Programming in Lensor Glint

lens_r

November 12, 2025

Contents

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 \ (back-slash), 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’ -> [102, 111, 111, 92, 110, 0]

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

"foo" -> [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 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.

1.10 Pointers and References

1.11 Templates

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