An Illustrated Guide to LLVM

or: an introduction to building simple and not-so-simple compilers with Rust and LLVM

Peter Marheine

2017-05-15

Rust Sydney

We're going to have something of a whirlwind tour through LLVM and why you should care, down into the guts of your operating system (assuming it's Linux) and doing FFI in Rust, building a compiler for an extremely simple programming language.

What?

What does LLVM do? Why should you care?

A library for building compilers

Formerly "Low level virtual machine"

- Compiler *backends*
  - Code generation, optimization
  - *Not* lexing, parsing

Supports both ahead-of-time and just-in-time

They dropped the initialism because it stopped making since as it gained features.

The library attempts to solve the problem of "I have an AST, how do I turn it into good code?". If you want to do frontend stuff, there are other good libraries:

- antlr
- yacc/bison
- nom
- combine

We'll discuss what a compiler backend does a bit later.

Industrial-grade

- Used in industry
  - Apple
  - Google
  - Others
- Mature
  - First release in 2003
  - ~5 million LOC

Basically all of Apple's compiler tooling is based on LLVM. They hired one of the creators a while back, though the project is technically owned by the LLVM foundation.

GCC is somehow 15 million LOC, though it's C rather than LLVM's C++.

Portable

Supports many systems:

- High-performance
  - x86
  - PowerPC
  - SPARC
- Embedded
  - ARM
  - PowerPC
  - SPARC
- GPUs
  - AMD GCN
  - Nvidia PTX
- Exotics
  - BPF
  - Hexagon
  - C

```
llvm-config --targets-built
```

- PTX: The intermediate language for CUDA.
- BPF: Berkeley packet filter. Supported by many Unix-like systems to operate over raw network packets. Linux supports loading BPF programs into the kernel.
- Hexagon: a Qualcomm DSP. Back in 2012 every SoC they sold contained more than one Hexagon core, so you probably carry around multiple Hexagons if your phone is built on a Qualcomm platform.
- C: literally a backend that emits C code, which you can theoretically feed into a compiler for a system that LLVM doesn't support.

Numerous frontends

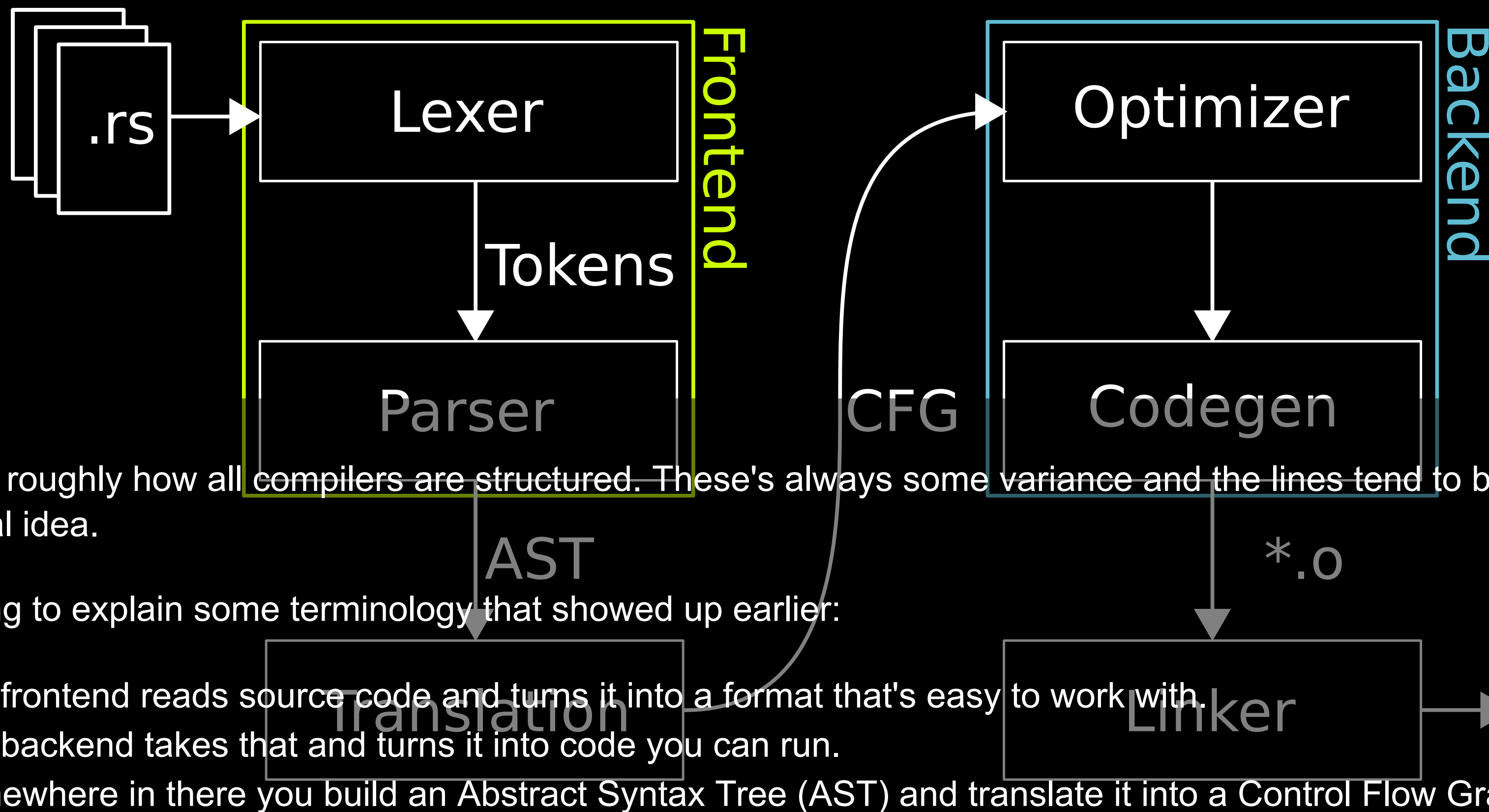
- Clang (C)
- GHC (Haskell)
- LDC (D)
- OpenJDK (Java)

.. and of course `rustc`.

OpenJDK doesn't usually use LLVM, but its no-assembly port ("Zero") can use LLVM as a native code generator ("Zero Shark").

How?

