C* - Language Proposal

Also written as `cstar` and pronounced **Sea Star**.

Authors

Name	UNI	Role
Shannon Jin	sj2802	Manager
Khyber Sen	ks3343	Language Guru
Ryan Lee	dbl2127	System Architect
Joanne Wang	jyw2118	Tester

Introduction

C* is a general-purpose systems programming language.
It is between the level of C and Zig on a semantic level,
and syntactically it also borrows a lot from Rust (pun intended).
It is meant primarily for programs that would otherwise be implemented in C
for the speed, simplicity, and explicitness of the language,
but want a few simple higher-level language constructs,
more expressiveness, and some safety, but not so many overwhelming
language features and implicit costs like in Rust, C++, or Zig.

It has manual memory management (no GC)
and uses LLVM as its primary codegen backend,
so it can be optimized as well as C, or even better in cases.
All of C*'s higher-level language constructs are zero-cost,
meaning none of those features give it any overhead over C,
which often lead to a highly-optimized style
where in C you would take less efficient shortcuts
(e.x. function pointers and type-erased generics)
and use dangerous constructs like `goto`.
In the future, it may also have a C backend
so that it can target any architecture where there is a C compiler.

While a general-purpose language, C* will probably have
the most advantages when used in systems and embedded programming.
It's expressivity and high-level features combined with
its relative simplicity, performance, and explicitness
is a perfect match for many of these low-level systems and embedded programs.

Language Features

A high-level overview of the important language features of C*.

Expression Oriented

C* is highly expression-oriented. Unlike C,
where many things are only statements, most things in C* are expressions.
Namely,

- Statements evaluate to the unit type `()`.
- Blocks evaluate to their last expression, which could be a statement (and thus `()`) or a trailing expression (with no `;`)
- Functions and closures themselves.
- `if`, `if/else`, `match` are all expressions.
- `for` evaluates to the `break` value, which is usually `()`.

Postfix Everything

Most unary operators and keywords can be used postfix as well.

- `.if {}`
- `.if {} else {}`
- `.match {}`
- `.for {}`
- `.*` for dereference
- `.&` for reference to
- `.&mut` for mutable reference to
- `.!` for negation
- `.$...()` for builtins, such as:
  - `.$cast<T>()` : convert to `T`, like an int to float cast, or an int widening cast
  - `.$ptr_cast<T>()` : cast a pointer like `*T` to `*U`
  - `.$bit_cast<T>()` : reinterpret the bits, like from `u32` to `f32`
  - `.$size_of()` : size of a type or value
  - `.$align_of()` : alignment of a type or value
  - `.$call(args)` : call a function or closure in a unified syntax

Combined with everything being an expression,
`match`, and having methods,
this makes it much easier to write programs in a very fluid style.

Furthermore, and perhaps most importantly in practice,
this makes autocompletion vastly better, because an IDE can narrow down
what you may type next based on the type of the previous expression.
This can't be done with postfix operators and functions (rather than methods).
You get to think in one forward direction, rather than
having to jump from some prefix keywords to some postfix methods and fields.

Algebraic Data Types

C* has `struct`s for product types and `enum`s for sum types.
This is very powerful combined with pattern matching.
`enum`s in particular, which are like tagged unions,
are much safer and correct compared to C unions.
These data types are also fully zero-cost; there is no automatic boxing,
and the same performance as C can be easily be achieved.
Sometimes even better, because the layout of compound types
is unspecified in C*.

For example, you can do this to make a copy-on-write string.

```
struct String {
    ptr: u8&,
    len: usize,
}

struct StringBuf {
    ptr: u8&,
    len: usize,
    cap: usize,
}

enum CowString {
    Borrowed(String),
    Owned(StringBuf),
}
```

Pattern Matching

Instead of having a `switch` statement like in C,
C* has a generalized `match` statement, which can be used to match
many more expressions, including integers (like in C), `enum` variants,
dereferenced pointers, slices, arrays, and strings.
Also, there is no fall-through, but `match` cases can be combined explicitly.

Furthermore, just like you can destructure to
pattern match in a `match` statement, you can also do the same
as a general statement, like in a `let`. It's like an unconditional `match`.

```
let cow = CowString.Borrowed("");
let len = match cow {
    Borrowed(s) => s.len(),
    Owned(s) => s.len(),
};
let String {ptr, len} = "";
```

Note that string literals are of the `String` type similarly defined as above,
and you can redeclare/shadow variables like `len`.

Generics

C* supports generic types and values,
but they are at this point unconstrained.
That is, they are like C++'s concept-less templates.
They are always monomorphic, except when the exact same code can be shared
(no boxing ever). They are not currently higher-kinded.
Types and functions can be generic over both types and values, like this:

```
enum Option<T> {
    None,
    Some(T),
}

struct InlineVec<T, N: u8> {
    array: [T; N],
    len: u8,
}
```

```
struct AllocatedVec<T> {
    ptr: Option<T&>,
    len: usize,
    cap: usize,
}

enum ShortVec<T, N: u8> {
    Inline(InlineVec<T, N>),
    Allocated(AllocatedVec<T>),
}

fn short_vec_len<T, N: u8>(v: ShortVec<T, N>&): usize = {
    v.match {
        Inline(InlineVec {len, _}) => len.$cast(),
        Allocated(AllocatedVec {len, _}) => len,
    }.*
}
```

Non-Null References

C* has references, `T&` and `T&mut`,
but they are always non-null valid pointers.
To express nullability, use `Option<T&>`, which uses
the `0` pointer representation for the `None` variant.
Nullability can also be nested with `Option`, like `Option<Option<T&>>`,
which can't easily be done in C with nullable pointers.

Monadic Error Handling

There are no exceptions in C*, just like C. It uses return values for error handling, similarly to C. But C* has much better support for this using the `Option` and `Result` types.

The definitions of these types are:

```
struct Option<T> {
    None,
    Some(T),
}

struct Result<T, E> {
    Ok(T),
    Err(E),
}
```

That is, `Option` represents an optional value, and `Result`
represents either a successful `Ok` value or an error `Err` value.

There is special syntactic support for using these two monadic types
for error-handling using the `.?` postfix operator in `try` blocks:

```
struct IndexError {
    index: usize,
}

fn get_by_index<T>(a: T[]&, i: usize): Result<T&, IndexError> = {
    if (i < a.len()) {
        Ok(a[i].unwrap())
    } else {
        Err(IndexError {index: i})
    }
}
```

```
fn get_two_by_index<T>(a: T[]&, i: usize, j: usize): Result<(T&, T&), IndexError> = try {
    let first = try {
        get_by_index(a, i).?
    };
    let second = get_by_index(a, j).?;
    (first, second)
}
```

This desugars to

```
fn get_two_by_index<T>(a: T[]&, i: usize, j: usize): Result<(T&, T&), IndexError> = {
    let first = try {
        get_by_index(a, i).match {
            Ok(i) => i,
            Err(e) => break Err(e),
        }
    };
    let second = get_by_index(a, j).match {
        Ok(i) => i,
        Err(e) => break Err(e),
    }
    Ok((first, second))
}
```

As you can see, without the try `.?` operator and `try` blocks,
doing all the error handling with just `match` quickly becomes tedious.
This is also kind of like a monadic `do` notation, except it is in C*
limited to just the monads `Option<T>`, and `Result<T, E>` (over `T`).

Note also that `try` blocks can be specified at the function level
as well as normal blocks.

Uncatchable Panics

While monadic error-handling with `Option` and `Result` is usually superior,
there are still cases where you have unrecoverable errors
(maybe you don't want to handle out of memory conditions),
or where you'd rather just end the program than handle the error.
In this case, you can `panic`, which will
print an error message and immediately `abort`.

To do this with an `Option` or `Result`,
you can just call `.unwrap()`, which will panic if
it was `None` or `Err` and return the `Some` or `Ok` value.

There is no language-supported unwinding.
`abort` is immediately called after a panic, and only the OS cleans things up.
Nothing is stopping you from calling `setjmp` and `longjmp` from C,
but no unwinding of `defer` statements is done,
and it may result in undefined behavior. There is no undefined behavior,
however, in a normal panic because you just simply `abort`.

Defer

To aid in resource handling, C* has a `defer` keyword.
`defer` defers the following statement or block until the function returns,
but will run it no matter where the function returns from

(but not `panic`s/`abort`s) (actually, the `defer` will run when
its block exits, but its easier to just think about function blocks first).

For example, you can use this to ensure
you correctly clean up resources in a function:

```
@extern @abi("C")
fn open(path: u8[]&, flags: i32): i32;

@extern @abi("C")
fn close(fd: i32): i32;

let O_RDWR: i32 = const { 2 };

fn open_file_in_dir(dir: u8[]&, filename: u8[]&): Result<i32, String> = try {
    let mut path = Vec.new(Mallocator());
    defer path.free();
    let path = path.&mut;
    try {
        if (dir.len() > 0) {
            path.extend(dir).?;
            path.push(b'/').?;
        }
        path.extend(filename).?;
        path.push(0).?;
    }.map_err(fn(_) = "alloc error").?;

    let path = path.as_slice();
    let fd = open(path, O_RDWR).match {
        -1 => Err("open failed"),
        fd => fd,
    }.?;
    defer println(f"opened {fd}");
    return fd;
}
```

In this example, you have to allocate a path to store
the directory and filename you combine, and then open
that path and return the file descriptor if it was successful.
You have to clean up the memory allocation, though, and do that
while still handling all the allocation errors and the open error.
The latter can be done elegantly with `try` and `.?`,
but if you mix in the `path.free()`, you'd have to run it before every
error return, which means you have to duplicate it and not use `.?` anymore.

Instead, you can use `defer` for this. No matter where you
return from the function, it will run its statement right before that.
You can also use `defer` for any statement, not just resource cleanup,
like logging for example.

However, sometimes you want to cancel a `defer`:

```
struct FilePair {
    fd1: i32,
    fd2: i32,
}

fn open_two_files(path1: u8[]&, path2: u8[]&): Result<FilePair, String> = try {
    let fd1 = open_file_in_dir(b"", path1).?;
    defer@close close(fd1);
    let fd2 = open_file_in_dir(b"", path2).?;
    defer@close close(fd2);
    println(f"opened {fd1} and {fd2}");
    undefer@close;
```

```
    FilePair {fd1, fd2}
}
```

In this example, you want open two files and return them if successfull.
If only one is successful, though, that's an error and you
should close the first one before returning the error.
In order to do that cleanly, you can use the `undefer` keyword,
which cancels an earlier labeled `defer`, in this case labeled `close`.

`defer` and `undefer` are actually syntax sugar
for something a bit more low-level and wordy:

```
fn open_two_files(path1: u8[]&, path2: u8[]&): Result<FilePair, String> = try {
    let fd1 = open_file_in_dir(b"", path1).?;
    let close1 = fn {fd1}() = { close(fd1); }
    let close1 = close1.$defer();
    let fd2 = open_file_in_dir(b"", path2).?;
    let close2 = fn {fd1}() = { close(fd1); }
    let close2 = close2.$defer();
    println(f"opened {fd1} and {fd2}");
    let close = [close2, close1].&[..];
    close.undo();
    FilePair {fd1, fd2}
}
```

That is, `.$defer()` places the closure on the stack and
returns a `Defer` struct, which can be undone with `Defer.undo()`
(`Defer[]&.undo()` just maps `Defer.undo()` over the array).
`Defer.undo()` sets a bit in the `Defer` struct that it's been undone.
Then when the stack unwinds, any none-undone `Defers` on the stack are run.

Comparison to Destructors

In many other languages, destructors are used for resource handling
instead of defer. This is more uniform, automatic, and safe,
since destructors run automatically when dropped out of scope.
If you have destructors, though, you also need moves in order to do
what we can do with `undefer`, but then you also need ownership,
which C* doesn't track. Furthermore, `defer` is a lot more
explicit and flexible. All the resource cleanup is written explicitly
so there are no hidden costs, which most programmers coming from C
will prefer. And since you can put any statement in a `defer`,
it's much more flexible than destructors.

Methods

C* has associated functions and simple methods,
though these are largely syntactic sugar.
To declare these for a type, simply write:

```
struct Person {
    first_name: String,
    last_name: String,
}

impl Hello {

    fn new(first_name: String, last_name: String): Self = {
        Self {first_name, last_name}
```

```
    }

    fn say_hi1(self: Self) = {
        print(f"Hi {self.first_name} {self.last_name}");
    }

    fn say_hi1(self: Self&) = {
        print(f"Hi {self.*.last_name}, {self.*.first_name}");
    }

    fn remove_last_name(self: Self&mut) = {
        self.*mut.last_name = "";
    }

}

fn main() {
    let mut person = Person.new("Khyber", "Sen");

    {
        person.say_hi1();
        person.&.say_hi2();
        person.&mut.remove_last_name();
        person.say_hi1();
    }
    {
        Person.say_hi1(person);
        Person.say_hi2(person.&);
        Person.remove_last_name(person.&mut);
        Person.say_hi1(person);
    }
}
```

In this example, we first declared a `struct Person`,
and then an `impl` block for `Person` to define
methods/associated functions for it.
Note that this `impl` block can be anywhere, even in other modules.

In the `impl` block, we first declared an associated function `Person.new`,
which is just a normal function but namespaced to `Person`.
Similarly, the other three methods are just normal functions, too,
as seen when we call them explicity in the second block in `main`.
But we can also use `.` syntax to call them,
which just allows us to explicitly name `Person`.

Inside an `impl` block, we can also use the `Self` type
as an alias to the type being implemented.
This is especially useful with generics.

Note that the `.&` and `Self&` are explicit,
because we wan't these kinds of possible costs to be noted explicitly.
For example, `Person.say_hi1` takes `Self` by value,
which means it must copy the `Person` every time.
If `Person` were a much larger struct,
this could be very expensive and we don't want to hide that information.
Also, the difference between `.&` and `.&mut`
is explicit to make mutability explicit everywhere.

Closures

In C*, you can also use anonymous closures.
These are similar to normal functions,
but they can "enclose" over values in the current scope.

For example,

```
impl <T, F> Option<T> {
    fn map(self: Self, f: F): F(T) = {
        match self {
            None => None,
            Some(t) => Some(f.$call(t)),
        }
    }
}

fn main() {
    try {
        let a = Some("hello").map(fn(s) = s.len()).?;
        let b = Some("world").map(fn {a}(s) = a + s.len()).?;
        let c = Some("□🤔").map(fn {n: b}(s) = n + s.len()).?;
        None.map(fn {a: a.&, b: b.&mut, n: c.&mut}(s) = {
            print(f"{s}: {a.*}, {b.*}, {n.*}");
            n.*mut++;
            b.*mut += n.*;
        });
        print(f"{s}: {a}, {b}, {c}");
    }
}
```

These are some example of how to create closures and how to call them.
In particular:

- Closures have a generic, unnamed type.
  So when we take a closure as a parameter, we need to use a generic
  (this is because closure type depend on what they capture).
  You can also apply a type to a function type to get its return type,
  like `F(T)`.
- We can call a closure using the unified calling syntax: `.$call()`.
  Normal function calls are `()`, and we want to be explicit
  when we're actually calling a closure, so `.$call()` is needed.
  `.$call()` also works on normal functions, though,
  since all functions can be implicitly converted to non-capturing closures.
- The closure syntax is very similar to function syntax, with a few differences:
  - The return expression does not have to be a block
    like in normal functions; it can directly use an expression.
    Note that functions effectively just return a block.
    That's how `try` blocks work, for example.
  - Argument and return types are inferred,
    though they can still be specified if you want.
    This is because they are more local,
    and thus documented types are not as necessary.
  - If you want to capture variables,
    you specify an anonymous struct literal before the `fn`.
    This follows the same normal rules for struct literals,
    but you don't have to specify the type, since the type is anonymous.
    Then that struct's fields are available within the closure as variables.

The way closures are implemented is by
creating an anonymous struct of the captured closure context.
Then there is a method on that struct that takes the closure arguments and
returns the closure body with the context struct destructured inside
(so its variables are in scope). This is what is called by `.$call()`.
Note that there are no indirect function calls, boxing,
or allocations involved in this, but it requires the use of generics.

If nothing is captured by a closure, though,
then it can be cast to a function pointer: `fn(T, U): R`,
which can be called indirectly and passed to C over FFI.
The same is true of normal functions.

Slices

C* also has slices. These are a pointer and length,
and are much preferred to passing the reference and length separately,
like you usually have to do in C.

They are implemented like this (not actually, but similarly):

```
struct Slice<T> {
    ptr: T&,
    len: usize,
}
```

But they can be written as `T[]&`. Actually, slices are unsized types,
so their type is just `T[]`, but usually `T[]&` is used and that
is what's equivalent to the above `Slice<T>`.

Unlike references like `T&`, slices can be indexed. By default,
using the indexing operator, this is bounds checked for safety,
but there are also unchecked methods for indexing. Usually, though,
bounds checking can be elided during sequential iteration,
so the performance hit is minimal, and can be side-stepped if really needed.

Slices can also be sliced to create subslices by indexing them with
a range (e.x. `[1..10]` or `[1..]`).
Again, this is bounds checked by default.

Strings

There are multiple types of strings in C* owing to
the inherent complexity of string-handling without incurring overhead.
The default string literal type is `String`, which is UTF-8 encoded and
wraps a `u8[]&`. This is a borrowed slice type and can't change size.
To have a growable string, there is the `StringBuf` type,
but there is no special syntactic support for this owned string.
`String`s are made of `char`s, unicode scalar values, when iterating
(even though they are stored as `u8[]&`). `char`s have literals like `c'\n'`.

Then there are byte strings, which are just `u8[]&` and
do not have to be UTF-8 encoded.
String literals for this are prefixed with `b`, like `b"hello"`
(and for char byte literals, a `b` prefix, too: `b'c'`).
The owning version of this is just a `Box<u8[]>`
(notice the unsized slice use), and
the growable owning version is just a `Vec<u8>`.

Furthermore, for easier C FFI, there is also `CString` and `CStringBuf`,
which are explicitly null-terminated. All other string types are
not null-terminated, since they store their own length,
which is way more efficient and safe.
Literal `CString`s have a `c` prefix, like `c"/home"`.

And finally, there are format strings. Written `f"n + m = {n + m}"`,
they can interpolate expressions within `{}`.
Types that can be used like this must have a `format` method (might change).
Format, or f-strings, don't actually evaluate to a string,
but rather evaluate to an anonymous struct that has methods to
convert it all at once into a real string. Thus, f-strings do not allocate.

Imports

Instead of using a preprocessor with `#include`s like in C, C* uses imports.
Each file is a module of its name, and it can be
imported to use in another file/module, or specific items from that module.
Short modules can also be declared inline with

```
mod name {

}
```

Structural Comments

Besides just using `//` for line comments and `///` for doc comments,
`/-` can be used for a sort of structural comment.
That is, it will comment out the next item,
whether that be the next expression, the next line, or the next function.
`/*` and `*/` can also be used for multi-line and nested comments.

C FFI

C* has no stable ABI, but can easily do C FFI by marking an item
(like a function or a struct) `extern "C"`.
C* constructs are automatically converted to their C equivalents:

C*	C	Notes
`()`	`void`	
`bool`	`_Bool`	
`u8`	`uint8_t`	
`i8`	`int8_t`	
`u16`	`uint16_t`	
`i16`	`int16_t`	
`u32`	`uint32_t`	
`i32`	`int32_t`	
`u64`	`uint64_t`	
`i64`	`int64_t`	
`u128`	`unsigned __int128`	
`i128`	`__int128`	

C*	C	Notes
`usize`	`size_t`	
`isize`	`ssize_t`	
`uptr`	`uintptr_t`	
`iptr`	`intptr_t`	
`f16`	`_Float16`	
`f32`	`float`	
`f64`	`double`	
`f128`	`_Float128`	
`T&`	`*T`	for argument types
`Option<T&>`	`*T`	for return types
`fn(T, U): R`	`R (*)(T, U)`	

There is also a `union {}` type available that
is for FFI with C `union`s. It is unknown which variant is active,
unlike `enum`s, which track that.

Examples

GCD

Here is how you write simple algorithms like GCD in C*:

```
fn gcd(a: i64, b: i64): i64 = {
    (fn gcd(a: u64, b: u64): u64 = {
        b.match {
            0 => b,
            _ => gcd(b, a % b),
        }
    })(a.abs(), b.abs()).$cast(i64)
}
```

Systems Programming

Here is an example program in C* for part of a simple HTTP/1.0 server,
equivalent to part0 of hw3 in Jae's OS class
(https://gist.github.com/RyanLee64/hash-redacted).
It showcases many of C*'s notable features,
like enums, methods, generics, defer, expression-orientedness,
postfix operators, pattern matching, closures, monadic error handling,
and byte, c, and format strings.

That code (the ported part) is ~230 LOC, while the C* below is only ~80 LOC,
and it is more correct in error handling and edge cases,
faster in places (though IO dominates here), and the business logic
stands out more (while less important aspects like errors, resource cleanup,

allocations, and string handling stay in the background).
That is, C* allows you to be simulatenously more expressive
while still staying correct and explicit,
and the performance is just as good if not better.

```
enum Status {
    Ok,
    NotImplemented,
    BadRequest,
    // rest skipped for brevity
}

struct RequestLine {
    method: u8[]&,
    uri: u8[]&,
    version: u8[]&,
}

impl RequestLine {
    fn check(self: Self&): Result<(), Status> = try {
        let Self {method, uri, version} = self.*;
        match (method, version) {
            (b"GET", b"HTTP/1.0" | b"HTTP/1.1") => {},
            _ => Err(Status.NotImplemented).?,
        }
        if uri.starts_with(b'/').! || uri.equals(b"/..") || uri.contains(b"/../") {
            Err(Status.BadRequest).?;
        }
    }
}

fn main(): Result<(), AnyError> = try {
    let (port, web_root) = std.env.argv().match {
        [_, port, web_root] => (port.parse<u16>().?, web_root),
        [program, ...] => Err(f"usage: {program} <server_port> <web_root>").?,
    };
    let server_socket = Socket.new(PF_INET, SOCK_STREAM, IPPROTO_TCP).?;
    defer server_socket.close();
    let server_socket = server_socket.&;
    server_socket.bind(SocketAddr {
        family: AF_INET,
        addr: InetAddr {
            addr: INADDR_ANY.to_big_endian(),
        },
        port: port.to_big_endian(),
    }).?;
    server_socket.listen(5).?;
    let mut request_line_buf = Vec.new();
    defer request_line_buf.free();
    let mut line_buf = Vec.new();
    defer line_buf.free();
    true.while try {
        let client_socket = server_socket.accept().?;
        defer@client_socket_close client_socket.close();
        let mut client_stream = fdopen(client_socket.fd, c"r").?;
        undefer@client_socket_close; // stream (`FILE *` in C) takes ownership
        defer client_stream.close();
        let line_or_status = try {
            // read and parse request line
            let line = client_stream.&mut.read_line(buf.&mut)
                .map_err(fn(_) = Status.BadRequest).?
                .split(fn(b) = " \t\r\n".contains(b)).match {
                    [method, uri, version] => RequestLine { method, uri, version },
                    _ => Err(Status.NotImplemented).?,
                };
            line.&.check().?;
            // read headers, skip them
            true.while {
                client_stream.&mut.read_line(buf.&mut)
                    .map_err(fn(_) = Status.BadRequest).?
                    .match {
                        "\n" | "\r\n" => break,
```

```
                        _ => {},
                    }
                }
                line
            }
            let (line, status) = line_or_status.match {
                Ok(line) => (line, Status.Ok),
                Err(status) => (RequestLine { method: b"", uri: b"", version: b"" }, status),
            };
            client_socket.write(f"HTTP/1.0 {status.code()} {status.reason()}\r\n\r\n").?;
            line_or_status.match {
                Ok(_) => handle_request(web_root, line.uri, client_socket).?,
                Err(_) => client_socket.write(f"<html><body>\n<h1>{status.code()} {status.reason()}</h1>\n</body>
            }
            eprintln(f"{client_socket.addr} \"{line.method} {line.uri} {line.version}\" {status.code()} {status.r
        }
    }
```

C* - Language Reference Manual

Github link: https://github.com/kkysen/cstar/blob/main/LRM.md

Table of Contents

Table of Contents

Overview

C* is a general-purpose systems programming language. It is between the level of C and Zig on a semantic level, and syntactically it also borrows a lot from Rust (pun intended). It is meant primarily for programs that would otherwise be implemented in C for the speed, simplicity, and explicitness of the language, but want a few simple

higher-level language constructs, more expressiveness, and some safety, but not so many overwhelming language features and implicit costs like in Rust, C++, or Zig.

It has manual memory management (no GC) and uses LLVM as its primary codegen backend, so it can be optimized as well as C, or even better in cases. All of C*'s higher-level language constructs are zero-cost, meaning none of those features give it any overhead over C, which often lead to a highly-optimized style where in C you would take less efficient shortcuts (e.x. function pointers and type-erased generics) and use dangerous constructs like goto. In the future, it may also have a C backend so that it can target any architecture where there is a C compiler.

While a general-purpose language, C* will probably have the most advantages when used in systems and embedded programming. It's expressivity and high-level features combined with its relative simplicity, performance, and explicitness is a perfect match for many of these low-level systems and embedded programs.

Table of Contents

A C* Program

A C* program is a top-level C* module.

Note that italics will be used here to refer to
placeholders for language items, not the items themselves.

Modules

Every C* file (by default using a `.cstar` extension)
must be UTF-8.
Each file is implicitly a module, though modules can also be declared
inline with the `mod` *`name`* `{}` keyword*.
Everything between the braces belongs to the module `name`.

A module is composed of a series of top-level items (aka declarations), which may be one of:

- `use`
- `let`
- `fn`
- `struct`
- `enum`
- `union`
- `impl`

These items may be proceeded by a single *`publicity`* modifier
and any number of annotations.

Comments may also appear anywhere.

C* is not whitespace sensitive, i.e.,
any consecutive sequence of whitespace may be replaced by
any other consecutive sequence of whitespace
without changing the meaning of the program.
A unicode character is considered whitespace if it matches the `\p{Pattern_White_Space}` unicode property.

Table of Contents

Identifiers

Identifiers in C* may be any UTF-8 string
in which the first characters is `_`, `$`, or matches the `\p{XID_Start}` unicode property,
and the remaining characters match the `\p{XID_Continue}` unicode property,
except for the following exceptions:

Identifiers may begin with `$` but are only definable by the compiler as intrinsics.

There are no keywords at the lexer level, but identifiers may not be a C* keyword.
They may also not be the boolean literals `true` or `false`.

`_` is a valid C* identifier at the syntactic level,
but has a special meaning and cannot be used everywhere.
That is, it can only be assigned to.

Examples:

```
// valid identifiers
let validWord: u32 = 2;
fn get_num() = {}
enum 小笼包 {}

// invalid identifier
let 2words = 2;
struct const {}
```

Table of Contents

Keywords

Keywords are reserved identifiers that cannot be used as regular identifiers for other purposes.

C* keywords:

- `use`
- `let`
- `mut`
- `pub`
- `try`
- `const`
- `impl`
- `fn`
- `struct`
- `enum`
- `union`
- `return`
- `break`
- `continue`
- `for`
- `while`
- `if`
- `else`
- `match`
- `defer`
- `undefer`

There are also reserved keywords:

- `trait`

Table of Contents

Comments

C* contains multiple types of comments

Table of Contents

`//` Single-Line Comments

Tokens followed by `//` until a `\n` newline are considered single-line comments.

Table of Contents

`///` Doc Comments

Tokens followed by `///` until a `\n` newline are considered doc comments. They are a form of single-line comments,
but may also be processed by tools for generating documentation.

Table of Contents

`/* */` Nested, Multi-Line Comments

Tokens followed by `/*` are considered multi-line comments.
They can be nested, and end at the next `*/` that is not a part
of an inner multi-line comment.
They also do not have to be multi-line,
and can comment out only part of a line.

Table of Contents

`/-` Structural Comments

`/-` denotes a structral comment. It comments out the next item in the AST, which could be the next expression, function, type definition, etc.

Example:

```
// This is a regular single line comment.

/// This is a doc comment for the function below.
fn foo() = {}

/* This is a multiline comment
Everything inside here is commented out until "* /"
*/

/* They can be /* nested */, too. */
fn /* and appear in-between things */ bar() = {}

/- let x = 25; // This comments out the entire let expression.
```

Table of Contents

`pub` Publicity

All top-level items (except `impl` blocks)
may be prefixed with a publicity modifier.

The syntax for this is `pub`.

Following the `pub`, there may also be a module path
within parentheses, like this: `( path )`.

If there is no publicity modifier, i.e. no `pub`,
then the publicity of the item is private, i.e. `pub(self)`.

Only public items may be `use`d from other modules.
Private items may only be used for the current module or its descendants.

Table of Contents

Annotations

All items may be prefixed with any number of annotations,
which annotate the item with certain metadata.

The syntax for this is `@ annotation`,
where `annotation` is the name of the annotation.
Note that annotations may be imported (`use`d)
or referred to with their fully-qualified path.

They may also have an `argument_list` after the annotation.
Having no `argument_list` is equivalent to having an empty, 0-length `argument_list`.
The `argument_list` is a normal C* `argument_list`,
except this one must be a compile-time constant.

The exact annotations available is still being decided,
but a few of them may be:

- `@extern`
- `@abi(" abi ")`, like `@abi("C")` or the default `@abi("C*")`
- `@inline`
- `@noinline`
- `@impl( type1 , ... , typeN )`
- `@align( alignment )`
- `@packed`
- `@allow(" warning_name ")`
- `@non_exhaustive`

For now, any available annotations will be implemented in the compiler,
though this could change in the future.

Annotations can also be applied to the current module.
In this case, they must appear before any other items in the module
and are prefixed with an extra `@`, like `@@allow("unused_variable")`.

Table of Contents

`use` Declarations

`use` declarations are used to import items/declarations
from other modules, such as the standard library,
external libraries, your own defined modules, or certain types.

Their syntax is *`use`* `=` use *`path`*,
where *`path`* `=` *`identitifier`* `.` *`path`*.

