Programming in Lensor Glint

lens_r

February 12, 2026

Contents

1 Preface

1.1 Hardware

Glint is a programming language. This book is meant to teach it (at least somewhat). A programming language is not useful in a vacuum (literally or figuratively); a programming language is a language that allows a programmer to instruct a computer with what to do (and when to do it).

A computer is a broad term, but, for the most part, simply refers to a machine that does "work" via a sequence of operations on binary (on or off) values. This sequence of operations is called *code*. A collection of code that a computer may run is called a *program*. So, a program is a collection of code, and code is a sequence of operations that tells the computer what to do.

A program *starts* at a specific point (usually called the entry point). A program is not required to end explicitly, and there are various routes through which a program may end up not running anymore. A program is said to be *running* on a given computer when operations from that program's code are currently executing on that computer.

A computer has to do operations *on* something. That is, an addition operation isn't useful unless you are actually operating on meaningful values. In the real world, computers have all sorts of different hardware to make this happen (i.e. CPU registers, ALUs, etc.), but we won't get into all of that right now.

For now, you just need to learn about something called *memory*. A computer has to operate on values, and store the result somewhere; it also

has to store code, and keep track of it's place. Computers often store those values in their *memory* (so they can recall it later).

A computer's memory may be implemented in the real world in all sorts of ways with differing amounts of limitations (i.e. size, speed, etc.), but the most important takeaway is that, while a computer's memory is made up of binary *bits* (values that may only be on or off), it is addressed via *bytes*. In hardware, a byte (the minimum addressable unit in memory) is *usually* eight bits.

```
MEMORY
| off on  off off off on  off on | ...
| BYTE 0                         | BYTE 1
```

In the example above, each `off` and `on` represents a bit (an actual stored value in memory that is either `on` or `off`). Every eight bits is labeled as a `byte`. If we were to access any value seen above within the computer's memory, we would do so by first *loading* the zero-eth byte from memory.

To `load` from memory means to copy the relevant set of bits from memory. Loading requires a memory address: the index of the byte we'd like to load.

The opposite of *loading from* memory is *storing into* memory. A value (or values) may be stored into memory, such that further loads from that memory result in the stored value(s). Because loading and storing both work with memory, they have the same limitations of not working directly on bits, but on sets of bits called *bytes*.

The operations that a computer performs (or, the *code* that a computer *executes*) operate directly on sets of bit values (on or off). These values may be written to or loaded from memory for later use/re-use.

As we touched on earlier, even a computer's code is stored in it's own memory. That is, a computer's operations are encoded in a binary format (such that they may be represented by values that may only be `on` or `off`), and those binary values are stored into the computer's memory. When the computer is started, the computer loads an operation from it's memory, executes it, and repeats. The computer keeps track of where in it's memory it is executing via a *program counter*. The program counter is also sometimes known as an *instruction pointer*. The program counter simply keeps track of which byte within memory the current operation is stored at. When the operation is finished, the computer may look at the next byte within memory to find the next operation.

There is also a more permanent form of memory known as *storage*. A computer's storage is, like memory, a set of binary values (on or off). In an

ideal world, memory and storage wouldn't be different. In the real world, storage is much slower (as it solves a different physical problem; it doesn't just store bit values while the computer is on, it stores them even without power). For the most part, computer *files* are meaningful, named portions of a computer's storage.

1.2 Software

Now that you know what a computer is, and a little bit about how a computer functions, we can now learn some basic conventions about how all modern computers use that hardware to do useful things.

Most computers come with code in their memory that may not ever be changed; it is locked into the hardware (physical parts of the computer machine). This code is called *firmware*, since it is bound to the physical hardware, yet it is still able to be changed without building a new physical chip. A computer's firmware usually handles doing initialisation procedures when the computer turns on, getting the computer to a known state. It does this so that the user of the computer may write software with a known starting state (it'd be quite difficult to guess where the computer has started executing code at, if the program counter's value was random every time the computer was started).

That code that the user runs on their computer is known as *software*. Most computers these days come with a conventional piece of software known as an *operating system* (often abbreviated *OS*). An *OS* is software that deals with the specifics of the particular hardware, such that *it* may provide a *more* known starting state to *other* software. An OS usually consists of a kernel program that "talks" to the hardware; in reality, it does whatever is needed to make hardware do a given operation whenever software requests that operation be completed (no actual "talking" involved).