Compiler structure



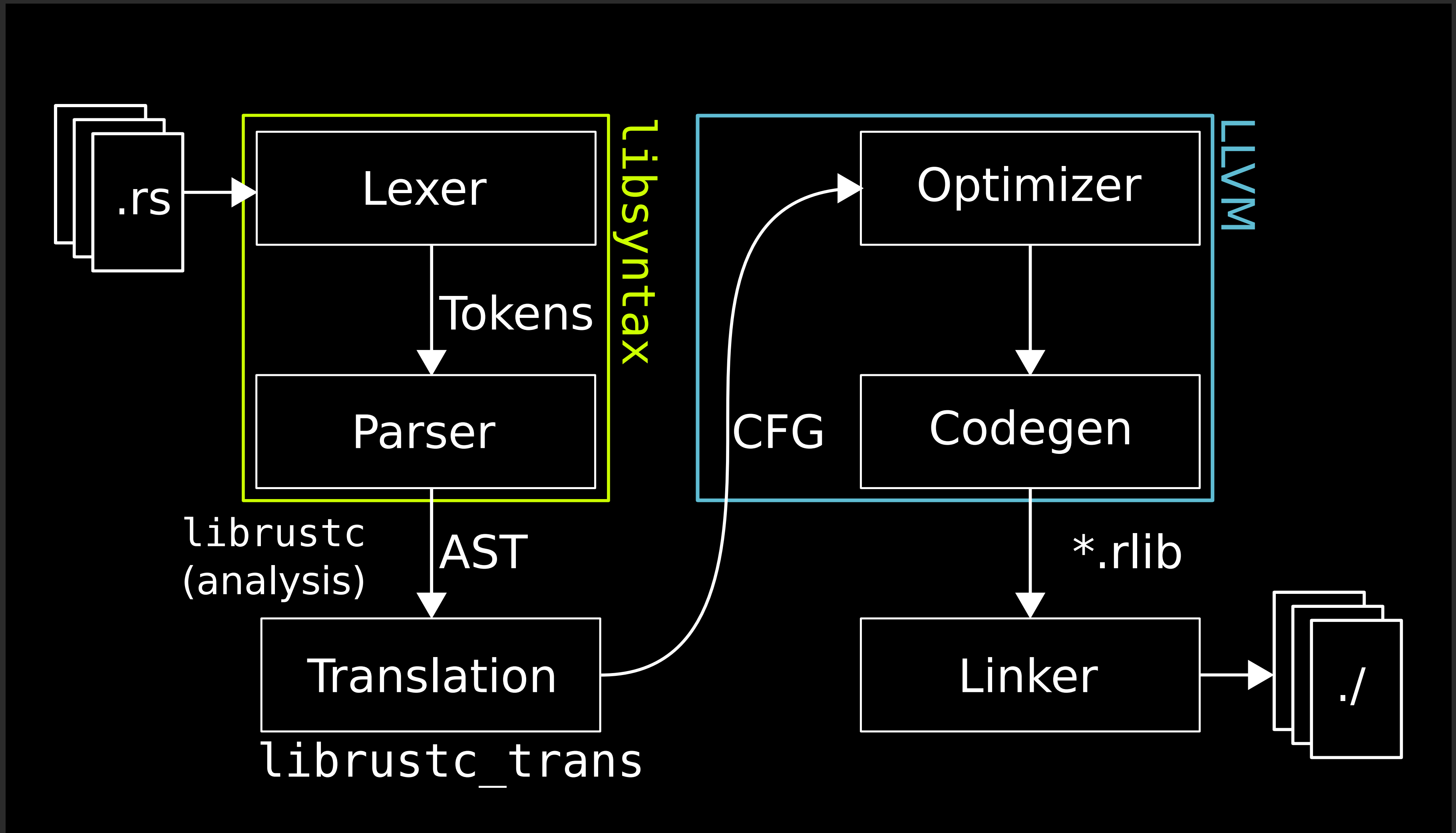
This is roughly how all compilers are structured. These's always some variance and the lines tend to blur, but it's the general idea.

Pausing to explain some terminology that showed up earlier:

- The frontend reads source code and turns it into a format that's easy to work with.
- The backend takes that and turns it into code you can run.
- Somewhere in there you build an Abstract Syntax Tree (AST) and translate it into a Control Flow Graph (CFG).

So we note that LLVM mostly provides the compiler *backend*, and there are numerous libraries that implement frontend-type things.

rustc



Roughly how these parts correspond to `rustc` subcrates.

Talking to LLVM

IR goes in..

```
fn add(x: i32, y: i32) -> i33 {
    (x as i33) + (y as i33)
}
```

```
target triple = "x86_64-unknown-linux"

define external i33 @add(i32 %x, i32 %y) nounwind readnone {
    %xe = sext i32 %x to i33
    %ye = sext i32 %y to i33
    %result = add i33 %x, %y
    ret i33 %result
}
```

The input to LLVM is IR, or "LLVM IR" (the LLVM Intermediate Representation). It's pretty easy to both read and write. If you're a program generating code you won't directly write IR (though you could if you really wanted), but it's easier to use the APIs to manage the code's structure rather than handle it as a blob of text.

This is a simple function that adds two signed 32-bit integers and returns a 33-bit result. We've explicitly marked the function as publicly visible (which will prevent some optimization) and attached attributes that tell the optimizer it will never throw an exception and does not read memory (so its result is only a function of its parameters). These are easy here so the optimizer could figure those out pretty trivally, but in other cases it might not be able to figure those out itself.

Use of a 33-bit integer here is just to illustrate that LLVM doesn't really care what our data types our. It understands integers and can work with arbitrary sizes.

The target triple definition tells the code generator things like what struct layouts look like and what calling convention to use. It will default if not specified, usually to your host system.

..code comes out

```
     .text
     .globl add
add:
     movsxd rcx, edi
     movsxd rax, esi
     add rax, rcx
     ret
```

Nothing very interesting here. If we generate x86 code for the previous function, it looks like this. Usually there's more debug information and whatnot, but that's not interesting to us. The i33s became 64-bit registers because the backend selected a register that was sufficiently large. If we wanted to do something like zero-extend an i33 to i64, the generated code might be rather interesting because the machine doesn't have an instruction to do that so the backend would have to synthesize something.