That is, it imports a path to an item to be used
without path qualification within the current scope.

`path` can also end in `.*`. The `*` indicates all items,
so this imports all items from the parent path.

Table of Contents

`let`s

A `let` binds an expression to a name.
That expression can either be a value or a type.

Normally (in expressions), `let` bindings can be shadowed,
but they cannot be at the module level.

Table of Contents

Value `let`s

For values, the syntax of this is `let mut` *`?`* *`identifier`* `:` *`type`* `=` *`expr`* `;` *`?`*.

The `mut` is optional. If there is no `mut`,
then the variable is an immutable const.
If there is a `mut`, then it is a mutable global variable.

In normal `let` bindings, *`expr`* can be any C* expression,
and the `:` *`type`* may be omitted where inferrable,
but at the top, global level, the *`expr`* must be constant evaluated
and the *`type`* must be annotated.
The way to do the former is by using a `const { ... }` block,
which evaluates the block to a constant at compile time.

A value `let` can also create zero, one, or multiple bindings
at once through destructuring a pattern.
If the pattern is tautological, i.e. the pattern always matches,
then the bindings are always created.
If the pattern may not match, then the `let` expression is a `bool` and may be used in `if`s or `match`es.
In this case, the `let` binding(s) are only created if the pattern matches and the `let` expression evaluated to `true`.
Note that `match`ing a non-tautological `let` is possible
but very un-idiomatic, since the binding could simply be done in the match itself. Thus, it is normally used with `if`.

See pattern matching for more info on patterns and destructuring.

Table of Contents

Type `let`s aka Type Aliases

For types, the syntax of this is `let` *`identifier generic_parameter_list?`* `=` *`type`* `;`.

The *`type`* here may be any type expression that
a value would be annotated with.
For example, this includes named types, tuples, arrays, slices, function pointers.

See below for info on the optional *`generic_parameter_list`*.

Note that this only creates an alias of the type,
but does not actually create a new type.
For example, the type alias cannot be used as a namespace
for methods or enum variants.

For example, you could have these type aliases:

```
let Option<T> = Result<T, ()>;
let Bool = Option<()>;
let Point = (f64, f64);
```

Table of Contents

`fn` Function Declarations

`fn` declarations declare functions.

The syntax of this is `fn` *`identifier generic_parameter_list? parameter_list`* `:` *`type`* `=` *`expr`*.

The *`identifier`* is the name of the function,
the *`generic_parameter_list`* optional generic parameters,
the *`parameter_list`* required normal (non-generic) parameters,
the *`type`* the return type of the function,
and the *`expr`* the return value of the function.

Generic Parameters

A *`generic_parameter_list`* is delimited by `<` `>` angle brackets
and contains `,` comma-separated generic parameters.
A trailing comma is allowed.

Each generic parameter is a generic type or a generic constant. If it is a generic constant, then it requires a `:` `type`* annotation.

Note that an empty *`generic_parameter_list`* like `<>`
is semantically distinct from no *`generic_parameter_list`* at all.
Generic functions are monomorphized (see generics for more).

Also, the `<` `>` angle brackets as used for generics
has higher precedence than the `<` `>` comparison operators.

Parameters

A *`parameter_list`* is delimited by `(` `)` parentheses
and contains a `,` comma-separated parameters.
A trailing comma is allowed.

Each parameter is a `let` binding except without the `let` keyword.
However, in function declarations, the parameters must have `: type` annotations.
Note that the similar function literals/values do not require this.

Return Type

The `: type` may be omitted if the type is the unit `()` type.

Return Value

The `expr` that the function returns may be any expression.
However, normally it is a `{ ... }` block,
which is necessary to include multiple statements in a function.
The block (like any) may also have modifiers,
like `try { ... }` or `const { ... }`.
Returning a `const { ... }` from a function in particular marks
that function as constant evaluatable*.

Normally a `;` is required to end the return value,
except if a block is used as the return value,
then it does not require the `;`.

A function return block is slightly special in that
`return` may be used within it, which is equivalent to a `break` from that top-level function block.

If a function is annotated with `@extern`,
then it must omit the `= expr` and end with a `;`.
In this case, only the function signature is specified
and the `@extern`ed function must be available as a function symbol at link time or else there will be a compile error.

Note that `@abi("C")` is usually specified along with `@extern`
because the default `@abi("C*")` is unstable.

In an `@extern @abi("C")` function,
the last (but not only) parameter may also be `...`,
which is a C varargs parameter and may be called with multiple arguments.
This is only for C FFI for functions like `syscall`,
which otherwise we'd need to implement with some assembly.

Note that `@extern` and `@abi("C")` may also be specified for an entire module,
in which case it applies to all items within that module.

Function Examples

For example, a non-generic function may look like this:

```
fn foo(_a: i32, b: usize, _c: String): usize = b * b;
```

or this:

```
fn string_len(c: String): usize = {
    c.len()
}
```

and a generic function may look like this:

```
fn equals<T>(a: T, b: T): bool = {
    a.equals(b)
}
```

Table of Contents

`struct` Declarations

`struct` declarations declare a `struct` type,
which is a product type of its field types.
All fields are always initialized.

The syntax of this is `struct` *`identifier generic_parameter_list?`* `{` *`fields`* `}`,
where *`identifier`* is the name of the `struct` type,
`generic_parameter_list` are its generic parameters,
and *`fields`* is a `,` comma-separated list of fields.
A trailing comma is allowed.
Zero fields is also allowed.

The syntax of each field is a value `let` without the `let` and the `=` *`expr`* `;`.
Each field may also be prefixed by a *`publicity`* modifier.

Note that `mut` can be specified for these fields,
in which case they are have interior mutability,
i.e., they can be mutated through a non-`mut` pointer to the struct.

By default, `struct`s use `@abi("C*")`,
which means their layout and alignment is unspecified and unstable.
This allows for fields to be rearranged for optimizations.
If `@abi("C")` is specified, however, then the fields are
layed out in memory in the order they appear in,
and C alignment and padding rules are used.

Table of Contents

`enum` Declarations

`enum` declarations declare an `enum` type,
which is a sum type of its variants.
That is, it is a discriminated union of variants,
each of which may have a value or not.
A value of an `enum` type is always one of its variants
and cannot be anything except those variants.
The discriminant value is stored.

The syntax of this is `enum` *`identifier generic_parameter_list?`* `{` *`variants`* `}`,
where *`identifier`* is the name of the `struct` type,
`generic_parameter_list` its generic parameters,
and *`variants`* is a `,` comma-separated list of variants.
A trailing comma is allowed.
Zero variants is also allowed, but note that this means that
the `enum` can never be instantiated because it has no variants.

Each variant may have a value or not.
If a variant does not have a value, then the syntax is `identifier`.
By default, the discriminant value of each variant is chosen by the compiler,
but this may be overridden for each variant
if all the variants of the `enum` have no value.
The syntax for this is `identifier = expr`,
where `expr` must be a `const { ... }` block
evaluating to the integer to be used for the discriminant.

If a variant does have a value, then the syntax is `identifier ( type )`.
Note that only one `type` is allowed here.
If you wish to include multiple types,
simple use a tuple or `struct` instead.

All variants of an `enum` implicity use `pub` as their publicity modifier, which cannot be changed.

By default, `enum`s use `@abi("C*")`,
which means their layout and alignment is unspecified and unstable.
This allows for the layout, including the discriminant, to be optimized.
Generally, though, the size of an `enum` type is the
size of the discriminant plus the size of the largest variant data.

If all the variants have no values,
then `@abi("C")` may be specified.
In this case, you must also specify the size of the enum
by adding a `: type` following the `identifier` name,
where the `type` is a primitive integer type.
In this case, all the variant discriminants must fit within that type.

The `@non_exhaustive` attribute can also be applied to an `enum` type,
in which case matching all the variants is no longer considered an exhaustive match,
and a catch-all `_ =>` match arm is required.

Table of Contents

`union` Declarations *

`union` declarations declare a `union` type,
which is a non-discriminated union similar to C `union`s.
It is meant for C FFI and thus defaults to `@abi("C")`.

The syntax of a `union` type declaration is
the same as a `struct` type declaration,
except the `struct` keyword is replaced by the `union` keyword.

The difference between the two is semantics.
The size of a union is the size of its largest field
and only one field may be active at any time.
Reading from an inactive field is undefined.

Table of Contents

`impl` Blocks

`impl` blocks define associated items for a type, which includes methods.

The syntax for this is `impl` *`generic_parameter_list?`* *`type`* `{` *`items`* `}`,
where *`type`* is the type you are defining associated items for,
`generic_parameter_list` is any generic parameters needed for *`type`*, and *`items`* are items like those in a module.

Within an `impl` block, there is an implicit type alias defined:
`let Self =` *`type`* `;`, where *`type`* is the same type being `impl`emented.

Items defined within an `impl` block are available
through the type as if it were a module.
The exception is methods, which may be called in another way as well.
A method is a function in an `impl` block whose first parameter
is `self: Self`.
The `: Self` may be inferred (an exception for function declarations).
To call a method, you may also call it using `.` syntax on a value of the `impl` *`type`*.
That is, *`value`* `.` *`method`* `(` *`args`* `)` is syntactic sugar
for *`type`* `.` *`method`* `(` *`value`* `,` *`args`* `)` where *`value`* `:` *`type`*.

Table of Contents

Type System

C* types can be split up into three kinds of types:

- primitive types
- compound types
  - built-in
  - user-defined

Table of Contents

Primitive Types

The primitive types in C* are:

- the `()` unit type
- integer types
- float types
- the `char`acter type

Table of Contents

`()` Unit Type

Table of Contents

`bool` Type

`bool` is the boolean type in C*,
except it is actually defined as an enum:

```
@allow("non_title_case_types")
enum bool {
    false = const { 0 },
    true = const { 1 },
}
```

Normally operator overloading is not allowed in C*.
The exception is `bool`, which defines the normal boolean operators.
See operators for details on them.

Table of Contents

Integer Types

Table of Contents

Float Types

Table of Contents

`char`acter Type

Table of Contents

Built-In Compound Types

The built-in compound types in C* are:

- reference types
- slice types
- array types
- pointer types
- tuple types
- function types

Table of Contents

Reference Types

In C*, you can have a reference to any type.
That reference is either immutable or mutable.

There is one exception to this.
type `.$bit_size_of()` must be a multiple of 8.
That is, bit fields like `u1` or `i5` may not be referenced.

The syntax for an immutable reference is *type* `&`,
and the syntax for a mutable reference is *type* `&mut`.

An immutable reference can be created using the postfix
`.&` reference operator from either an immutable or mutable binding.
A mutable reference can be created using the postfix
`.&mut` mutable reference operator, but only from a mutable binding.

Both immutable and mutable references can be dereferenced
using the postfix `.*` dereference operator.
This creates a temporary, unnamed, non-copied, immutable binding.
A mutable reference can also be dereferenced mutably
using the postfix `.*mut` mutable dereference operator.
This is the same as the `.*` deference operator,
except the resultant temporary is mutable.

Note that references can only be created by referencing an existing value.
Thus, null references are impossible to create.
Instead, `Option` should be used, like `Option<T&>`.

Table of Contents

Slice Types

In C*, you can also have a slice of a type, a contiguous collection of values of the same type. The number of values is only known at runtime.

The syntax for this is `type []`.

A slice `T[]` is similar to the struct

```
struct SliceT {
    len: usize,
    ptr: T&,
}
```

but there are a few important differences.
Slices store their values inline.
They are thus unsized (i.e. dynamically sized) (`.$size_of()` is non-`const` for them).
However, references to slices are sized.
They are so-called fat pointers, i.e. the length and raw pointer both constitute the reference.

Slices are the only fundamentally unsized types.
Other compounds may only contain at most one unsized type,
and if they do, then they themselves are unsized.
Like slices, references to any unsized type are fat pointers.

To access the values of a slice,
the `[]` index operator may be used: `value [ index ]`,
where `index` is a value of an unsigned integer type
and `value` is a reference to a value of slice type.
Note that if you have a slice reference,
it must be derefenced before indexing the slice directly.

Indexing a slice reference `T[]&` evaluates to `Result<T&, IndexBoundsError>`,
and indexing a mutable slice reference `T[]&mut` evaluates to `Result<T&mut, IndexBoundsError>`.
Thus, it is always bounds checked.
To panic on an out-of-bounds index, simply `.unwrap()`
the `Result` to get the `T&` or `T&mut`,
which can then be dereference to access.
To elimiate bounds checking, the `Result` can instead be `.unwrap_unchecked()` to get the `T&` or `T&mut`
without checking if there was an error,
thus eliminating the bounds check.

Bounds checking can also be eliminated in many other safe ways.
Bounds checking is usually only a problem when it is done for many elements of a slice when it only needs to be done once.
For this case, multiple elements can be indexed using a slice pattern (see patterns),
or an iterator can be used, which will eliminate redundant bounds checking.

Slices can also be sliced to yield a smaller view of the original slice.
This is also done by the same `[]` indexing operator,

except now the syntax is `value [ range ]`,
where `range` is a value of range type.

Slicing a slice reference `T[]&` evaluates to `Result<T[]&, SliceBoundsError>`,
and slicing a mutable slice reference `T[]&mut` evaluates to `Result<T[]&mut, SliceBoundsError>`.

Table of Contents

Array Types

In C*, there also arrays of a type,
which, like slices, are a contiguous collection of values of the same type,
but unlike slices, have a length known at compile time and not stored at runtime.
Thus, they are sized unliked slices.

The syntax for this type is `type [ size ]`,
where `size` is a const of an unsized integer type.

Arrays can also be indexed and sliced,
but since the length is known at compile time,
if the index or range is also known at compile time,
then indexing and slicing always succeeds at runtime
(i.e. there is no `Result`) yielding another array,
or else is a compile error.
The same syntax is used for indexing and slicing as is for slices.

To explicitly turn an array into a slice reference,
`.$cast<T[]>()` can be used.

Table of Contents

Pointer Types

In C*, you can have a pointer to any type,
That reference is either immutable or mutable.

There is one exception to this.
`type .$bit_size_of()` must be a multiple of 8.
That is, bit fields like `u1` or `i5` may not be referenced.

The syntax for an immutable reference is `type *`,
and the syntax for a mutable reference is `type *mut`.

A pointer can point to 0, 1, or any number of the pointee type.

A pointer can only be created from
an explicit cast from a reference type
and through the return type of an `@extern` function.
It is just meant primarily for FFI.

A pointer cannot be dereferenced directly.
It must be explicitly cast to one of these types to be dereferenced:

- a reference if it points to 1 pointee type
- a slice if it points to any number of pointee types of runtime-known amount
- an array if it points to any number of pointee types of compile-time-known amount
- `None` if it is a null pointer

Table of Contents

Tuple Types

In C*, you can also have a contiguous collection values of different types, i.e. a heterogenous array of sorts.
This is called a tuple and its length must be known at compile time.

The syntax for this type is `( types )`,
where `types` is a list of `,` comma-separated `type` s.
A trailing `,` comma is allowed.
However, in a single-element tuple, a trailing comma is required
to differentiate from general parentheses.

The elements of a tuple can be accessed as fields like in a `struct`.
In fact, a tuple is syntax sugar for an anonymous `struct`
with all public fields, though there is one caveat.
The fields of a tuple are decimal integer literals (the index),
which would not otherwise be allowed as an identifier for a field name.
Note that like `struct` s, tuple elements may be not layed out in memory in order.

Table of Contents

Function Types

The type of a function `fn(a: A, b: B): C` is `fn(A, B): C`.

The syntax for this is `fn tuple_type : type`,
where `tuple_type` is a tuple type of the arguments
and `type` is the return type.

Other postfix type modifiers (e.x. `*`, `&`, `[]`)
applied at the end by default apply to the return type.
To apply them to the entire function type,
the function type must be parenthesized,
like `(fn(A): B)&`.

Table of Contents

User-Defined Compound Types

The user-defined compound types in C* are:

- `struct` types
- `enum` types
- `union` types

They correspond to the item declarations of the same name.

Table of Contents

`struct` Types

See `struct` declarations for more.

Table of Contents

`enum` Types

See `enum` declarations for more.

Table of Contents

`union` Types

See `union` declarations for more.

Table of Contents

Destructive Moves

Passing a variable (to a function, to another variable, etc.)
are done by moving destructively.
That is, a simple `memcpy` to the new location.
There are no move constructors or anything like that.
Clones must be explicit with a `.clone()` call for `Clone` types (`@impl(Clone)`).
The exception is `Copy` types (`@impl(Copy)`),
for which clones are implicit.

Table of Contents

Expressions

Almost everything that is not a type in C* is an expression.
This includes all control flow constructs.

Table of Contents

Literals

C* Literals:

- unit
- bool
- int
- float
- char
- string
- struct
- tuple
- array
- enum
- union
- function
- closure
- range

Table of Contents

Unit Literals

In C*, every expression has a type. Even statements that return "nothing",
they really return unit, or `()`.
The type of this unit literal is also called unit and written `()` as well.

Table of Contents

Boolean Literals

There are two boolean literals of type `bool`: `true` and `false`.
These are actually enum variants of the `enum bool`.
See the `bool` Type.

Table of Contents

Number Literals

In C*, number literals are composed of 4 (potentially optional) parts (in order):

- the integral part
- the floating part (optional)
- the exponent (optional)
- the suffix (optional)

For each of the integral part, floating part, and exponent,
they contain an optional sign, optional base,
and then a series of one or more digits.
Note that each part may specify a different base.

The sign may be `+` for positive numbers, `-` for negative numbers, or nothing, which defaults to `+`.

The base and corresponding digits may be:

Prefix	Name	Base	Digits
none	decimal	10	`0-9`
`0b`	binary	2	`0-1`
`0o`	octal	8	`0-8`
`0x`	hexadecimal	16	`0-9`, `A-F`

The series of digits may also be separated by
any number of `_` underscores between the digits.
It cannot begin or end with `_` underscores, however.

If there is a floating part, then a decimal point `.`
separates it from the preceeding integral part.
The floating part may not have a sign and is always positive (in itself).

If there is an exponent, then an `e` precedes it.

The (optional) suffix contains the type of number and a bit size.

The type of number may be:

- `u`: unsigned integer
- `i`: signed integer
- `f`: floating-point number

The bit size is usually a literal power of 2 number,
but may be any positive integer for integer types.

It may also be a word whose bit size is architecture-dependent.

For integers (`u` and `i`), the common bit sizes are:

- `8`
- `16`
- `32`
- `64`
- `128`
- `size` (bit size necessary to store an array index)
- `ptr` (bit size necessary to store a pointer
  or the difference between them)

For floats (`f`), the bit sizes are:

- `16`
- `32`
- `64`
- `128`

These suffixes are the primitive number types.
Thus, in total, they are (with their C equivalent for FFI):

C*	C
`u8`	`uint8_t`
`i8`	`int8_t`
`u16`	`uint16_t`
`i16`	`int16_t`
`u32`	`uint32_t`
`i32`	`int32_t`
`u64`	`uint64_t`
`i64`	`int64_t`
`u128`	`unsigned __int128`
`i128`	`__int128`
`usize`	`size_t`
`isize`	`ssize_t`
`uptr`	`uintptr_t`
`iptr`	`intptr_t`
`f16`	`_Float16`
`f32`	`float`
`f64`	`double`
`f128`	`_Float128`

Integers always use 2's-complement
and floats always are IEEE 754 floating point numbers.

If the type is a float, then it must contain
a `.` decimal point and a floating part.
If the type is an integer, then it must not.
Both can contain exponents, though for integers,
the exponent (in scientific notation) cannot cause
the integer to exceed its finite size.

If there is no suffix type, then the type is inferred.
If there is a `.` decimal point, then the type must be a float, and vice versa with integers.
If there is a `-` sign for the integral part,
then the type must be a float or a signed integer.
To infer the bit size of the number,
general type inference is used.
If it cannot be unambiguously inferred,
then it is an error and the user must
explicitly specify the suffix type.

Table of Contents

Character Literals

In C*, character literals are of type `char` and are denoted with single `''` quotes.
They are unicode scalar values,
which are slightly different from unicode code points.
This means they are always 32 bits on all architectures.

For the actual char literal within the quotes,
it may be any unicode scalar value,
but some characters need to be or may be escaped.
The ascii values that must be escaped are:

- `\n` : newline
- `\r` : carriage return
- `\t` : tab
- `\0` : null char
- `\\` : backslash
- `\'` : single quote

Other ascii values may also be escaped as well using the syntax `\x7F`,
where `7F` is the hexadecimal value of the ascii character,
from 0 to 127 (aka `0x7F`).
Thus it may only be two digits.

Unicode scalar values can also be escaped with the syntax `\u{7FFF}`.
The hexadecimal value is the 24-bit unicode character code.

Character literals can also be prefixed with a `b`: `b' '`,
in which case they are byte literals, i.e. a `u8`.
The required ascii escapes are the same,
though the `\xFF` escape can now go up to 255 (aka `0xFF`),
and there may not be unicode escapes
(since it's only a `u8` byte literal now).

Table of Contents

String Literals

There are multiple types of strings in C* owing to
the inherent complexity of string-handling without incurring overhead.
The default string literal type is `String`, which is UTF-8 encoded and
wraps a `*[u8]`. This is a borrowed slice type and can't change size.
To have a growable string, there is the `StringBuf` type,
but there is no special syntactic support for this owned string.
`String`s are made of `char`s, unicode scalar values, when iterating
(even though they are stored as `*[u8]`).

Then there are byte strings, which are just `*[u8]` and
do not have to be UTF-8 encoded.
String literals for this are prefixed with `b`, like `b"hello"`.
The owning version of this is just a `Box<[u8]>`
(notice the unsized slice use), and
the growable owning version is just a `Vec<u8>`.

Furthermore, for easier C FFI, there is also `CString` and `CStringBuf`,
which are explicitly null-terminated. All other string types are
not null-terminated, since they store their own length,
which is way more efficient and safe.
Literal `CString`s have a `c` prefix, like `c"/home"`.

And finally, there are format strings. Written `f"n + m = {n + m}"`,
they can interpolate expressions within `{}`.
Format, or `f`-strings, don't actually evaluate to a string,
but rather evaluate to an anonymous struct that has methods to
convert it all at once into a real string. Thus, `f`-strings do not allocate.

For the character literals allowed in C* strings,
that depends on the string type, which are:

Prefix	Name	Type
none	string	`String`
`b`	byte-string	`*[u8]`
`r`	raw-string	type without the `r`
`c`	c-string	`CString`
`f`	f-string	anonymous struct with methods

All of these string prefixes can be combined with each other,
except for `r` and `f`, since f-strings require escaping,
which goes against raw strings.

For `r` raw strings, no escapes are allowed.

For normal UTF-8 strings (which includes the `r`, `c`, and `f` modifiers),
the string must contain character literals, except there are no single `'` quotes anymore,
double `"` quotes delimit strings,
and double quotes must escaped (`\"`) instead of single quotes (`\'`).
Obviously the escapes don't apply to raw `r` strings.
For `f`-strings, braces must also be escaped: `\{` and `\}`,

since they are used to delimit expressions within the string.
And for `c`-strings, they must not contains any `\0` null characters.

For byte `b` strings, the string must contains byte literals.
The other string modifiers apply in the same way,
and again, double quotes (`\"`) must be escaped instead of single quotes (`\'`).

Table of Contents

Struct Literals

Struct literals are literals that create a value of a struct type.
That is, if we have a struct `Example`:

```
struct Example {
    a: u32,
    b: f64,
    c: String,
}
```

then we can create a value of type `Example` with the struct literal

```
Example {
    a: 0,
    b: 0.0,
    c: "",
}
```

That is, we first have the struct type name, an open `{` brace,
the list of fields and their values, and then a closing `}` brace.
The fields are separate by `,` commas (a trailing `,` comma is allowed),
and `:` colons separate the field name and its value.

If the name of a field and its value expression are the same,
then the `:` colon and value may be omitted, like so:

```
let c = "";
Example {
    a: 0,
    b: 0.0,
    c,
}
```

Furthermore, `..` can be used to spread the fields of another struct into a struct literal, like so:

```
struct SmallExample {
    a: u32,
    b: f64,
}

let x = SmallExample {
    a: 0,
    b: 0.0,
};

Example {
    ..x,
```

```
    c: "",
}
```

Note that the struct type does not have to be the same,
but the fields that are being spread must match between the struct types in name and type.

Table of Contents

Tuple Literals

C* has tuples, but they are simply shorthand and syntax sugar for structs.
A tuple type is a finite, heterogenous list of types,
such as `(i32, usize, String)`,
and its field names are unsigned integers (`.0` , `.1` , and `.2` for this tuple).
This is the only difference between tuples and desugaring them to structs:
struct field names must be valid C* identifiers,
but tuple field names begin with digits.
Otherwise, they are exactly the same.
The tuple type with 0 element types, `()` , is also valid,
but it is equivalent to the `()` unit type.

Tuple literals mirror tuple types.
The field names are unnamed (unlike struct literals),
so it is just a `,` comma separated list of values of any type delimited by open `(` and close `)` parentheses.
There may be a trailing `,` comma separator,
and for 1-element tuple literals, this trailing `,` comma
is required to distinguish it from using `()` parentheses for associating general expressions.

Table of Contents

Array Literals

In C*, arrays are finite, homogenous lists of a single type.
There are delimited by open `[` and close `]` brackets,
as opposed to `()` parentheses for tuples.
Their values are also `,` comma separated.
Trailing `,` commas are allowed but never required,
unlike in 1-element tuple literals.

Array types are denoted `[T; N]` , where `T` is any type
and `N: usize` .

Table of Contents

Enum Literals

In an enum, such as

```
enum Example {
    A,
    B(i32),
}
```

there are two possible forms of enum literals
depending on if the variant has any data or not.

In the case of the variant `A`, which has no data attached,
the enum literal `Example.A` (or just `A` if `A` is imported)
is a value of type `Example`.

In the case of the variant `B`, which has data attached,
the enum literal `Example.B` is a function of type `fn(i32): Example`
that returns the `B` variant with the given data attached.
Thus, `Example.B(0)` or `Example.B(100)` is normally written,
though the function can also be referred to by itself.

Table of Contents

Union Literals

Union literals are the same as struct literals
except only one field may be specified.

Table of Contents

Function Literals

In C*, there is very little difference between function declarations
and function literals (using them as values).

In function declarations, they are written

```
PUBLICITY fn FUNC_NAME GENERIC_ARGS ARGS = BODY_EXPRESSION
```

such as

```
fn foo<T>(t: T): T = { t * t }
```

In function literals, there is no more publicity modifier
and the function name is optional,
since it usually specified as the let binding instead if named:

```
fn<T>(t: T): T = { t * t }
```

Furthermore, type inference of function arguments and return type
is allowed for function literals, since they cannot be public declarations.
If the types are ambiguous, though, type annotations are still required of course.

The type of a function literal is unique and opaque,
but can be casted to a function pointer like `fn(T): T`.

Note that annotations like `@abi("C")` can still be applied
to function literals just like function declarations.

Table of Contents

Closure Literals

Closure literals are very similar to function literals—in fact,
they are a superset of function literals—except they also have a closure context.
That is, they can "enclose" over values in the current scope.

The syntax for a closure literal is simply a normal function literal with an anonymous struct literal, the closure context, following the `fn`.

The closure context is an anonymous struct literal
in that it has no named struct type. That is, instead of

```
Example {a: 0, b: 0.0, c: ""}
```

it would just be

```
{a: 0, b: 0.0, c: ""}
```

The fields in this closure context struct
are then immediately available within the function body
as if they were immediately destructured.

The type of a closure literal is unique and opaque.
Unlike function literals (in which there is no context),
the type of closure literals cannot be casted to a bare function pointer.
The closure function corresponds to a method on the closure context struct,
and as such, cannot be casted to a function pointer
since there is an implicit `*Self` argument.
Thus, the only way to accept a closure as an argument is by using generics,
which ensures there is no pointer indirection
and the closure can be inlined into the call site.

Table of Contents

Range Literals

Range literals denote an integer range.
There are a few different forms of ranges,
which we will define in terms of set interval notation
as to what integers the range includes.
Here, `n` refers to the parent length that the range applies to.

Range	Interval
`a..b`	`[a, b)`
`a..`	`[a, n)`
`..b`	`[0, b)`
`..`	`[0, n)`
`a..=b`	`[a, b]`
`..=b`	`[0, b]`
`a..+b`	`[a, a + b)`

Range	Interval
`a..+=b`	`[a, a + b]`
`a..-b`	`[a, n - b)`
`a..-=b`	`[a, n - b]`
`..-b`	`[0, n - b)`
`..-=b`	`[a, n - b]`

Table of Contents

Function Calls

TODO

Table of Contents

Method Calls

TODO

Table of Contents

Blocks

TODO

Table of Contents

Control Flow

TODO

Table of Contents

Pattern Matching

TODO

Table of Contents

Conditionals

TODO

Table of Contents

`match`

TODO
patterns

Table of Contents

`if`

`if` evaluates a block conditionally.

The syntax for this is *`expr`* `.if` *`block`*.
It is syntax sugar for a `match`:

`expr` `.match { true =>` *`block`* `, false => (), }`

Table of Contents

`else`

An `else` may immediately follow an `if` expression,
in which case the whole thing becomes an if-else expression.

The syntax for this is *`expr`* `.if` *`block`* `else` *`block`*.
It is syntax sugar for a `match`:

`expr` `.match { true =>` *`block`* `, false =>` *`block`* `, }`,
where the *`block`* are in the same order as in the if-else expression.

Normally the *`expr`* following an `else` must be a *`block`*,
but it can also be another if expression.

Table of Contents

Labels

TODO

Table of Contents

Loops

TODO

Table of Contents

`while`

TODO

Table of Contents

`for`

A `for` loop allows you to iterate through an iterator.
An iterator is just a type `Iter` that has
a `fn next(self: Self) -> Option<T>` method,
where `T` is the element type we are iterating over.