The programs that an OS runs are usually called "user programs", and the known state that the OS provides to software called "userspace". User programs are said to run within userspace.

Glint is a programming language meant to write user programs, so, Glint code runs within userspace of some OS on some computer. This is important to understand how to get the computer to do what you want it to do. *If you want to shoot an arrow and hit a target, you have to know where you are standing.*

1.3 Tooling

Now, the weird part is that we actually *use* computers to do (or at least help with) the work of producing code. This means there is actually *software* we use when writing *software*.

One such piece of software is a *compiler* program. A compiler produces code in a file, given some input. The format that a compiler inputs and outputs is not set in stone, and varies depending on your particular OS, hardware, etc. A compiler is usually referred to by the *language* the compiler accepts as input. For example, a *C compiler* would be a compiler that takes as input the language *C*, and outputs code in some other (possibly machine-ready) format.

A compiler will often tell you when a program is ill-formed; a good compiler will tell you where and how. An ill-formed program is like a sentence that doesn't follow grammar rules: the sentence will never convey any meaning until the rules are followed, as it is fundamentally flawed. Some problems in programs mean the program will never work; these are usually called *errors*. Other problems are just things that are likely to be misinterpreted or often cause confusion (like a sentence that starts with "Because"); these are usually called *warnings*.

The file a compiler produces is called an *object* file. An object file is a type of file that stores code and data, and necessary information about that code and data.

An object file must be turned into an executable file via a *linker* program before a computer may load it into memory and actually execute the operations. An executable file is just a binary format that an OS supports loading, like *the Executable and Linkable Format (ELF)*. A linker's main job is to convert named portions of memory to actual memory addresses.

2 Programming in Lensor Glint

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

RECOMMENDATION TO READERS: It is my belief that you will learn faster, and better understand the concepts presented, if you have a computer with *some* implementation of a Glint compiler present on the system. As you read, test yourself by trying to write Glint programs that you believe to follow the rules as you currently understand them. The feedback the compiler gives you when trying to build the programs you write is invaluable in quickly correcting small errors in your current mental model.

I recommend GNU Emacs (or Vim) as an editor, but you may use whatever editor tooling you find the most comfortable.

2.1 Introduction, Overview

2.1.1 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 `:^)`.

2.1.2 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).

;; Variable Declaration
my/array : [byte];

my/array += ‘8‘; ;; append
my/array ~= ‘5‘; ;; prepend

;; Iteration
for b in my/array,
  b += 1;

;; Emit "69".
print my/array, ‘\n‘;

;; Some Glint tooling (namely glintdoc) makes use of special comments that
```

```
;; precede declarations. They usually contain @ directives.

;; @returns the answer to the universe.
my/cool-function! : int()
  42;

;; Emit "420".
print 10 my/cool-function!, `\n`;

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

2.1.3 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.

2.1.4 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?

2.2 Simple Values

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

2.2.1 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.

Integer literals are always of `int` type.

2.2.2 Fractionals

Glint supports decimal fractional literal formats. The whole part of a decimal fractional literal may not begin with `0`.

```
0.0;
0.5;
1.0;
1.0005;
82921.5;
12398.812;
```

Fractional literals are always of `float` type.

2.2.3 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.

Byte literals are always of `byte` type.

2.2.4 Boolean

Logical values that denote veracity of an expression.

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

Boolean values are always of `s1` (one-bit signed integer) type.

2.2.5 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 ’ (U+0027, apostrophe), and escape-able strings with " (U+0022, quotation mark).

An inescapable string will contain the contents between the two ’ (apostrophes), 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]

For historical reasons, and compatibility with other languages and tooling, Glint strings have a NUL byte (value of zero) appended. So, the empty string literal ’’ contains the bytes `[0]`, and is of type `[byte 1]`.

String literals are always a fixed array of `byte`, with the length set long enough to store the string and the implicit NUL byte.

2.3 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*.

2.3.1 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).

2.3.2 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).