More complex

Testing the Collatz conjecture:

```
fn collatz(x: u32) -> bool {
    if x == 1 {
        return true;
    }

    let next = if x % 2 == 0 {
        x / 2
    } else {
        (3 * x) + 1
    }
    collatz(next)
}
```

`collatz(u32)` in IR

```
define "fastcc" i1 @collatz(i32 %x) {
    %finished = icmp eq i32 %x, 1
    br i1 %finished, label %Base, label %Continue
Base:
    ret i1 1

Continue:
    %next = alloca i32
    %odd = urem i32 %x, 2
    %odd1 = trunc i32 %odd to i1
    br i1 %odd1, label %Odd, label %Even
```

There are a few things in here that will look strange to experienced assembly programmers, but they're important to LLVM.

- Conditional branches always have two targets. This is because a branch (conditional or not) is classified as a terminator, always going at the end of a basic block. We'll discuss this more shortly.
- Value types are strict. We need to explicitly truncate an i32 to i1 to do a comparison (conditional br requires an i1 condition).
- We use the alloca instruction to get some memory with automatic storage duration. You do this basically any time you want a mutable local variable. Discussed further later.

Note as well that our type is still just `i32`. LLVM doesn't care about signedness of values, it's your instruction choice that matters (`urem` here for unsigned remainder) because LLVM guarantees two's complement representation of integers. Compare to C, which does not and it's the source of plenty of exciting UB.

`collatz(u32)` continued

```
Odd:
     %halved = udiv i32 %x, 2
     store i32 %halved, i32* %next
     br label %Recurse
Even:
     %larger = mul i32 %x, 3
     %larger1 = add i32 %larger, 1
     store i32 %larger1, i32* %next
     br label %Recurse

Recurse:
     %nextval = load i32, i32* %next
     %result = musttail call i1 @collatz(i32 %nextval)
     ret i1 %result
}
```

We have two branches of an if/else here which isn't too exciting, but note that we compute values and then store to the temporary we allocated on the stack.

The recursive call is marked as `tail`, which is a hint to the optimizer that we want to make a tail call which can be guaranteed not to consume lots of stack space. This is why we gave the function definition (previously) the `fastcc` calling convention, because only some calling conventions are compatible with tail calls. We can also go `musttail` to require that it generate a tail call, which would be an error if it's not possible, which in a frontend would require good analysis or language invariants.

Again we note that every basic block ends with a terminator, either a branch or return in this case.

Leaving overflow in add and mul defined, now `nsw` or `nuw` flags on those. Usually depends on the semantics of your language, since overflow if you enable either or both of those flags causes UB if the result becomes externally visible.

SSA

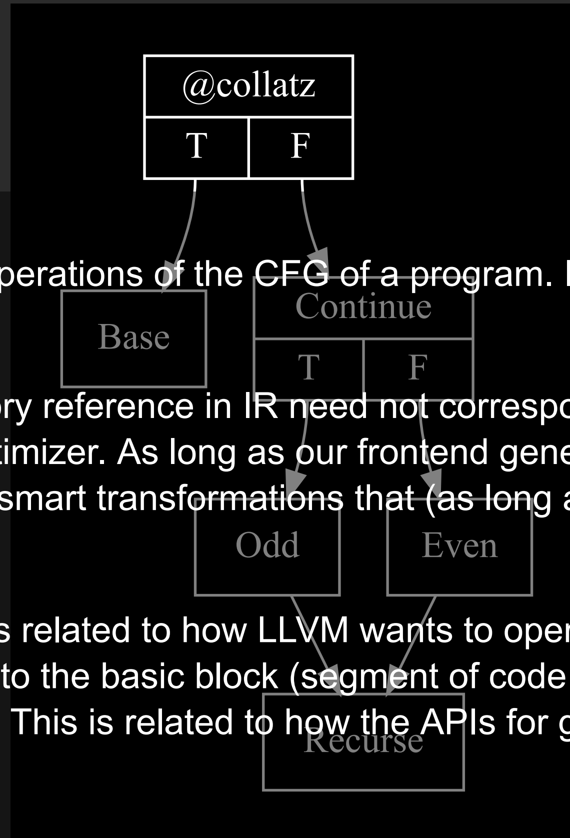
Why not this?

```
    %nextval = %halved
    br label %Recurse
Even:
    // ...
    %nextval = %larger1
Recurse:
    // use nextval
```

Single static assignment

- Every value has exactly one assignment
- Allows the system to work with a true CFG

Control flow graph



Native format for optimization.

LLVM uses SSA to simplify a number of operations of the CFG of a program. It's certainly possible to not use SSA, but it's much easier to do it this way.

The key thing to recognize is that a memory reference in IR need not correspond to an actual memory read or write, but we can leave details like that up to the optimizer. As long as our frontend generates reasonably sane code, LLVM's optimizer is capable of doing a lot of very smart transformations that (as long as your code doesn't break the rules) will make your code efficient.

Recall how a branch was a terminator? It's related to how LLVM wants to operate over a CFG. You could build a CFG with fallthrough from a conditional branch to the basic block (segment of code that has only one path through it) that follows lexically, but it would be less clear. This is related to how the APIs for generating code are designed as well, which we'll touch on later.

We'll be discussing not breaking the rules a bit later.

`opt` can generate the CFGs for provided IR: `opt -S -dot-cfg collatz.ll; dot -Tsvg cfg.collatz.dot > cfg.collatz.svg` for instance, generating this CFG which I've trimmed down some.

Speaking Merthese

Now that we've discussed some of how LLVM works and how we can make it do our bidding, let's actually do some of what this talk promised to do and start building a compiler.

Our target language here is a very easy esoteric language, "Merthese."

Token	Action
m	Print "merth"
e	Print "\n"
r	Print " "
t	Print random [a-z] [0, 13.4) times
h	Jump to after the next 'h'
_	Do nothing

Planning

Primitive operations

- Print string
  - `fn print(s: *mut u8, len: u8)`
- Random integer
  - `fn rand_inrange(max: u8) -> u8`

These are not supported by any CPU..

We have two primitive operations, basically. We need to generate random integers with custom range, and print characters to stdout.

