Programming in Lensor Glint

lens_r

November 15, 2025

Contents

1 Programming in Lensor Glint **2**
1.1 Introduction, Overview . . . 2
1.1.1 History . . . 2
1.1.2 Example Program . . . 3
1.1.3 Program Structure . . . 3
1.1.4 Glint Exercise . . . 4
1.2 Simple Values . . . 4
1.2.1 Integers . . . 4
1.2.2 Bytes . . . 5
1.2.3 Boolean . . . 5
1.2.4 Strings . . . 5
1.3 Variables . . . 5
1.3.1 Declaration Forms . . . 6
1.3.2 Type Decay . . . 7
1.3.3 Scopes . . . 7
1.3.4 Glint Exercise . . . 8
1.4 Assignment and Basic Expressions . . . 8
1.4.1 Assignment . . . 8
1.4.2 Arithmetic . . . 8
1.4.3 Bitwise . . . 9
1.4.4 Implicit Casting . . . 9
1.4.5 Explicit Casting . . . 9
1.4.6 Glint Exercise . . . 10
1.5 Glint's Print . . . 10
1.5.1 Glint Exercise . . . 10
1.6 Arrays . . . 10
1.6.1 Fixed Arrays . . . 10

1.6.2 Dynamic Arrays 10
1.6.3 Array Views 10
1.6.4 Initializing Array Views 10
1.6.5 Initializing Fixed Arrays 10
1.6.6 Initializing Dynamic Arrays 10
1.7 Conditionals and Control Flow 10
1.7.1 `if` 10
1.7.2 `while` 10
1.7.3 `for` 11
1.7.4 `cfor` 11
1.7.5 `match` 11
1.8 Declared Types 11
1.8.1 Supplanted Members 11
1.9 Basic Functions 11
1.10 Pointers and References 13
1.11 Templates 14
1.12 Function Overloading 14
1.13 Operator Overloading 15
1.14 Modules 15
1.14.1 Writing A Module Vs A Program 15
1.14.2 Exporting Variables 15
1.14.3 Importing Modules 15
1.15 Example Programs 15
1.15.1 Recursive Fibonacci 15
1.15.2 Euclid GCD 16

1 Programming in Lensor Glint

NOTE: Glint is a language, Lensor is the specification of the language described by this book.

1.1 Introduction, Overview

1.1.1 History

Glint started in 2023 when Lens_r decided he wanted to make a compiler, from scratch, for fun. That compiler needed to compile a language, and so he made one up. At first, it had no name, and was referred to as "the language the FUNCompiler compiles"; as you may have guessed, that grew tedious rather fast, and we needed a name.

That's when Glint got it's name: *Intercept*. Yes, it was named something else for the first year or so. It was renamed Glint when the language itself was cleaned up and made more opinionated toward important design goals, and the language kind of outgrew it's initial version, looking nothing like it used to (and as such, felt deserving of a new name).

The hope is that, even if this language doesn't change your perceptions entirely, or blow your mind, you'll be able to see a glint of promise in an otherwise dark world `:^)`.

1.1.2 Example Program

```
;; Comments may be placed in the source to explain the program and/or make
;; it easier to understand. They begin with ;; and end with a newline or
;; end of file (EOF).

;; Some Glint tooling (namely glintdoc) makes use of special comments that
;; precede declarations. They usually contain @ directives.

foo : int;

;; @returns updated value of 'foo'.
;; @see foo;
getset_foo : int()
  foo := ++foo;

;; The final expression in the program is returned.
69;
```

1.1.3 Program Structure

A Glint program consists of:

Header The first portion of the file designates module status and imported dependencies.

Comments Non-code portions of the file that are used to describe, annotate, and document the code portions of the file. Comments begin at `;;` and go until a newline or EOF.

Expressions Code portions of the file that are used to tell the computer what to do; they are said to "return" a value of some type upon evaluation/execution.