2.3.3 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'
```

The following is a non-exhaustive list of expressions which have containing expressions that are parsed in a different scope.

- Block Expression
- If Expression
- Else Clause
- For/CFor
- Function Body
- Each Match Expression

2.3.4 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
```

2.3.5 Glint Exercise

Exercise Mission

- Recognize declarations of any form.
- Differentiate variable names, types, and initializing expressions.
- Reason about the values variables have at different points in the control flow of the program.

```
x : int;
y : int = 420;

;; y := y - x;
y -= x;

x := 351;

z :: y - x;
```

The following questions relate to the above example.

1. What is the name of the first variable declared?
2. How many variables are declared?
3. What is the value of variable 'y' when the program starts, and when the program finishes?
4. What is the initial value of 'z'?

5. What is the type of 'z'?

6. What is the value of the variable 'x' when the program starts, and when the program finishes?

2.4 Basic Expressions

2.4.1 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 (`%`).

2.4.2 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.

2.4.3 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`.

2.4.4 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;
```

2.4.5 Glint Exercise

TODO

2.5 Glint's Print

Often, a program will need to "print" values, such that a human running the program may see them. We still use the word "print", but we are not referring to printing onto paper; now-a-days, we have software terminals which are much faster and cheaper to run than a dedicated printer `:^)`.

In `Glint`, the `print` keyword acts as a pseudo-function, and inserts code for each argument passed to it according to the type of the argument.

- `void` arguments signal an error.
- `byte` arguments get printed immediately using `putchar` or similar.
- `[byte]`, `[byte view]`, and fixed byte arrays (i.e. `[byte 4]`) arguments get each byte printed using `putchar` or similar.
- Otherwise, a call to `format`, a function that returns a dynamic byte array, is inserted, the contents of that returned dynamic byte array are printed, and then the dynamic array is freed.

Since byte literals are of `byte` type, they get printed "verbatim". The following Glint program would print `420` and then a newline (line feed character).

```
print '4', '2', '0', '\n';
```

2.5.1 Glint Exercise

```
1  x :: 42 + 27;
2  print x, " gallons of milk on the wall...\n"
```

1. What is the output of the above program?
2. What *would* the output of the above program be, if `x` were initialized to `3`?

2.6 Arrays

An array is a (contiguous) collection of a given element type.

A fixed array has the amount of elements known at compile-time. This is like arrays in the C language.

A dynamic array has the amount of elements known at runtime. This is like vectors in the C++ language.

An array view is a non-owning view of a contiguous collection of elements of a given type.

2.6.1 Fixed Arrays

A fixed array has the amount of elements known at compile-time. This means that the compiler is able to check every compile-time known subscript to be in range. You should use this type of array if the amount of elements never changes.

The type of a string literal is a fixed array with an element type of `byte`. The size of the fixed array is the amount of bytes needed to contain the contents of the string literal, plus one for the implicit terminating NUL byte.

2.6.2 Dynamic Arrays

A dynamic array may have any amount of elements; any given instance of a dynamic array type stores a value regarding it's size.

A dynamic array's `data` pointer is initialized using the return value of `malloc` (A function provided by the C standard library). Glint is (mainly) intended to write userspace programs, and as such defaults to defining `main` (such that a C runtime may call it). Therefore, Glint implementations may rely on the presence of any C runtime functions (like `malloc`) to implement their features.

That's right! Dynamic arrays are *always* default initialized, and `data` *always* points to some memory of size `capacity * byte_size_of_element_type`.

Therefore, the elements of a dynamic array are available via a subscriptable pointer at the `data` member. The first element of dynamic array `d` is accessible via `@d.data[0]`; there is a short-hand that subscripting a dynamic array directly is equivalent to subscripting it's `data` member. That means the first element is usually accessed using `@d[0]`.

You can think of a dynamic array type as a shorthand for defining the following struct, where `T` is the element type of the dynamic array.

```
[T];
;; equivalent to
struct {
  data : T.ptr;
  size : uint;
  capacity : uint;
};
```

Where `size` is the first invalid index at `data`. `capacity` refers to the size of memory located at `data`. If the memory at `data` is not large enough to store an inserted element, `data` is reallocated and all of the old elements are copied to the new memory.

An element may be prepended to (added to the front of) a dynamic array using the `~=` operator.