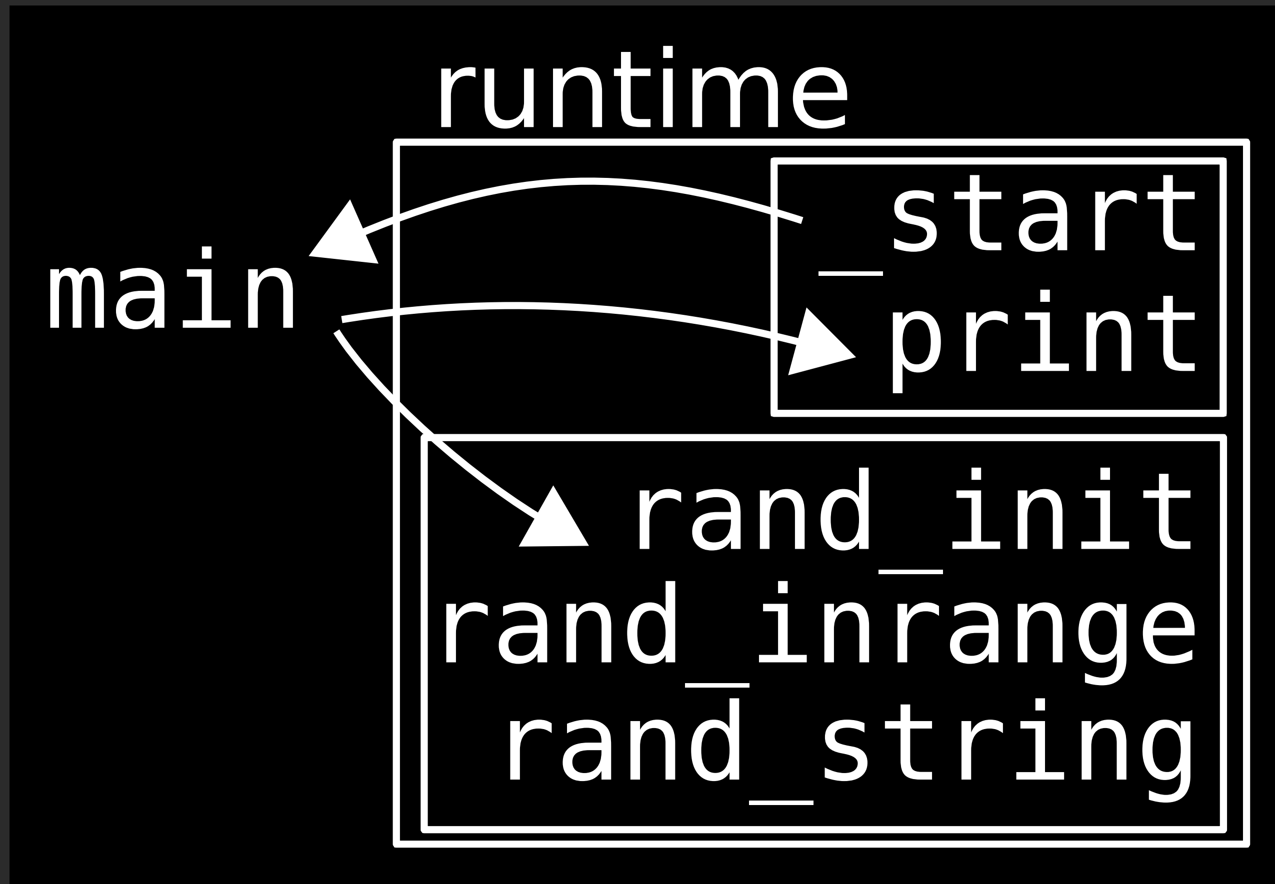
How this happens is entirely up to the OS, and LLVM doesn't know how to do it for us, so we're going to need a runtime library.

Runtime library

- Statically linked
  - Better optimization (inlining!)
  - Self-contained
- Written in IR
  - Because we can
  - More portable

A few arbitrary choices for our runtime. It means binaries we create are easier to copy and run elsewhere, and we'll try to avoid system-specific code as much as possible. The "more portable" note for writing things in IR is not really a very good reason, but that's okay.

Program structure



_start

- Initialize RNG
  - Open /dev/urandom
  - Read bytes
  - Close file
- Call main
- exit(0)

We need to do syscalls

Here's where things get kind of arcane.

You might be noting that LLVM is portable, so how do we define do non-portable things like making syscalls? The answer is we can generate system-specific IR. Designing your program such that the non-portable parts are nicely separated from the portable ones is still necessary- LLVM will not magically make the same code work on different machines without being at least somewhat aware of the differences, but it will help you do it.

```
extern fn syscall(nr: i64, ...) -> i64;
```

```
define private i64 @syscall2(i64 %nr, i64 %p1, i64 %p)
         inlinehint {
     %1 = call i64 asm sideeffect "syscall",
               "={rax},{rax},{rdi},{rsi}"
               (i64 %nr, i64 %p1, i64 %p2)
     ret i64 %1
}
```

- RAX: nr
- RDI: p1
- RSI: p2
- Result: RAX

First, the general signature for the kernel's syscall interface. You pass the syscall number you want, and up to 6 parameters.

The actual implementation (for x86_64 linux here) is load the specified registers, then execute `syscall` to trap into the OS.

The inline assembly syntax is rather arcane. We marked the invocation as `sideeffect` to prevent the compiler from optimizing out a syscall if we don't use the output, because otherwise it might notice that we don't use the output and helpfully omit the syscall, making the program run faster.

extern fn open(path: *mut u8, flags: c_int) -> c_int

```
@__NR_open = private constant i64 2

define private i32 @open(i8 *%path0, i32 %flags0) {
     %nr = load i64, i64* @__NR_open
     %path = ptrtoint i8* %path0 to i64
     %flags = zext i32 %flags0 to i64
     %out0 = call i64 @syscall2(i64 %nr, i64 %path, i64 %flags)
     %out = trunc i64 %out0 to i32
     ret i32 %out
}
```

Again, we need to know a few things about how the C API maps to the linux syscall interface. You get them from reading the source, mostly.

Define the syscall number we want (2) as a module-level constant. There's no particular reason we couldn't just put a 2 in @open, but this way is clearer. This is the first time we've seen a global. They work basically like they do in C.

More system-specific things here. We happen to know that a C int is 32 bits, and the cast from a pointer to i64 is system-specific. Most of this function is just knowing the correct syscall number, and casting your parameters.

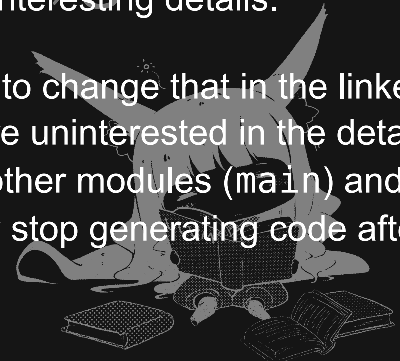
Turns out syscalls are boring.
Back to `_start`.

```
declare void @main()
define void @exit(i32 %code) noreturn {}

define void @_start() noreturn {
    // initialize RNG
    call void @main()
    call void @exit(i32 0)
    unreachable
}
```

Feels like C: declare external functions, glue them together.