1.1.4 Glint Exercise

```
 1  ;; Declare some variables with initial values.
 2  a : int 42;
 3  b : byte 27;
 4
 5  ;; @returns the sum of parameters x and y.
 6  foo : int(x : int, y : int)
 7    x + y; ;; adds x and y
 8
 9  ;; Invocation of function foo with parameters a and b.
10  foo a, b;
```

The following questions relate to the above example. If you don't know the answer to one, just keep reading and move on, it's okay; you can come back whenever you are ready.

1. On what line does the first *expression* appear?
2. On what line does the comment with an @ directive appear?
3. Is there an expression on line 7?
4. What would the result of the entire program be?
5. Is there a comment on line 7?

Extra credit:

1. How many parameters does `foo` take?
2. What kind of type is `foo` of?

1.2 Simple Values

or, Literals and Constants, or, Self-evaluating Expressions.

1.2.1 Integers

Glint supports decimal, hexadecimal, octal, and binary integer literal formats; all except decimal are accessible through a special prefix that begins with `0`. A decimal number literal may not begin with `0`.

A hexadecimal number literal is prefixed with `0x`. `0xf` has the value of `f` in hexadecimal, or `15` in decimal.

An octal number literal is prefixed with `0o`. `0o25` has the value of `25` in octal, or `21` in decimal.

A binary number literal is prefixed with `0b`. `0b10` has the value of `10` in binary, or `2` in decimal.

1.2.2 Bytes

Glint makes no distinction between character and integer data types (unlike Pascal, and like C).

Glint provides a way to get the encoded byte value of a character as an integer value in the program, and that is to wrap the character in ‘ (back-ticks).

`‘0‘` is equivalent to `48`.

Often, it can be messy to have control characters appear in the source code, so Glint provides an "escape" syntax, such that other, non-control characters may represent the control characters.

Relevant escape characters include `\n` for a line feed, `\\` for a \ (back-slash), and `\t` for a tab stop.

1.2.3 Boolean

Logical values that denote veracity of an expression.

`true` and `false` keywords to access these values.

1.2.4 Strings

A string allows the Glint programmer to include (textual) data directly from the source code into the final program as an array of byte values.

Inescapable strings are wrapped with ’ , and escape-able strings with ".

An inescapable string will contain the contents between the two ’, verbatim. \ is not special within an inescapable string. An inescapable string may span any amount of newlines.

`’foo\n’` -> [102, 111, 111, 92, 110, 0]

An escape-able string accepts the "escape" syntax that byte literals also accept.

`"foo\n"` -> [102, 111, 111, 10, 0]

1.3 Variables

A variable is a name we give to some Glint construct. In most cases, a variable refers to data at some address in memory; as such, it is sort of like

a labeled, specialty box. The box may only contain the type of data the box is declared to contain, and the box may be referenced by it's name. Any value may go in the box, as long as it fits both in the box and through the opening.

A variable is a named location in the computer's memory at the time of evaluation. You may store data at this named location for easier retrieval later on. The data is stored on a computer, and, as such, is in a binary format (just bits). However, Glint makes use of types of data that the hardware cannot understand, allowing the Glint programmer to write more idiomatic source code. You can think of types as an interpretation of the binary bits that are actually on your computer hardware when the code is running. The type of data that is stored at this location is declared by the programmer either explicitly or implicitly in a *declaration expression*.

1.3.1 Declaration Forms

A declaration expression may take on several different *forms*. All of these forms accomplish the same thing (declaring a variable), but in different ways and with different requirements.

The most standard form of declaration expression is an explicitly-typed variable.

```
x : int;
```

This sort of expression may be read as, "x is an integer".

This form may be built upon to include an initializing expression, placed directly after the type expression. In this way, it is equivalent in syntax to an explicit cast expression preceded by an identifier.

```
x : int 69;
```

This may be read as, "x is an integer with the value sixty-nine".

There is another form of declaration, though, that allows us to shorten a lot of code, and reduce duplicated type names in declarations: the type-inferred declaration expression form.

```
x :: 69;
```

This may be read as "x has the value of 69". The type of x will be set to whatever the type of the initializing expression is (in this case, `int`).

Uninitialized variables are default initialized, which, for the most part, means they have a value of zero (except for dynamic arrays, but, we'll get to that).