The syntax for this is *`expr`* `.for` *`binding block`*,
where the *`expr`* is a value that has
a `.into_iter()` method returning the iterator,
the *`binding`* is the binding for the element name,
and *`block`* is the block of the `for` loop.

It is syntax sugar for:

`{ let iter = expr .into_iter(); true.while { let binding = iter.next().?; block } }`

Table of Contents

`defer`

TODO

Table of Contents

Error Handling

TODO

Table of Contents

`try`

TODO
error handling

Table of Contents

Panicking

In C*, all fallible functions and operations return either
`Result` or `Option` to indicate an error or exceptional case.
Normally errors are handled by bubbling up the error with `.?`
or handling the error directly in a `match` or other `Option` / `Result` methods.
However, in certain cases you either don't care about
handling the exceptional case or you can determine that
the error case is statically impossible but the compiler cannot.
In this case, you may wish to simply get the `Some` or `Ok` value
out of the `Option` or `Result`.
This can be done by panicking on a `None` or `Err`.

Panicking in C* means the program will immediately print out an error message
and then `abort`, i.e., calls the libc function `abort`.
No cleanup or unwinding is done in this case.
In particular, `defer`s on the stack are not run because the stack is not unwound.
Because of this, panicking should only be done under extreme circumstances,
such as statically determining the error case is impossible.
If you want unwinding and `defer`s to run,
simply use `.?` to bubble up the errors.

The way to panic is to call `.unwrap()` on a `Result`.
This is the only fundamental way to panic in C*.
All other functions that panic or may panic ultimately call `Result.unwrap`.
For example, `Option.unwrap` converts the `Option`
into a `Result` and then calls `.unwrap()` on it.
The same is true for `Option.expect` and `Result.expect`,
which allow you to set an error message to be printed.

The error message that `Result.unwrap` prints to `stderr` is implementation defined,
but it calls `E.error_message` to obtain the error message of the `e: E` in `Err(e)`.
Thus, to `.unwrap()` a `Result<T, E>`, `E` must have such a `.error_message()` method.

It may also print a (function call) stack trace or error return trace,
but that is not guaranteed.

There is one other option as well besides panicking.
If you know for certain that the error case is impossible,
you may call `Result.unwrap_unchecked()`.
This does not panic if the `Result` is `Err`,
but it is undefined behavior.

Table of Contents

Operators

Operator	Arity	In-Place	Type	Description	Example
`+`	binary	no	arithmetic	addition	`2 + 2`, `4.0 + 2.0`
`-`	binary	no	arithmetic	subtraction	`2 - 2`, `4.2 - 2.2`
`*`	binary	no	arithmetic	multiplication	`2 * 2`, `4.0 * 2.0`
`/`	binary	no	arithmetic	division	`2 / 2`, `4.0 / 2.0`
`%`	binary	no	arithmetic	modulus	`2 % 2`
`-`	unary	no	arithmetic	negation	`-a`
`==`	binary	no	relational	equal to	`a == 2`
`!=`	binary	no	relational	not equal to	`a != 2`
`>`	binary	no	relational	greater than	`a > 2`
`<`	binary	no	relational	less than	`a < 2`
`>=`	binary	no	relational	greater than or equal to	`a >= 2`
`<=`	binary	no	relational	less than or equal to	`a <= 2`
`&&`	binary	no	logical	and	`a && b`
`||`	binary	no	logical	or	`a || b`
`!`, `.!`	unary	no	logical	not	`!a`
`&`	binary	no	bitwise	and	
`|`	binary	no	bitwise	or	
`^`	binary	no	bitwise	xor	
`~`, `.~`	unary	no	bitwise	not	
`<<`	binary	no	bitwise	left shift	
`>>`	binary	no	bitwise	right shift	
`[]`	binary	no	indexing	index a slice	`a[1]`
`+=`	binary	yes	arithmetic	addition	

Operator	Arity	In-Place	Type	Description	Example
`-=`	binary	yes	arithmetic	subtraction	
`*=`	binary	yes	arithmetic	multiplication	
`/=`	binary	yes	arithmetic	division	
`%=`	binary	yes	arithmetic	modulus	
`&&=`	binary	yes	logical	and	
`||=`	binary	yes	logical	or	
`&=`	binary	yes	bitwise	and	
`|=`	binary	yes	bitwise	or	
`^=`	binary	yes	bitwise	xor	
`<<=`	binary	yes	bitwise	left shift	
`>>=`	binary	yes	bitwise	right shift	
`++`	unary	yes	arithmetic	increment	
`--`	unary	yes	arithmetic	decrement	
`.&`	unary	no	reference	reference	
`.&mut`	unary	no	reference	mutable reference	
`.*`	unary	no	reference	dereference	
`.*mut`	unary	no	reference	mutable dereference	
`.?`	unary	no	control flow	try	

Arithmetic operators operate on expressions of the same number type
and evaluate to the same number type as well.
`.$cast<>()` can be used here when the operands are of different type.
`%`, `++`, and `--` are not allowed for floats.

Relational operators operate on expressions of the same type
and evaluate to a `bool`.

Logical operators operate on `bool` expressions and evaluate to a `bool`.

Bitwise operators operate on expressions of the same number type
and evaluate to the same number type as well.
The except is the shift operators: `<<`, `>>`, `<<=`, and `>>=`,
whose right operand is the minimum unsigned integer type
that may be shifted by (i.e. the bit size of the left operand).
Otherwise it would be UB.
For example, if the left operand is `u64`, then the right operand is `u6`.
For signed integer types as the left operand,
the sign bit is extended when shifting.

For indexing operators, see slices
and arrays, which may be indexed.

In-place `operator=`s evalute to `()`.

Table of Contents

Generics

Generics in C* are always monomorphized.

TODO

Table of Contents

Constant Evaluation

TODO

Table of Contents

Builtin Functions

TODO

Table of Contents

Lang Types

Lang types are standard library types that the compiler knows about and may use.
They are:

- `Option`
- `Result`

For example, they are used for the `.?` try operator.

Table of Contents

`Option`

```
enum Option<T> {
    Some(T),
    None,
}
```

Table of Contents

`Result`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Table of Contents

List of Annotations

TODO

Table of Contents

Current Restrictions and Unimplemented Features

The following features are currently unimplemented:

- non ASCII source code (normally UTF-8 is allowed) -- this will be super low priority for us
- targets other than `x86_64-linux-gnu`
- user-defined `mod`ules, except for:
  - the implicit single-file module
  - those defined by the compiler or in the standard library
- `pub` publicity modifiers (everything will be public for now)
- any and all generic programming
- explicit `enum` discriminants set to user-decided constants
- `use` declarations except for the standard prelude,
  which is implicitly `use`d
- strings and characters except for byte ones, i.e.:
  - `b`yte string literals
  - `b`yte literals
- growable string types (in the standard library)
- type aliases except for:
  - those implemented by the compiler
- most attributes except for:
  - `@extern` and `@abi("C")` for functions (for calling libc)
  - all other annotations are allowed but ignored
- `...` trailing varargs parameter for `@extern @abi("C")` functions
  unless it's needed for the standard library (using libc)
- `union`s since they're only for C FFI
- tuples since they're just sugar for structs [2]
- `if`, `else`, `for`, which are just sugar for `match` and `while` [2]
- non-temporary unsized types (slices must be references)
- const generics
- const evaluation other than constant literals
- `mut` fields for interior mutability
- `struct` spread `..` syntax and `field: field` => `field` sugar [2]
- the only `Copy` types are primitive types

The following features we hope to implement but will come at the end:

- generics except for `Option` and `Result` (which will definitely be done)
- `defer` and `undefer` (`undefer` more likely to skip)
- closures and function pointers

2

may add this back if we have time since it's just sugar

Table of Contents

Grammar

Much of the grammar is specified above using `*italics*` and in words,
but here is the `ocamlyacc` grammar:

```
TODO
```

Old Stuff Below

Statements and Expressions

Table of Contents

Statements

Due to the expression oriented nature of C* all control flow statements are
themselves expressions.

Table of Contents

If-Else Statements

If-Else statements execute one of two cases. The first consists of typical
C-style semantics wherein we have:

```
if (expr1)
    statement1
else
    statement2
```

Both `statement1` and `statement2` must evaluate to the unit type. Like
C the `else` part of the If-Else control flow block is optional. In addition to
the C-style control flow we also can have:

```
if (expr1)
    expr2
else
    expr3
```

In both cases the expressions in the `if` statement are evaluated and in the case
they evaluate to a non-zero value the flow of execution continues down that
path otherwise the body of the `else` statement is executed.

C* utilizes the same mechanism to eliminate ambiguity relating to a "dangling-else". An `else` is grouped to the nearest `if`. In the case of:

```
let i: i32 = 6;
let j: i32 = 7;
```

```
if(i > 4)
    if(j > i)
        println!("j is greater than i!");
    else
        println("j is less than or equal to i!");
```

While the indentation and print statements make clear which `if` the `else` clause is grouped with it should be clear that barring the use of additional brackets to direct control flow the `else` is grouped to the nearest `if` above it.

Table of Contents

For Statements

For statements can execute over a range in the case of:

```
for season in seasons.iter()
    println!(season);
```

In addition to the use of an explicit iterator it is also possible to use a range literal to bound the execution of the body of a for loop in the case of:

```
let mut day_ = 1;
for x in 1..365{
    println!("Day {} of 365", x);
}
```

Table of Contents

While Statements

Execution of the body of a while statement continues until the expression labeled `expr1` evaluates to zero. For example:

```
while(expr1){
    statement1;
}
```

Similiar to `if` statements due to the expression oriented nature of C* `statement1` must evaluate to the unit type and it is possible to replace `statement1` with `expr2`.

Table of Contents

Defer

To aid in resource handling, C* has a `defer` keyword.
`defer` defers the following statement or block until the function returns,
but will run it no matter where the function returns from
(but not `panic`s/`abort`s) (actually, the `defer` will run when
its block exits, but its easier to just think about function blocks first).

For example, you can use this to ensure
you correctly clean up resources in a function:

```
extern "C" fn open(path: *u8, flags: i32): i32;
extern "C" fn close(fd: i32): i32;

fn open_file_in_dir(dir: *[u8], filename: *[u8]): Result<i32, String> try = {
    let mut path = Vec.new(Mallocator());
    defer path.free();
    try {
        if (dir.len() > 0) {
            path.extend(dir).?;
            path.push(b'/').?;
        }
        path.extend(filename).?;
        path.push(0).?;
    }.map_err(fn(_) "alloc error").?;

    let path = path.as_ptr();
    let fd = open(path, O_RDWR).match {
        -1 => Err("open failed"),
        fd => fd,
    }.?;
    defer println(f"opened {fd}");
    return fd;
}
```

In this example, you have to allocate a path to store
the directory and filename you combine, and then open
that path and return the file descriptor if it was successful.
You have to clean up the memory allocation, though, and do that
while still handling all the allocation errors and the open error.
The latter can be done elegantly with `try` and `.?`,
but if you mix in the `path.free()`, you'd have to run it before every
error return, which means you have to duplicate it and not use `.?` anymore.

Instead, you can use `defer` for this. No matter where you
return from the function, it will run its statement right before that.
You can also use `defer` for any statement, not just resource cleanup,
like logging for example.

However, sometimes you want to cancel a `defer`:

```
struct FilePair {
    fd1: i32,
    fd2: i32,
}

fn open_two_files(path1: *[u8], path2: *[u8]): Result<FilePair, String> try = {
    let fd1 = open_file_in_dir(b"", path1).?;
    close: defer close(fd1);
    let fd2 = open_file_in_dir(b"", path2).?;
    close: defer close(fd2);
    println(f"opened {fd1} and {fd2}");
    undefer close;
    FilePair {fd1, fd2}
}
```

In this example, you want open two files and return them if successfull.
If only one is successful, though, that's an error and you
should close the first one before returning the error.
In order to do that cleanly, you can use the `undefer` keyword,
which cancels an earlier labeled `defer`, in this case labeled `close`.

`defer` and `undefer` are actually syntax sugar
for something a bit more low-level and wordy:

```
fn open_two_files(path1: *[u8], path2: *[u8]): Result<FilePair, String> try = {
    let fd1 = open_file_in_dir(b"", path1).?;
    let close1 = {fd1} fn() close(fd1);
    let close1 = close1.$defer());
    let fd2 = open_file_in_dir(b"", path2).?;
    let close2 = {fd1} fn() close(fd1);
    let close2 = close2.$defer());
    println(f"opened {fd1} and {fd2}");
    let close = [close2, close1];
    close.undo();
    FilePair {fd1, fd2}
}
```

That is, `.$defer()` places the closure on the stack and
returns a `Defer` struct, which can be undone with `Defer.undo()`
(`[Defer].undo()` just maps `Defer.undo()` over the array).
`Defer.undo()` sets a bit in the `Defer` struct that it's been undone.
Then when the stack unwinds, any none-undone `Defers` on the stack are run.

Table of Contents

Expressions and Operators

Table of Contents

Unary Operators

Unary operators are operators that can act on an expression. C* uses the unary operators "-" and "!" to represent negation and the logical not repectively. "-" negates a number literal such as

```
let x = -2
```

The logical not "!" represents negation for bool literals or boolean expressions such as

```
let a = true
let b = !a
```

where b returns the value of false.

Table of Contents

Binary Operators

A binary operator acts on two expressions and can be show as follows:

Binary operator = expr * operator * expr

Table of Contents

Assignment operator

The assignment operator stores values into vairables. It uses the keyword "let" and the = symbol so that the left side variable stores the expression on the right.

Ex.

```
let a = 23 // a stores the value 23
```

Table of Contents

Arithmetic Operator

- The addition operator "+" adds two values of the same type. Automatic type conversion is applied when adding two number literals and can also be applied to string addition.

Ex.

```
1 + 2 // 3
12.3 + 10 // 22.3
"string" + "test" // "stringtest"
```

- The subtraction operator "-" subtracts two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 - 2 // -1
12.3 - 10 // 2.3
```

- The multiplication operator "*" multiplies two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 * 2 // 2
12.3 * 10 // 123
```

- The division operator "/" divides two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 / 2 // .5
12.3 / 10 // 1.23
```

- The modulus operator "%" takes the modulus of two values of the same type. Automatic type conversion is applied when adding two number literals.

Ex.

```
1 % 2 // 1
12.3 % 10 // 2.3
```

Table of Contents

Relational Operators

Relational operators represent how the operands relate to each other. Each expression using a relational operator has two values as inputs and outputs either true or false. The relational operators are: ==, !=, <,>, <=,>=, &, |.

```
1 < 2 // true
1 > 2 // false
1 != 2 // true
1 == 2 // false
true | false // true
true & false // false
```

Table of Contents

Functions

Functions are a type of statement that can be declared one of two ways:

```
fn name(parameters): return type = body
```

or

```
fn name(parameters): return type = { body }
```

It takes in a list of parameters and returns a value based on the expression. Functions can be written with or without specifying the return type.

Ex.

```
fn hello(): string = "hello world"

fn adding(a, b): = { return a + b }
```

Table of Contents

Pattern Matching

Instead of having a `switch` statement like in C,
C* has a generalized `match` statement, which can be used to match
many more expressions, including integers (like in C), `enum` variants,
dereferenced pointers, slices, arrays, and strings.
Also, there is no fall-through, but `match` cases can be combined explicitly.

Furthermore, just like you can destructure to
pattern match in a `match` statement, you can also do the same
as a general statement, like in a `let`. It's like an unconditional `match`.

```
let cow = CowString::Borrowed("");
let len = match cow {
    Borrowed(s) => s.len(),
    Owned(s) => s.len(),
};
let String {ptr, len} = "";
```

Note that string literals are of the `String` type similarly defined as above,
and you can redeclare/shadow variables like `len`.

Table of Contents

Methods

C* has associated functions and simple methods,
though these are largely syntactic sugar.
To declare these for a type, simply write:

```
struct Person {
    first_name: String,
    last_name: String,
}

impl Hello {

    fn new(first_name: String, last_name: String): Self = {
        Self {first_name, last_name}
    }

    fn say_hi1(self: Self) = {
        print(f"Hi {self.first_name} {self.last_name}");
    }

    fn say_hi1(self: *Self) = {
        print(f"Hi {self.last_name}, {self.first_name}");
    }

    fn remove_last_name(self: *mut Self) = {
        self.last_name = "";
    }

}

fn main() {
    let mut person = Person.new("Khyber", "Sen");

    {
        person.say_hi1();
        person.&.say_hi2();
        person.&mut.remove_last_name();
        person.say_hi1();
    }
    {
        Person.say_hi1(person);
        Person.say_hi2(person.&);
        Person.remove_last_name(person.&mut);
        Person.say_hi1(person);
    }
}
```

In this example, we first declared a `struct Person`,
and then an `impl` block for `Person` to define
methods/associated functions for it.
Note that this `impl` block can be anywhere, even in other modules.

In the `impl` block, we first declared an associated function `Person.new`,
which is just a normal function but namespaced to `Person`.
Similarly, the other three methods are just normal functions, too,
as seen when we call them explicity in the second block in `main`.
But we can also use `.` syntax to call them,
which just allows us to explicitly name `Person`.

Inside an `impl` block, we can also use the `Self` type
as an alias to the type being implemented.
This is especially useful with generics.

Note that the `.&` and `*Self` are explicit,
because we wan't these kinds of possible costs to be noted explicitly.
For example, `Person.say_hi1` takes `Self` by value,
which means it must copy the `Person` every time.
If `Person` were a much larger struct,
this could be very expensive and we don't want to hide that information.
Also, the difference between `.&` and `.&mut`
is explicit to make mutability explicit everywhere.

Table of Contents

Postfix

Most unary operators and keywords can be used postfix as well.

- `.if {}`
- `.if {} else {}`
- `.match {}`
- `.for {}`
- `.*` for dereference
- `.&` for pointer to
- `.&mut` for mutable pointer to
- `.!` for negation
- `.@()` for builtins, like as (casting), size_of, etc.
  - `.$cast(T)` : convert to `T`, like an int to float cast, or an int widening cast
  - `.$ptr_cast<T>()` : cast a pointer like `*T` to `*U`
  - `.$bit_cast<T>()` : reinterpret the bits, like from `u32` to `f32`
  - `.$size_of()` : size of a type
  - `.$align_of()` : alignment of a type
  - `.$call(func)` : call a function or closure in a unified syntax

Combined with everything being an expression,
`match`, and having methods,
this makes it much easier to write programs in a very fluid style.

Furthermore, and perhaps most importantly in practice,
this makes autocompletion vastly better, because an IDE can narrow down
what you may type next based on the type of the previous expression.
This can't be done with postfix operators and functions (rather than methods).
You get to think in one forward direction, rather than
having to jump from some prefix keywords to some postfix methods and fields.

Table of Contents

Slices

C* also has slices. These are a pointer and length,
and are much preferred to passing the pointer and length separately,
like you usually have to do in C.

They are implemented like this (not actually, but similarly):

```
struct Slice<T> {
    ptr: *T,
    len: usize,
}
```

But they can be written as `*[T]`. Actually, slices are unsized types,
so their type is just `[T]`, but usually `*[T]` is used and that
is what's equivalent to the above `Slice<T>`.

Unlike pointers like `*T`, slices can be indexed. By default,
using the indexing operator, this is bounds checked for safety,
but there are also unchecked methods for indexing. Usually, though,
bounds checking can be elided during sequential iteration,
so the performance hit is minimal, and can be side-stepped if really needed.

Slices can also be sliced to create subslices by indexing them with
a range (e.x. `[1..10]` or `[1..]`).
Again, this is bounds checked by default.

Table of Contents

Monadic Error-Handling

There are no exceptions in C*, just like C. It uses return values for error handling, similarly to C. But C* has much
better support for this using the `Option` and `Result` types.

The definitions of these types are:

```
enum Option<T> {
    None,
    Some(T),
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

That is, `Option` represents an optional value, and `Result`
represents either a successful `Ok` value or an error `Err` value.

There is special syntactic support for using these two monadic types
for error-handling using the `.?` postfix operator in `try` blocks:

```
struct IndexError {
    index: usize,
}

fn get_by_index<T>(a: *[T], i: usize): Result<T, IndexError> {
    if (i < a.len()) {
        Ok(a[i])
```

```
    } else {
        Err(IndexError {index: i})
    }
}

struct IndexPair {
    first: usize,
    second: usize,
}

fn get_two_by_index<T>(a: *[T], i: usize, j: usize): Result<T, IndexError> try = {
    let first = try {
        get_by_index(a, i).?
    };
    let second = get_by_index(a, j).?;
    IndexPair {first, second}
}
```

This desugars to

```
fn get_two_by_index<T>(a: *[T], i: usize, j: usize): Result<T, IndexError> ={
    let first = try {
        get_by_index(a, i).match {
            Ok(i) => i,
            Err(e) => return Err(e),
        }
    };
    let second = get_by_index(a, j).match {
        Ok(i) => i,
        Err(e) => return Err(e),
    }
    Ok(IndexPair {first, second})
}
```

As you can see, without the try `.?` operator and `try` blocks,
doing all the error handling with just `match` quickly becomes tedious.
This is also kind of like a monadic `do` notation, except it is in C*
limited to just the monads `Option<T>`, and `Result<T, E>` (over `T`).

Note also that `try` blocks can be specified at the function level
as well as normal blocks.

Table of Contents

Uncatchable Panics

While monadic error-handling with `Option` and `Result` is usually superior,
there are still cases where you have unrecoverable errors
(maybe you don't want to handle out of memory conditions),
or where you'd rather just end the program than handle the error.
In this case, you can `panic`, which will
print an error message and immediately `abort`.

To do this with an `Option` or `Result`,
you can just call `.unwrap()`, which will panic if
it was `None` or `Err` and return the `Some` or `Ok` value.

There is no language-supported unwinding.
`abort` is immediately called after a panic, and only the OS cleans things up.
Nothing is stopping you from calling `setjmp` and `longjmp` from C,
but no unwinding of `defer` statements is done,

and it may result in undefined behavior. There is no undefined behavior, however, in a normal panic because you just simply `abort`.

Table of Contents

Operator Precedence

The table below shows the operator precedence for binary and unary operators from lowest precedence to highest precedence.

Operator	Description	Associativity
;	sequencing	Left
=	assignment	Right
.	access	Left
		or
&	and	Left
== !=	equality/inequality	Left
< > <=>=	comparison	Left
+-	addition/subtraction	Left
*/	multiplication/division	Left
-	negation	Right
!	logical NOT	Right
?	conditional	Left

In C* generics have a higher precedence than comparison thus removing ambiguity from "< >".

Table of Contents

Examples

Table of Contents

GCD

Here is how you write simple algorithms like GCD in C*:

```
fn gcd(a: i64, b: i64): i64 = {
    (fn gcd(a: u64, b: u64): u64 = {
        match b {
            0 => b,
            _ => gcd(b, a % b),
        }
    })(a.abs(), b.abs()).$cast(i64)
}
```

Systems Programming

Here is an example program in C* for part of a simple HTTP/1.0 server,
equivalent to part0 of hw3 in Jae's OS class
(https://gist.github.com/RyanLee64/hash-redacted).
It showcases many of C*'s notable features,
like enums, methods, generics, defer, expression-orientedness,
postfix operators, pattern matching, closures, monadic error handling,
and byte, c, and format strings.

That code (the ported part) is ~230 LOC, while the C* below is only ~80 LOC,
and it is more correct in error handling and edge cases,
faster in places (though IO dominates here), and the business logic
stands out more (while less important aspects like errors, resource cleanup,
allocations, and string handling stay in the background).
That is, C* allows you to be simulatenously more expressive
while still staying correct and explicit,
and the performance is just as good if not better.

```
enum Status {
    Ok,
    NotImplemented,
    BadRequest,
    // rest skipped for brevity
}

struct RequestLine {
    method: *[u8],
    uri: *[u8],
    version: *[u8],
}

impl RequestLine {
    fn check(self: *Self): Result<(), Status> try = {
        let Self {method, uri, version} = self.*;
        match (method, version) {
            (b"GET", b"HTTP/1.0" | b"HTTP/1.1") => {},
            _ => Err(Status.NotImplemented).?,
        }
        if uri.starts_with(b'/').! || uri.equals(b"/..") || uri.contains(b"/../") {
            Err(Status.BadRequest).?;
        }
    }
}

fn main(): Result<(), AnyError> try = {
    let (port, web_root) = std.env.argv().match {
        [_, port, web_root] => (port.parse<u16>().?, web_root),
        [program, ...] => Err(f"usage: {program} <server_port> <web_root>").?,
    };
    let server_socket = Socket.new(PF_INET, SOCK_STREAM, IPPROTO_TCP).?;
    defer server_socket.&.close();
    server_socket.&.bind(SocketAddr {
        family: AF_INET,
        addr: InetAddr {
            addr: INADDR_ANY.to_be(),
        },
        port: port.to_be(),
    }).?;
    server_socket.&.listen(5).?;
    let mut request_line_buf = Vec.new();
    defer request_line_buf.free();
    let mut line_buf = Vec.new();
    defer line_buf.free();
```

```
    loop try {
        let client_socket = server_socket.&.accept().?;
client_socket_close:
        defer client_socket.&.close();
        let mut client_stream = fdopen(client_socket.fd, c"r").?;
        undefer client_socket_close; // stream (`FILE *` in C) takes ownership
        defer client_stream.&.close();
        let line_or_status = try {
            // read and parse request line
            let line = client_stream.&mut.read_line(buf.&mut)
                .map_err(fn(_) Status.BadRequest).?
                .split(fn(b) " \t\r\n".contains(b)).match {
                    [method, uri, version] => RequestLine { method, uri, version },
                    _ => Err(Status.NotImplemented).?,
                };
            line.&.check().?;
            // read headers, skip them
            loop {
                client_stream.&mut.read_line(buf.&mut)
                    .map_err(fn(_) Status.BadRequest).?
                    .match {
                        "\n" | "\r\n" => break,
                        _ => {},
                    }
            }
            line
        }
        let (line, status) = match line_or_status {
            Ok(line) => (line, Status.Ok),
            Err(status) => (RequestLine { method: b"", uri: b"", version: b"" }, status),
        };
        client_socket.write(f"HTTP/1.0 {status.code()} {status.reason()}\r\n\r\n").?;
        match line_or_status {
            Ok(_) => handle_request(web_root, line.uri, client_socket).?,
            Err(_) => client_socket.write(f"<html><body>\n<h1>{status.code()} {status.reason()}</h1>\n</body>
        }
        eprintln(f"{client_socket.addr} \"{line.method} {line.uri} {line.version}\" {status.code()} {status.r
    }
}
```