We get the general idea. Syscalls are important, but they differ from each other with uninteresting details.

`_start` is the magic name of what Linux will run when our program starts. It's possible to change that in the linker, but we have no reason to. Putting aside RNG setup which we've touched on already and are uninterested in the details of, this function is very simple. New things we're seeing here are declaring functions from other modules (`main`) and `unreachable`, which tells LLVM that instruction will never be executed so it can simply stop generating code after the call to `exit`.



Writing some Rust

We've been writing IR long enough- let's write some Rust now.

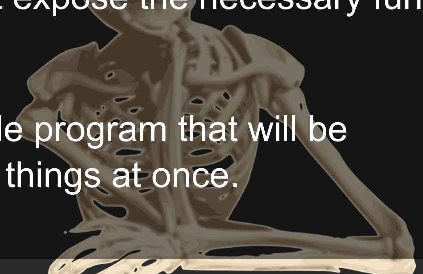
Skeleton

```
extern crate llvm_sys as llvm;

fn main() {
    unsafe {
        LLVM_InitializeNativeTarget();
        LLVM_InitializeNativeAsmPrinter();
        LLVM_InitializeNativeAsmParser();

        let ctxt = LLVMContextCreate();
        /* Use ctxt */
        LLVMContextDispose(ctxt);
    }
}
```

A skeleton for a program that uses LLVM, targeting the host machine.

`llvm-sys` is bindings to the LLVM C API. It also has a C++ API (because it's implemented in C++) and some other languages (Go, OCaml). For some tasks it's necessary to write C++ because they don't expose the necessary functions in the C API.

A context is essentially an instance of LLVM. You can create multiple contexts in a single program that will be independent, which would be important if you wanted to independently compile several things at once.



Create `main`

```
declare void @main()
```

```
let main_name = b"main\0".as_ptr() as *const _;
let main_module =
        LLVMModuleCreateWithNameInContext(main_name, ctxt);

let ty_void = LLVMVoidType();
let ty_fn_main = LLVMFunctionType(ty_void,
        /* ParamTypes */ ptr::null_mut(),
        /* ParamCount */ 0,
        /* IsVarArg   */ 0);
let main_function = LLVMAddFunction(main_module,
        main_name,
        ty_fn_main);
```

Given the runtime we already described (TODO add a copy of the earlier figure in here?), our compiler needs to generate a `main` function. When linked with the runtime we described previously, this becomes a complete program.

First we create a module to contain our function. This works much like C, where a module is the unit of compilation and you link multiple modules together. A module is the equivalent of a single .c or .ll file. You need a module to put code in.

Then we need to create a function called main. In order to create it, we need to tell LLVM what its type is. It returns `void` and takes no parameters, so we get a reference to the opaque LLVM type that represents `void` and construct a function returning that and taking no parameters. The parameter types argument to the function to construct a function type is a C array, so it's actually a pointer. In this case we say there are zero elements so we can provide a null pointer.

Finally, we add the function definition to our module with the name 'main'.

Emitting IR

```
fn LLVMPrintModuleToFile(M: LLVMModuleRef,
                         Filename: *const c_char,
                         ErrorMessage: *mut *mut c_char)
                         -> LLVMBool

fn LLVMPrintModuleToString(M: LLVMModuleRef) -> *mut c_char
```

Could we use io::Write instead?

At this point we should have something approximating usable IR. How can we sanity check it?

The easiest answer is dumping textual IR and reading it. LLVM gives us two functions that could do this, but they're both a little icky. The first can only write to a named file, while the second returns a C string which may have surprising semantics.

If we examine the implementation of PrintModuleToString we see that gives you ownership of a `strdup`'d string so we could probably use `libc::free` on it, but it's still not a very strong guarantee of correctness. Plus wasting memory on a string that we could stream to output is silly.

Dropping to C++

```
extern "C" typedef int (*cb_t)(const void *, size_t, void *);

class raw_callback_ostream : public llvm::raw_ostream {
    cb_t callback;
    void *callback_data;

public:
    raw_callback_ostream(cb_t cb, void *cb_data) { /* ... */ }

private:
    void write_impl(const char *p, size_t sz) override {
        callback(p, sz, callback_data);
        offset += sz;
    }
};
```

This is a point where the C API doesn't do what we want, so we'll write a little C++ and expose it to our Rust code. Knowing what to extend kind of requires reading the LLVM source or otherwise knowing what it does.

Here the interface involves extending `raw_ostream` and implementing `write_impl`. We'll just store a callback and support passing a pointer to it. By extending `raw_ostream` we can bitshift a module to it and get IR.

A C function to expose it:

```
extern "C" void PrintModuleIR(LLVMModuleRef M,
                              cb_t cb,
                              void *cb_data) {
    raw_callback_ostream out(cb, cb_data);
    out << *llvm::unwrap(M);
}
```

So we implement a C-API function that takes a callback and uses it to emit IR.

Rust adapter from `Write` to callbacks

```
extern "C" fn module_ir_printer<W>(src: *const u8,
                                    size: libc::size_t,
                                    state: *mut libc::c_void)
                                    -> libc::c_int
        where W: std::io::Write {
    let (src, out) = unsafe {
        (slice::from_raw_parts(src, size),
         &mut *(state as *mut W))
    };
    let _res = out.write_all(src);
    0
}
```

Safe wrapper

```
fn dump_module_ir<W>(module: LLVMModuleRef, mut out: W)
        where W: std::io::Write {
    unsafe {
        PrintModuleIR(module,
                      module_ir_printer::<W>,
                      &mut out as *mut W as *mut libc::c_void);
    }
}
```

Then in Rust we can define a function that's generic over `Writer`s so we can stream IR for a module to anywhere using a safe interface.

Emit some IR

```
let main_function = LLVMAddFunction(main_module,
                                    main_name,
                                    ty_fn_name);
dump_module_ir(main_module, std::io::stderr());
```

```
; ModuleID = 'main'
source_filename = "main"

declare void @main()
```

Now recalling that we were generating a module and wanted to inspect its IR, we can do that by dumping IR to stderr. With our 'main' module that contained a 'main' function, we get this out.