1.3.2 Type Decay

One of the more confusing parts of any type system is declaration type decay; as long as you aren't doing anything too weird, you'll never run into problems. But, if you are trying to pass functions as parameters to other functions, it becomes relevant. For now, just understand that some types are quite complex in Glint, but quite simple in the underlying implementation, and that means that sometimes you want the simple underlying implementation rather than all the complexities that come with the Glint type (if that makes any sense).

1.3.3 Scopes

A *scope* is a concept of a meaningful collection of declarations. For example, the body of a function is within a new scope, such that any variable declarations are not accessible from outside the function.

A Glint programmer may open a scope manually by using a block expression; the contents of which is within a new scope.

Whenever a new scope is created (or *opened*), it is given a parent scope; if a name lookup doesn't match in the enclosing scope, the parent will be searched until either the name is found or there are no more parent scopes.

```
x : int;

{
  y : int;
}

y; ;; Error! Unknown symbol y'
```

The scope structure of the above program might look something like the following. (NOTE: Simplification for explanation purposes: leaves out global and top level scope).

```
root
|-- x
‘--block
   ‘-- y
```

When we "exit" the block expression, the scope is exited as well; further expressions are once again within the parent scope.

If we were to lookup `x` from the `block` scope, we would not find it in the enclosing scope, search the parent (`root`), and find it. This means that `x` is accessible from the block scope, and it would not be an error to use it there.

On the contrary, if we were to lookup `y` from the `root` scope, we would again not find it in the enclosing scope; but, there is no parent of the `root` scope, so we are done searching. Because we exhausted the search of scopes and did not find a matching declaration, that means this symbol is not valid at this point in the program. That is why the above use of `y`, outside of the block expression, is invalid.

In Glint, shadowing is not allowed, so you cannot actually declare a variable of the same name as another if both would be valid at that point.

```
x : int;
{ x : int; } ;; Error! Redeclaration of 'x'
```

1.3.4 Glint Exercise

1.4 Assignment and Basic Expressions

1.4.1 Assignment

Assignment is the operation of setting a variable's value, such that further accesses to the variable result in the set value. Assignment uses the `:=` operator (like Pascal). Assignment destructively modifies the values in memory associated with the given variable. After assignment, any value that was held in the memory *before* the assignment is destroyed, and can not be accessed.

```
x : int;

;; This is an assignment!
x := 69;
```

Often, you want to use the variable itself in the calculation of the new value of the variable; there are short-hands for lots of operations mixed with assignment.

```
x :: 42;
x += 27; ;; Now, x = 69
```

1.4.2 Arithmetic

Glint programs often need to do basic math; for this, there are the arithmetic operators. These include the standard addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulo (`%`).

1.4.3 Bitwise

On computers, data is represented as bits. Bitwise operations allow you to operate directly on the data bits, rather than on sets of bits with some applied semantics (i.e. an 8-bit signed byte). Bitwise operations include AND, OR, XOR, and NOT.

1.4.4 Implicit Casting

In Glint, the philosophy is that you should have to write less code *when you can*. That means that, as long as everything is clear to the compiler, you don't need to mark up your program with meaning that is already implied. This reduces the code the Glint programmer has to write, and, in turn, reduces chances that they may introduce a bug.

An integer is implicitly convertible to a boolean, and vice versa.

For integer to integer casts, it is valid if the bitwidth we are casting from is less than or equal to the bitwidth we are casting to, OR, if the values are known at compile time, if data is preserved through performing the cast.

A struct is implicitly convertible to any supplanted member.

A fixed array is implicitly convertible to an array view.

A dynamic array is implicitly convertible to an array view.

Functions are implicitly convertible to their corresponding function pointers.

Functions with no parameters are implicitly convertible to their return type.

Pointers are convertible to `void.ptr`.

1.4.5 Explicit Casting

To treat a variable's memory as if it were of another type, you can explicitly cast a value. To do this, simply invoke the type with a single argument of the type you'd like to cast.