Table of Contents

Project Timeline / Git Log

```
commit 29e1ef5c2f82d7119244685d01509846cf3ed95f
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 22:09:09 2021 -0500

    Serialized mdpdfs.

 justfile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 645c46eb981421516dea2e723aff32f7e2bf9757
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 22:04:47 2021 -0500

    Added extensions for generated pdfs.

 justfile | 4 +---
 1 file changed, 1 insertion(+), 3 deletions(-)

commit b07ec2b6fb829385175c150a3f2470eefc8ab062
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 21:34:25 2021 -0500

    Typo.
```

```
 justfile | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)

commit a1263f6c22e1f371b1b52bf1678527d8917c54ce
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 21:32:25 2021 -0500

    Some other report changes.

 docs/proposal.md | 2 +-
 justfile         | 9 ++++++---
 2 files changed, 7 insertions(+), 4 deletions(-)

commit cb6aba81534b43156a4ec70531c30e9ad3d60832
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 21:22:23 2021 -0500

    Fixed some old `.@intrinsics` to `.$instrinsics`.

 docs/LRM.md | 22 ++++++++++-----------
 1 file changed, 11 insertions(+), 11 deletions(-)

commit eb1efb82eb45e76a8ff08d1752240a46d40fe3f0
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 21:09:07 2021 -0500

    Fixed pdf generation.

 justfile | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

commit f2397211eec49e8ef8c719d6e43cb5a8a1536c3e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 20:59:39 2021 -0500

    Added rest of report generation and moved it to `report/`.

 .gitignore      |  1 +
 docs/.gitignore |  3 ---
 justfile        | 39 +++++++++++++++++++++++++++++++--------
 3 files changed, 32 insertions(+), 11 deletions(-)

commit 9e21747a7c65099de2652ef66533202d85a49b7b
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 20:39:12 2021 -0500

    Added mdpdf (markdown to pdf) install to setup.

 setup.sh | 5 +++++
 1 file changed, 5 insertions(+)

commit ae432225f020e929cea7ab9949dcb6147fb2185c
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 20:38:57 2021 -0500

    Removed notes from top of LRM.

 docs/LRM.md | 76 ------------------------------------------------------------
 1 file changed, 76 deletions(-)

commit adf4b62202333457bb6e1b982ef1f56487ab3b51
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 20:09:42 2021 -0500

    Added generated git log for project timeline log.

 docs/.gitignore | 1 +
 justfile        | 9 +++++++++
 2 files changed, 10 insertions(+)

commit 5f528de45fd058b8c37f518bad254e30b33ee0ac
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 14:16:04 2021 -0500
```

```
    Added code listing markdown generator script.

 docs/.gitignore |   2 +
 justfile        | 232 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 234 insertions(+)

commit e2e73c393a305fb3d30ba3f053c42a2fdf503f6d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 12:55:32 2021 -0500

    Moved docs into separate directly, but kept a LRM symlink for old URLs.

 LRM.md                          | 2396 +---------------------------------------
 README.md                       |    4 +-
 docs/LRM.md                     | 2395 ++++++++++++++++++++++++++++++++++++++
 proposal.md => docs/proposal.md |    0
 4 files changed, 2398 insertions(+), 2397 deletions(-)

commit 70d0c6958d4df56f84a27739a2f2576a90103db5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 12:54:17 2021 -0500

    Fixed a couple mistakes in the proposal.

 proposal.md | 10 +++++-----
 1 file changed, 5 insertions(+), 5 deletions(-)

commit f82b6c5483c715827e4a8d8b472c793fdd0f970f
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 12:53:51 2021 -0500

    Restored to a working state.

 src/codegen.ml  |  8 +++++---
 src/compiler.ml | 13 +++++++++----
 2 files changed, 14 insertions(+), 7 deletions(-)

commit 8e7208f0af6e2a4326f6a099b4e199e204ec84e6
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 22 05:21:49 2021 -0500

    Working on codegen.

 src/codegen.ml | 211 ++++++++++++++++++++++++++++++++++++++++++++++++++++++---
 src/lir.ml     |  14 +++-
 2 files changed, 212 insertions(+), 13 deletions(-)

commit cb6068ca0384421603a89a80bf6bebbb2112f11f
Author: Khyber Sen <kkysen@gmail.com>
Date:   Tue Dec 21 07:28:52 2021 -0500

    Working on codegen.

 LICENSE                          |   2 +-
 src/ast.ml                       |   2 +
 src/ast.mli                      |   2 +
 src/codegen.ml                   | 165 +++++++++++++++++++++++++-------------
 src/codegen.mli                  |   4 +
 src/compiler.ml                  | 105 +++++++++++++++----------
 src/compiler.mli                 |   1 +
 src/driver.ml                    |   3 +-
 src/emitType.ml                  |  26 +++---
 src/emitType.mli                 |   1 +
 src/lir.ml                       | 131 +++++++++++++++++++++++++++++++
 test/codegen/a.cstar.lir.json    |   7 ++
 test/{ => end-to-end}/.gitignore |   0
 13 files changed, 335 insertions(+), 114 deletions(-)

commit 60e30d762d4670028bf556ef9bbf301081d7bfdd
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Dec 20 02:04:38 2021 -0500

    Removed libcstar which didn't really have anything anyways.  Only using libc.
```

```
 justfile           |  7 +------
 libcstar/Makefile  | 20 --------------------
 libcstar/println.c |  5 -----
 libcstar/println.h |  6 ------
 4 files changed, 1 insertion(+), 37 deletions(-)

commit 387bb2cf60bf811b63050bda495c5c1daadfad46
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Dec 20 02:02:27 2021 -0500

    Commented out things until it compiled and worked.

 justfile                    |   6 +
 src/ast.ml                  | 298 ++++++++-----------
 src/ast.mli                 | 425 ++++++++++++++-------------
 src/compiler.ml             |  18 +-
 src/cstar.ml                | 174 -----------
 src/driver.ml               |  48 ++--
 src/driver.mli              |   1 +
 src/emitType.ml             |   2 +-
 src/parser.mly              | 686 ++++++++++++++++++++++---------------------
 src/token.ml                |   1 +
 test/end-to-end/empty.cstar |   2 +-
 11 files changed, 744 insertions(+), 917 deletions(-)

commit a9615867dbeae25a1190c4d80c2d302733ae6d9b
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Dec 19 18:28:08 2021 -0500

    Wrote the parser, but a bunch of shift/reduce and reduce/reduce conflicts.  But otherwise, it's complete

 LRM.md                      | 153 +++++++++++++--
 setup.sh                    |   2 +-
 src/ast.ml                  | 350 ++++++++++++++++++++-------------
 src/compiler.ml             |   7 +-
 src/lexer.mll               |  10 +
 src/parser.mly              | 470 ++++++++++++++++++++++++++++++++++++++++++--
 src/token.ml                |  17 +-
 src/token.mli               |  13 +-
 test/end-to-end/empty.cstar |   1 +
 test/end-to-end/hello.cstar |   2 +-
 10 files changed, 848 insertions(+), 177 deletions(-)

commit a8ab9e85bc40d1366e0804bf3ecf2d5ea4221e75
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Dec 16 20:41:14 2021 -0500

    Set up tokens in `parser.mly`.

 src/ast.ml       |   7 ++-
 src/ast.mli      |   7 ++-
 src/compiler.ml  | 120 +++++++++++++++++++++++++++++++++++++++++++++++++--------
 src/compiler.mli |   6 +++
 src/cstar.ml     | 133 ++++++++++++++++++++++++++++-------------------------
 src/parser.mly   |  84 ++++++++++++++++++++++++++----------
 src/token.ml     |   6 ---
 src/token.mli    |   6 ---
 8 files changed, 247 insertions(+), 122 deletions(-)

commit 3491c59d5035440e106168c90a19020f8b618efb
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 12 18:00:05 2021 -0500

    forgot to push most recent codegen

 src/codegen.ml | 33 ++++++++++++++++++++++++---------
 1 file changed, 24 insertions(+), 9 deletions(-)

commit 905918adc1b4fb2e26b88350edc257e88c4cd020
Merge: bc33cf1 5ce5e9d
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 12 17:53:04 2021 -0500

    Merge branch 'main' of github.com:kkysen/cstar into main
```

```
commit bc33cf1cbfbed04f9b5f872f410e1ec77289c4f0
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 12 17:52:51 2021 -0500

    libcstar skeleton compiled through just and Makefile

 justfile           |  7 ++++++-
 libcstar/Makefile  | 20 ++++++++++++++++++++
 libcstar/println.c |  5 +++++
 libcstar/println.h |  6 ++++++
 4 files changed, 37 insertions(+), 1 deletion(-)

commit 5ce5e9ddd890fcaff0ea720459bea9c6b62820e5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Dec 12 14:58:45 2021 -0500

    Clarified readme a bit more.

 README.md | 5 ++++-
 1 file changed, 4 insertions(+), 1 deletion(-)

commit d3d629cf4e8eb7c87ca79b3afc2ceccaf97f92bc
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Dec 12 14:44:13 2021 -0500

    Clarified things more in the readme instructions.

 README.md | 35 +++++++++++++++++++++++-----------
 1 file changed, 24 insertions(+), 11 deletions(-)

commit d90f108a4a27723b65694bfbd721c9d1ecdf8cc5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Dec 12 14:38:04 2021 -0500

    Added `just build` to `./setup.sh build` so it puts `cstar` on `$PATH`.

 setup.sh | 10 +++++++---
 1 file changed, 7 insertions(+), 3 deletions(-)

commit fed2fb26025c4ac0cf40fd48f835bcf57479f5e8
Author: shannonjin <shannonj112@gmail.com>
Date:   Sun Dec 12 12:53:19 2021 -0500

    add aditional steps to building

 README.md | 3 +++
 1 file changed, 3 insertions(+)

commit 985b7db118f1ace0046a58adb1bbfe9855ac77a9
Merge: ca0cbe3 ce75c70
Author: shannonjin <shannonj112@gmail.com>
Date:   Sun Dec 12 12:48:51 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit ca0cbe352aaec45da03a9783111614eba0733bef
Author: shannonjin <shannonj112@gmail.com>
Date:   Sun Dec 12 12:48:45 2021 -0500

    update steps

 README.md | 32 ++++++++++++++++++++++++++++++++
 1 file changed, 32 insertions(+)

commit 2a153e757acd3d24e19695222f4a74f24fdaa24b
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 12 01:25:42 2021 -0500

    integer and float types

 src/codegen.ml | 42 ++++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 41 insertions(+), 1 deletion(-)
```

```
commit ce75c704d4d60fb0a71b90abc8f1d15e6bcfcb36
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Dec 12 00:29:47 2021 -0500

    Added keywords to the lexer.

 src/lexer.mll | 11 +++++++++--
 src/token.ml  | 55 +++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/token.mli | 29 +++++++++++++++++++++++++++++
 3 files changed, 93 insertions(+), 2 deletions(-)

commit 48221357865c4516f09089538014dcf3c5ab7d97
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Dec 11 02:31:00 2021 -0500

    statrting codegen files

 src/codegen.ml  | 31 +++++++++++++++++++++++++++++++
 src/codegen.mli |  0
 2 files changed, 31 insertions(+)

commit 8bb9baa447f1d7a299905bab6cdd2ae96e0949bd
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 19:35:05 2021 -0500

    Added number literals to the lexer.

    Had to change the grammar to remove an ambiguity.
    Hex literals need to use uppercase 'A-F's now,
    and scientific notation needs to use a lowercase 'e'.

    Also, ocamllex is quite shitty in that you can't have recursive rules.
    You can combine regexes with variables, but you can't access
    all their named capture groups because it's not recursive.

 LRM.md                      |  4 +--
 src/lexer.mll               | 75 ++++++++++++++++++++++++++++++-----------------
 src/token.ml                | 71 +++++++++++++++++++++++++++++++++++++++++++++-
 src/token.mli               |  8 ++++-
 test/end-to-end/hello.cstar | 22 +++++++++++++
 5 files changed, 146 insertions(+), 34 deletions(-)

commit 9d8a45617345c48ffca7cc0c425385b9d08d05f5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 16:49:04 2021 -0500

    Fixed block comment value in lexer.

 src/lexer.mll | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 3e59b85aa7460c830a5dca6f1486331b019694b5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 03:09:42 2021 -0500

    Fixed cstar hello world (missing newline).

 test/end-to-end/hello.cstar | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 00f06999ad89886b92bd4361a1668d0d041e81f9
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 03:07:22 2021 -0500

    Fixed a couple issues with the lexer (save whitespace and make token list forward (was backward before)).

 src/cstar.ml  | 2 +-
 src/lexer.mll | 2 +-
 src/token.ml  | 2 +-
 src/token.mli | 2 +-
 src/util.ml   | 2 +-
 5 files changed, 5 insertions(+), 5 deletions(-)

commit 2c38e41133346e3e1fb48880df470530f5e9739cf
```

```
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 02:57:47 2021 -0500

    Added identifiers to the lexer, so now hello world successfully lexes!

 src/lexer.mll | 19 ++++++++++++++++---
 1 file changed, 16 insertions(+), 3 deletions(-)

commit 7597388182baff58fed4e80156cb57c3b2dc402d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 02:42:25 2021 -0500

    Added the real lexer (not finished) into `Compiler.Lexer` so it actually runs.

 src/compiler.ml |  9 +++++++--
 src/driver.ml   | 16 ++++------------
 src/lexer.mll   |  1 +
 src/util.ml     | 16 ++++++++++++++++
 src/util.mli    |  4 ++++
 5 files changed, 32 insertions(+), 14 deletions(-)

commit a73573354ab42467ce6e705982be7b2360bee35e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 02:30:03 2021 -0500

    Finished comments and string/char literals in the lexer.

 src/cstar.ml  |  1 -
 src/lexer.mll | 65 ++++++++++++++++++++++++++++++++---------------------------
 src/token.ml  | 10 ++++-----
 src/token.mli | 10 ++++-----
 4 files changed, 43 insertions(+), 43 deletions(-)

commit a82ad384d57f4dde3ab1c97e40b5aba1f6f65550
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 8 02:29:35 2021 -0500

    Put `parser.mly` back into a compiling state by commenting out some stuff.

 src/parser.mly | 32 +++++++++++++++++++-------------
 1 file changed, 19 insertions(+), 13 deletions(-)

commit f0a2d732424d949623652ee5214ade72e976c4ba
Author: Khyber Sen <kkysen@gmail.com>
Date:   Tue Dec 7 17:57:30 2021 -0500

    Added `just watch-and-run`.

 justfile | 5 +++++
 setup.sh | 1 +
 2 files changed, 6 insertions(+)

commit 969e037f1da8610e4a70d86ed19d37def4967c27
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Dec 5 00:57:03 2021 -0500

    Fixed `cstar compile-raw` description.

 src/driver.ml | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit d3407d84de52e620429dc02542481f1e8e35b4af
Merge: 897882a 29e9a1a
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Dec 4 19:53:05 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 897882a97dd3fb73ce97e5ba66dfd3406b257854
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Dec 4 19:52:59 2021 -0500

    Fixed the esy install and llvm patches. Now `./setup.sh dev` should fully work on a fresh Ubuntu/Debian.
```

```
 patches/.gitignore            |  1 +
 patches/llvm-install.sh.patch |  4 +++-
 setup.sh                      | 12 ++++++++++--
 3 files changed, 14 insertions(+), 3 deletions(-)

commit 29e9a1af7b81af508cca406b398dfa1a079374c0
Author: shannonjin <shannonj112@gmail.com>
Date:   Sat Dec 4 17:11:22 2021 -0500

    Basic parser

 src/parser.mly | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 01321bc2f42ea01d2be9819f6cfc4a9569d00c53
Author: shannonjin <shannonj112@gmail.com>
Date:   Sat Dec 4 17:09:23 2021 -0500

    test

 src/parser.mly | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 1e4ed2f6cb76ac3c3609c91ba21f0569f80eaf3f
Merge: 34efd8b 07af991
Author: Shannon <shannon@dyn-160-39-207-146.dyn.columbia.edu>
Date:   Sat Dec 4 16:56:35 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 34efd8b631fe5b1e45bb8937f327b96fcc3337d5
Author: Shannon <shannon@dyn-160-39-207-146.dyn.columbia.edu>
Date:   Sat Dec 4 16:56:03 2021 -0500

    Basic starter parser

 src/parser.mly | 38 ++++++++++++++++++++++++++++++++++-----
 1 file changed, 33 insertions(+), 5 deletions(-)

commit 07af991135054eeda07103 96c895a2567a0c8bdc
Author: Shannon <shannon@dyn-160-39-207-146.dyn.columbia.edu>
Date:   Sat Dec 4 16:53:27 2021 -0500

    Basic starter parser

 src/parser.mly | 38 ++++++++++++++++++++++++++++++++++-----
 1 file changed, 33 insertions(+), 5 deletions(-)

commit 06878cc276fd4aa318a06d0570263fd9f1f54cb4
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 16:27:31 2021 -0500

    Typo.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 818b27020937afb3ae6a18f1a85047a91a40f3af
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 16:05:24 2021 -0500

    Added `cmake` dependency for building llvm bindings.

 setup.sh | 2 ++
 1 file changed, 2 insertions(+)

commit 57c90ee988eb3e255b17539c11facbd2f20ce6f9
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 15:22:03 2021 -0500

    Forgot to add `install-llvm` to build deps.

 setup.sh | 1 +
 1 file changed, 1 insertion(+)
```

```
commit fe54d8efb5890c9248ab0ad212110663a27fce9c
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 14:37:19 2021 -0500

    Added `wget` for vscode.

 setup.sh | 1 +
 1 file changed, 1 insertion(+)

commit 99c18f370b6e6779d9c96e41e28cb2cc96b957e4
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 14:32:44 2021 -0500

    Trying to fix `link`.

 setup.sh | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

commit 75099335eeaf71cfba6756e532a2714ed176a672d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 14:28:54 2021 -0500

    Don't make infinite recursive links.

 setup.sh | 5 ++++-
 1 file changed, 4 insertions(+), 1 deletion(-)

commit 067091ab83b9c055082955b8c43bff05716df0df
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 14:28:39 2021 -0500

    Prefer `apt` over `brew` since `brew` sometimes doesn't work.

 setup.sh | 8 ++++----
 1 file changed, 4 insertions(+), 4 deletions(-)

commit 1067d6e05311b99bb3fd17552ccf068603996 50e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 14:12:54 2021 -0500

    Made `build-essential` check for `cc` actually.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 4eb2922bdf572744e99ccf2acab178b32d68bb72
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 14:06:32 2021 -0500

    Fixed `cached-install` when there's no exe name.

 setup.sh | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)

commit 20552b03a7513e5313f7e8338c2d88b71e32420d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 13:52:44 2021 -0500

    Fixed build-essential install.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 23e83c1fc92f0663372decd5217ce262fa788c0e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 13:48:56 2021 -0500

    Added `build-essential` install for a C compiler.

 setup.sh | 1 +
 1 file changed, 1 insertion(+)

commit f1fa3dee374fbfe6c5a453a3290e6ff7be676b03
```

```
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 13:41:37 2021 -0500

    Stopped using `install-in-parallel` since it messes with stdin.

 setup.sh | 6 ++++--
 1 file changed, 4 insertions(+), 2 deletions(-)

commit e483ce32b5f1291db5c3d3f46776d081501ba547
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 13:37:41 2021 -0500

    Trying to fix cargo installation.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 95b69005aee6ee84912b6bac7acd3d42b6fa9838
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 13:32:41 2021 -0500

    Added `-y` to package installs.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 259c314316c314c8223547ed56e2735ccd2ffd64
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 13:19:55 2021 -0500

    Fixed `esy-install` when `esy install` fails so we can patch llvm and re-run it.

 setup.sh | 3 +--
 1 file changed, 1 insertion(+), 2 deletions(-)

commit 81eb0547f707d921cce4dbd1265d58ebd70c3c5d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Dec 3 11:47:11 2021 -0500

    Fixed `package-install-raw args`.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit c1ddce3e40b47147896101bf449733f726a9469d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Dec 2 19:45:47 2021 -0500

    Driver commands are run in-process by default now.
    Also added a few new options:

    * `--exe-extension`: use `.cstar.exe` as the extension for executables
                         (easier for cross-compability with windows and
                         gitignor'ing all executables)
    * `--run-driver-commands-in-new-processes`: there's still a reason clang does this

 src/driver.ml    | 201 ++++++++++++++++++++++++++++++++++---------------------------
 src/driver.mli   |  20 +++++-
 src/emitType.ml  |  29 +++++---
 src/emitType.mli |   6 +-
 test/.gitignore  |   1 +
 5 files changed, 164 insertions(+), 93 deletions(-)

commit 3f352804ec9419bbcb3937bb85ebe2dcbcfb1ab5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Dec 2 18:41:57 2021 -0500

    Added `just path` as a slight-shortcut for `just setup path`.

 README.md | 5 +++--
 justfile  | 2 ++
 2 files changed, 5 insertions(+), 2 deletions(-)

commit e3961ab316c240c8ccb2cbad7fd30daafa6815cd
```

```
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Dec 2 18:39:22 2021 -0500

    Set up serializable staged compilation for cstar stages (llvm stages already done before).

 src/ast.ml       |   7 ++-
 src/ast.mli      |   6 ++-
 src/compiler.ml  | 160 ++++++++++++++++++++++++++++++++++++++++++++++++++-----
 src/compiler.mli |  25 ++++++++-
 src/cstar.ml     |  96 +++++++++++++++++----------------
 src/driver.ml    |  52 +++++++++---------
 src/emitType.ml  |  53 +++++++++++-------
 src/emitType.mli |   5 +-
 src/token.ml     |   6 +++
 src/token.mli    |   6 +++
 10 files changed, 310 insertions(+), 106 deletions(-)

commit 43fe58df7f57da759320cedb7720b08254c830fd
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Dec 2 12:41:38 2021 -0500

    Print all tokens as a json array, not lines of json tokens.

 src/cstar.ml | 8 ++++++--
 1 file changed, 6 insertions(+), 2 deletions(-)

commit 1ea83dd60e08c2e448f2bc4a15a77c3e299a5bbc
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Dec 2 12:41:09 2021 -0500

    Use `Yojson.Safe.Util.to_assoc` instead of a `match` and `failwith`.

 src/stringMap.ml | 24 +++++++++++-------------
 1 file changed, 11 insertions(+), 13 deletions(-)

commit bfc0126070837b99134bc18ec7179194a7384273
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:45:25 2021 -0500

    Derived yojson for `ast`.

 src/ast.ml        | 108 +++++++++++++++++++++++++++---------------------------
 src/ast.mli       | 106 ++++++++++++++++++++++++++---------------------------
 src/cstar.ml      |  12 ++++--
 src/stringMap.ml  |  19 ++++++++++
 src/stringMap.mli |   4 ++
 5 files changed, 140 insertions(+), 109 deletions(-)

commit d2501b61ec88a898d69b05f78b328bebb9885828
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:45:10 2021 -0500

    `just expand` now saves the output in a temporary so you can run things on it.

 justfile | 20 ++++++++++++++++----
 1 file changed, 16 insertions(+), 4 deletions(-)

commit 5fbb45558b11f29d78119529d57426c1bfb50247
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:07:46 2021 -0500

    Derived yojson for `token`.

 src/token.ml  | 18 +++++++++---------
 src/token.mli | 18 +++++++++---------
 2 files changed, 18 insertions(+), 18 deletions(-)

commit 461c162d6dcb52aa0321984318ad3c335480190e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:07:26 2021 -0500

    You can set `debug=1` for `setup.sh` to print its commands as it goes.

 setup.sh | 4 ++++
```

```
 1 file changed, 4 insertions(+)

commit de01f2897b543fe19fd5fd5a6bdd3f7295e0d3a3
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:07:02 2021 -0500

    Added `just watch`, which builds and watches for changes to rebuild.

 justfile | 4 ++++
 setup.sh | 1 +
 2 files changed, 5 insertions(+)

commit 1248985748ecb0e4009d7d1e5ceee67097ee146a
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:06:33 2021 -0500

    Added `bat` as a dependency since I use it in `just expand`.

 setup.sh | 1 +
 1 file changed, 1 insertion(+)

commit d04bf29463e97c6789981a06d3a28628255a241d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:06:04 2021 -0500

    Fixed a bunch of bugs in `setup.sh` around `package-install`.

 setup.sh | 27 +++++++++++++++++------------
 1 file changed, 15 insertions(+), 12 deletions(-)

commit fa805fb0e38722eba57f99341a11ca98aaeaf3e1
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:03:52 2021 -0500

    Added `just expand <path>`, which expands the ppx of an ocaml file.

 justfile | 6 ++++++
 1 file changed, 6 insertions(+)

commit 351bfa450d278c7e652ca443dd00ab663f122e48
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:02:16 2021 -0500

    Found a better way to look up the right `cstar.exe` path in a generic way (so for any path).

 justfile | 13 ++++++++++---
 1 file changed, 10 insertions(+), 3 deletions(-)

commit 40b67cddec6eba8f01d9177a7732f83d3ac3f605
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Dec 1 19:01:31 2021 -0500

    Added `ppx_yojson_conv` for `[@@deriving yojson]` so we can easily print things as json (and (de)serializ

 package.json | 1 +
 src/dune     | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit 0086652371c4876dbd6ef9ac6596d014a74fbe77
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 28 02:30:21 2021 -0500

    Refactored out `emit_type` into `emitType.ml`, now as `EmitType.t`.

 src/driver.ml    | 135 +++++++++++++++++-----------------------------------------
 src/driver.mli   |  14 +-----
 src/emitType.ml  |  91 ++++++++++++++++++++++++++++++++++++++++++
 src/emitType.mli |  29 +++++++++++++
 4 files changed, 142 insertions(+), 127 deletions(-)

commit f700ff5bcdf495506b6e49ff64d78df4c24af893
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 28 02:10:04 2021 -0500
```

```
    Refactored a bunch of things into `driver.ml`.

 src/cstar.ml   | 363 +-----------------------------------------------------
 src/driver.ml  | 362 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/driver.mli |  15 +++
 3 files changed, 378 insertions(+), 362 deletions(-)

commit c721938c243a9fb150697e70fcb758150ad01a89
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 28 02:00:08 2021 -0500

    Refactored the codegen into `compiler.ml`.

 src/compiler.ml  | 43 +++++++++++++++++++++++++++++++++++++++++++
 src/compiler.mli |  1 +
 src/cstar.ml     | 43 +-----------------------------------------
 3 files changed, 45 insertions(+), 42 deletions(-)

commit 366826abc198ed8a33f0ec9d63a27e388c892289
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 24 06:09:11 2021 -0500

    Changed the hardcoded hello world to use the llvm ocaml bindings instead of hardcoding the llvm ir string

 src/cstar.ml | 50 ++++++++++++++++++++++++++++++++++---------------
 1 file changed, 35 insertions(+), 15 deletions(-)

commit 5b8cd52d55f77012b782baee9136d0aa848bce57
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 24 06:08:46 2021 -0500

    Instead of `--save-temps obj`, we let `--save-temps ''` mean that.

 src/cstar.ml | 5 +++++
 1 file changed, 5 insertions(+)

commit 74ff1a315ad7a045ca1863264e01ff912212c494
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 24 06:07:08 2021 -0500

    Fixed `just cstar-path` so only the newest `cstar.exe` is linked.

 justfile | 6 ++++--
 1 file changed, 4 insertions(+), 2 deletions(-)

commit 6dc42954d5480844d51ef9bb0ae35dc0a6425c34
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 24 06:06:46 2021 -0500

    Added llvm libraries to dune so we can actually use them (llvm's split up into a bunch).

 src/dune | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit eaa04abd87780686b0f053f225ed2e53690fbe34
Author: Khyber Sen <kkysen@gmail.com>
Date:   Tue Nov 23 15:47:00 2021 -0500

    Added llvm ocaml bindings dependency to esy, along with a patch needed since esy breaks on the llvm depen

 package.json                  |  1 +
 patches/llvm-install.sh.patch | 19 +++++++++++++++++++
 setup.sh                      | 16 ++++++++++++++--
 3 files changed, 34 insertions(+), 2 deletions(-)

commit 2f4ec00b507912a7c70f1310801b4fd7a9907b7c
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 19 15:02:46 2021 -0500

    Added `eval "$(./setup.sh path)"` or `eval "$(just setup path)"` to easily add things to $PATH.

 README.md | 5 ++---
 setup.sh  | 7 +++++++-
 2 files changed, 8 insertions(+), 4 deletions(-)
```

```
commit 3dd4f546346a4c90068ec6f93f017b36bb13b293
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 19 14:56:05 2021 -0500

    Improved setup script and made it work on macOS.

 setup.sh | 194 +++++++++++++++++++++++++++++++++++++++++++++++++--------------
 1 file changed, 153 insertions(+), 41 deletions(-)

commit 2455bb4ad9e16844e62f025ce429f3e4a8a22c32
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 19 13:17:25 2021 -0500

    Removed llvm again from esy since it's not working rn.

 package.json | 1 -
 1 file changed, 1 deletion(-)

commit 4a043ce12306f68b5a9c227ae55a06d01fe044b7
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 19 13:16:54 2021 -0500

    Removed `llvm.sh`, which should be in `.gitignore`.

 .gitignore |  3 ++-
 llvm.sh    | 82 --------------------------------------------------------------
 2 files changed, 2 insertions(+), 83 deletions(-)

commit 02a6195cd8daea039fbaba942c78e53c2680232d
Merge: 3fc3eec aaf5ca5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 19 13:14:08 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 3fc3eecb88a262fae858fd24dee90175260f4915
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 19 13:14:01 2021 -0500

    Fixed `quote` func in `justfile` for empty strings.

 justfile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit aaf5ca5acd84d1f8d66b6bd732726813a9e5581d
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Fri Nov 19 13:13:39 2021 -0500

    installation changes

 justfile |  2 +-
 llvm.sh  | 82 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 setup.sh |  6 ++---
 3 files changed, 86 insertions(+), 4 deletions(-)

commit e6fb4ca27e3b1d45eac7438eef127aa34148ad3a
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 14 02:41:01 2021 -0500

    Added llvm 13 ocaml bindings.

 package.json | 1 +
 1 file changed, 1 insertion(+)

commit 23e7ec6e5d587b86debc4e330d7dbb8fde805d22
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 07:21:55 2021 -0500

    Now each sub driver command exits if it had an error.

 src/cstar.ml | 37 ++++++++++++++++++++++---------------
 1 file changed, 22 insertions(+), 15 deletions(-)
```

```
commit 2f13e2dc6fa0800226f4af8b6f2c01b2ed4f7761
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 07:01:46 2021 -0500

    Got hello world to work! (totally hardcoded though lol)

 src/cstar.ml | 24 ++++++++++++++++++++++--
 1 file changed, 22 insertions(+), 2 deletions(-)

commit 209c40ce68fc946244e70cbf182464df5038f541
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 06:45:47 2021 -0500

    Got the compiler driver working now (i.e. llvm is invoked correctly in stages); this is how clang does it

 justfile     |   5 +-
 src/cstar.ml | 199 +++++++++++++++++++++++++++++++++++++++++++++++++++++------
 2 files changed, 183 insertions(+), 21 deletions(-)

commit 9c1ae91e0c00822d1b7187586bc990e427ecfc87
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 03:59:30 2021 -0500

    Forgot obj files in the pipeline.

 src/cstar.ml | 5 ++++-
 1 file changed, 4 insertions(+), 1 deletion(-)

commit 9c9f4a0cd28dc7bc3e6b9f07d66d73b4734c4c07
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 03:57:15 2021 -0500

    Started setting up the compilation pipeline (src -> ast/ir -> bc -> asm -> exe).

 src/cstar.ml | 129 +++++++++++++++++++++++++++++++++++++++++++++--------------
 1 file changed, 98 insertions(+), 31 deletions(-)

commit 0003d473435428cf9667260ecff7aa3e894ce7f1
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 02:44:19 2021 -0500

    Added `--emit [exe|asm|bc|ir|ast]`.

 src/cstar.ml | 84 +++++++++++++++++++++++++++++++++++++++++++++++++++++++-----
 1 file changed, 77 insertions(+), 7 deletions(-)

commit 8a2fc0dfb873abad88ce63eed938152768c224e3
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 01:40:59 2021 -0500

    Moved cargo installs to install-build, not install-dev, since `just build` is required to build.

 setup.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 4eb322feecd658a10f2b30f7c06592ff3dd79a46
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 13 01:38:11 2021 -0500

    Added simple command line parsing and shell autocompletion.

 README.md    | 13 +++++--------
 justfile     |  5 +++++
 package.json |  8 +++-----
 src/cstar.ml | 55 ++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 src/dune     |  9 +++++----
 5 files changed, 72 insertions(+), 18 deletions(-)

commit f6242c121b5614101be12257e7c43acf87ea3687
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 20:44:13 2021 -0500

    `./setup.sh dev` now makes `cstar` available in `./bin/`, too.
```

```
 setup.sh | 1 +
 1 file changed, 1 insertion(+)

commit 23108cd383593215e79c2068099a7d0a299eca5e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 20:40:57 2021 -0500

    `just build` now makes `cstar` available in `$PATH`.

 dune-project | 1 +
 justfile     | 6 +++++-
 package.json | 9 ++++++++-
 src/dune     | 3 ++-
 4 files changed, 16 insertions(+), 3 deletions(-)

commit 0d9facaeee7a90424e86d41bb1f303d43359ee0d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 20:28:37 2021 -0500

    Improved `install-tooling.sh` (now `setup.sh`) and now it installs llvm 13, too.

 .gitignore         |   1 +
 README.md          |   8 ++-
 install-tooling.sh | 118 -----------------------------------------
 justfile           |   7 +--
 setup.sh           | 151 +++++++++++++++++++++++++++++++++++++++++++++++++++++
 5 files changed, 160 insertions(+), 125 deletions(-)

commit 284642043c10c8f38a9f337d0c523a1343df8997
