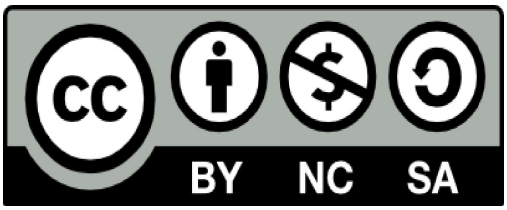


This book assumes some basic knowledge of Rust language. Please take a look at the official Rust book.

The accompanying codes and materials for this book are available in GitHub. To follow along, make sure you have

- Rust toolchain installed
- Cloned the repository

```
git clone https://github.com/ehsanmok/create-your-own-lang-with-rust
cd create-your-own-lang-with-rust
```

- LLVM installed to run and test locally `cargo test --tests`
  - Easiest option is LLVM v14.0 (Debian/Ubuntu or macOS)
  - Otherwise, in `Cargo.toml` you'd need to change the `inkwell = { ..., branch = "master", features = ["your-llvm-version"] }` with LLVM version on your system (output of `llvm-config --version`)

Motivations and Goals

This book arises from my frustration of not finding modern, clear and concise teaching materials that are readily accessible to beginners like me who wants to learn a bit on how to create their own programming language.

The following are my guidelines

> "If you don't know how *compilers* work, then you don't know how computers work" [1]

> "If you can't explain something in simple terms, you don't understand it" [2]

Pedagogically, one of the most effective methods of teaching is co-creating interactively. Introducing the core aspects around the *simplest example* (here, our calculator language) helps a lot to build knowledge and confidence. For that, we will use mature technologies instead of spending tone of time on partially reinventing-the-wheel and bore the reader.

Here is the outline of the contents

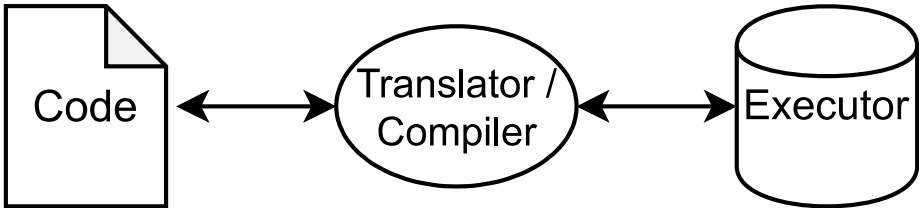
- Crash Course on Computing which we briefly set up the definitions and foundations
- We create our first programming language `Calc` that supports simple integer addition and subtraction. The simplicity allows us to touch a lot of important topics. We will use PEG to define our grammar, pest to generate our `CalcParser` and explain what AST is and interpreting the AST means. Next, we will introduce JIT compilation and use inkwell to JIT compile our `Calc` language from its AST. To show an alternative compilation approach, we will create a Virtual Machine and a Runtime environment and discuss its features. Finally, we will write a simple REPL for our `Calc` language and test out different execution paths.
- TODO: We will create `Firstlang`, a statically typed language, by gradually working our way up from our `Calc`
- TODO: Object system and minimal object oriented programming support
- TENTATIVE: Create a mini standard library
- TODO: Resources

Donation

If you have found this book useful, please consider donating to any of the organizations below

- Child Foundation
- Black Lives Matter
- Food Bank of Canada

Here is a bird's-eye view of a computer program execution



All these three components are intertwined together and learning their connections is crucial in understanding what makes *Computing* possible. Informally, a *language* is a structured text with syntax and semantics. A *Source Code* written in a programming language needs a translator/compiler of *some sort*, to translate it to *another* language/format. Then an executor of *some sort*, to execute/run the translated commands with the goal of matching the syntax (and semantics) to *some form* of output.

Elements of Computing

Instructions and the Machine Language

If you want to create a "computer" from scratch, you need to start by defining an *abstract model* for your computer. This abstract model is also referred to as **Instruction Set Architecture (ISA)** (instruction set or simply *instructions*). A CPU is an *implementation* of such ISA. A standard ISA defines its basic elements such as *data types*, *register* values, various hardware supports, I/O etc. and they all make up the *lowest-level language* of computing which is the **Machine Language Instructions.**

Instructions are comprised of *instruction code* (aka *operation code*, in short **opcode** or p-code) which are directly executed by CPU. An opcode can either have operand(s) or no operand. For example, in an 8-bits machine where instructions are 8-bits an opcode *load* might be defined by the 4-bits **0011** following by the second 4-bits as operand with **0101** that makes up the instruction **00110101** in the Machine Language while the opcode for *incrementing by 1* of the previously loaded value could be defined by **1000** with no operand.