To cast integer `42` to `byte`:

```
byte 42;
```

1.4.6 Glint Exercise

1.5 Glint's Print

1.5.1 Glint Exercise

1.6 Arrays

1.6.1 Fixed Arrays

1.6.2 Dynamic Arrays

1.6.3 Array Views

1.6.4 Initializing Array Views

1.6.5 Initializing Fixed Arrays

1.6.6 Initializing Dynamic Arrays

1.7 Conditionals and Control Flow

1.7.1 `if`

`if` evaluates a condition expression, and, if the result is truthy, evaluates a body expression. Optionally, when an `else` clause is present, a different body expression will be evaluated if the result of evaluating the condition expression is falsy.

```
if condition, body;

if condition,
  then
else otherwise;
```

1.7.2 `while`

`while` evaluates a condition expression, and, if the result is truthy, evaluates a body expression. After evaluating the body expression, it will jump back to evaluating the condition expression.

```
while condition, body;
```

1.7.3 `for`

`for` is a loop structure that will iterate over anything with `data` and `size` members (like dynamic arrays or array views). It provides a reference to the corresponding element for each iteration.

```
for element in container, body;
```

1.7.4 `cfor`

`cfor` is a control structure that mimics C's `for`.

```
cfor init; condition; increment; body;
```

1.7.5 `match`

`match` selects a control flow depending on which member of a sum type is held.

`match` must handle every possible member a sum type may have.

```
match value_of_sum_type, {
  .member0 expr;
  .member1 expr;
};
```

1.8 Declared Types

1.8.1 Supplanted Members

1.9 Basic Functions

A *function* is a concept that programming borrows from mathematics.

Remember $f(x)$ in math class? That is a *function application*. What it means is that, `f` is some operation (or sequence of operations) that we would like to apply to `x`. Hence, *application*.

In programming, we've extended math to suit our use-case; variables have types associated with them, for example (not really a thing in math: numbers are numbers).

Functions are no different; programming extends them to better fit the common use-cases.

Just like in math, Glint functions may have any amount of parameters.

Remember `f(x)` from before? `x` is a *function parameter*. `f` is said to be a function with a single parameter. A parameter is a variable (as in, it may

vary) value that may change upon different invocations of the function. You may think of it like a name for a placeholder value that will be filled in at some later point.

On Parameters vs Arguments Parameters appear in the function declaration and definition, while arguments appear in each function invocation. An argument is the value that a parameter takes on during a particular invocation of a function.

Functions do not *just* execute an operation, or sequence of operations, though. They apply operations in order to form a result value, which is then said to be "returned from" the function, upon application. In Glint, the result of evaluating the last expression in a function is the return value, unless otherwise explicitly returning via `return`.

A function's return value is constrained to be of the declared return type of the function (such that uses of the function may rely upon a binary format of the returned value).

Basic functions allow the programmer to de-duplicate code, increase readability, and form custom control flow.

You should write a function when you find yourself writing the same set of expressions over and over, with just a few values swapped out for another in each set. Instead, extract that set of expressions into a function body, then invoke that function each time you need to do that set of operations.

Basic function example:

```
foo : [int 2] !{ 69, 42 };
bar : [int 2] !{ 27, 42 };
baz : [int 2] !{ 64920, 42 };

@foo[1] *= 10;
@bar[1] *= 10;
@baz[1] *= 10;
```

With a function:

```
foo : [int 2] !{ 69, 42 };
bar : [int 2] !{ 27, 42 };
baz : [int 2] !{ 64920, 42 };

ten_times_second : void(x : [int 2])
  @x[1] *= 10;
```

```
ten_times_second foo;
ten_times_second bar;
ten_times_second baz;
```

As you can see, when proper names are picked, functions increase the ability for code readers to understand what the code is trying to do (vs what it may actually be doing). They also allow for easier refactors; imagine you want to also multiply the first value of every array by 3. That would be one expression, one extra line of Glint code to write in the function body. Without a function, you'd have to write and edit that line of code where every invocation is! For this simple example, it would be doable, but, with a large and growing codebase, that becomes more and more difficult.

