := imaginary

    func user ! str(user String out-str)
        self.real.str(user out-str)!
        out-str.append(copy ' ')!
        if self.imaginary > 0
            out-str.append(copy '+')!
        else
            out-str.append(copy '-')!
        out-str.append(copy ' ')!
        var String imaginary-str
        if self.imaginary > 0
            self.imaginary.str(user imaginary-str)!
        else
            (- self.imaginary).str(user imaginary-str)!
        out-str.concat(user imaginary-str)!
        out-str.append(copy 'i')!

func ! usage-example()
    var Complex complex(copy 5, copy 3)
    var String complex-str
    complex.str(user complex-str)!
    sys.println(user complex-str)!
```

2.21 Serialization

This is not supported yet in *TL5*.

Lumi is planned to support 3 layout types for converting structures to buffers, and vice versa:

- **encoding** - same layout as the structure platform-specific memory layout
- **serializing** - platform-independent and CPU efficient implicit layout
- **packing** - explicit, platform-independent, user defined layout

type	CPU	memory	cross-platform	explicit
encoding	best	medium	no	no
serializing	medium	worst	yes	no
packing	worst	best	yes	yes

2.22 Asynchronous IO

This is not supported yet in *TL5*.

Lumi is planned to have a built-in support for asynchronous IO and running parallel fibers (a.k.a coroutines / user-space-threads) in a single thread.

The syntax is not decided yet…

INDEX

A

append() (*String method*), 29
Array (*built-in class*), 28

B

Bool (*built-in class*), 27
Buffer (*built-in class*), 28
built-in function
    step(), 50
    sys.exit(), 33
    sys.getchar(), 33
    sys.getenv(), 33
    sys.getline(), 33
    sys.print(), 33
    sys.println(), 33
    sys.system(), 33
Byte (*built-in class*), 27

C

Char (*built-in class*), 27
clear() (*String method*), 29
close() (*File method*), 31
concat() (*String method*), 29

E

equal() (*String method*), 29

F

File (*built-in class*), 30
FileReadBinary() (*File method*), 30
FileReadText() (*File method*), 30
FileReadWriteBinary() (*File method*), 30
FileReadWriteText() (*File method*), 30
FileWriteBinary() (*File method*), 30
FileWriteText() (*File method*), 30
find() (*String method*), 29
flush() (*File method*), 31
Func (*built-in class*), 27

G

get() (*File method*), 31
get() (*String method*), 29
getline() (*File method*), 32

H

has() (*String method*), 29

I

Int (*built-in class*), 25

L

length() (*Array method*), 28
length() (*String method*), 29
Long (*built-in class*), 26

N

new() (*String method*), 29

P

put() (*File method*), 32

R

read() (*File method*), 32
Real (*built-in class*), 27

S

set() (*String method*), 29
step()
    built-in function, 50
str() (*Int method*), 26
String (*built-in class*), 28
sys.exit()
    built-in function, 33
sys.getchar()
    built-in function, 33
sys.getenv()
    built-in function, 33
sys.getline()
    built-in function, 33
sys.print()
    built-in function, 33
sys.println()

built-in function, 33

sys.system()

built-in function, 33

T

tell() (*File method*), 31

W

write() (*File method*), 32