It's a valid module, hooray. It's just a function definition though: we need to put some code into it yet.

Using Builders

```
let bb_main =
    LLVMAppendBasicBlockInContext(ctxt,
                                  main_function,
                                  b"\0".as_ptr() as *const _);
let b = LLVMCreateBuilderInContext(ctxt);

LLVMPositionBuilderAtEnd(b, bb_main);
LLVMBuildRetVoid(b);
LLVMDisposeBuilder(b);
```

```
define void @main() {
    ret void
}
```

In order to put code in a function, you must first create a basic block (remember how the CFG is the unit of code that LLVM likes to operate over?). Then you can use a `Builder` to make instructions, and position it at arbitrary positions in basic blocks. It's usually easiest to be strictly append-only.

Manual testing

At this point we've written a runtime library in IR, and written a Rust program that can generate a minimal `main` function. Let's try to manually put them together into a program we can run, then apply those lessons to writing code to do it.

The pieces so far

linux-x86_64.ll

`_start` and `print` functions for x86_64 Linux

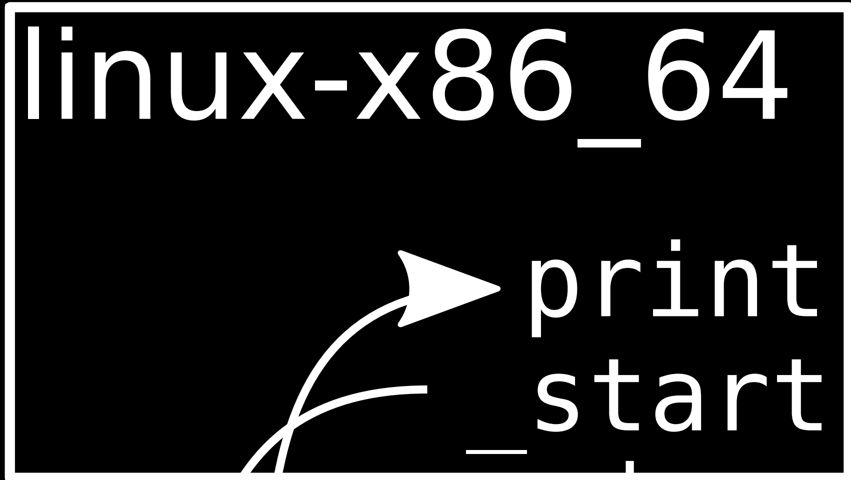
random.ll

`rand_inrange` function and RNG impl

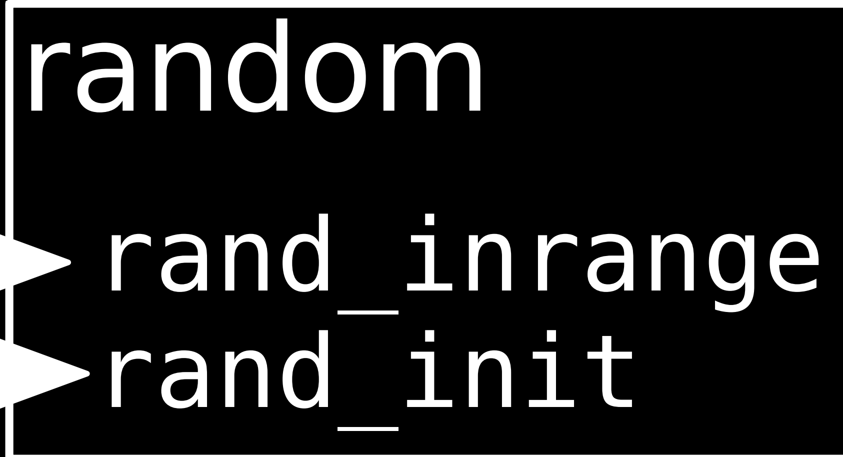
main.ll

Generated `main` function

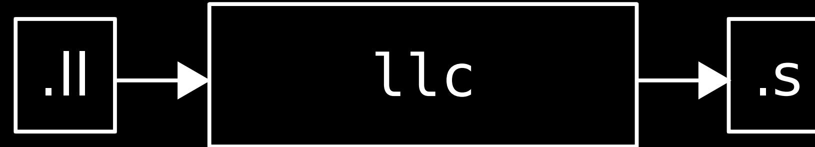
Recall that our runtime implements a few primitive operations and provides the overall program entry point. In this implementation (much of which I've elided here), they're split into three files of IR.

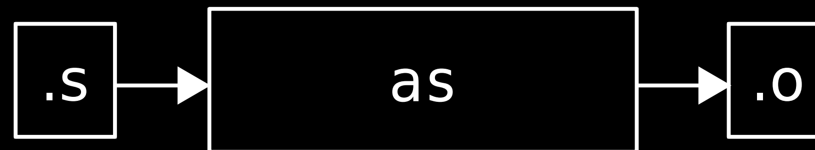

linux-x86_64
print
_start

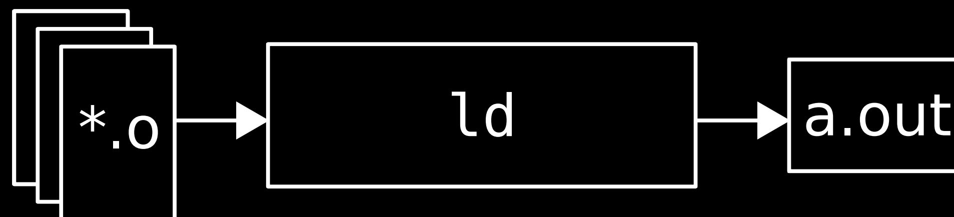
main
main


random
rand_inrange
rand_init

Code generation







Most of the transformations LLVM implements have command line programs that can do that transformation over IR, which is great for testing and debugging. To turn IR into assembly, we use `llc`.

`llc` emits assembly, which we then need to assemble and link. LLVM prefers to leave these tasks up to your platform's tools, so in this case (Linux) it's GNU binutils.

```
llc linux-x86_64.ll
as -o linux-x86_64.o linux-x86_64.s

llc random.ll
as -o random.o random.s

llc main.ll
as -o main.o main.s

ld main.o linux-x86_64.o random.o
./a.out
```

Optimization

The generated code is inefficient!