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 14:54:57 2021 -0500

    Added a hello world example written in C*, C, and LLVM IR.

 test/end-to-end/hello.c     |  6 ++++++
 test/end-to-end/hello.cstar |  5 +++++
 test/end-to-end/hello.ll    | 25 +++++++++++++++++++++++++
 3 files changed, 36 insertions(+)

commit 2dd0eb8138de480ed37e94003add34391067148d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 14:46:48 2021 -0500

    Renamed `tests` dir to `test` to be consistent with `src`.

 {tests => test}/lexer/fail-comment.cstar        | 0
 {tests => test}/lexer/fail-str1.cstar           | 0
 {tests => test}/lexer/test-comment.cstar        | 0
 {tests => test}/lexer/test-enum.cstar           | 0
 {tests => test}/lexer/test-escape-seqs.cstar    | 0
 {tests => test}/lexer/test-function1.cstar      | 0
 {tests => test}/lexer/test-gcd.cstar            | 0
 {tests => test}/lexer/test-str1.cstar           | 0
 {tests => test}/lexer/test-var-assignment.cstar | 0
 {tests => test}/parser/generic-fail.cstar       | 0
 {tests => test}/parser/test-gcd.cstar           | 0
 11 files changed, 0 insertions(+), 0 deletions(-)

commit 241a3f1dcaced811e540f8d5bbf7d94c1363dd38
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 14:46:07 2021 -0500

    Simplified `install-tooling.sh` and separated `build` and `dev` modes.

    Also removed mold.
    Made sure everything currently builds with `esy`.

 README.md          |  8 +++++++
 install-tooling.sh | 62 ++++++++++++++++++++--------------------------------------
 justfile           |  9 +++++---
 src/lexer.mll      |  4 ++--
 4 files changed, 39 insertions(+), 44 deletions(-)

commit 74e9112a27699a46da09137533424519efe1f037
```

```
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 14:26:56 2021 -0500

    Specified reference operator does not work on bit-field like types.

 LRM.md | 4 ++++
 1 file changed, 4 insertions(+)

commit 71abf8d56e07cb1158f7185fb94089cab7c8a4b0
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 12 13:25:18 2021 -0500

    Added Option/Result to the LRM and removed some parentheses in examples.

 LRM.md      | 22 ++++++++++++++++++-----
 proposal.md |  4 ++--
 2 files changed, 19 insertions(+), 7 deletions(-)

commit d4542cdce5a97c4634a39dd12ef79ea918438974
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 22:03:10 2021 -0500

    Typo.

 LRM.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 0cd7773934443007f582c9958c23d62d1c690f26
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 22:01:44 2021 -0500

    Added panicking to the LRM.

 LRM.md | 40 ++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 39 insertions(+), 1 deletion(-)

commit 32dc8be68f7d0dc4578c53872fdeb3b2f6a79069
Merge: db4092d bf1daba
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 22:00:50 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit bf1daba6320565e1f28863fdae70e640613a561e
Merge: 852f291 6e71c0c
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Thu Nov 11 20:43:44 2021 -0500

    Merge branch 'main' of github.com:kkysen/cstar into main

commit 852f29187d10d357e0655443f021d84a32254a85
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Thu Nov 11 20:43:39 2021 -0500

    tokenize number

 src/lexer.mll | 1 -
 1 file changed, 1 deletion(-)

commit db4092d67eef46a3440fe39053dcd10ef75566f6
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 19:07:36 2021 -0500

    Added lang types section (Option, Result).

 LRM.md | 21 +++++++++++++++++++++
 1 file changed, 21 insertions(+)

commit fcfd086bf1a614ca4b7f0d090a828f3d554c265f
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 19:07:14 2021 -0500

    Added reference and control flow (i.e. try) operators.
```

```
 LRM.md | 81 +++++++++++++++++++++++++++++++++++-------------------------------------
 1 file changed, 43 insertions(+), 38 deletions(-)

commit 6e71c0c2c8eaa7575b5dea496077e095f8e5939f
Merge: 57ea970 1dfe3e4
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 16:24:09 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 57ea970b00ad116f722e71e76e1c2bc461c1d1d3
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 16:23:43 2021 -0500

    Clarified the only `Copy` types are primitive types for now.

 LRM.md | 1 +
 1 file changed, 1 insertion(+)

commit 1dfe3e431b042018943c7b6b5746e6d10a23b53d
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Thu Nov 11 16:20:51 2021 -0500

    tokenize numbers

 src/lexer.mll | 11 +++++++++++
 1 file changed, 11 insertions(+)

commit 1bac96166625e294088682e16111493882197eeb
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 15:35:34 2021 -0500

    Always install ccache for tooling, not just when bootstrapping.

 install-tooling.sh | 1 +
 1 file changed, 1 insertion(+)

commit 006b217dc86934ceb5a69557f46e38250c8279a6
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 15:12:25 2021 -0500

    Updated the growable/mutable string part we're not implementing.

 LRM.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 71143fc0fa64435488526967c3cbb840bec2df8b
Merge: 41c625a 5482c28
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 15:07:39 2021 -0500

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 41c625a1cc78368a419976e93b75053b10b24381
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 15:07:26 2021 -0500

    Updated the unimplemented features in the LRM to combine redundant things and remove ones that we will im

 LRM.md | 40 +++++++++++++++-------------------------
 1 file changed, 15 insertions(+), 25 deletions(-)

commit 85785d70183bd88d38d437d71777d50312e35920
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 15:06:31 2021 -0500

    Updated range literals in the LRM to clarify the end length as `n` instead of using infinities.

 LRM.md | 11 +++++++----
 1 file changed, 7 insertions(+), 4 deletions(-)

commit 2607f42ba65780e2972b689460526373ba37f791
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 11 15:06:06 2021 -0500
```

```
    Explained destructive moves in the LRM (the fundamentals at least).

 LRM.md | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

commit 5482c2844d023e9bdf133375a6420fd8a83d536c
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Thu Nov 11 14:45:48 2021 -0500

    small LRM changes

 LRM.md | 1 +
 1 file changed, 1 insertion(+)

commit 36f73876c8eeccf345e1bf8bfbc98cf3dac3a5dd
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Thu Nov 11 14:00:34 2021 -0500

    spell check on cut features

 LRM.md | 8 ++++----
 1 file changed, 4 insertions(+), 4 deletions(-)

commit e6337300e758a52151efc2639c77ce8117a181c2
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Thu Nov 11 13:58:33 2021 -0500

    added features being cut/scaled back

 LRM.md | 29 +++++++++++++++++++++++++---
 1 file changed, 26 insertions(+), 3 deletions(-)

commit 388b46ef873de4e9f881f861a8352442c9aa3c0c
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 03:56:09 2021 -0500

    Added a TODO grammar section where we put the `parser.mly` `ocamlyacc` grammar directly.

 LRM.md | 9 +++++++++
 1 file changed, 9 insertions(+)

commit fd5f02a040e1fb07a007e00f3c71996f66685b92
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 03:12:39 2021 -0500

    Added `...` trailing varargs parameter purely for C FFI.

 LRM.md | 11 ++++++++++-
 1 file changed, 10 insertions(+), 1 deletion(-)

commit bb8a66ce61d06035838a1226c430fe66bf896cbe
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 03:00:28 2021 -0500

    Forgot to explain the types of indexing operators.

 LRM.md | 3 +++
 1 file changed, 3 insertions(+)

commit e6149720710 3a1fbf706916aa7b3a29eb488f699
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 02:59:24 2021 -0500

    Added bitshift operators and explained the types all operators operate on.

 LRM.md | 30 ++++++++++++++++++++++++++++++
 1 file changed, 30 insertions(+)

commit 7e92f97b9cd3d8663a5eed509876f15cf2447fd9
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 02:58:51 2021 -0500

    Added arbitrary integer bit sizes (e.x. `u1`, `i100`, `u6`).
```

```
    Note that LLVM implements this natively.

 LRM.md | 9 +++++----
 1 file changed, 5 insertions(+), 4 deletions(-)

commit 7e638540dd4b941b16058f3a6798e5aa97b1dd58
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 02:35:14 2021 -0500

    Specified we're only handing ascii, not full UTF-8 source code for now.

 LRM.md | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

commit a5595ecc84fc79b5dff0eb3a8a198172fbccac1b
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 02:34:59 2021 -0500

    Added `if`, `else`, and `for` exprs to the LRM, which are syntax sugar.

 LRM.md | 51 +++++++++++++++++++++++++++++++++++++++++++++++++++--------
 1 file changed, 43 insertions(+), 8 deletions(-)

commit f0e5a8820dc7dc8c9e2e574c9bb305fc01397200
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 02:12:35 2021 -0500

    Removed names and roles from the LRM since it's an LRM.

 LRM.md | 7 -------
 1 file changed, 7 deletions(-)

commit 076d8df165c69162e26dd4addaabcb5ed120d2df
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Nov 8 01:35:29 2021 -0500

    Outlined the rest of the LRM.

    Also made a few other important changes:
    * Builtins begin with `$` instead of `@` now
      since `@` may clash with annotations for free function builtins.
    * `bool` is now an `enum` with special-cased operator overloading

 LRM.md      | 390 +++++++++++++++++++++++++++++++++---------------------------
 proposal.md |  34 +++---
 2 files changed, 232 insertions(+), 192 deletions(-)

commit 0ff646781190016ac1fb986f5935fffb5d342eed
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 7 22:01:49 2021 -0500

    Added generic parameters to `struct` and `enum` declarations in the LRM.

 LRM.md | 10 ++++++----
 1 file changed, 6 insertions(+), 4 deletions(-)

commit 4a53fc57c29f41ac2b8cac0250d8bbf3367bbfbb
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 7 22:00:01 2021 -0500

    Added user-defined compound types that just link to their declaration sections.

 LRM.md | 3 +++
 1 file changed, 3 insertions(+)

commit 0c1076343c90cd98122227bb68dcb335821d1703
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 7 21:56:46 2021 -0500

    Added array and tuple types to the LRM.

 LRM.md      | 31 +++++++++++++++++++++++++++++++
 proposal.md |  2 +-
 2 files changed, 32 insertions(+), 1 deletion(-)
```

```
commit 4b6220c6edc3f897fcd5ec13ee1dec8eaf9504fe
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 7 19:57:29 2021 -0500

    Typo.

 LRM.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 3996261a0884f1403dd15824c34712c41d95e6e3
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Nov 7 19:56:21 2021 -0500

    Added reference and slice types to the LRM (and changed their syntax to make more uniform with expression

 LRM.md      | 111 ++++++++++++++++++++++++++++++++++++++++++++----
 proposal.md | 141 +++++++++++++++++++++++++++++-----------------------------
 2 files changed, 172 insertions(+), 80 deletions(-)

commit 9d58ef919daabd89378da8084ffc05777442664a
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 6 20:44:07 2021 -0400

    Added an overview of a C* program and its top-level items.  Also started the type system section (just pl

 LRM.md | 558 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++------
 1 file changed, 511 insertions(+), 47 deletions(-)

commit 3126c5cb3c040a8847e164feaafbcec60cab3197
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Nov 6 20:43:19 2021 -0400

    Fixed a failing test that was failing for the wrong reason.

 tests/parser/generic-fail.cstar | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 2a7a66c18f54e7324cefb50a4e7a02dac5056a01
Merge: 3142733 66638ad
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 14:32:50 2021 -0400

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 31427331a4d9628d8bc53010d1fbb78868883cd7
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 14:32:43 2021 -0400

    Updated table of contents with links and back links after each section.

 LRM.md | 212 ++++++++++++++++++++++++++++++++++++++++++++++++++---------------
 1 file changed, 165 insertions(+), 47 deletions(-)

commit 66638ad6cd379333686624f2df1af732f4203deb
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 13:59:37 2021 -0400

    Explained (a bit, more later) the `_` identifier in the LRM.

 LRM.md | 4 ++++
 1 file changed, 4 insertions(+)

commit c9ab01cc9cf3b1ddc2ea5e36ed7863f7c8cc6514
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 13:57:01 2021 -0400

    Renamed scanner to lexer.

 tests/{scanner => lexer}/fail-comment.cstar        | 0
 tests/{scanner => lexer}/fail-str1.cstar           | 0
 tests/{scanner => lexer}/test-comment.cstar        | 0
 tests/{scanner => lexer}/test-enum.cstar           | 0
 tests/{scanner => lexer}/test-escape-seqs.cstar    | 0
```

```
 tests/{scanner => lexer}/test-function1.cstar      | 0
 tests/{scanner => lexer}/test-gcd.cstar            | 0
 tests/{scanner => lexer}/test-str1.cstar           | 0
 tests/{scanner => lexer}/test-var-assignment.cstar | 0
 9 files changed, 0 insertions(+), 0 deletions(-)

commit deb34e969def4b0a8ea19218911fa4f34b8a001a
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 13:56:45 2021 -0400

    Updated the syntax in some of the tests.

 tests/scanner/fail-comment.cstar     |  6 +++---
 tests/scanner/test-comment.cstar     |  4 ++--
 tests/scanner/test-enum.cstar        | 10 +++++-----
 tests/scanner/test-escape-seqs.cstar |  8 +-------
 tests/scanner/test-function1.cstar   |  7 ++-----
 tests/scanner/test-gcd.cstar         | 14 ++++++++------
 tests/scanner/test-str1.cstar        |  2 +-
 7 files changed, 22 insertions(+), 29 deletions(-)

commit 548c0c6757b5f986a735541ed6f690569b5d7dd7
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 04:41:25 2021 -0400

    Added enum and union literals to the LRM, which we forgot before.

 LRM.md | 27 +++++++++++++++++++++++++++
 1 file changed, 27 insertions(+)

commit 4666c66e5291738a3328d2095da79b2d499930e0
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 04:27:36 2021 -0400

    Updated range literals in the LRM.

 LRM.md | 21 ++++++++++++++-------
 1 file changed, 14 insertions(+), 7 deletions(-)

commit 3df0ec01de61abacbbff94a23502da23a63fa973
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 04:19:15 2021 -0400

    Added function and closure literals to the LRM.

    I also moved the closure context to after the `fn`
    instead of before, because this should make parsing
    and disambiguation much easier.

 LRM.md      | 76 ++++++++++++++++++++++++++----------------------------------
 proposal.md | 10 ++++----
 2 files changed, 37 insertions(+), 49 deletions(-)

commit 356e1b567da9879ae14bb3b97ef56424cb67099c
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 03:57:33 2021 -0400

    Updated function literals in the LRM.

 LRM.md | 27 ++++++++++++++++++++++++++-
 1 file changed, 26 insertions(+), 1 deletion(-)

commit 97ebf91104l0e3fba1e95136dd4df400239debfa
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 01:40:55 2021 -0400

    Added array literals to the LRM, which we forgot before.

 LRM.md | 12 ++++++++++++
 1 file changed, 12 insertions(+)

commit da6ba9f98e191b77c671550fbcd56ab911547f3a
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 01:40:00 2021 -0400
```

```
    Updated tuple literals in the LRM.

 LRM.md | 40 +++++++++++++++++++++++++++-------------
 1 file changed, 27 insertions(+), 13 deletions(-)

commit 828c0d772bb10516fccc8759aa1777e8b61daaee
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 01:23:47 2021 -0400

    Fixed a few typos.

 LRM.md | 12 ++++++------
 1 file changed, 6 insertions(+), 6 deletions(-)

commit 2750cdaffde18247d2fb327557e6fe6d02fd6f8f
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 01:20:48 2021 -0400

    Updated struct literals.

 LRM.md | 53 ++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 52 insertions(+), 1 deletion(-)

commit bb3a99be45cc9024bc081d978163c6438f15de8f
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Nov 5 01:11:47 2021 -0400

    Updated string literals in the LRM, specifying in detail what characters and escapes are allowed.

 LRM.md | 44 ++++++++++++++++++++++++++++++++++++--------
 1 file changed, 36 insertions(+), 8 deletions(-)

commit 312ea721ff6423bf3b6d6efea100f44802c05255
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 4 17:50:42 2021 -0400

    Updated character literals in the LRM.

 LRM.md | 40 ++++++++++++++++++++++++++++++++++------
 1 file changed, 34 insertions(+), 6 deletions(-)

commit 6ae6d030f047c58d04e1a8d63fbbc208e8ee02fd
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Nov 4 17:35:59 2021 -0400

    Updated unit, bool, and number (int/float) literals in the LRM, especially numbers, which needed a lot mo

 LRM.md | 151 +++++++++++++++++++++++++++++++++++++++++++++++++++++------------
 1 file changed, 125 insertions(+), 26 deletions(-)

commit 18b6ee8c751730b2cbe5006cc3c707d0a39f96d5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 18:41:07 2021 -0400

    Added missing ``s to string literals that I had in the proposal.

 LRM.md | 36 +++++++++++++++++++++++++++++-------
 1 file changed, 29 insertions(+), 7 deletions(-)

commit cdcd37def4a4f77cc3876fca1951ad70fc7f6fd9
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 18:24:16 2021 -0400

    Accidentally flipped the operator in-place values.

 LRM.md | 58 +++++++++++++++++++++++++++++-----------------------------
 1 file changed, 29 insertions(+), 29 deletions(-)

commit eca1ad499c6fddfc58301f95a7936c69ef3bb071
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 18:22:38 2021 -0400

    Fixed some escape typos.
```

```
 LRM.md | 11 ++++++-----
 1 file changed, 6 insertions(+), 5 deletions(-)

commit ce1c209cfbe2395ed3f170f3a7e5577d17a5d498
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 18:15:28 2021 -0400

    Updated comments, identifiers, keywords, and operators in the LRM.

 LRM.md | 164 ++++++++++++++++++++++++++++++++++++++++++++++-------------------
 1 file changed, 117 insertions(+), 47 deletions(-)

commit e29eb2fe03765e3ebcd23d1143343d50c623d5a4
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 18:14:06 2021 -0400

    Removed `loop` and replaced it with `while (true)`.

 proposal.md | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

commit e1c9ec6956a18f2f184c25724e8deb06c4d960f2
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 16:50:14 2021 -0400

    Updated the proposal with syntax changes (mostly `=` for `fn`s and `@label`s.

 proposal.md | 97 +++++++++++++++++++++++++++++++++---------------------------
 1 file changed, 53 insertions(+), 44 deletions(-)

commit edf057b29f9dc95f70f3862a212a2716eddac968
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 16:39:47 2021 -0400

    Removed the proposal from the readme (since it's in the proposal instead) and added links to the proposal

 README.md | 773 +-----------------------------------------------------------
 1 file changed, 2 insertions(+), 771 deletions(-)

commit fbb7fd75512cc09bda7a59ab35fcbb71954617c8
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Nov 3 16:27:19 2021 -0400

    Renamed proposal to lowercase.

 Proposal.md => proposal.md | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit bbfb20d2626f366b3098c369c60d4e4d1cb9ecfa
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 23:18:49 2021 -0400

    updated toc

 LRM.md | 7 +++++--
 1 file changed, 5 insertions(+), 2 deletions(-)

commit 6b9f2e6e47a718b9d4ebc708d2f959c2724b392a
Merge: 94ba834 75fd56b
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Oct 31 23:09:56 2021 -0400

    Merge branch 'main' of github.com:kkysen/cstar into main

commit 94ba834c8c6987eecc31806c1c962d8ac5fabcbf
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Oct 31 23:08:23 2021 -0400

    control flow for LRM

 LRM.md | 58 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 57 insertions(+), 1 deletion(-)
```

```
commit 75fd56b5b6bb4856742c59be3bc806f42b18a405
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 23:04:52 2021 -0400

    added link to LRM

 LRM.md | 2 ++
 1 file changed, 2 insertions(+)

commit 9fd392f86b5e7d8cd47a67d1f723ddaf533eed86
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 22:58:25 2021 -0400

    added operators to LRM

 LRM.md    | 316 ++++++++++++++++++++++++++++++++++++++++++++++++++++++--------
 README.md | 243 +----------------------------------------------------
 2 files changed, 277 insertions(+), 282 deletions(-)

commit b77a08fc5854ea8da609309d610604919e3b9d3a
Merge: ed99db8 e9cff80
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 21:47:40 2021 -0400

    Merge branch 'main' of github.com:kkysen/cstar into main

commit ed99db86d7dd02ffe55d8bdc535d0c19449d6a7f
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 21:47:32 2021 -0400

    added literals to LRM

 LRM.md | 74 +++++++++++++++++++++++++++++-------------------------------------
 1 file changed, 33 insertions(+), 41 deletions(-)

commit e9cff80988685f7b3fb8ac0a0ac71014d2fd84b7
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Oct 31 20:21:30 2021 -0400

    recurisve hook for string lexer, testing infrastructure setup

 .gitignore                              |   4 +
 runtests.sh                             | 182 ++++++++++++++++++++++++++++++++
 src/lexer.mll                           |   1 +
 tests/parser/generic-fail.cstar         |   5 +
 tests/parser/test-gcd.cstar             |   1 +
 tests/scanner/fail-comment.cstar        |   6 ++
 tests/scanner/fail-str1.cstar           |   2 +
 tests/scanner/test-comment.cstar        |   8 ++
 tests/scanner/test-enum.cstar           |   6 ++
 tests/scanner/test-escape-seqs.cstar    |   7 ++
 tests/scanner/test-function1.cstar      |   7 ++
 tests/scanner/test-gcd.cstar            |   6 ++
 tests/scanner/test-str1.cstar           |   1 +
 tests/scanner/test-var-assignment.cstar |   2 +
 14 files changed, 238 insertions(+)

commit e0774773fb2faafa02a4053ca15bb67b0ab9c430
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 20:18:18 2021 -0400

    update LRM

 LRM.md | 49 ++++++++++++++++++++++++++++++++++++++++++-------
 1 file changed, 42 insertions(+), 7 deletions(-)

commit a37247d9fd5d42c4e10e1af48b678580d3c48fa9
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Sun Oct 31 19:39:12 2021 -0400

    LRM and Proposal

 LRM.md      | 634 ++++++++++++++++++++++++++++++++++++++++++++++++
 Proposal.md | 780 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
 2 files changed, 1414 insertions(+)

commit 7ca9d6ec12d537f0161fb441aec9de0081dd9846
Merge: 09f8d87 58e0548
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Oct 31 14:39:41 2021 -0400

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 09f8d87efac6b372926ed769b70749d7ddc1731d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Oct 31 04:41:42 2021 -0400

    Added an example ast.

 src/cstar.ml | 104 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++--
 1 file changed, 102 insertions(+), 2 deletions(-)

commit 5f146e86216029aed4d6f86201d94d54aa30a8a5
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sun Oct 31 04:41:26 2021 -0400

    Added the `Self` type and the function return type, which I forgot.
    Also removed the extern field, since it'll be an annotation.

 src/ast.ml  | 13 +++++++++----
 src/ast.mli | 13 +++++++++----
 2 files changed, 18 insertions(+), 8 deletions(-)

commit 58e054848902c7a0f786b3bc6767e62d9af90dee
Merge: ebc18a5 a2c5d27
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Fri Oct 29 19:28:00 2021 -0400

    Merge branch 'main' of github.com:kkysen/cstar into main

commit ebc18a55a4ed72da6f2a9132c2186bfbe43b329a
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Fri Oct 29 19:27:57 2021 -0400

    rule for lexing strings

 src/lexer.mll | 23 +++++++++++++++++++++-
 src/token.ml  |  1 +
 src/token.mli |  1 +
 3 files changed, 24 insertions(+), 1 deletion(-)

commit a2c5d27702ee256187816b3eff086f74ac8c7e88
Merge: 002abda 28f904c
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Oct 29 15:24:59 2021 -0400

    Merge branch 'main' of https://github.com/kkysen/cstar into main

commit 002abda810966aee21bc49cc0eed1b30f64319de8
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Oct 29 15:24:53 2021 -0400

    Added `[@@deriving show]` to the tokens and ast.
    Also fixed any compiler errors/warnings for the ast.

    I do have to duplicate types from `.mli` to `.mli` files
    b/c ppx import is incompatible with ppx deriving, though.

    Added some printing examples in `cstar.ml`.

 cstar.opam        |   0
 justfile          |   1 +
 package.json      |   9 +-
 src/ast.ml        | 355 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/ast.mli       | 196 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/cstar.ml      |  53 +++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/dune          |   4 +-
 src/stringMap.ml  |  18 +++++++++++++++++
```

```
 src/stringMap.mli |   7 ++
 src/token.ml      |   6 +-
 src/token.mli     |   7 +-
 11 files changed, 551 insertions(+), 105 deletions(-)

commit 28f904cc02dc82f5dd31278d068063aa69da8682
Author: JoanneWang-2 <69273871+JoanneWang-2@users.noreply.github.com>
Date:   Fri Oct 29 10:35:08 2021 -0400

    LRM in progress

 README.md | 241 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 241 insertions(+)

commit 52486600d7af1f4a98d840bf0fdd240ed3519d0d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Oct 29 02:08:53 2021 -0400

    Added `deriving show`s to the tokens.

 src/dune      |  4 +++
 src/token.ml  | 89 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/token.mli | 15 +++++++++--
 3 files changed, 106 insertions(+), 2 deletions(-)

commit ab0bcef5bb1e0f6bdf2479aa5cefe0b398b9c424
Author: Khyber Sen <kkysen@gmail.com>
Date:   Fri Oct 29 02:08:28 2021 -0400

    Added an `esy` `package.json` for declaring dependencies, since `dune` doesn't do that.  Now `esy` should

 .gitignore         |  4 ++++
 install-tooling.sh | 26 ++++++++++++++++++++++++--
 package.json       | 14 ++++++++++++++
 src/cstar.ml       | 22 ++++++++++++++++++++++
 src/lexer.mll      | 14 +-------------
 5 files changed, 65 insertions(+), 15 deletions(-)

commit aa227a0fc725cafd3cb2c4dc14f51e6c72da5084
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Oct 28 08:20:54 2021 -0400

    Added bool literals to the ast.

 src/ast.mli | 1 +
 1 file changed, 1 insertion(+)

commit ddf03eb3a12fca51cdfe96d917982443495003bb
Author: Khyber Sen <kkysen@gmail.com>
Date:   Wed Oct 27 19:35:56 2021 -0400

    Finished most of the ast (except patterns).

 src/ast.mli    | 159 +++++++++++++++++++++++++++++++++++++++++++++++++--------
 src/parser.mly |   6 +--
 src/token.mli  |   8 +--
 3 files changed, 143 insertions(+), 30 deletions(-)

commit 97e04ad25e4ca9052aeb3590c200dcdfd072d2f7
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Oct 25 23:29:26 2021 -0400

    Parser template.

 src/parser.mly | 17 +++++++++++++++++
 1 file changed, 17 insertions(+)

commit 8571793da01b946094ed2a1eefee2f12baceb9e1
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Oct 25 23:29:17 2021 -0400

    Worked on the ast together (vscode live share).

 src/ast.mli | 220 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
```

```
 1 file changed, 218 insertions(+), 2 deletions(-)

commit 20ef5d049e1995b637cb89b0ee9e194e53b27684
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Oct 25 20:54:54 2021 -0400

    Started the lexer.

 src/ast.ml     |  1 +
 src/ast.mli    |  4 +++
 src/dune       |  2 ++
 src/lexer.mll  | 78 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/parser.mly |  0
 src/token.mli  |  9 ++++++-
 6 files changed, 93 insertions(+), 1 deletion(-)

commit cb5f93d49d144ca857d259956edf7e547542b481
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Oct 25 04:14:44 2021 -0400

    Changed how number literals work to reduce ambiguities:
      A raw int can have an optional base prefix + digits.
      Then an int is just a raw int + type suffix.
      And floats are just a raw int + '.' + raw int + type suffix.

    This removes ambiguities between hexadecimal floats after the floating point,
    because an 'f' could also be a field/method name.
    Now an extra base prefix is required in this case,
    which starts with '0' so disambiguates it.
    Also, this has the added benefit of allowing you to switch bases
    between the integral and floating parts of a float.

 src/token.mli | 11 ++++++++---
 1 file changed, 8 insertions(+), 3 deletions(-)

commit f0dfa4d71e0c83b84b1aefa83948ebb66bb27e1e
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Oct 25 03:56:07 2021 -0400

    Added the lexer token definition.

 src/token.ml  |  0
 src/token.mli | 71 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 71 insertions(+)

commit dd49aa32539db02851c80788a766eab595a2f6ac
Author: Khyber Sen <kkysen@gmail.com>
Date:   Mon Oct 25 03:55:42 2021 -0400

    Added a `.ocamlformat`.

 .ocamlformat | 20 ++++++++++++++++++++
 1 file changed, 20 insertions(+)

commit db09ebf4526e5027dbf4da68b61af191967d2381
Author: Khyber Sen <kkysen@gmail.com>
Date:   Tue Oct 12 04:26:39 2021 -0400

    Added `just trace-exec`, which traces all the exec calls in a command, recursively.

 install-tooling.sh |   1 -
 justfile           | 143 +++++++++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 143 insertions(+), 1 deletion(-)

commit 1ba74e2225afc9c7fb4261f75248deae77fafb4b
Author: Khyber Sen <kkysen@gmail.com>
Date:   Tue Oct 12 02:45:22 2021 -0400

    Added `install-mold-ld`, which adds a symlink to `mold` as `ld` so that the directory can be passed to `g

 install-tooling.sh | 12 ++++++++++++
 1 file changed, 12 insertions(+)

commit da7b1f557f7a1bcfbc913bbf30e9b16ee49b59c8
```

```
Author: Khyber Sen <kkysen@gmail.com>
Date:   Tue Oct 12 00:00:44 2021 -0400

    Reworded some ambiguous "thing"s in the readme/proposal.

 README.md | 8 ++++----
 1 file changed, 4 insertions(+), 4 deletions(-)

commit 22d5094f544c4b6151f0736641f5573b77c3fbd9
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Oct 9 05:23:23 2021 -0400

    Sped up parts of `just install-tooling` and now it also installs the vscode extension.

 install-tooling.sh | 24 ++++++++++++++++++++----
 justfile           |  2 +-
 2 files changed, 21 insertions(+), 5 deletions(-)

commit f80497c5e2d8253b910c3a7215f4b25df3828ba4
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Oct 9 05:04:50 2021 -0400

    Set up the basic `dune` files for the build system.

 .ocamlformat | 0
 dune-project | 1 +
 src/cstar.ml | 1 +
 src/dune     | 2 ++
 4 files changed, 4 insertions(+)

commit 199d15d528d8990638e8c75c86e936d78f09b866
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Oct 9 05:04:32 2021 -0400

    Added a `justfile` and an `install-tooling.sh` file.

    The `justfile` contains a bunch of short convenient recipes.

    The `install-tooling.sh` file (invoked as `just install-tooling`),
    installs and tooling needed to build and develop the project.

 install-tooling.sh | 84 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
 justfile           | 28 ++++++++++++++++++
 2 files changed, 112 insertions(+)

commit b91a46c9f2a924292f2c87c4e0f66e27e3e34ea3
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Oct 9 05:01:16 2021 -0400

    Added a `.gitignore`.

 .gitignore | 3 +++
 1 file changed, 3 insertions(+)

commit c5b1bedcc0758ec6e57d192afee9f7f20c3e024d
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Oct 9 05:00:55 2021 -0400

    Updated LICENSE to all of us.

 LICENSE | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit f195d752664426655447d41e1fd63be83c9ed010
Author: Khyber Sen <kkysen@gmail.com>
Date:   Sat Oct 9 03:00:11 2021 -0400

    Split markdown paragraphs into editor-sized lines so that they have better line diffs in the future.

 README.md | 342 ++++++++++++++++++++++++++++++++++++++++++++++++++++------------
 1 file changed, 277 insertions(+), 65 deletions(-)

commit 634d992cb6a065c1404c8a1ae07be70960e251d5
Author: Khyber Sen <kkysen@gmail.com>
```

```
Date:   Sat Oct 9 02:35:38 2021 -0400

    Fixed some typos.

 README.md | 12 ++++++------
 1 file changed, 6 insertions(+), 6 deletions(-)

commit 24f1c16477025aaeb2517a88c3278514a0cdc199
Author: HackMD <no-reply@hackmd.io>
Date:   Sat Oct 9 03:35:10 2021 +0000

    Finished Proposal

    Finished the C* language proposal, including a longer systems programming example from the OS hw.

 README.md | 505 ++++++++++++++++++-----------------------------------------
 1 file changed, 129 insertions(+), 376 deletions(-)

commit 045d782e690bd31bd01bdef68c5321bf6f622923
Author: HackMD <no-reply@hackmd.io>
Date:   Fri Oct 8 16:52:31 2021 +0000

    First Draft

    Finished most of the proposal except for the larger examples at the end.
    The intro is mostly done, but could use some more work.
    The lang features I think are pretty much done, maybe just touch up a bit.
    Still need the larger example at the end.
    I also left in the language reference stuff I had (mistakenly) started writing,
    as well as the notes we had before outlining things.

 README.md | 815 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 815 insertions(+)

commit 9a9707c960c666d1c890cdd876b5c02d0f638d59
Author: HackMD <no-reply@hackmd.io>
Date:   Fri Oct 8 03:01:36 2021 +0000

    Initial rough draft

    Super rough draft, just laid out the main features and sections of the proposal.  Need to fill in.

 README.md | 2 --
 1 file changed, 2 deletions(-)

commit 44f8605e10a60ff43bbe09ee95429b80a3a74b80
Author: Khyber Sen <kkysen@gmail.com>
Date:   Thu Oct 7 20:51:02 2021 -0400

    Initial commit

 LICENSE   | 21 +++++++++++++++++++++
 README.md |  2 ++
 2 files changed, 23 insertions(+)
```