```
d : [byte];
d ~= ‘9‘;
d ~= ‘6‘;

;; Outputs 69, then a newline
print d, ‘\n‘;
```

An element may be appended to (added to the back of) a dynamic array using the `+=` operator.

```
d : [byte];
d += ‘6‘;
d += ‘9‘;

;; Outputs 69, then a newline
print d, ‘\n‘;
```

Finally, when a dynamic array is declared, memory is allocated; that means that, after the dynamic array is done being used, the programmer must free it by using the unary minus operator `-`.

```
d : [byte];
d += ‘6‘;
d += ‘9‘;

;; Outputs 69, then a newline
print d, ‘\n‘;

;; Free the memory associated with the dynamic array.
-d;
;; Any further uses of ’d’ would be a program error reported by the
;; compiler.
```

A dynamic array is compatible with a ranged for loop expression (more on that later).

2.6.3 Array Views

You can think of a view type as a shorthand for defining the following struct, where `T` is the element type.

```
[T view];
;; equivalent to
struct {
  data : T.ptr;
  size : uint;
};
```

A view is meant to act as the minimal representation of a contiguous set of elements of a given type; it contains a pointer to the contiguous memory where all the elements are stored, as well as a value denoting how many elements are stored there.

A view is compatible with a ranged for loop expression (more on that later).

2.7 Conditionals and Control Flow

2.7.1 `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;
```

2.7.2 `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;
```

2.7.3 `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;
```

2.7.4 `cfor`

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

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

2.7.5 `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;
};
```

2.8 Declared Types

2.8.1 Structs

The most important declared type there is, the `struct`. A struct is a way to group other types together in memory, and reference each type's value by a name.

```
struct {
  x : int;
  y : uint;
};
```

Given an instance of the above struct, setting the value of `x` would **not** alter the value of `y`, and vice versa. They are unique, named points in memory.

To access these points, you use the member access operator, `.`, with the object you are trying to access on the left hand side, and the name of the member you are trying to reference on the right hand side. A member access, as it references a unique point in memory, may be assigned to.

```
A :: struct {
  x : int;
  y : int;
};

foo : A;

foo.x := 70;
foo.y := -1;

foo.x + foo.y;
```

1. Supplanted Members

   A struct may have another struct (or sum) type supplanted within it; this means that the supplanted type's members are now accessible as if they were members of the struct the member resides in. In reality, the compiler does generate a unique name for the member and generates code to access that member by name; this construct is simply for the programmer's convenience.

```
A :: struct {
  x : int;
  y : int;
};

B :: struct {
  supplant A;
  z : int;
};

foo : B;
foo.x;
foo.y;
foo.z;
```

2.8.2 Unions

A union is a way to store multiple types in one place in memory. A union is only as big as it's largest member.

```
union {
  x : int;
  y : uint;
};
```

Given an instance of the above struct, **setting the value of `x` would alter the value of `y`**, and vice versa. They *are* named points in memory, but they are not unique. You could say a union gives multiple names to the same position in memory.

If you can't picture why that'd be useful, then you don't ever need to use unions; they are really just there so that people who want to solve problems a certain way are able to do it that way.

2.8.3 Enumerations

Enumerations, often shortened to just *enums*, are named constant values.

Let's say you wanted the number `1` to be called `Success`, and the number `7` to be called `Erased`, to make your Glint code more readable, or understandable.

```
Code :: enum(int) {
```

```
  Success :: 1;
  Erased :: 7;
};

Code.Success; ;; = 1
Code.Erased; ;; = 7
```

Enumerators (in Glint) are *just named constants*. As such, the values of an enumerator must be known at compile time. The underlying value type may be specified within parentheses.

If you come from Rust (you crustacean, you), see sum types for what you may call enums. If you come from C or C++, not too much has changed.

2.8.4 Sums

A sum is a way to store **one of** a group of types in one position.

A sum type is a lot like a union, except that when you access a member of a sum type, it is verified that the member you are accessing is *actually* currently stored in the memory where that member is stored. You see, each member of a sum type is stored at the same position in memory (like a union), but the sum type also keeps track of what member was last assigned to, and, as such, what kind of value is stored in it's member memory.