- Less so if optimization is turned on
  - `llc -O2`
- Better: LTO
  - `llvm-link *.ll -o a.bc`
  - `llc -O2 a.bc`, etc

We can do better (but not right now)

If we inspect the assembly generated by LLC, it's not very good. Turns out it doesn't optimize at all by default, but it can't inline across modules either because the code for them is generated independently! It's obvious to us that the function to initialize the RNG for instance is only ever called from `_start`, so it could be trivially inlined.

We can do link-time optimization too, with `llvm-link` that takes a collection of IR and makes them all one module, preserving symbols (functions or globals) with external visibility and renaming private ones as necessary to avoid collisions.

`llvm-link` emits a `.bc` file, which we haven't encountered yet. It's LLVM bitcode, which is basically just IR in a format that's easier for the machine to read but infeasible for humans. The `llvm-dis` utility can disassemble bitcode into textual IR if you want to examine a bitcode file.

This still isn't perfect, because symbols with external visibility must remain visible, so even if the optimizer chooses to inline uses of a function for instance, it still needs to provide a standalone copy for a hypothetical external caller. We can't make all our functions private because then we couldn't link the modules together correctly, so for now this is the best that's possible. We'll consider it again later.

More Rust

Using the runtime

```
let ty_void = LLVMVoidTypeInContext(ctxt);
let ty_i8 = LLVMIntTypeInContext(ctxt, 8);
let ty_i8p = LLVMPointerType(ty_i8, 0);

let param_types = [ty_i8p, ty_i8];
let ty_rt_print = LLVMFunctionType(
        ty_void,
        param_types.as_ptr() as *mut _,
        param_types.len(),
        0);

let rt_print = LLVMAddFunction(
        llmod,
        b"print\0".as_ptr() as *const _,
        ty_rt_print);
```

```
declare void print(i8*, i8)
```

Other functions left as an exercise for the reader.

So we wrote a runtime and now need to generate code that refers to it. As discussed earlier, we need to generated function declarations, that at link time are resolved as references to the function.

As we saw earlier, simply adding a function to a module will make it a declaration, and it magically becomes a definition if you add code to it. So given known signatures for our runtime support functions, we just need to create matching functions.

As before, we create types and string them together to make a function describing the one we know exists. We see a few new functions here, but they're mostly obvious. `LLVMIntType` takes a parameter for the number of bits in the integer type, and `LLVMPointerType` takes the type you want a pointer to and an address space number. LLVM pointers can exist in multiple address spaces, which is useful for some more unusual machines. The semantics are target-defined, but the default (and correct choice unless you know you need something different) is address space 0.

Being wrong

`LLVMVoidType` ↔ `LLVMVoidTypeInContext`

Implicit global context ↔ explicit

- Mixing contexts is *wrong* and can cause miscompilation
- Tools to help prevent bugs?

If you're particularly alert, you'll notice we used `LLVMVoidTypeInContext` here, but `LLVMVoidType` in an earlier example. What's up with that?

We're mixing the global context with an explicit one. You never want to do this. Anecdote: I was updating the `merthc` we've been writing for LLVM 4.0 and `llvm-sys 40` (it was originally written for LLVM 3.6) and it was generating wrong code. I eventually discovered that I was doing exactly this, which made it omit function definitions at link time that matched declarations I needed, causing it to emit things that didn't link correctly.

So if subtle bugs like this can cause miscompilation, what tools are available to catch them?

Bug swatting

- `LLVMVerifyModule`
  - Print message or abort
- Debug assertions
  - `LLVM_ENABLE_ASSERTIONS`
  - Usually not enabled in binary releases
- Manual inspection
  - As done earlier

Filling in `main`

Parsing code

```
let code: Iterator<char>;

while let Some(c) = code.next() {
    match c {
        'm' => { /* Print "merth" */ },
        'e' => { /* Print newline */ },
        'r' => { /* Print space */ },
        't' => { /* Random string */ },
        'h' => { loop { match code.next() {
            Some('h') | None => break,
            _ => continue,
        } } },
        _ => { /* Do nothing */ },
    }
}
```

The core loop for parsing the input code to our compiler. Nothing very special here. If we wanted we could do something like generate code between 'h's and jump over it, but this is much easier.

m is for merth

m → print("merth", 5)

```
let b: Builder;

let v_5i8 = LLVMConstInt(ty_i8, 5, 0);
let v_merth = LLVMBuildGlobalStringPtr(
        b,
        b"merth\0".as_ptr() as *const _,
        b"MERTH\0".as_ptr() as *const _);

LLVMBuildCall(b,
              rt_print,
              [v_merth, v_5i8].as_ptr() as *mut _,
              2,
              b"\0".as_ptr() as *const _);
```

As seen before, we need to build calls to runtime functions for most operations. Here we need to call `print` with two constant parameters, so we learn how to generate constant values. They work just like non-constant values for the most part, but are immutable.

Since strings aren't primitive types, we need to make a global one and work with pointers to it. The type of that is `i8*`.

e is for newline

e → print("\n", 1)

```
let v_newline = ptr_to_const(
        llmod,
        ty_i8,
        LLVMConstInt(ty_i8, 10, 0),
        b"NEWLINE\0");

LLVMBuildCall(b,
              rt_print,
              [v_newline, v_1i8].as_ptr() as *mut _,
              2,
              b"\0".as_ptr() as *const _);
```

Create a const '\n' and call `print` with a pointer to it. Easy.

ptr_to_const

```
fn ptr_to_const(llmod: LLVMModuleRef,
                ty: LLVMTypeRef,
                value: LLVMValueRef,
                name: &[u8])
                -> LLVMValueRef {

    let g = LLVMAddGlobal(llmod, ty, name.as_ptr() as *const _);
    LLVMSetInitializer(g, value);
    LLVMSetGlobalConstant(g, 1 /* true */);
    LLVMConstInBoundsGEP(g, [v_const_0i8].as_ptr() as *mut _, 0)
}
```

- Save a byte vs `GlobalStringPtr`
- GEP: GetElementPointer
  - Inbounds: must not be out of bounds

We can save a total of two bytes in our binary by creating the newline and space parameters to `print` as plain old `i8` values instead of strings so we don't get implicit null terminators.

The new thing we get here is a 'GEP', which gives you a value which is a pointer to something inside another value, usually an array or struct but used for any values.

r is for space

This space intentionally left blank.