Code Listing

Code Listing - Table of Contents

- `src/driver.ml`
- `src/driver.mli`
- `src/emitType.ml`
- `src/emitType.mli`
- `src/lexer.mll`
- `src/lir.ml`
- `src/parser.mly`
- `src/stringMap.ml`
- `src/stringMap.mli`
- `src/token.ml`
- `src/token.mli`
- `src/util.ml`
- `src/util.mli`

`src/ast.ml`

```
type publicity =
  | Public
  | PublicIn of string
  | Private
[@@deriving show, yojson]

[@@@warning "-39"] (* from yojson *)
type mutability = {mut : bool} [@@deriving show, yojson]
[@@@warning "+39"]

type doc_comment = {lines : string list} [@@deriving show, yojson]

type label = {label_name : string} [@@deriving show, yojson]

type number_literal = Token.number_literal [@@deriving show, yojson]

type char_literal = Token.char_literal [@@deriving show, yojson]

type string_literal = Token.string_literal [@@deriving show, yojson]

type arithmetic_binary_op =
  | Add
  | Subtract
  | Multiply
  | Divide
  | Modulo
  | And
  | Or
  | BitAnd
  | BitOr
  | BitXor
  | LeftShift
  | RightShift
[@@deriving show, yojson]

type comparison_op =
  | Equal
  | NotEqual
  | LessThan
  | LessThanOrEqual
  | GreaterThan
  | GreaterThanOrEqual
[@@deriving show, yojson]

type binary_op =
  | ArithmeticBinaryOp of arithmetic_binary_op
  | AssigningArithmeticBinaryOp of arithmetic_binary_op
  | ComparisonOp of comparison_op
  | Assign
[@@deriving show, yojson]

type unary_op =
```

```
  | Negate (* - *)
  | Not (* ! *)
  | BitNot (* ~ *)
[@@deriving show, yojson]

type sign =
  | Plus
  | Minus
[@@deriving show, yojson]

type range_options = {
    inclusive : bool
  ; sign : sign option (* e.x. start..+length *)
}[@@deriving show, yojson]

type goto_kw =
  | Return
  | Break
  | Continue
[@@deriving show, yojson]

type path = {
    path_pars : string list
}
[@@deriving show, yojson]

type use = {
  use_path : path
}
[@@deriving show, yojson]

type named_type = {
  type_name : string
}
[@@deriving show, yojson]

and pointer_type = {
  pointee : type_
  ; pointer_mutability : mutability
}
[@@deriving show, yojson]

and reference_type = {
  referent : type_
  ; reference_mutability : mutability
}
[@@deriving show, yojson]

and slice_type = {
  slice_element_type : type_
}
[@@deriving show, yojson]

and array_type = {
  array_element_type : type_
  ; array_length : expr
}
[@@deriving show, yojson]

and tuple_type = {
  elements : type_ list
}
[@@deriving show, yojson]

and func_type = {
    func_args : tuple_type
  ; return_type : type_
}
[@@deriving show, yojson]

and generic_type = {
    name : string
  ; args : tuple_type
}
```

```
[@@deriving show, yojson]

and type_ =
  | InferredType
  | NamedType of named_type
  | PointerType of pointer_type
  | ReferenceType of reference_type
  | SliceType of slice_type
  | ArrayType of array_type
  | TupleType of tuple_type (* empty () is the unit type *)
  | FuncType of func_type
  | GenericType of generic_type
[@@deriving show, yojson]

and pattern =
| IdentifierPattern of string * mutability
| NumPattern of number_literal
| CharPattern of char_literal
| StringPattern of string_literal
| RestPattern
[@@deriving show, yojson]

and if_expr = {
    then_case : block_expr
  ; else_case : block_expr option
}
[@@deriving show, yojson]

and match_arm = {
    match_pattern : pattern
  ; match_condition : expr option
  ; match_arm_value : expr
}
[@@deriving show, yojson]

and match_expr = {
    match_arms : match_arm list
}
[@@deriving show, yojson]

and for_expr = {
    for_element : variable
  ; for_block : block_expr
  ; for_label : label option
}
[@@deriving show, yojson]

and statement =
  | Expr of expr
  | Item of item
[@@deriving show, yojson]

and block_expr = {
    statements : statement list
  ; trailing_semicolon : bool
}
[@@deriving show, yojson]

and func_call_expr = {
    func : expr
  ; call_generic_args : type_ list
  ; call_args : expr list
}
[@@deriving show, yojson]

and anon_func_signature = {
    signature_args : variables
  ; signature_return_type : type_
}
[@@deriving show, yojson]

and func_literal = {
    anon_signature : anon_func_signature
  ; func_literal_value : expr
```

```
}
[@@deriving show, yojson]

and closure_literal = {
    closure_context : struct_literal
  ; closure_func : func_literal
}
[@@deriving show, yojson]

and struct_literal_field =
| Explicit of string * expr
| Implicit of string
| Spread of expr
[@@deriving show, yojson]

and struct_literal = {
    struct_literal_name : string
  ; struct_literal_fields : struct_literal_field list
}
[@@deriving show, yojson]

and tuple_literal = {
  tuple_elements : expr list
} [@@deriving show, yojson]

and array_literal = {
  array_elements : expr list
} [@@deriving show, yojson]

and range_literal = {
    start : expr option
  ; stop : expr option
  ; options : range_options
}
[@@deriving show, yojson]

and literal =
  | Number of number_literal
  | Char of char_literal
  | String of string_literal
  | Range of range_literal
  | Struct of struct_literal
  | Tuple of tuple_literal
  | Array of array_literal
  | Func of func_literal
  | Closure of closure_literal
[@@deriving show, yojson]

and unary_expr = {
    unary_op : unary_op
  ; unary_value : expr
}
[@@deriving show, yojson]

and binary_expr = {
    binary_op : binary_op
  ; left : expr
  ; right : expr
}
[@@deriving show, yojson]

and postfix_expr =
  | Dereference of mutability (* .* *)
  | Reference of mutability (*.&, .&mut *)
  | Try (* .? *)
  | PostFixBinaryOp of binary_op
  | FieldAccess of string
  | ElementAccess of number_literal
  | MethodCall of func_call_expr
  | GoTo of label option * goto_kw
  | Defer of label option
  | Match of match_expr
  | If of if_expr
  | While of block_expr
```

```
  | For of for_expr
[@@deriving show, yojson]

and expr =
  | Variable of string
  | Literal of literal
  | UnaryOp of unary_expr
  | BinaryOp of binary_expr
  | Index of expr
  | FuncCall of func_call_expr
  | PostFixExpr of expr * postfix_expr
  | UnDefer of label option
  | Block of block_expr
[@@deriving show, yojson]

and annotation = {
    annotation_path : path
  ; annotation_args : tuple_literal
}
[@@deriving show, yojson]

and metadata = {
    publicity : publicity
  ; annotations : annotation list
  ; doc_comment : doc_comment
}
[@@deriving show, yojson]

and variable = {
    variable_name : string
  ; variable_mutability : mutability
  ; variable_type : type_
}
[@@deriving show, yojson]

and variable_with_metadata = {
    variable : variable
  ; metadata : metadata
}
[@@deriving show, yojson]

and variables = variable_with_metadata list

and let_ = {
    let_variable : variable
  ; let_value : expr
}
[@@deriving show, yojson]

and func_decl_signature = {
    func_name : string
  ; func_signature : anon_func_signature
}
[@@deriving show, yojson]

and func_decl = {
    func_decl_signature : func_decl_signature
  ; func_value : expr option
}
[@@deriving show, yojson]

and fields = variables

and struct_decl = {
    struct_name : string
  ; struct_fields : fields
}
[@@deriving show, yojson]

and variant_data =
  | TupleVariant of tuple_type
  | StructVariant of fields
[@@deriving show, yojson]
```

```
and variant = {
    variant_name : string
  ; variant_data : variant_data option
}
[@@deriving show, yojson]

and enum_decl = {
    enum_name : string
  ; enum_variants : variant list
}
[@@deriving show, yojson]

and union_decl = {
    union_name : string
  ; union_fields : variables
}
[@@deriving show, yojson]

and inner_item =
  | Use of use
  | Let of let_
  | FuncDecl of func_decl
  | StructDecl of struct_decl
  | EnumDecl of enum_decl
  | UnionDecl of union_decl
  | Impl of module_
  | Mod of module_
[@@deriving show, yojson]

and item = {
    item_metadata : metadata
  ; inner_item : inner_item
}
[@@deriving show, yojson]

and module_body = {
  module_items : item list
}
[@@deriving show, yojson]

and module_ = {
    module_name : string
  ; module_body : module_body
}
[@@deriving show, yojson]

type ast = {
    path : string
  ; module_ : module_
}
[@@deriving show, yojson]
```

Code Listing - Table of Contents

src/ast.mli

```
type publicity =
  | Public
  | PublicIn of string
  | Private
[@@deriving show, yojson]

[@@@warning "-39"] (* from yojson *)
type mutability = {mut : bool} [@@deriving show, yojson]
[@@@warning "+39"]

type doc_comment = {lines : string list} [@@deriving show, yojson]

type label = {label_name : string} [@@deriving show, yojson]
```

```
type number_literal = Token.number_literal [@@deriving show, yojson]

type char_literal = Token.char_literal [@@deriving show, yojson]

type string_literal = Token.string_literal [@@deriving show, yojson]

type arithmetic_binary_op =
  | Add
  | Subtract
  | Multiply
  | Divide
  | Modulo
  | And
  | Or
  | BitAnd
  | BitOr
  | BitXor
  | LeftShift
  | RightShift
[@@deriving show, yojson]

type comparison_op =
  | Equal
  | NotEqual
  | LessThan
  | LessThanOrEqual
  | GreaterThan
  | GreaterThanOrEqual
[@@deriving show, yojson]

type binary_op =
  | ArithmeticBinaryOp of arithmetic_binary_op
  | AssigningArithmeticBinaryOp of arithmetic_binary_op
  | ComparisonOp of comparison_op
  | Assign
[@@deriving show, yojson]

type unary_op =
  | Negate (* - *)
  | Not (* ! *)
  | BitNot (* ~ *)
[@@deriving show, yojson]

type sign =
  | Plus
  | Minus
[@@deriving show, yojson]

type range_options = {
    inclusive : bool
  ; sign : sign option (* e.x. start..+length *)
}[@@deriving show, yojson]

type goto_kw =
  | Return
  | Break
  | Continue
[@@deriving show, yojson]

type path = {
    path_pars : string list
}
[@@deriving show, yojson]

type use = {
  use_path : path
}
[@@deriving show, yojson]

type named_type = {
  type_name : string
}
[@@deriving show, yojson]
```

```
and pointer_type = {
  pointee : type_
  ; pointer_mutability : mutability
}
[@@deriving show, yojson]

and reference_type = {
  referent : type_
  ; reference_mutability : mutability
}
[@@deriving show, yojson]

and slice_type = {
  slice_element_type : type_
}
[@@deriving show, yojson]

and array_type = {
  array_element_type : type_
  ; array_length : expr
}
[@@deriving show, yojson]

and tuple_type = {
  elements : type_ list
}
[@@deriving show, yojson]

and func_type = {
    func_args : tuple_type
  ; return_type : type_
}
[@@deriving show, yojson]

and generic_type = {
    name : string
  ; args : tuple_type
}
[@@deriving show, yojson]

and type_ =
  | InferredType
  | NamedType of named_type
  | PointerType of pointer_type
  | ReferenceType of reference_type
  | SliceType of slice_type
  | ArrayType of array_type
  | TupleType of tuple_type (* empty () is the unit type *)
  | FuncType of func_type
  | GenericType of generic_type
[@@deriving show, yojson]

and pattern =
| IdentifierPattern of string * mutability
| NumPattern of number_literal
| CharPattern of char_literal
| StringPattern of string_literal
| RestPattern
[@@deriving show, yojson]

and if_expr = {
    then_case : block_expr
  ; else_case : block_expr option
}
[@@deriving show, yojson]

and match_arm = {
    match_pattern : pattern
  ; match_condition : expr option
  ; match_arm_value : expr
}
[@@deriving show, yojson]

and match_expr = {
```

```
    match_arms : match_arm list
}
[@@deriving show, yojson]

and for_expr = {
    for_element : variable
  ; for_block : block_expr
  ; for_label : label option
}
[@@deriving show, yojson]

and statement =
  | Expr of expr
  | Item of item
[@@deriving show, yojson]

and block_expr = {
    statements : statement list
  ; trailing_semicolon : bool
}
[@@deriving show, yojson]

and func_call_expr = {
    func : expr
  ; call_generic_args : type_ list
  ; call_args : expr list
}
[@@deriving show, yojson]

and anon_func_signature = {
    signature_args : variables
  ; signature_return_type : type_
}
[@@deriving show, yojson]

and func_literal = {
    anon_signature : anon_func_signature
  ; func_literal_value : expr
}
[@@deriving show, yojson]

and closure_literal = {
    closure_context : struct_literal
  ; closure_func : func_literal
}
[@@deriving show, yojson]

and struct_literal_field =
| Explicit of string * expr
| Implicit of string
| Spread of expr
[@@deriving show, yojson]

and struct_literal = {
    struct_literal_name : string
  ; struct_literal_fields : struct_literal_field list
}
[@@deriving show, yojson]

and tuple_literal = {
  tuple_elements : expr list
} [@@deriving show, yojson]

and array_literal = {
  array_elements : expr list
} [@@deriving show, yojson]

and range_literal = {
    start : expr option
  ; stop : expr option
  ; options : range_options
}
[@@deriving show, yojson]
```

```
and literal =
  | Number of number_literal
  | Char of char_literal
  | String of string_literal
  | Range of range_literal
  | Struct of struct_literal
  | Tuple of tuple_literal
  | Array of array_literal
  | Func of func_literal
  | Closure of closure_literal
[@@deriving show, yojson]

and unary_expr = {
    unary_op : unary_op
  ; unary_value : expr
}
[@@deriving show, yojson]

and binary_expr = {
    binary_op : binary_op
  ; left : expr
  ; right : expr
}
[@@deriving show, yojson]

and postfix_expr =
  | Dereference of mutability (* .* *)
  | Reference of mutability (*.&, .&mut *)
  | Try (* .? *)
  | PostFixBinaryOp of binary_op
  | FieldAccess of string
  | ElementAccess of number_literal
  | MethodCall of func_call_expr
  | GoTo of label option * goto_kw
  | Defer of label option
  | Match of match_expr
  | If of if_expr
  | While of block_expr
  | For of for_expr
[@@deriving show, yojson]

and expr =
  | Variable of string
  | Literal of literal
  | UnaryOp of unary_expr
  | BinaryOp of binary_expr
  | Index of expr
  | FuncCall of func_call_expr
  | PostFixExpr of expr * postfix_expr
  | UnDefer of label option
  | Block of block_expr
[@@deriving show, yojson]

and annotation = {
    annotation_path : path
  ; annotation_args : tuple_literal
}
[@@deriving show, yojson]

and metadata = {
    publicity : publicity
  ; annotations : annotation list
  ; doc_comment : doc_comment
}
[@@deriving show, yojson]

and variable = {
    variable_name : string
  ; variable_mutability : mutability
  ; variable_type : type_
}
[@@deriving show, yojson]

and variable_with_metadata = {
```

```
    variable : variable
  ; metadata : metadata
}
[@@deriving show, yojson]

and variables = variable_with_metadata list

and let_ = {
    let_variable : variable
  ; let_value : expr
}
[@@deriving show, yojson]

and func_decl_signature = {
    func_name : string
  ; func_signature : anon_func_signature
}
[@@deriving show, yojson]

and func_decl = {
    func_decl_signature : func_decl_signature
  ; func_value : expr option
}
[@@deriving show, yojson]

and fields = variables

and struct_decl = {
    struct_name : string
  ; struct_fields : fields
}
[@@deriving show, yojson]

and variant_data =
  | TupleVariant of tuple_type
  | StructVariant of fields
[@@deriving show, yojson]

and variant = {
    variant_name : string
  ; variant_data : variant_data option
}
[@@deriving show, yojson]

and enum_decl = {
    enum_name : string
  ; enum_variants : variant list
}
[@@deriving show, yojson]

and union_decl = {
    union_name : string
  ; union_fields : variables
}
[@@deriving show, yojson]

and inner_item =
  | Use of use
  | Let of let_
  | FuncDecl of func_decl
  | StructDecl of struct_decl
  | EnumDecl of enum_decl
  | UnionDecl of union_decl
  | Impl of module_
  | Mod of module_
[@@deriving show, yojson]

and item = {
    item_metadata : metadata
  ; inner_item : inner_item
}
[@@deriving show, yojson]

and module_body = {
```

```
  module_items : item list
}
[@@deriving show, yojson]

and module_ = {
    module_name : string
  ; module_body : module_body
}
[@@deriving show, yojson]

type ast = {
    path : string
  ; module_ : module_
}
[@@deriving show, yojson]
```

Code Listing - Table of Contents

src/codegen.ml

```
open Core
module LL = Llvm
module LLAnalysis = Llvm_analysis
open Lir
module Scope = Map.Make (String)

type scope = LL.llvalue Scope.t

type num_kind =
  | UnSigned
  | Signed
  | Float

let type_num_kind (t : type_) : num_kind option =
  match t with
  | IntType t ->
      Some
        (match t.unsigned with
        | true -> UnSigned
        | false -> Signed)
  | FloatType _ -> Some Float
  | PointerType _ -> Some UnSigned
  | _ -> None
;;

let float_type_bits (t : float_type) : int =
  match t with
  | F32 -> 32
  | F64 -> 64
;;

let compile ~(lir : lir) ~(ctx : LL.llcontext) ~(mod_ : LL.llmodule) : unit =
  let rec compile_type (t : type_) : LL.lltype =
    match t with
    | UnitType -> LL.void_type ctx
    | IntType t -> LL.integer_type ctx t.bits
    | FloatType t -> (
        match t with
        | F32 -> LL.float_type ctx
        | F64 -> LL.double_type ctx)
    | PointerType t -> t |> compile_type |> LL.pointer_type
    | ArrayType (t, len) -> (t |> compile_type |> LL.array_type) len
    | TupleType ts -> ts |> Array.map ~f:compile_type |> LL.struct_type ctx
    | FuncType {func_args; func_return_type} ->
        let args = func_args |> Array.map ~f:compile_type in
        let return_type = func_return_type |> compile_type in
        LL.function_type return_type args
  in

  let declare_global (g : global) : LL.llvalue =
```

```
  let {global_name = name; global_type = t; global_value = _} = g in
  let t = compile_type t in
  let g = LL.declare_global t name mod_ in
  g
in

let declare_func (f : func) : LL.llvalue =
  let {func_name = name; func_type = t; func_decl = decl} = f in
  let t = compile_type (FuncType t) in
  let f =
    (match decl with
    | Some _ -> LL.define_function
    | None -> LL.declare_function)
      name
      t
      mod_
  in
  f
in

let create_scope (values : LL.llvalue list) : scope =
  values
  |> Sequence.of_list
  |> Sequence.map ~f:(fun v -> (LL.value_name v, v))
  |> Scope.of_sequence_exn
in

let {globals; functions; _} = lir in

let global_scope =
  create_scope
    (let globals = globals |> List.map ~f:declare_global in
     let functions = functions |> List.map ~f:declare_func in
     globals @ functions)
in

let compile_const (lit : literal) (t : type_) : LL.llvalue =
  let t = compile_type t in
  match lit with
  | Int i -> LL.const_int t i
  | Float f -> LL.const_float t f
in

let rec compile_expr (expr : expr) ~(scope : scope) ~(irb : LL.llbuilder)
    : LL.llvalue
  =
  let {type_ = t; value} = expr in
  let value =
    match value with
    | Literal lit -> compile_const lit t
    | Var name -> Scope.find_exn scope name
    | UnaryOp (op, expr) ->
        let value = compile_expr expr ~scope ~irb in
        let t = LL.type_of value in
        let op =
          match op with
          | Negate -> LL.build_neg
          | Not -> LL.build_icmp LL.Icmp.Ne (LL.const_int t 0)
          | BitNot -> LL.build_not
          | AddressOf ->
              fun v n irb ->
                let var = LL.build_alloca t n irb in
                let store = LL.build_store v var irb in
                ignore store;
                var
          | Dereference -> LL.build_load
        in
        let value = op value "" irb in
        value
    | BinaryOp (lhs, op, rhs) ->
        let kind = type_num_kind expr.type_ in
        let lhs = compile_expr lhs ~scope ~irb in
        let rhs = compile_expr rhs ~scope ~irb in
        let op =
```

```
            match op with
            | Assign -> fun v p _n irb -> LL.build_store v p irb
            | Arithmetic op -> (
                let kind = Option.value_exn kind in
                match op with
                | Add -> LL.build_add
                | Subtract -> LL.build_sub
                | Multiply -> LL.build_mul
                | Divide -> (
                    match kind with
                    | UnSigned -> LL.build_udiv
                    | Signed -> LL.build_sdiv
                    | Float -> LL.build_fdiv)
                | Modulo -> (
                    match kind with
                    | UnSigned -> LL.build_urem
                    | Signed -> LL.build_srem
                    | Float -> LL.build_frem)
                | And | BitAnd -> LL.build_and
                | Or | BitOr -> LL.build_or
                | BitXor -> LL.build_xor
                | LeftShift -> LL.build_shl
                | RightShift -> (
                    match kind with
                    | UnSigned -> LL.build_lshr
                    | Signed -> LL.build_ashr
                    | Float -> failwith "float shift impossible"))
            | Comparison op -> (
                let kind = Option.value_exn kind in
                match kind with
                | UnSigned ->
                    let pred =
                      match op with
                      | Equal -> LL.Icmp.Eq
                      | NotEqual -> LL.Icmp.Ne
                      | LessThan -> LL.Icmp.Ult
                      | LessThanOrEqual -> LL.Icmp.Ule
                      | GreaterThan -> LL.Icmp.Ugt
                      | GreaterThanOrEqual -> LL.Icmp.Uge
                    in
                    LL.build_icmp pred
                | Signed ->
                    let pred =
                      match op with
                      | Equal -> LL.Icmp.Eq
                      | NotEqual -> LL.Icmp.Ne
                      | LessThan -> LL.Icmp.Slt
                      | LessThanOrEqual -> LL.Icmp.Sle
                      | GreaterThan -> LL.Icmp.Sgt
                      | GreaterThanOrEqual -> LL.Icmp.Sge
                    in
                    LL.build_icmp pred
                | Float ->
                    let pred =
                      match op with
                      | Equal -> LL.Fcmp.Oeq
                      | NotEqual -> LL.Fcmp.One
                      | LessThan -> LL.Fcmp.Olt
                      | LessThanOrEqual -> LL.Fcmp.Ole
                      | GreaterThan -> LL.Fcmp.Ogt
                      | GreaterThanOrEqual -> LL.Fcmp.Oge
                    in
                    LL.build_fcmp pred)
        in
        let value = op lhs rhs "" irb in
        value
    | Cast expr ->
        let value = compile_expr expr ~scope ~irb in
        let u = expr.type_ in
        let nop v _t _n _b = v in
        let op =
          match (u, t) with
          | (IntType u, IntType t) -> (
              match (u.unsigned, t.unsigned) with
```

```
                | (true, true) ->
                    if u.bits < t.bits
                    then LL.build_zext
                    else if u.bits > t.bits
                    then LL.build_trunc
                    else nop
                | (false, false) ->
                    if u.bits < t.bits
                    then LL.build_sext
                    else if u.bits > t.bits
                    then LL.build_trunc
                    else nop
                | (_, _) -> LL.build_bitcast)
            | (FloatType u, FloatType t) ->
                let ubits = float_type_bits u in
                let tbits = float_type_bits t in
                if ubits < tbits
                then LL.build_fpext
                else if ubits > tbits
                then LL.build_fptrunc
                else nop
            | (IntType u, FloatType _t) -> (
                match u.unsigned with
                | true -> LL.build_uitofp
                | false -> LL.build_sitofp)
            | (FloatType _u, IntType t) -> (
                match t.unsigned with
                | true -> LL.build_fptoui
                | false -> LL.build_fptosi)
            | (IntType _u, PointerType _v) -> LL.build_inttoptr
            | (PointerType _u, IntType _v) -> LL.build_ptrtoint
            | (_, _) -> LL.build_bitcast
          in
          let value = op value (compile_type t) "" irb in
          value
      | Call {callee; call_args = args} ->
          let callee = compile_expr callee ~scope ~irb in
          let args = args |> Array.map ~f:(compile_expr ~scope ~irb) in
          let value = LL.build_call callee args "" irb in
          value
      | If {condition; then_case; else_case} -> (
        let _condition = compile_expr condition ~scope ~irb in
        ignore then_case;
        ignore else_case;
        failwith "TODO: if"
      )
      | GoTo _expr -> (
        failwith "TODO goto"
      )
      | Block exprs ->
          exprs
          |> List.fold ~init:None ~f:(fun _ expr ->
                 Some (compile_expr expr ~scope ~irb))
          |> Option.value_exn
    in
    value
  in

  let compile_global (g : global) : unit =
    let {global_name = name; global_type = t; global_value = value} = g in
    let g = Scope.find_exn global_scope name in
    match value with
    | Some value ->
        let value = compile_const value t in
        LL.set_initializer value g
    | None ->
        LL.set_externally_initialized true g;
        ()
  in

  let compile_func_decl (decl : func_decl) (f : LL.llvalue) : unit =
    let {arg_names; func_value} = decl in
    let entry = LL.entry_block f in
    let irb = LL.builder_at_end ctx entry in
```

```
    let scope =
      Array.zip_exn arg_names (LL.params f)
      |> Array.fold ~init:global_scope ~f:(fun scope (name, param) ->
             LL.set_value_name (name ^ ".param") param;
             let t = LL.type_of param in
             let local = LL.build_alloca t name irb in
             let store = LL.build_store param local irb in
             ignore store;
             Scope.add_exn scope ~key:name ~data:local)
    in
    let ret_val = compile_expr func_value ~scope ~irb in
    ignore ret_val;
    (* failwith "TODO" *)
  in

  let compile_func (f : func) : unit =
    let {func_name = name; func_type = _; func_decl = decl} = f in
    let f = Scope.find_exn global_scope name in
    (match decl with
    | Some decl -> compile_func_decl decl f
    | None -> ());
    LLAnalysis.assert_valid_function f;
    ()
  in

  (* globals |> List.iter ~f:compile_global; *)
  (* functions |> List.iter ~f:compile_func; *)
  ignore compile_global;
  ignore compile_func;

  let i8 = LL.i8_type ctx in
  let i32 = LL.i32_type ctx in
  let i64 = LL.i64_type ctx in
  let puts =
    let type_ = LL.function_type i32 [|LL.pointer_type i8|] in
    LL.declare_function "puts" type_ mod_
  in
  let main =
    let type_ = LL.function_type i32 [||] in
    let func = LL.define_function "main" type_ mod_ in
    let entry = LL.entry_block func in
    let irb = LL.builder_at_end ctx entry in
    let hello_world_const = LL.const_stringz ctx "Hello, World!" in
    let hello_world_global = LL.define_global "" hello_world_const mod_ in
    let zero_i64 = LL.const_int i64 0 in
    let hello_world_local =
      LL.build_in_bounds_gep hello_world_global [|zero_i64; zero_i64|] "" irb
    in
    let (_ : LL.llvalue) = LL.build_call puts [|hello_world_local|] "" irb in
    let zero_i32 = LL.const_int i32 0 in
    let (_ : LL.llvalue) = LL.build_ret zero_i32 irb in
    func
  in
  LLAnalysis.assert_valid_function main;
  ()
;;
```