1.10 Pointers and References

We've talked a little about how variables (for the most part) are named locations in the computer's memory. When we say memory, we mean it's RAM (*Random Access Memory*). RAM is (usually) a linear set of addressable bytes that the CPU may read from and write to.

When we say *addressable*, we mean that there is a unique way of referring to each and every byte within that linear set. Mostly, this is done simply by index, and we call this index a *memory address*.

A *pointer* is simply a memory address; an index into that linear set of bytes that the CPU can do whatever it wants with. In Glint, pointers have an associated underlying type; the type tells Glint how to interpret the value in memory at that memory address. So, a value of type `int.ptr` (a pointer to an integer) will be some index into the linear memory array; we don't really care what it is. If we were to `load` the value at that memory address (technically the next eight values, since an `int` is eight bytes), we would then have a value of type `int`.

In Glint, subscripting is the act of offsetting a pointer by some multiple of the size of a given type. So, to continue with our last example, subscripting a value of type `int.ptr` by one would increase the value by eight, since that is the size in bytes of an integer. The idea is, the first subscript of a pointer gets the address where the next element in contiguous memory would be, if there were one. This applies to every value, not just the first, so the zero-eth subscript leaves the pointer unchanged, while the second subscript increases the pointer by the size of two elements (sixteen bytes in the case of `int.ptr`).

Do keep in mind that, unlike most other languages, Glint's subscript does **NOT** dereference. That is, the act of subscripting does not load any values

in Glint; it simply calculates the memory address. In order to load a value from a memory address, you have to dereference it. In Glint, that is done via the unary prefix operator `@` (i.e. get the value "at" the given memory address).

1.11 Templates

Templates, at their core, are expressions which evaluate to other expressions.

A template is a special form of expression that results in an expression upon evaluation. You can think of it as code that generates code.

This is often useful when you have some expression (or set of expressions) that end up being duplicated throughout a function, with just a few values swapped out for different ones here and there.

First, let's look at the most basic template expression: the *identity* template.

```
template(x : expr) x;
```

This template expression, when invoked with an expression, would evaluate to that argument expression unchanged.

Often, you want to use a template more than once; for convenience, a Glint programmer may name a template expression. This may also improve readability. Remember that templates are NOT functions! They are their own type of invocable (callable) expression.

```
identity :: template(x : expr) x;

;; Evaluates to '69420'.
identity 69420;
```

1.12 Function Overloading

Glint supports *function overloading*, which is the idea of giving multiple definitions to a single, named function; each definition differs in their signature (return type and/or parameter types), and the compiler determines which definition to call at each call site based on the types of the argument expressions.

```
;; [0]
make_byte : byte(v : int)
  byte v;
```

```
;; [1]
make_byte : byte(v : byte)
  v;

make_byte 69; ;; calls [0]
make_byte '0'; ;; calls [1]
```

1.13 Operator Overloading

1.14 Modules

1.14.1 Writing A Module Vs A Program

A Glint program is a collection of Glint source code that may be compiled and linked to an executable. A Glint program may import Glint modules.

A Glint module is a collection of Glint source code that does not compile to executable, but rather a library. This library then may be used by a linker to include it's code in a separate executable. This library is also used by the Glint compiler to resolve the Glint module's metadata that produced it when compiling code that imports the module.

Programs and modules may import modules, but neither programs nor modules may import programs.

1.14.2 Exporting Variables

1.14.3 Importing Modules

1.15 Example Programs

1.15.1 Recursive Fibonacci

```
fib : uint(n : uint) {
  if n <= 1, return n;
  (fib n - 1) + (fib n - 2);
};

;; Calculate 7th number in the Fibonacci sequence, and return that.
fib 7;
```

Writing this example actually caught a bug in the compiler implementation regarding IR generation; see commit 9d152973, if interested.

1.15.2 Euclid GCD

```
gcd : uint(a : uint, b : uint) {
  if b = 0, return a;
  gcd b, a % b;
};

;; Calculate the greatest common denominator of 32 and 80, and return
;; that.
gcd 32, 80;
```