'r' works just like 'e' but with a different value. We don't need to look at it.

t is for randomness

- [0, 13.4) times [a-z]
- Runtime: `rand_inrange` + `rand_string`

```
let len = rand_inrange(13.4 as i8 + 1);
rand_string(len);
```

Nothing too special. Note that `rand_inrange` returns a value [0, n) so if we want a string that may be up to 13.4 characters long we need to add one to make the range inclusive.

For instance, if we wanted to generate a value in range [0, 1) we'd need to pass 1 instead of 0 because otherwise it will underflow and give us [0, 255].

```
let v_len = LLVMBuildCall(
        b,
        rt_rand_inrange,
        [LLVMConstAdd(
             LLVMConstFPToUI(
                 LLVMConstReal(LLVMFloatTypeInContext(ctxt),
                               13.4),
                 ty_i8),
             v_1i8)
        ].as_ptr() as *mut _,
        1,
        b"\0".as_ptr() as *const _);

LLVMBuildCall(rt_rand_string, [v_len]);
```

FP is slow, do it all as const for speed.

We've started paraphrasing some of this code, like this `BuildCall` instance. We've seen how it works.

We've introduced non-integer values. LLVM understands a few floating-point types, mostly the usual `float` and `double` types, `f32` and `f64` as used in Rust. If you're being unusual it also supports `half` (16-bit), `fp128`, `fp128`, `x86_fp80` and `ppc_fp128`.

You can also do some arithmetic with const values. Nice.

Codegen & optimization

Load runtime

```
static RT_SOURCES: &'static [&'static [u8]] = &[
    include_bytes!("../runtime/random.ll")
];

let mbuf = LLVMCreateMemoryBufferWithMemoryRange(
        code.as_ptr() as *const _,
        code.len() - 1 as libc::size_t,
        b"\0".as_ptr() as *const _,
        /* RequiresNullTerminator */ 1);

let module: LLVMModuleRef;
let err_msg: *mut i8;
LLVMParseIRInContext(ctxt, mbuf, &mut module, &mut err_msg);
/* Error checking here */
```

To make everything self-contained, choose to embed the runtime sources in our binary. Then create a memory buffer that LLVM knows how to manage, and ask it to parse the IR.

Platform runtime

```
static RT_TARGET_SOURCE: phf::Map<
    &'static str,
    &'static [u8]
> = ...;
```

Use `phf` for excessively efficient lookup tables (built at compile-time)

```
let target = LLVMGetDefaultTargetTriple();
RT_TARGET_SOURCES.get(target);
```

Nothing real special here, but `phf` is worth commenting on if you're not familiar with it. I've written this to generate the runtime map at build time.

We use the LLVM default target to determine what machine we're building for, which will usually be the same as the one you're running on. We could support cross-compilation if we wanted to be fancy.

Linking

```
let main_module: LLVMModuleRef;

for module in rt_modules {
    // Destroys module
    LLVMLinkModules2(main_module, module);
}
```

Easy as `llvm-link`

Codegen

```
let triple = "x86_64-unknown-linux";
let target: LLVMTargetRef;
LLVMGetTargetFromTriple(triple, &mut target, ptr::null_mut());

let tm = LLVMCreateTargetMachine(target,
                                 triple,
                                 "", "",
                                 LLVMCodeGenLevelAggressive,
                                 LLVMRelocDefault,
                                 LLVMCodeModelDefault);
```

Get a target, make a target machine

```
let mbuf: LLVMMemoryBufferRef;
LLVMTargetMachineEmitToMemoryBuffer(tm, llmod, ty,
                                    ptr::null_mut(),
                                    &mut mbuf);

let mut w: io::Write;
let code: &[u8] = slice::from_raw_parts(
        LLVMGetBufferStart(mbuf),
        LLVMGetBufferSize(mbuf)
);
w.write_all(code);

LLVMDisposeMemoryBuffer(mbuf);
```

Emit code to memory,
write to a file.

Not pictured: linker invocation
(as subprocess)

We could probably do something similar to what we did earlier for emitting IR directly to an `io::Write` but with code, but that was a lot of effort.

Optimization

```
let fpm = LLVMCreateFunctionPassManagerForModule(llmod);
let mpm = LLVMCreatePassManager();

let pmb = LLVMPassManagerBuilderCreate();
LLVMPassManagerBuilderSetOptLevel(pmb, 2);
LLVMPassManagerBuilderUseInlinerWithThreshold(pmb, 225);
LLVMPassManagerBuilderPopulateModulePassManager(pmb, mpm);
LLVMPassManagerBuilderPopulateFunctionPassManager(pmb, fpm);
LLVMPassManagerBuilderDispose(pmb);
```

Pass manager wrangles optimizer passes

Including: DCE, GVN, constant propagation, LICM, loop unrolling, inlining..

Because the optimization process is rather involved, LLVM gives us a pass manager that wrangles passes. In a quick examination of the documentation, I see probably more than 100 different passes.

There are a few handles to control optimization. Here we tell it to do medium inlining (equivalent to -O2 with Clang) and use a magic (arbitrary) inlining threshold.

There are two optimization flavors, per-function and per-module. They're run separately.

Running passes

```
LLVMInitializeFunctionPassManager(fpm);

let mut func = LLVMGetFirstFunction(llmod);
while func != ptr::null_mut() {
    LLVMRunFunctionPassManager(fpm, func);
    func = LLVMGetNextFunction(func);
}

LLVMFinalizeFunctionPassManager(fpm);
```

Iterate over functions, optimizing each

```
LLVMRunPassManager(mpm, llmod);
```

Running pass managers: not too weird.

LTO

- Link modules together, then optimize
  - ..retaining external symbols
- Internalize symbols first

```
let mut func = LLVMGetFirstFunction(llmod);
while func != ptr::null_mut() {
    let func_name = CStr::from_ptr(LLVMGetValueName(func));
    if func_name != "_start" {
        LLVMSetLinkage(func, LLVMLinkage::LLVMPrivateLinkage);
    }

    func = LLVMGetNextFunction(func);
}
```

We'll just iterate over functions, setting them to private linkage so the compiler will not be required to retain things other than `_start`. As long as you're linking everything your program needs, this is fine.

This should come before codegen and linking, but is otherwise the same.

References & advertising

LLVM

- llvm.org

`llvm-sys`

- Rust → LLVM C library bindings
- crates.io:llvm-sys

Reference `merthc`

- bitbucket.org/tari/merthc

Me

Incidental foxes

- きつねさんでもわかるＬＬＶＭ

Ferris the Rustacean

- Karen Rustad Tölva (public domain)

Ask me questions now.