Code Listing - Table of Contents

src/codegen.mli

```
module LL = Llvm
open Lir

val compile : lir:lir -> ctx:LL.llcontext -> mod_:LL.llmodule -> unit
```

Code Listing - Table of Contents

src/compiler.ml

```
open Core
module LL = Llvm
module LLAnalysis = Llvm_analysis
module LLTarget = Llvm_target

module type RawStage = sig
  type input

  type output

  val from_file : path:string -> unit -> input

  val to_file : path:string -> output -> unit

  val compile : input -> output
end

module type Stage = sig
  type input

  type output

  val compile : input -> output

  val compile_file : input_path:string -> output_path:string -> unit
end

module MakeStage : functor (RawStage : RawStage) -> Stage =
functor
  (RawStage : RawStage)
  ->
  struct
    type input = RawStage.input

    type output = RawStage.output

    let from_file = RawStage.from_file

    let to_file = RawStage.to_file

    let compile = RawStage.compile

    let compile_file ~(input_path : string) ~(output_path : string) : unit =
      () |> from_file ~path:input_path |> compile |> to_file ~path:output_path
    ;;
  end

let json_deserializer (deserializer : Yojson.Safe.t -> 'a)
    : path:string -> unit -> 'a
  =
 fun ~path () -> Yojson.Safe.from_file ~fname:path path |> deserializer
;;

let json_serializer (serializer : 'a -> Yojson.Safe.t)
    : path:string -> 'a -> unit
  =
 fun ~path a -> a |> serializer |> Yojson.Safe.to_file path
;;

type src = {
    path : string
  ; code : string
}
[@@deriving show, yojson]

type token_src = {
    src : src
  ; tokens : Token.token list
}
[@@deriving show, yojson]
```

```
type desugared_ast = {path : string} [@@deriving show, yojson]

type typed_ast = {path : string} [@@deriving show, yojson]

module Lex = MakeStage (struct
  type input = src

  type output = token_src

  let from_file ~(path : string) () = {path; code = In_channel.read_all path}

  let to_file = json_serializer yojson_of_token_src

  let compile (src : src) : token_src =
    let lexbuf = Lexing.from_string src.code in
    let tokens =
      Util.list_from_fn (fun () ->
          match Lexer.token lexbuf with
          | Token.EOF -> None
          | token -> Some token)
    in
    {src; tokens}
  ;;
end)

module Parse = MakeStage (struct
  type input = token_src

  type output = Ast.ast

  let from_file = json_deserializer token_src_of_yojson

  let to_file = json_serializer Ast.yojson_of_ast

  let translate_token (token : Token.token) : Parser.token =
    match token with
    | Token.EOF -> Parser.EOF
    | Token.WhiteSpace s -> Parser.WhiteSpace s
    | Token.Comment comment -> (
        match comment with
        | Token.Structural -> Parser.StructuralComment
        | Token.Line s -> Parser.LineComment s
        | Token.Block s -> Parser.BlockComment s)
    | Token.Identifier s -> Parser.Identifier s
    | Token.Literal literal -> (
        match literal with
        | Token.Number n -> Parser.NumberLiteral n
        | Token.Char c -> Parser.CharLiteral c
        | Token.String s -> Parser.StringLiteral s)
    | Token.Keyword kw -> (
        match kw with
        | Token.KwMod -> Parser.KwMod
        | Token.KwUse -> Parser.KwUse
        | Token.KwLet -> Parser.KwLet
        | Token.KwMut -> Parser.KwMut
        | Token.KwPub -> Parser.KwPub
        | Token.KwTry -> Parser.KwTry
        | Token.KwConst -> Parser.KwConst
        | Token.KwImpl -> Parser.KwImpl
        | Token.KwFn -> Parser.KwFn
        | Token.KwStruct -> Parser.KwStruct
        | Token.KwEnum -> Parser.KwEnum
        | Token.KwUnion -> Parser.KwUnion
        | Token.KwReturn -> Parser.KwReturn
        | Token.KwBreak -> Parser.KwBreak
        | Token.KwContinue -> Parser.KwContinue
        | Token.KwFor -> Parser.KwFor
        | Token.KwWhile -> Parser.KwWhile
        | Token.KwIf -> Parser.KwIf
        | Token.KwElse -> Parser.KwElse
        | Token.KwMatch -> Parser.KwMatch
        | Token.KwDefer -> Parser.KwDefer
        | Token.KwUndefer -> Parser.KwUndefer
```

```
        | Token.KwIn -> Parser.KwIn
        | Token.KwTrait -> Parser.KwTrait)
    | Token.SemiColon -> Parser.SemiColon
    | Token.Colon -> Parser.Colon
    | Token.Comma -> Parser.Comma
    | Token.Dot -> Parser.Dot
    | Token.DotDot -> Parser.DotDot
    | Token.OpenParen -> Parser.OpenParen
    | Token.CloseParen -> Parser.CloseParen
    | Token.OpenBrace -> Parser.OpenBrace
    | Token.CloseBrace -> Parser.CloseBrace
    | Token.OpenBracket -> Parser.OpenBracket
    | Token.CloseBracket -> Parser.CloseBracket
    | Token.At -> Parser.At
    | Token.QuestionMark -> Parser.QuestionMark
    | Token.ExclamationPoint -> Parser.ExclamationPoint
    | Token.Equal -> Parser.Equal
    | Token.EqualEqual -> Parser.EqualEqual
    | Token.NotEqual -> Parser.NotEqual
    | Token.LessThan -> Parser.LessThan
    | Token.GreaterThan -> Parser.GreaterThan
    | Token.LessThanOrEqual -> Parser.LessThanOrEqual
    | Token.GreaterThanOrEqual -> Parser.GreaterThanOrEqual
    | Token.LeftShift -> Parser.LeftShift
    | Token.RightShift -> Parser.RightShift
    | Token.Arrow -> Parser.Arrow
    | Token.Plus -> Parser.Plus
    | Token.Minus -> Parser.Minus
    | Token.Times -> Parser.Times
    | Token.Divide -> Parser.Divide
    | Token.And -> Parser.And
    | Token.Or -> Parser.Or
    | Token.AndAnd -> Parser.AndAnd
    | Token.OrOr -> Parser.OrOr
    | Token.Caret -> Parser.Caret
    | Token.Percent -> Parser.Percent
    | Token.Tilde -> Parser.Tilde
    | Token.Pound -> Parser.Pound
    | Token.DollarSign -> Parser.DollarSign
  ;;

  let parse_token (lexbuf : Lexing.lexbuf) : Parser.token =
    lexbuf |> Lexer.token |> translate_token
  ;;

  let compile (token_src : token_src) : Ast.ast =
    let {src; tokens} = token_src in
    let {path; code} = src in
    let lexbuf = Lexing.from_string code in
    (* let body = Parser.module_body parse_token lexbuf in *)
    ignore parse_token;
    ignore lexbuf;
    let body = {Ast.module_items = []} in
    let name = Filename.basename path in
    let module_ = {Ast.module_name = name; Ast.module_body = body} in
    let ast = {Ast.path; Ast.module_} in
    ignore tokens;
    ast
  ;;
end)

module Desugar = MakeStage (struct
  type input = Ast.ast

  type output = desugared_ast

  let from_file = json_deserializer Ast.ast_of_yojson

  let to_file = json_serializer yojson_of_desugared_ast

  let compile (ast : Ast.ast) : desugared_ast =
    let {path; Ast.module_} = ast in
    ignore module_;
    {path}
```

```
  ;;
end)

module TypeCheck = MakeStage (struct
  type input = desugared_ast

  type output = typed_ast

  let from_file = json_deserializer desugared_ast_of_yojson

  let to_file = json_serializer yojson_of_typed_ast

  let compile (ast : desugared_ast) : typed_ast = {path = ast.path}
end)

module Lower = MakeStage (struct
  type input = typed_ast

  type output = Lir.lir

  let from_file = json_deserializer typed_ast_of_yojson

  let to_file = json_serializer Lir.yojson_of_lir

  let compile (ast : typed_ast) : Lir.lir =
    Lir.(
      let u64 = IntType {bits = 64; unsigned = true} in
      let i64 = IntType {bits = 64; unsigned = false} in
      ignore u64;
      ignore i64;
      {path = ast.path; globals = []; functions = [
      (* {
        func_name = "gcd";
        func_type = {
          func_args = [|i64; i64|];
          func_return_type = i64;
        };
        func_decl = Some {
          arg_names = [|"a"; "b"|];
          func_value = {
            type_ = i64;
            value = Literal (Int 0);
          };
        };
      };
      {
        func_name = "gcd'";
        func_type = {
          func_args = [|u64; u64|];
          func_return_type = u64;
        };
        func_decl = Some {
          arg_names = [|"a"; "b"|];
          func_value = {
            type_ = u64;
            value = Literal (Int 0);
          };
        };
      } *)
    ]})
  ;;
end)

module CodeGen = MakeStage (struct
  type input = Lir.lir

  type output = LL.llcontext * LL.llmodule

  let from_file = json_deserializer Lir.lir_of_yojson

  let to_file ~(path : string) ((ctx, mod_) : LL.llcontext * LL.llmodule) : unit
    =
    LLAnalysis.assert_valid_module mod_;
    LL.print_module path mod_;
```

```
    LL.dispose_module mod_;
    LL.dispose_context ctx;
    ()
  ;;

  let compile (lir : Lir.lir) : LL.llcontext * LL.llmodule =
    LL.enable_pretty_stacktrace ();
    let ctx = LL.create_context () in
    let mod_ = LL.create_module ctx lir.path in
    let target_triple = LLTarget.Target.default_triple () in
    LL.set_target_triple target_triple mod_;
    Codegen.compile ~lir ~ctx ~mod_;
    (ctx, mod_)
  ;;
end)
```

Code Listing - Table of Contents

src/compiler.mli

```
module type Stage = sig
  type input

  type output

  val compile : input -> output

  val compile_file : input_path:string -> output_path:string -> unit
end

module Lex : Stage
module Parse : Stage
module Desugar : Stage
module TypeCheck : Stage
module Lower : Stage
module CodeGen : Stage

type src = {
    path : string
  ; code : string
} [@@deriving show, yojson]

type token_src = {
    src : src
  ; tokens : Token.token list
}
[@@deriving show, yojson]

type desugared_ast = {path : string} [@@deriving show, yojson]

type typed_ast = {path : string} [@@deriving show, yojson]
```

Code Listing - Table of Contents

src/cstar.ml

```
let main () : unit =
  Driver.run ();
  ()
;;

let () = main ()
```

Code Listing - Table of Contents

src/driver.ml

```
open Core

let range ~(min : int) ~(max : int) : int list =
  let n = max - min in
  if n < 0 then [] else List.init n ~f:(fun i -> i + min)
;;

let _range_inclusive ~(min : int) ~(max : int) : int list =
  range ~min ~max:(max + 1)
;;

let quote_arg (arg : string) : string =
  if String.contains arg ' ' then "\"" ^ arg ^ "\"" else arg
;;

let argv_to_string (argv : string list) : string =
  argv |> List.map ~f:quote_arg |> String.concat ?sep:(Some " ")
;;

type raw_compile_args = {
    src_path : string
  ; src_type : EmitType.t
  ; out_path : string
  ; out_type : EmitType.t
}
[@@deriving show]

let run_subprocess ~(program : string) ~(argv : string list) ~(use_path : bool)
    : unit
  =
  let pid = Unix.fork_exec ~prog:program ~argv ?use_path:(Some use_path) () in
  let (_, status) = Unix.wait ?restart:(Some true) (`Pid pid) in
  status
  |> Result.map_error ~f:(fun e ->
         let cmd = argv |> argv_to_string in
         let exit_message = Error e |> Unix.Exit_or_signal.to_string_hum in
         let message = cmd ^ ": " ^ exit_message in
         failwith message)
  |> Result.ok_exn
;;

let compile_file_raw ~(args : raw_compile_args) : unit =
  let {src_path; src_type; out_path; out_type} = args in
  if EmitType.is_llvm src_type && EmitType.is_llvm out_type
  then (
    let args =
      match (src_type, out_type) with
      (* | (Src, Ast) | (Ast, Ir) -> failwith "C* ast not currently supported" *)
      (* prefer delegating to clang since it knows how to invoke llvm better *)
      | (Ir, Bc) -> ["clang"; "-x"; "ir"; "-emit-llvm"; "-c"]
      | (Bc, Asm) -> ["llc"; "--filetype=asm"]
      | (Asm, Obj) -> ["clang"; "-c"]
      | (Obj, Exe) -> ["clang"; "-fuse-ld=lld"]
      | _ -> failwith "invalid src and out llvm types for compile-raw"
    in
    let argv = args @ ["-o"; out_path; src_path] in
    let program = args |> List.find ~f:(fun _ -> true) |> Option.value_exn in
    run_subprocess ~program ~argv ~use_path:true;
    ())
  else (
    let compile_file =
      match (src_type, out_type) with
      | (Src, Tokens) -> Compiler.Lex.compile_file
      | (Tokens, Ast) -> Compiler.Parse.compile_file
      | (Ast, DesugaredAst) -> Compiler.Desugar.compile_file
      | (DesugaredAst, TypedAst) -> Compiler.TypeCheck.compile_file
      | (TypedAst, Lir) -> Compiler.Lower.compile_file
      | (Lir, Ir) -> Compiler.CodeGen.compile_file
      | _ -> failwith "invalid src and out cstar types for compile-raw"
    in
```

```
    compile_file ~input_path:src_path ~output_path:out_path;
    ())
;;

let run_raw_compile
    ~(args : raw_compile_args)
    ~(print : bool)
    ~(in_new_process : bool)
    : unit
  =
  let print_command argv = argv |> argv_to_string |> print_endline in

  if print || in_new_process
  then (
    let {src_path; src_type; out_path; out_type} = args in
    let argv =
      [
        "cstar"
      ; "compile-raw"
      ; "--src"
      ; src_path
      ; "--src-type"
      ; EmitType.to_string src_type
      ; "--output"
      ; out_path
      ; "--out-type"
      ; EmitType.to_string out_type
      ]
    in
    if print
    then print_command argv
    else run_subprocess ~program:Sys.executable_name ~argv ~use_path:false)
  else compile_file_raw ~args
;;

let compile_file
    ~(src_path : string)
    ~(src_type : EmitType.t option)
    ~(out_path : string option)
    ~(out_type : EmitType.t option)
    ~(temps_dir : string option)
    ~(print_driver_commands : bool)
    ~(no_exe_extension : bool)
    ~(run_driver_commands_in_new_processes : bool)
    : unit
  =
  let src_type =
    src_type
    |> Util.value_or_thunk ~default:(fun () ->
           EmitType.detect_exn ~path:src_path ~no_exe_extension)
  in
  let out_type =
    out_type
    |> Util.value_or_thunk ~default:(fun () ->
           match out_path with
           | Some out_path ->
               EmitType.detect_exn ~path:out_path ~no_exe_extension
           | None -> Exe)
  in
  let out_path =
    out_path
    |> Util.value_or_thunk ~default:(fun () ->
           let (dir_and_stem, _) = Filename.split_extension src_path in
           let ext = EmitType.extension out_type ~no_exe_extension in
           dir_and_stem ^ ext)
  in

  if String.equal src_path out_path
  then failwith "src and out path are the same";

  (* match compare_emit_type src_type out_type with | -1 -> () | 0 -> ( (* file
     can't be the same; just do a simple copy then *)

     ) | 1 -> failwith "can't decompile"; *)
```

```
  (* TODO at least match llvm's -save-temps=obj, too*)
  let temps_dir =
    temps_dir
    |> Util.value_or_thunk ~default:(fun () ->
           Filename.temp_dir (Filename.basename src_path) ".cstar")
  in
  let temps_dir =
    match temps_dir with
    | "" -> Filename.dirname out_path
    | _ -> temps_dir
  in
  let src_name =
    let base = Filename.basename src_path in
    let (stem, _) = Filename.split_extension base in
    stem
  in
  let temp_path emit_type =
    Filename.concat
      temps_dir
      (src_name ^ EmitType.extension emit_type ~no_exe_extension)
  in

  let raw_compile_args =
    range ~min:(EmitType.to_enum src_type) ~max:(EmitType.to_enum out_type)
    |> List.map ~f:(fun i ->
           (EmitType.of_enum_exn i, EmitType.of_enum_exn (i + 1)))
    |> List.map ~f:(fun (src, out) ->
           {
             src_path =
               (if EmitType.equal src src_type then src_path else temp_path src)
           ; src_type = src
           ; out_path =
               (if EmitType.equal out out_type then out_path else temp_path out)
           ; out_type = out
           })
  in

  if List.is_empty raw_compile_args
  then failwith "nothing to compile and cannot decompile";

  raw_compile_args
  |> List.iter ~f:(fun args ->
         run_raw_compile
           ~args
           ~print:print_driver_commands
           ~in_new_process:run_driver_commands_in_new_processes);

  (* TODO: clean up temp dir *)
  ()
;;

let generate_completions (shell : string option) : unit =
  let shell =
    shell
    |> Option.bind ~f:(fun _ ->
           Sys.getenv "SHELL" |> Option.map ~f:Filename.basename)
    |> Option.value ~default:"bash"
  in
  let env_var = "COMMAND_OUTPUT_INSTALLATION_" ^ String.uppercase shell in
  let env = `Extend [(env_var, "1")] in
  let (_ : never_returns) =
    Unix.exec
      ~prog:Sys.executable_name
      ~argv:["cstar"]
      ?use_path:(Some false)
      ?env:(Some env)
      ()
  in
  ()
;;

let make_cmd () : Core.Command.t =
  let completions =
```

```
    Command.basic
      ~summary:"generate autocompletions"
      Command.Let_syntax.(
        let%map_open shell = anon (maybe ("shell" %: string)) in
        fun () -> generate_completions shell)
  in
  let compile =
    Command.basic
      ~summary:"compile a C* source file"
      Command.Let_syntax.(
        let%map_open src_path = anon ("source_file" %: Filename.arg_type)
        and src_type =
          flag
            "--src-type"
            (optional EmitType.arg)
            ~doc:"src-type type of source if not inferred"
        and out_path =
          flag
            ?aliases:(Some ["-o"])
            "--output"
            (optional Filename.arg_type)
            ~doc:"output output file"
        and out_type =
          flag
            ?aliases:(Some ["--emit"])
            "--out-type"
            (optional EmitType.arg)
            ~doc:"out-type what to output/emit"
        and temps_dir =
          flag
            "--save-temps"
            (optional Filename.arg_type)
            ~doc:"save-temps save all temporaries"
        and print_driver_commands =
          flag
            "--print-driver-commands"
            no_arg
            ~doc:"print-driver-commands print a dry run of the driver commands"
        and exe_extension =
          flag
            "--exe-extension"
            no_arg
            ~doc:"exe-extension use a `.cstar.exe` extension for executables"
        and run_driver_commands_in_new_processes =
          flag
            "--run-driver-commands-in-new-processes"
            no_arg
            ~doc:"run-driver-commands-in-new-processes as it says"
        in
        fun () ->
          compile_file
            ~src_path
            ~src_type
            ~out_path
            ~out_type
            ~temps_dir
            ~print_driver_commands
            ~no_exe_extension:(not exe_extension)
            ~run_driver_commands_in_new_processes)
  in
  let compile_raw =
    Command.basic
      ~summary:
        "compile a single stage of a C* source file; this is what the driver \
         invokes"
      Command.Let_syntax.(
        let%map_open src_path =
          flag "--src" (required Filename.arg_type) ~doc:"src src file"
        and src_type =
          flag
            "--src-type"
            (required EmitType.arg)
            ~doc:"src-type type of source if not inferred"
        and out_path =
```

```
          flag "--output" (required Filename.arg_type) ~doc:"output output file"
        and out_type =
          flag
            "--out-type"
            (required EmitType.arg)
            ~doc:"out-type what to output/emit"
        in
        fun () ->
          compile_file_raw ~args:{src_path; src_type; out_path; out_type})
  in
  Command.group
    ~summary:"the C* compiler"
    ~readme:(fun () -> "See README.md")
    [
      ("completions", completions)
    ; ("compile", compile)
    ; ("compile-raw", compile_raw)
    ]
;;

let run () : unit = make_cmd () |> Command.run ~version:"0.1" ~build_info:""
```

Code Listing - Table of Contents

src/driver.mli

```
type raw_compile_args = {
    src_path : string
  ; src_type : EmitType.t
  ; out_path : string
  ; out_type : EmitType.t
}
[@@deriving show]

val compile_file_raw : args:raw_compile_args -> unit

val compile_file
  :  src_path:string
  -> src_type:EmitType.t option
  -> out_path:string option
  -> out_type:EmitType.t option
  -> temps_dir:string option
  -> print_driver_commands:bool
  -> no_exe_extension:bool
  -> run_driver_commands_in_new_processes:bool
  -> unit

val run : unit -> unit
```

Code Listing - Table of Contents

src/emitType.ml

```
open Core

type t =
  | Src
  | Tokens
  | Ast
  | DesugaredAst
  | TypedAst
  | Lir
  | Ir
  | Bc
  | Asm
  | Obj
```

```
  | Exe
[@@deriving show, eq, ord, enum]

let of_enum_exn (i : int) : t = i |> of_enum |> Option.value_exn

let all : t list =
  [Src; Tokens; Ast; DesugaredAst; TypedAst; Ir; Bc; Asm; Obj; Exe]
;;

let to_string (this : t) : string =
  match this with
  | Src -> "src"
  | Tokens -> "tokens"
  | Ast -> "ast"
  | DesugaredAst -> "desugared-ast"
  | TypedAst -> "typed-ast"
  | Lir -> "lir"
  | Ir -> "ir"
  | Bc -> "bc"
  | Asm -> "asm"
  | Obj -> "obj"
  | Exe -> "exe"
;;

let of_string (s : string) : t =
  all
  |> List.find ~f:(fun it -> String.equal s (to_string it))
  |> Option.value_exn ?message:(Some "invalid emit type")
;;

let extensions (this : t) ~(no_exe_extension : bool) : string list =
  let base = "cstar" in
  let json = "json" in
  match this with
  | Src -> []
  | Tokens -> [base; "tokens"; json]
  | Ast -> [base; "raw"; "ast"; json]
  | DesugaredAst -> [base; "desugared"; "ast"; json]
  | TypedAst -> [base; "typed"; "ast"; json]
  | Lir -> [base; "lir"; json]
  | Ir -> [base; "ll"]
  | Bc -> [base; "bc"]
  | Asm -> [base; "s"]
  | Obj -> [base; "o"]
  | Exe -> (
      match no_exe_extension with
      | true -> []
      | false -> [base; "exe"])
;;

let extension (this : t) ~(no_exe_extension : bool) : string =
  this
  |> extensions ~no_exe_extension
  |> (fun a -> "" :: a)
  |> String.concat ?sep:(Some ".")
;;

let detect_by_extension (path : string) ~(no_exe_extension : bool) : t option =
  all
  |> List.find ~f:(fun t ->
         String.is_suffix ~suffix:(extension t ~no_exe_extension) path)
;;

(* TODO For example, llvm bitcode starts with `BC\0xC0\0xD\0xE` (`BC0xC0DE`). *)
let detect_by_magic (_path : string) : t option = None

let detect ~(path : string) ~(no_exe_extension : bool) : t option =
  [detect_by_extension ~no_exe_extension; detect_by_magic]
  |> List.fold ~init:(Ok ()) ~f:(fun acc f ->
         match acc with
         | Ok () -> (
             match f path with
             | Some emit -> Error emit
             | None -> Ok ())
```

```
          | Error emit -> Error emit)
  |> Result.error
;;

let detect_exn ~(path : string) ~(no_exe_extension : bool) : t =
  detect ~path ~no_exe_extension
  |> Option.value_exn ?message:(Some "couldn't detect file type")
;;

let is_llvm (this : t) : bool =
  match this with
  | Src -> false
  | Tokens -> false
  | Ast -> false
  | DesugaredAst -> false
  | TypedAst -> false
  | Lir -> false
  | Ir -> true
  | Bc -> true
  | Asm -> true
  | Obj -> true
  | Exe -> true
;;

let arg : t Command.Arg_type.t =
  all
  |> List.map ~f:(fun it -> (to_string it, it))
  |> String.Map.of_alist_exn
  |> Command.Arg_type.of_map
;;
```

Code Listing - Table of Contents

src/emitType.mli

```
open Core

type t =
  | Src
  | Tokens
  | Ast
  | DesugaredAst
  | TypedAst
  | Lir
  | Ir
  | Bc
  | Asm
  | Obj
  | Exe
[@@deriving show, eq, ord, enum]

val of_enum_exn : int -> t

val all : t list

val to_string : t -> string

val of_string : string -> t

val extension : t -> no_exe_extension:bool -> string

val detect : path:string -> no_exe_extension:bool -> t option

val detect_exn : path:string -> no_exe_extension:bool -> t

val is_llvm : t -> bool

val arg : t Command.Arg_type.t
```

Code Listing - Table of Contents

src/lexer.mll

```
{
  open Token

  let of_char = Core.String.of_char
}

(* https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=%5B%3AXID_Start%3A%5D&abb=on&g=&i=
   But only ascii for now
*)
let xid_start = ['a'-'z' 'A'-'Z']
(* https://util.unicode.org/UnicodeJsps/list-unicodeset.jsp?a=%5B%3AXID_Continue%3A%5D&abb=on&g=&i=
   But only ascii for now
*)
let xid_continue = xid_start | ['0'-'9' '_']

let identifier = ['_' '$'] | xid_start) xid_continue*

(* let binary_digit = ['0'-'1']
let octal_digit = ['0'-'7']
let decimal_digit = ['0'-'9']
let hex_digit = ['0'-'9' 'A'-'F' 'a'-'f']

let raw_binary_int = "0b" ((binary_digit | '_')* binary_digit as digits)
let raw_octal_int = "0o" ((octal_digit | '_')* octal_digit as digits)
let raw_hex_int = "0x" ((hex_digit | '_')* hex_digit as digits)
let raw_decimal_int = (decimal_digit | (decimal_digit (decimal_digit | '_')* decimal_digit)) as digits

let sign = ['+' '-']? as sign
let raw_int = raw_binary_int | raw_octal_int | raw_decimal_int | raw_hex_int
let num_suffix = identifier? as suffix

let integral = sign raw_int
let floating = '.' raw_int
let exponent = ['e' 'E'] sign raw_int

let num = integral floating? exponent? num_suffix *)

let sign = ['+' '-']?
let digit = ['0'-'9' 'A'-'F']
let raw_int = ('0' ['b' 'o' 'x'] '_'?)? (digit | (digit (digit | '_')* digit))
let integral = (sign raw_int as integral)
let floating = '.' (raw_int as floating)
let exponent = 'e' (sign raw_int as exponent)
let num = integral floating? exponent? (identifier as suffix)?

rule token = parse
  | eof { EOF }
  (* simple literal tokens *)
  | ';' { SemiColon }
  | ':' { Colon }
  | ',' { Comma }
  | '.' { Dot }
  | ".." { DotDot }
  | '(' { OpenParen }
  | ')' { CloseParen }
  | '{' { OpenBrace }
  | '}' { CloseBrace }
  | '[' { OpenBracket }
  | ']' { CloseBracket }
  | '@' { At }
  | '?' { QuestionMark }
  | '!' { ExclamationPoint }
  | '=' { Equal }
  | "==" { EqualEqual }
  | "!=" { NotEqual }
  | '<' { LessThan }
  | '>' { GreaterThan }
  | "<=" { LessThanOrEqual }
```

```
  | ">=" { GreaterThanOrEqual }
  | "=>" { Arrow }
  | "<<" { LeftShift }
  | ">>" { RightShift }
  | '+' { Plus }
  | '-' { Minus }
  | '*' { Times }
  | '/' { Divide }
  | '&' { And }
  | '|' { Or }
  | "&&" { AndAnd }
  | "||" { OrOr }
  | '^' { Caret }
  | '%' { Percent }
  | '~' { Tilde }
  | '#' { Pound }
  | '$' { DollarSign }
  (* whitespace *)
  | ([' ' '\n' '\r' '\t' '\x0B' '\x0C']+) as s { WhiteSpace s }
  (* comments *)
  (* Only match the actual structural "slashdash" comment,
   * since we need to fully parse to know what it comments out.
   *)
  | "/-" { Comment Structural }
  (* Could also be a doc comment if /// so definitely store the comment string. *)
  | "//" ([^ '\n' '\r']* as s) { Comment (Line s) }
  (* Need to do this recursively since block comments can be nested. *)
  | "/*" { Comment (Block (block_comment 0 "" lexbuf)) }
  (* string/char literals *)
  | (['b']? as prefix) '\'' { Literal (Char {prefix; unescaped = unescape_char_literal "" lexbuf}) }
  | (['b' 'c' 'r' 'f']? as prefix) '"' { Literal (String {prefix; unescaped = unescape_string_literal "" lexb
  (* keyword and identifiers
  ideally keywords would be done in the parser
  (i.e., they'd all be identifiers here in the lexer)
  due to context dependencies, but that's harder in ocamlyacc *)
  | (identifier) as s {
    Token.keyword_of_string s
    |> Option.map (fun kw -> Keyword kw)
    |> Option.value ~default:(Identifier s)
  }
  (* number literals *)
  | num { Literal (Number ({
      integral = integral |> parse_raw_int_literal
    ; floating = floating |> Option.map parse_raw_int_literal
    ; exponent = exponent |> Option.map parse_raw_int_literal
    ; suffix = suffix |> Option.value ~default:""
  })) }
  | _ as c { failwith (Printf.sprintf "illegal character: %c" c) }


(* https://stackoverflow.com/questions/7117975/how-to-deal-with-nested-comments-in-fslex
   How do I get the comment value, or do I even need to?
*)
and block_comment depth comment = parse
  | "*/" as s { match depth with
    | 0 -> comment
    | _ -> block_comment (depth - 1) (comment ^ s) lexbuf
  }
  | "/*" as s { block_comment (depth + 1) (comment ^ s) lexbuf }
  | _  as c { block_comment depth (comment ^ of_char c) lexbuf }

and decode_char_escape = parse
  | ['\\' '\'' '"'] as c { c }
  | 'n' { '\n' }
  | 'r' { '\r' }
  | 't' { '\t' }
  | 'b' { '\b' }
  | 'v' { '\x0B' }
  | 'f' { '\x0C' }
  | _ as c { failwith (Printf.sprintf "illegal escape: %c" c) }

and unescape_char_literal s = parse
  | '\'' { s }
  | '\\' { unescape_char_literal (s ^ of_char (decode_char_escape lexbuf)) lexbuf }
```

```
  | _ as c { unescape_char_literal (s ^ of_char c) lexbuf }

(* modified from: https://github.com/realworldocaml/examples/blob/v1/code/parsing/lexer.mll *)
and unescape_string_literal s = parse
  | '"' { s }
  | '\\' { unescape_string_literal (s ^ of_char (decode_char_escape lexbuf)) lexbuf }
  | _ as c { unescape_string_literal (s ^ of_char c) lexbuf }
```

