Programming in Lensor Glint

lens_r

November 14, 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 . . . 3
1.2 Simple Values . . . 3
1.2.1 Integers . . . 4
1.2.2 Bytes . . . 4
1.2.3 Booleans . . . 4
1.2.4 Strings . . . 4
1.3 Variables . . . 5
1.3.1 Declaration Forms . . . 5
1.3.2 Type Decay . . . 6
1.3.3 Glint Exercise . . . 6
1.4 Assignment and Basic Expressions . . . 6
1.4.1 Assignment . . . 6
1.4.2 Arithmetic . . . 7
1.4.3 Bitwise . . . 7
1.4.4 Implicit Casting . . . 7
1.4.5 Explicit Casting . . . 7
1.4.6 Glint Exercise . . . 7
1.5 Glint's Print . . . 7
1.5.1 Glint Exercise . . . 7
1.6 Arrays . . . 7
1.6.1 Fixed Arrays . . . 7
1.6.2 Dynamic Arrays . . . 7

1.6.3 Array Views . . . 7
1.6.4 Initialing Array Views . . . 7
1.6.5 Initializing Fixed Arrays . . . 7
1.6.6 Initializing Dynamic Arrays . . . 7
1.7 Conditionals and Control Flow . . . 7
1.8 Declared Types . . . 7
1.8.1 Supplanted Members . . . 7
1.9 Basic Functions . . . 7
1.10 Pointers and References . . . 9
1.11 Templates . . . 10
1.12 Function Overloading . . . 10
1.13 Modules . . . 10
1.13.1 Writing A Module Vs A Program . . . 10
1.13.2 Exporting Variables . . . 11
1.13.3 Importing Modules . . . 11
1.14 Example Programs . . . 11
1.14.1 Recursive Fibonacci . . . 11
1.14.2 Euclid GCD . . . 11

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:

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

1.2 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 \ (backslash), and `\t` for a tab stop.

1.2.3 Booleans

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.

Inescapeable strings are wrapped with `’` , and escapeable strings with `"`.

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

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

An escapeable 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 labelled, 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 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).

```
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

1.4.5 Explicit Casting

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 Initialing Array Views

1.6.5 Initializing Fixed Arrays

1.6.6 Initializing Dynamic Arrays

1.7 Conditionals and Control Flow

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.

1.12 Function Overloading

1.13 Modules

1.13.1 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.

1.13.2 Exporting Variables

1.13.3 Importing Modules

1.14 Example Programs

1.14.1 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.

1.14.2 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;
```