```
sum {
  x : int;
  y : int;
  z : byte;
};
```

Given an instance of the above struct, **setting the value of `x` would alter the value of `y`**, and vice versa. The catch is that, you would never be able to access `y` after setting `x`; doing so would actually cause the program to exit.

```
A :: sum {
  x : int;
  y : byte;
};

foo : A;
foo.x; ;; program exits here
```

The only way to get your Glint program to continue execution when accessing a sum's member is to access the *right* one. The *right* one is whichever one is currently stored in the sum type. Note that this is NOT calculated at compile-time; each sum type stores a value in the computer's memory *at run-time* indicating which member is valid (if any). The compiler automatically generates code to update this value accordingly whenever assigning to any sum type's member.

```
A :: sum {
  x : int;
  y : byte;
};

foo : A;
foo.x := 69;
foo.y; ;; program exits here
```

```
A :: sum {
  x : int;
  y : byte;
};

foo : A;
foo.x := 69;
foo.y := foo.x;
foo.y;
foo.x; ;; program exits here
```

The members of a given instance of a sum type may not co-exist at any one time. However, over the "lifetime" of an instance of a sum type, any amount of members may be valid.

You may query if a given member is valid using the `has` unary prefix operator.

```
A :: sum {
  x : int;
  y : byte;
};

foo : A;
```

```
foo.x := 69;
foo.y := foo.x;

out : int;

if (has foo.x), out := foo.x;
if (has foo.y), out := foo.y;

out;
```

To avoid long chains of `if` expressions used in conjunction with `has`, as seen above, the `match` expression may be used in turn. In fact, there are more benefits to using `match`, such as compile-time guarantee that every possible member has a control flow to handle it.

```
A :: sum {
  x : int;
  y : byte;
};

foo : A;
foo.x := 69;
foo.y := foo.x;

match foo, {
  .x out := foo.x;
  .y out := foo.y;
};
```

The match expression will execute exactly one of it's named body expressions: the one matching the name of the value currently stored in the instance of the given sum type.

The program would not compile if there were no body expression named `x` or if there were no body expression named `y`; every declared member within the sum type must have an expression associated with it in a `match` expression on that sum type.

2.9 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.

NOTE: Function arguments are evaluated "in order"; that is, the order they appear *in the function's type declaration*.

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).

NOTE: There is a special built-in type named `void` which declares the absence of a valid binary value. Therefore, it is an incomplete type and uses of it are invalid. However, it is most used as the return type of a function that doesn't return anything; that way, the compiler knows not to expect a value to return, and you don't have to return an `int` for no reason or something (making future users of your module question why in the world you are returning zero from a function every time).

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.

2.9.1 Function Attributes

2.10 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).

2.11 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;
```

2.12 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]
```

2.13 Operator Overloading

Operators are tokens, or sequences of tokens, that may be placed somewhere around an expression in order to alter the result of the expression in some way. Operators may be defined by how many inputs they operate on (*arity*), as well as where the operator is placed in relation to the inputs (prefix, infix, or postfix). An operator that comes before the single expression it operates on would be a *unary prefix operator*. An operator that sits between the two expressions it operates on would be a *binary infix operator*.

For simple types like pointers or integers, an operator will map quite neatly to one (or a minimal amount of) machine instruction(s), so there is

really no advantage to defining functions for these operations; instead, we insert the instructions directly.

However, in Glint, a programmer may declare their own types, as well as variables with values of those declared types. The programmer may want to take advantage of these operators for their own types; to accomplish this, Glint inserts calls to specially-named functions when these custom types are used in these operators that normally wouldn't support them. The programmer may then define these functions themselves, customizing the code that runs when these operators are used.

TODO: syntax example

2.14 Templated Types

A templated type is a template expression that defines a type.