Code Listing - Table of Contents

src/lir.ml

```
(* In LIR (low-level cstar IR), everything is monomorphized and all symbols,
   globals and functions, have unique names now.

   Platform-dependent types like usize have been expanded. Types and other
   semantic analyses are all checked by now. Everything is now mutable.
   References are now pointers, slices are structs structs are now tuples (llvm
   aggregate types), TODO enums unions.

   `main` is guaranteed to be defined.*)

type int_type = {
    bits : int
  ; unsigned : bool
}
[@@deriving yojson]

type float_type =
  | F32
  | F64
[@@deriving yojson]

type func_type = {
    func_args : type_ array
  ; func_return_type : type_
}
[@@deriving yojson]

and type_ =
  | UnitType
  | IntType of int_type
  | FloatType of float_type
  | PointerType of type_
  | ArrayType of type_ * int
  | TupleType of type_ array
  | FuncType of func_type
[@@deriving yojson]

type literal =
  | Int of int
  | Float of float
[@@deriving yojson]

type global = {
    global_name : string
  ; global_type : type_
  ; global_value : literal option
}
[@@deriving yojson]

type unary_op =
  | Negate
  | Not
  | BitNot
  | AddressOf
  | Dereference
[@@deriving yojson]

type arithmetic_binary_op =
```

```
  | Add
  | Subtract
  | Multiply
  | Divide
  | Modulo
  | And
  | Or
  | BitAnd
  | BitOr
  | BitXor
  | LeftShift
  | RightShift
[@@deriving yojson]

type comparison_op =
  | Equal
  | NotEqual
  | LessThan
  | LessThanOrEqual
  | GreaterThan
  | GreaterThanOrEqual
[@@deriving yojson]

type binary_op =
  | Assign
  | Arithmetic of arithmetic_binary_op
  | Comparison of comparison_op
[@@deriving yojson]

type label = string [@@deriving yojson]

type call_expr = {
    callee : expr
  ; call_args : expr array
}
[@@deriving yojson]

and if_expr = {
    condition : expr
  ; then_case : expr
  ; else_case : expr
}
[@@deriving yojson]

and raw_expr =
  | Literal of literal
  | Var of string
  | UnaryOp of unary_op * expr
  | BinaryOp of expr * binary_op * expr
  | Cast of expr
  | Call of call_expr
  | If of if_expr
  | GoTo of expr
  | Block of expr list (* non-empty *)
[@@deriving yojson]

and expr = {
    type_ : type_
  ; value : raw_expr
}
[@@deriving yojson]

type func_decl = {
    arg_names : string array
  ; func_value : expr
}
[@@deriving yojson]

type func = {
    func_name : string
  ; func_type : func_type
  ; func_decl : func_decl option
}
[@@deriving yojson]
```

```
type lir = {
    path : string
  ; globals : global list
  ; functions : func list
}
[@@deriving yojson]
```

Code Listing - Table of Contents

src/parser.mly

```
%{
    open Ast
%}

%token EOF
%token <string> WhiteSpace
%token StructuralComment
%token <string> LineComment
%token <string> BlockComment
%token <string> Identifier
%token <Token.number_literal> NumberLiteral
%token <Token.char_literal> CharLiteral
%token <Token.string_literal> StringLiteral
%token KwMod
%token KwUse
%token KwLet
%token KwMut
%token KwPub
%token KwTry
%token KwConst
%token KwImpl
%token KwFn
%token KwStruct
%token KwEnum
%token KwUnion
%token KwReturn
%token KwBreak
%token KwContinue
%token KwFor
%token KwWhile
%token KwIf
%token KwElse
%token KwMatch
%token KwDefer
%token KwUndefer
%token KwIn
%token KwSelf
%token KwTrait
%token SemiColon
%token Colon
%token Comma
%token Dot
%token DotDot
%token OpenParen
%token CloseParen
%token OpenBrace
%token CloseBrace
%token OpenBracket
%token CloseBracket
%token At
%token QuestionMark
%token ExclamationPoint
%token Equal
%token EqualEqual
%token NotEqual
%token LessThan
%token GreaterThan
%token LessThanOrEqual
```

```
%token GreaterThanOrEqual
%token LeftShift
%token RightShift
%token Arrow
%token Plus
%token Minus
%token Times
%token Divide
%token And
%token Or
%token AndAnd
%token OrOr
%token Caret
%token Percent
%token Tilde
%token Pound
%token DollarSign

%start module_body
%type <Ast.module_body> module_body

// TODO precedence

%nonassoc Equal
%nonassoc DotDot
%left OrOr
%left AndAnd
%nonassoc EqualEqual NotEqual LessThan GreaterThan LessThanOrEqual GreaterThanOrEqual
%left Or
%left Caret
%left And
%left LeftShift RightShift
%left Plus Minus
%left Times Divide Modulo
%left Dot
%left At

%%

// trailing comma required for 1-element tuple
// tuple_elements:
// | { [] }
// | expr Comma expr { [$1; $3] }
// | expr Comma tuple_elements { $1 :: $3 }

// tuple:
// | OpenParen tuple_elements CloseParen { {elements = $2} }

// array_elements:
// | { [] }
// | expr { [$1] }
// | expr Comma array_elements { $1 :: $3 }

// array:
// | OpenBracket array_elements CloseBracket { {elements = $2} }

// struct_initializer:
// | Identifier Colon expr { Explicit ($1, $3) }
// | Identifier { Implicit $1 }
// | DotDot expr { Spread $2 }

// struct_initializers:
// | { [] }
// | struct_initializer { [$1] }
// | struct_initializer Comma struct_initializers { $1 :: $3 }

// struct:
// | Identifier OpenBrace struct_initializers CloseBrace { {
//     name = $1;
//     fields = $3;
// } }

// range_op:
// | DotDot { () }
```

```
// range_options:
// | { {inclusive = false; sign = None} }
// | Equal { {inclusive = true; sign = None} }
// | Plus { {inclusive = false; sign = Plus} }
// | Plus Equal { {inclusive = true; sign = Plus} }
// | Minus { {inclusive = false; sign = Minus} }
// | Minus Equal { {inclusive = true; sign = Minus} }

// range:
// | expr range_op range_options expr { {start = $1; stop = $3; options = $2} }
// | expr range_op range_options { {start = $1; stop = None; options = $2} }
// | range_op range_options expr { {start = None; stop = $3; options = $2} }

// literal:
// | NumberLiteral { Number $1 }
// | CharLiteral { Char $1 }
// | StringLiteral { String $1 }
// | tuple { Tuple $1 }
// | array { Array $1 }
// | struct { Struct $1 }
// | range { Range $1 }
// TODO { Func $1 }
// TODO { Closure $1 }

// path:
// | Identifier { [$1] }
// | Identifier Dot path { $1 :: $3 }

publicity:
| { Private }
| KwPub { Public }
// | KwPub OpenParen KwIn WhiteSpace path CloseParen { PublicIn $5 }

// annotation:
// | At path { {
//     path = $2;
//     args = [];
// } }
// | At path tuple { {
//     path = $2;
//     args = $3;
// } }

// annotations:
// | { [] }
// | annotation WhiteSpace annotations { $1 :: $2 }

metadata:
// | annotations publicity { {
//     publicity = $2;
//     annotations = $1;
//     doc_comments = {
//         lines = [];
//     };
// } }
| publicity { {
    publicity = $1;
    annotations = [];
    doc_comment = {
        lines = [];
    };
} }

// use:
// | KwUse path SemiColon { {
//     path = $2;
// } }

// mut:
// | { {mut = false} }
// | KwMut { {mut = true} }

type_:
```

```
| Identifier { NamedType {
    type_name = $1;
} }
// | type_ Times mut { PointerType {
//     pointee = $1;
//     mutability = $3;
// } }
// | type_ And mut { ReferenceType {
//     referent = $1;
//     mutability = $3;
// } }
// | type_ OpenBracket CloseBracket { SliceType {
//     element_type = $1;
// } }
// TODO: eventually allow length to be an expr if it's const
// | type_ OpenBracket NumberLiteral CloseBracket { ArrayType {
//     element_type = $1;
//     array_length = Literal (Number $3);
// } }
| tuple_type { TupleType $1 }
// | KwFn tuple_type Colon type_ { FuncType {
//     args = $2;
//     return_type = $4;
// } }
// TODO GenericType
// | OpenParen type_ CloseParen { $2 } // as parentheses

// single-element tuple requires trailing comma
// but otherwise it's optional
inner_tuple_type:
| { [] }
| type_ Comma type_ { [$1; $3] }
| type_ Comma inner_tuple_type { $1 :: $3 }

tuple_type:
| OpenParen inner_tuple_type CloseParen { {
    elements = $2;
} }

type_annotation:
| { InferredType }
| Colon type_ { $2 }

// variable:
// | mut Identifier type_annotation { {
//     name = $2;
//     type_ = $3;
//     mutability = $1;
// } }

// meta_variable:
// | metadata variable { {
//     variable = $2;
//     metadata = $1;
// } }

// let_:
// | KwLet variable Equal expr SemiColon { {
//     variable = $2;
//     value = $4;
// } }

// unary_op:
// | Minus { Negate }
// | ExclamationPoint { Not }
// | Tilde { BitNot }

// arithmetic_binary_op:
// | Plus { Add }
// | Minus { Subtract }
// | Times { Multiply }
// | Divide { Divide }
// | Percent { Modulo }
// | AndAnd { And }
```

```
// | OrOr { Or }
// | And { BitAnd }
// | Or { BitOr }
// | Caret { BitXor }
// | LeftShift { LeftShit }
// | RightShift { RightShift }

// comparison_op:
// | EqualEqual { Equal }
// | NotEqual { NotEqual }
// | LessThan { LessThan }
// | LessThanOrEqual { LessThanOrEqual }
// | GreaterThan { GreaterThan }
// | GreaterThanOrEqual { GreaterThanOrEqual }

// binary_op:
// | arithmetic_binary_op { ArithmeticBinaryOp $1 }
// | arithmetic_binary_op Equal { AssigningArithmeticBinaryOp $1 }
// | comparison_op { ComparisonOp $1 }
// | Equal { Assign }

// else:
// | { None }
// | KwElse block { Some $2 }

// if:
// | KwIf block else { {
//     then_case = $2;
//     else_case = $3;
// } }

// pattern:
// | mut Identifier { IdentifierPattern ($2, $1) }
// | NumberLiteral { NumPattern $1 }
// | CharLiteral { CharPattern $1 }
// | StringLiteral { StringPattern $1 }
// | DotDot { RestPattern }

// pattern_condition:
// | { None }
// | KwIf WhiteSpace expr { Some $3 }

// match_arm:
// | pattern pattern_condition Arrow expr { {
//     match_pattern = $1;
//     match_condition = $2;
//     match_arm_value = $4;
// } }

// match_arms:
// | { [] } // empty match on empty enum (a never type) would be valid
// | match_arm Comma { [$1] }
// | match_arm Comma match_arms { $1 :: $3 }

// match:
// | KwMatch OpenBrace match_arms CloseBrace { {
//     match_arms = $3;
// } }

// label:
// | WhiteSpace { None }
// | At Identifier WhiteSpace { Some {label_name = $2} }

// goto_kw:
// | KwReturn { Return }
// | KwBreak { Break }
// | KwContinue { Continue }

// postfix_expr:
// | Times mut { Dereference $2 }
// | And mut { Reference $2 }
// | QuestionMark { Try }
// | binary_op { BinaryOp $1 }
// | Identifier { FieldAccess $1 }
```

```
// | NumberLiteral { ElementAccess $1 }
// | Identifier tuple { MethodCall {
//     func = $1;
//     generic_args = [];
//     args = $2;
// } }
// | goto_kw label { GoTo ($2, $1) }
// | KwDefer label { Defer $2 }
// | match { Match $1 }
// | if { If $1 }
// | KwFor label variable block { For {
//     for_element = $3;
//     for_block = $4;
//     for_label = $2;
// } }
// | KwWhile label block { While ($2, $3) }

// blockless_expr:
// | Identifier { Variable $1 }
// | literal { Literal $1 }
// | unary_op expr { UnaryOp {
//     unary_op = $1;
//     unary_value = $2;
// } }
// | expr binary_op expr { BinaryOp {
//     binary_op = $2;
//     left = $1;
//     right = $3;
// } }
// | OpenBracket expr CloseBracket { Index $2 }
// // separate func and method call to differentiate
// // method calls and field function pointer calls
// | expr tuple { FuncCall {
//     func = $1;
//     call_generic_args = [];
//     call_args = $2;
// } }
// | expr Dot postfix_expr { PostFixExpr ($1, $3) }
// | KwUndefer label { UnDefer $2 }

statements:
| { {statements = []; trailing_semicolon = true} }
// | expr { {statements = [Expr $1]; trailing_semicolon = false} }
// | item statements {
//     let {statements; trailing} = $2 in
//     let statements = (Item $1) :: statements in
//     {statements; trailing_semicolon}
// }
// | expr SemiColon statements {
//     let {statements; trailing} = $3 in
//     let statements = (Expr $1) :: statements in
//     {statements; trailing_semicolon}
// }

block:
| OpenBrace statements CloseBrace { $2 }

// expr:
// | blockless_expr { $1 }
// | block { Block $1 }
// | OpenParen expr CloseParen { $2 }

// if it's not a {} block, ends with a ;
terminating_expr:
// | blockless_expr SemiColon { $1 }
| block { Block $1 }

variables:
| { [] }
// | meta_variable Comma { [$1] }
// | meta_variable Comma variables { $1 :: $3 }

func_args:
| variables { $1 }
```

```
anon_func_signature:
| OpenParen func_args CloseParen type_annotation { {
    signature_args = $2;
    signature_return_type = $4;
} }

func_decl_signature:
| KwFn WhiteSpace Identifier anon_func_signature { {
    func_name = $3;
    func_signature = $4;
} }

func_decl:
| func_decl_signature SemiColon { {
    func_decl_signature = $1;
    func_value = None;
} }
| func_decl_signature Equal terminating_expr { {
    func_decl_signature = $1;
    func_value = Some $3;
} }

// fields:
// | variables { $1 }

// struct_decl:
// | KwStruct WhiteSpace Identifier OpenBrace fields CloseBrace { {
//     struct_name = $3;
//     struct_fields = $5;
// } }

// variant_data:
// | { None }
// | tuple_type { Some (TupleVariant $1) }
// | fields { Some (StructVariant $1) }

// variant:
// | Identifier variant_data { {
//     variant_name = $1;
//     variant_data = $2;
// } }

// variants:
// | { [] }
// | variant Comma { [$1] }
// | variant Comma variants { $1 :: $3 }

// enum_decl:
// | KwEnum WhiteSpace Identifier OpenBrace variants CloseBrace { {
//     enum_name = $3;
//     enum_variants = $5;
// } }

// union_decl:
// | KwUnion WhiteSpace Identifier OpenBrace fields CloseBrace { {
//     union_name = $3;
//     union_fields = $5;
// } }

// module_or_impl:
// | Identifier OpenBrace module_body CloseBrace { {
//     module_name = $1;
//     module_body = $3;
// } }

// impl:
// | KwImpl WhiteSpace module_or_impl { $3 }

// mod:
// | KwMod WhiteSpace module_or_impl { $3 }

inner_item:
// | use { Use $1 }
```

```
// | let_ { Let $1 }
| func_decl { FuncDecl $1 }
// | struct_decl { StructDecl $1 }
// | enum_decl { EnumDecl $1 }
// | union_decl { UnionDecl $1 }
// | impl { Impl $1 }
// | mod { Mod $1 }

item:
| metadata inner_item { {
    item_metadata = $1;
    inner_item = $2;
} }

items:
| { [] }
| item items { $1 :: $2 }

module_body:
| items { {module_items = $1} }
;
```

Code Listing - Table of Contents

src/stringMap.ml

```
module S = Map.Make (String)

let pp_map
    (pp : Format.formatter -> 'a -> unit)
    (fmt : Format.formatter)
    (this : 'a S.t)
    : unit
  =
  this
  |> S.to_seq
  |> Format.pp_print_seq
       (fun fmt (k, v) ->
         Format.fprintf fmt "%s: " k;
         pp fmt v)
       fmt
;;

type 'a t = {map : 'a S.t [@polyprinter pp_map]} [@@deriving show]

let yojson_of_t (to_yojson : 'a -> Yojson.Safe.t) (this : 'a t) : Yojson.Safe.t =
  this.map
  |> S.to_seq
  |> Seq.map (fun (k, v) -> (k, to_yojson v))
  |> List.of_seq
  |> fun entries -> `Assoc entries
;;

let t_of_yojson (of_yojson : Yojson.Safe.t -> 'a) (json : Yojson.Safe.t) : 'a t =
  json
  |> Yojson.Safe.Util.to_assoc
  |> List.to_seq
  |> Seq.map (fun (k, v) -> (k, of_yojson v))
  |> S.of_seq
  |> fun map -> {map}
;;
```

Code Listing - Table of Contents

src/stringMap.mli

```
module S : Map.S with type key = string

type 'a t = {map : 'a S.t} [@@deriving show]

val yojson_of_t : ('a -> Yojson.Safe.t) -> 'a t -> Yojson.Safe.t

val t_of_yojson : (Yojson.Safe.t -> 'a) -> Yojson.Safe.t -> 'a t

(* val pp : Format.formatter -> 'a t -> unit

val show : 'a t -> string *)
```

Code Listing - Table of Contents

src/token.ml

```
open Core

type comment =
  | Structural (* /- ... *)
  | Line (* // *) of string
  | Block (* /* */ *) of string
[@@deriving show, yojson]

type number_base =
  | Binary (* 0b *)
  | Octal (* 0o *)
  | HexaDecimal (* 0x *)
  | Decimal
[@@deriving show, yojson]

let base_num ~(base : number_base) : int =
  match base with
  | Binary -> 2
  | Octal -> 8
  | HexaDecimal -> 16
  | Decimal -> 10
;;

let base_ranges ~(base : number_base) : (char * char) list =
  match base with
  | Binary -> [('0', '1')]
  | Octal -> [('0', '7')]
  | HexaDecimal -> [('0', '9'); ('A', 'F')]
  | Decimal -> [('0', '9')]
;;

let is_digit ~(base : number_base) (c : char) : (unit, string) result =
  let ranges = base_ranges ~base in
  let is_valid =
    ranges |> List.exists ~f:(fun (low, high) -> Char.between c ~low ~high)
  in
  if is_valid
  then Ok ()
  else (
    let regex =
      ranges
      |> List.map ~f:(fun (low, high) -> String.of_char_list [low; '-'; high])
      |> String.concat ?sep:(Some "")
    in
    let msg =
      Printf.sprintf
        "'%c' is not a valid digit of base %d (%s); must be in /%s/"
        c
        (base_num ~base)
        (show_number_base base)
        regex
    in
    Error msg)
```

```
;;

type sign =
  | Unspecified
  | Positive
  | Negative
[@@deriving show, yojson]

type raw_int_literal = {
    sign : sign
  ; base : number_base
  ; digits : string
}
[@@deriving show, yojson]

let parse_raw_int_literal (s : string) : raw_int_literal =
  let s = s |> String.to_list in
  let (sign, s) =
    match s with
    | '+' :: s -> (Positive, s)
    | '-' :: s -> (Negative, s)
    | s -> (Unspecified, s)
  in
  let (base, s) =
    match s with
    | '0' :: 'b' :: s -> (Binary, s)
    | '0' :: 'o' :: s -> (Octal, s)
    | '0' :: 'x' :: s -> (HexaDecimal, s)
    | s -> (Decimal, s)
  in
  let digits =
    s
    |> List.filter ~f:(fun c -> not (Char.equal c '_'))
    |> List.map ~f:(fun c ->
           match is_digit ~base c with
           | Ok () -> c
           | Error msg -> failwith msg)
    |> String.of_char_list
  in
  {sign; base; digits}
;;

type number_literal = {
    integral : raw_int_literal
  ; floating : raw_int_literal option (* is a float if it has a floating part *)
  ; exponent : raw_int_literal option
  ; suffix : string (* i32, f64, u1, x1000, usize, "" *)
}
[@@deriving show, yojson]

type char_literal = {
    prefix : string
  ; unescaped : string
}
[@@deriving show, yojson]

type string_literal = {
    prefix : string
  ; unescaped : string
}
[@@deriving show, yojson]

(* TODO format_string_literal *)

type literal =
  | Number of number_literal
  | Char of char_literal
  | String of string_literal
[@@deriving show, yojson]

type keyword =
  | KwMod
  | KwUse
  | KwLet
```

```
  | KwMut
  | KwPub
  | KwTry
  | KwConst
  | KwImpl
  | KwFn
  | KwStruct
  | KwEnum
  | KwUnion
  | KwReturn
  | KwBreak
  | KwContinue
  | KwFor
  | KwWhile
  | KwIf
  | KwElse
  | KwMatch
  | KwDefer
  | KwUndefer
  | KwIn (* not always *)
  | KwTrait (* reserved for future *)
[@@deriving show, yojson]

let keyword_of_string (s : string) : keyword option =
  match s with
  | "mod" -> Some KwMod
  | "use" -> Some KwUse
  | "let" -> Some KwLet
  | "mut" -> Some KwMut
  | "pub" -> Some KwPub
  | "try" -> Some KwTry
  | "const" -> Some KwConst
  | "impl" -> Some KwImpl
  | "fn" -> Some KwFn
  | "struct" -> Some KwStruct
  | "enum" -> Some KwEnum
  | "union" -> Some KwUnion
  | "return" -> Some KwReturn
  | "break" -> Some KwBreak
  | "continue" -> Some KwContinue
  | "for" -> Some KwFor
  | "while" -> Some KwWhile
  | "if" -> Some KwIf
  | "else" -> Some KwElse
  | "match" -> Some KwMatch
  | "defer" -> Some KwDefer
  | "undefer" -> Some KwUndefer
  | "in" -> Some KwIn
  | "trait" -> Some KwTrait
  | _ -> None
;;

type token =
  | EOF
  | WhiteSpace of string (* ' \n\r\t', ... *)
  | Comment of comment
  | Literal of literal
  | Keyword of keyword
  | Identifier of string
  | SemiColon (* ; *)
  | Colon (* : *)
  | Comma (* , *)
  | Dot (* . *)
  | DotDot (* .. *)
  | OpenParen (* ( *)
  | CloseParen (* ) *)
  | OpenBrace (* { *)
  | CloseBrace (* } *)
  | OpenBracket (* [ *)
  | CloseBracket (* ] *)
  | At (* @ *)
  | QuestionMark (* ? *)
  | ExclamationPoint (* ! *)
  | Equal (* = *)
```

```
  | EqualEqual (* == *)
  | NotEqual (* != *)
  | LessThan (* < *)
  | GreaterThan (* > *)
  | LessThanOrEqual (* <= *)
  | GreaterThanOrEqual (* >= *)
  | LeftShift (* << *)
  | RightShift (* >> *)
  | Arrow (* => *)
  | Plus (* + *)
  | Minus (* - *)
  | Times (* * *)
  | Divide (* / *)
  | And (* & *)
  | Or (* | *)
  | AndAnd (* && *)
  | OrOr (* || *)
  | Caret (* ^ *)
  | Percent (* % *)
  | Tilde (* ~ *)
  | Pound (* # *)
  | DollarSign (* $ *)
[@@deriving show, yojson]
```

Code Listing - Table of Contents

src/token.mli

```
type comment =
  | Structural (* /- ... *)
  | Line (* // *) of string
  | Block (* /* */ *) of string
[@@deriving show, yojson]

type number_base =
  | Binary (* 0b *)
  | Octal (* 0o *)
  | HexaDecimal (* 0x *)
  | Decimal
[@@deriving show, yojson]

type sign =
  | Unspecified
  | Positive
  | Negative
[@@deriving show, yojson]

type raw_int_literal = {
    sign : sign
  ; base : number_base
  ; digits : string
}
[@@deriving show, yojson]

(*
ocamllex is kind of shitty.
Can't define recursive rules, so have to do the lexing/parsing twice.
*)
val parse_raw_int_literal : string -> raw_int_literal

type number_literal = {
    integral : raw_int_literal
  ; floating : raw_int_literal option (* is a float if it has a floating part *)
  ; exponent : raw_int_literal option
  ; suffix : string (* i32, f64, u1, x1000, usize, "" *)
}
[@@deriving show, yojson]

type char_literal = {
    prefix : string
```

```
  ; unescaped : string
}
[@@deriving show, yojson]

type string_literal = {
    prefix : string
  ; unescaped : string
}
[@@deriving show, yojson]

(* TODO format_string_literal *)

type literal =
  | Number of number_literal
  | Char of char_literal
  | String of string_literal
[@@deriving show, yojson]

type keyword =
  | KwMod
  | KwUse
  | KwLet
  | KwMut
  | KwPub
  | KwTry
  | KwConst
  | KwImpl
  | KwFn
  | KwStruct
  | KwEnum
  | KwUnion
  | KwReturn
  | KwBreak
  | KwContinue
  | KwFor
  | KwWhile
  | KwIf
  | KwElse
  | KwMatch
  | KwDefer
  | KwUndefer
  | KwIn (* not always *)
  | KwTrait (* reserved for future *)
[@@deriving show, yojson]

val keyword_of_string : string -> keyword option

type token =
  | EOF
  | WhiteSpace of string (* ' \n\r\t', ... *)
  | Comment of comment
  | Literal of literal
  | Keyword of keyword
  | Identifier of string
  | SemiColon (* ; *)
  | Colon (* : *)
  | Comma (* , *)
  | Dot (* . *)
  | DotDot (* .. *)
  | OpenParen (* ( *)
  | CloseParen (* ) *)
  | OpenBrace (* { *)
  | CloseBrace (* } *)
  | OpenBracket (* [ *)
  | CloseBracket (* ] *)
  | At (* @ *)
  | QuestionMark (* ? *)
  | ExclamationPoint (* ! *)
  | Equal (* = *)
  | EqualEqual (* == *)
  | NotEqual (* != *)
  | LessThan (* < *)
  | GreaterThan (* > *)
  | LessThanOrEqual (* <= *)
```

```
  | GreaterThanOrEqual (* >= *)
  | LeftShift (* << *)
  | RightShift (* >> *)
  | Arrow (* => *)
  | Plus (* + *)
  | Minus (* - *)
  | Times (* * *)
  | Divide (* / *)
  | And (* & *)
  | Or (* | *)
  | AndAnd (* && *)
  | OrOr (* || *)
  | Caret (* ^ *)
  | Percent (* % *)
  | Tilde (* ~ *)
  | Pound (* # *)
  | DollarSign (* $ *)
[@@deriving show, yojson]
```

Code Listing - Table of Contents

src/util.ml

```
(* https://github.com/janestreet/base/blob/master/src/option.ml#L108 Like
   https://doc.rust-lang.org/std/option/enum.Option.html#method.unwrap_or_else *)
let value_or_thunk (o : 'a option) ~(default : unit -> 'a) : 'a =
  match o with
  | Some x -> x
  | None -> default ()
;;

let list_from_fn (f : unit -> 'a option) : 'a list =
  let rec next list =
    match f () with
    | None -> list
    | Some e -> next (e :: list)
  in
  next [] |> List.rev
;;
```

Code Listing - Table of Contents

src/util.mli

```
val value_or_thunk : 'a option -> default:(unit -> 'a) -> 'a

val list_from_fn : (unit -> 'a option) -> 'a list
```

Code Listing - Table of Contents