```
vector :: struct (T : type) {
  data : T.ptr;
  size : uint;
  capacity : uint;
};

my_vec_of_int : vector int;
```

2.15 Templated Functions

A templated function is *like* a template expression that defines a function declaration.

The catch here is that the template arguments are all types (because, well, templating a function on a value seems an awful lot like you may just want to use a parameter instead `:^)`). Because of that, the syntax has removed the `(T1 : type, T2 : type)` boilerplate in favor of automagically generating them from the amount of unique `auto` types in the signature.

The `auto` type allows for partial definition of a type, just like C++20's concepts.

```
vector_push : void(vec : (vector auto.element).ref, val : auto.element) {
  if vec.size + 2 > vec.capacity,
    vector_grow vec;

  @vec.data[vec.size] := val;
  vec.size += 1;
```

```
};
```

Now, upon invocation of `vector_push`, the compiler will first look for a defined function with the signature matching the given arguments (i.e. `auto.element` is `int`). If it exists, it will call it. If it doesn't, it will use the templated function's body as a template body and expand it with the given type(s). It will then call that defined function.

```
vector_push :: template(element : type)
  vector_push_impl : void(vec : (vector element), val : element)
    ...;

;; Invocations could be thought of like this
(vector_push int) my_vec_of_int 69;
```

Basically, a templated function is shorthand to define a template that defines a unique function for every unique set of types passed in place of `auto` in the parameters.

2.16 Modules

2.16.1 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.

2.16.2 Exporting Variables

A Glint module may export any amount of variables, such that those variables are accessible by any Glint program or module that imports the module. This is done via the `export` keyword, a storage specifier. Storage specifiers come before the name, within a declaration expression.

```
module Foo;

export foo : int 69;
```

Within Glint, you may now put `import Foo;` at the top of a file, and subsequent expressions will execute as if `foo : int 69;` was at the top of the file (but without having to execute any code for that to happen!).

In practice, this most often makes the symbol associated with the variable "visible" to the linker in the generated object file. This means that other object files may reference the symbol to access, operate on, and interact with the variable.

2.16.3 Importing Modules

```
import Foo;

foo; ;; equals int 69
```

When building a Glint program that imports other Glint programs, the compiler needs to know details about the imported module. These details are stored in a built module interface (BMI) known as `.gmeta` files. These files contain Glint module metadata that is loaded by the compiler when a module is imported. This metadata includes the name of the module, exported declarations (types, functions, templates, etc). This metadata is required to be able to properly ensure the Glint program importing the module is using it as expected.

When you build a module `Foo`, the `Foo.gmeta` file will be generated (alongside any output you've asked LCC to generate from the IR that the Glint frontend generates). When building a program that imports Glint module `Foo`, the compiler searches it's include directories (specified via `-I` on the command line) for the `Foo.gmeta` metadata file. If this file cannot be found in any of the include directories, or if it is of an unexpected file format, the Glint compiler issues an error and stops compilation, as it cannot continue to build a program that imports a module it knows nothing about.

3 Types

Just some extra information on the type sub-language. A lot of the first portion of the book goes into expressions, and types are used within them, but the types themselves are not fully expanded upon in one area. That's why this exists.

3.1 Built-in Types

```
;; architecture-sized integer. 64 bits on x86_64, for example.
```

```
int;
;; architecture-sized unsigned integer. Same size as 'int'.
uint;
;; architecture-sized fractional number.
float;
;; Boolean: binary value 'true' or 'false'.
bool;
;; The nothing type. Declares absence of something (i.e. a function that
;; doesn't return anything).
void;
```

3.2 FFI Types

FFI stands for Foreign Function Interface. These types allow you to define variables as if they had a type from another language. For now, C is the only supported "foreign" (read: non-Glint) language by the Lensor Compiler Collection Glint implementation.

```
;; The convention is to take the primitive integer type from C, and
;; prepend the 'c' character, codepoint U+0063. Where 'unsigned' would
;; appear, instead prepend 'cu'.
cshort;     ;; 'short'
cint;       ;; 'int'
clong;      ;; 'long'
clonglong;  ;; 'long long'
;; Unsigned
cushort;    ;; 'unsigned short'
cuint;      ;; 'unsigned int'
culong;     ;; 'unsigned long'
culonglong; ;; 'unsigned long long'
```

3.3 Arbitrary Integers

Instead of having lots of different integer types that relate to different ranges of numbers they may represent, we have a single "arbitrary integer" type where you are able to set the bitwidth manually.

Arbitrary integers may be signed or unsigned.

An unsigned arbitrary integer starts with `u`.

A signed arbitrary integer starts with `s`.

The bitwidth is specified directly following the sign specifier.

Here are some valid arbitrary integer types.

```
s64;
s32;
s16;
s8;

u64;
u32;
u16;
u8;

u5;
s9;
s69;
;; ...
```

3.4 Types with One Element

3.4.1 Pointers

You can make a pointer to any type by appending `.ptr`.

This is a pointer to a signed integer type.

```
int.ptr;
```

Pointers may be loaded from, which yields the underlying type.

```
value : int
  69;
pointer : int.ptr
  &value;
loaded_value : int
  @pointer;
```

Pointers may be stored into, which destrucively modifies the underlying value.

```
value : int
  69;
pointer : int.ptr
  &value;

@pointer := 420;
```

```
value = 69; ;; false!!!
```

As you can see, a pointer doesn't store an `int` value, it stores somewhere an `int` may be found (upon loading from it). This also lets you change the value where that `int` may be found (upon storing into it).

3.4.2 References

You can make a reference to *almost* any type by appending `.ref`.

This is a reference to a signed integer type.

```
int.ref;
```

Like a pointer, a reference doesn't store the underlying value, but a location where that value may be found. The difference is that a reference does not require *explicit* loading and storing via the `@` operator; you can just use it like it's the underlying type.

```
value : int
  69;

reference : int.ref
  &value;

reference -= 27;

value = 42; ;; true!!!

reference_example : uint()
```

3.5 Composite Types

A composite type is a type that is made up of other types; that is, these types allow you to group types together.

3.5.1 Struct Types

A struct type is a sequential set of typed values that may be accessed via a given name. These are called the struct's *members*.

For an instance of the following struct...

```
struct {
  x : int;
  y : uint;
};
```

If `x` was modified, `y` would be unchanged, and vice versa. A struct may have as many members as you could feasibly want, and their types are not required to be unique. You may have two `int` typed values in one struct with different names.

3.5.2 Union Types

A union type is an non-type-safe way of converting between types. A union declaration looks a lot like a struct, but the underlying semantics are very different. For one, a union is only large enough to store it's largest member, and all the members share an address. This means that, if one member is changed, all the others are destructively modified! For most use cases, you should use a sum type instead. But, this is here if you know what you are doing and need to do it your way. You are the Glint programmer, after all, you are in total control.

For a real world use case: use unions to convert between a type and it's binary format easily.

```
union {
  value : int;
  bytes : [byte (sizeof int)];
};
```

3.5.3 Sum Types

A sum type is a type-safe way of storing a group of types in one location. A sum type may only have one of it's members valid at a time, and, each time an access is performed, Glint will safely (and forcefully) exit the program before any memory errors may occur iff the access is not to the valid member.

A sum type's members are much like a union type's: only one member is valid at a given point in the program, and all the members share an address. However, the sum type has a hidden `tag` value that stores which member is currently valid (if any). Whenever an access occurs, this tag may be checked against, and an access to an invalid value will error. This means that sum types do not allow you to convert between types, like a union, but they do allow you to have one variable have "one of a group of" types.

3.6 `sizeof` and `alignof`

`sizeof T` where `T` is any valid type will return the size **IN BITS** of the given type.

`alignof T` where `T` is any valid type will return the size **IN BYTES** of the given type.

3.6.1 On bits vs bytes

Bit alignment is not something that is supported by any real machine or language. By definition, a byte is the minimum addressable unit in any computer. So, when it comes to alignment of an address, bytes are relevant. However, when it comes to the size of a type in Glint, bits are definitely relevant, since you may have a `u27` or a `s42`.

4 Postface

4.1 Example Programs

4.1.1 Hello, World!

```
print "Hello, World!";
```

4.1.2 Factorial

```
factorial : uint(n : uint) {
  result :: 1;

  cfor
      i : uint 1;
      i <= n;
      i += 1;
    result *= i;

  result;
};

;; Calculate the factorial of 5, and return that.
factorial 5;
```

4.1.3 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.

4.1.4 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;
```