Since *opcodes are like atoms of computing*, they are presented in an opcode table. An example of that is Intel x86 opcode table.

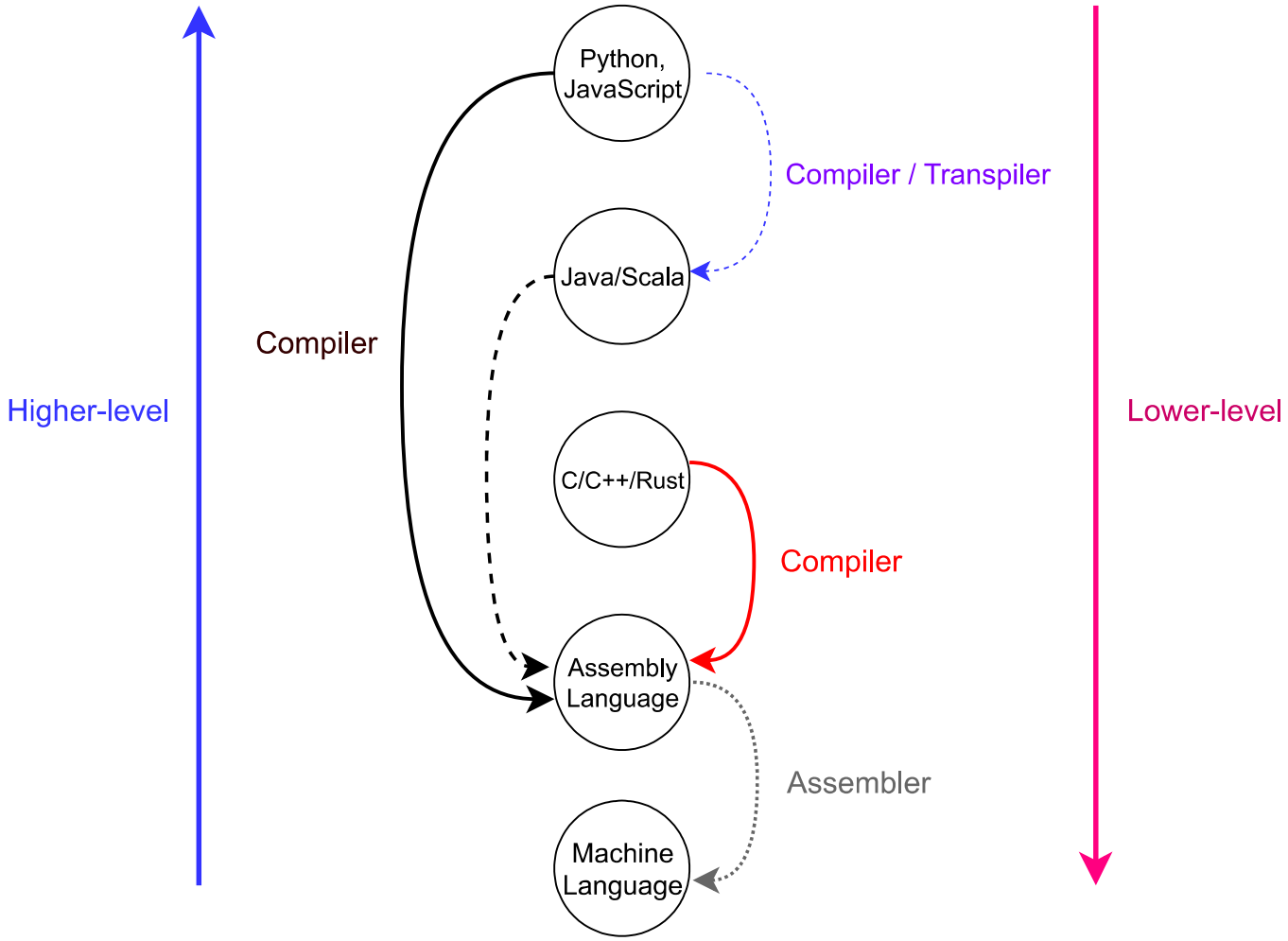
Assembly Language

Since it's hard to remember the opcodes by their bit-patterns, we can assign *abstract* symbols to opcodes matching their operations by name. This way, we can create Assembly language from the Machine Language. In the previous Machine Language example above,

00110101 (means load the binary **0101**), we can define the symbol **LOAD** referring to **0011** as a higher level abstraction so that **00110101** can be written as **LOAD 0101**.

The utility program that translates the Assembly language to Machine Language is called **Assembler**.

Compiler



Compiler is any program that translates (maps, encodes) a language A to language B. Each compiler has two major components

- **Frontend:** deals with mapping the source code string to a structured format called **Abstract Syntax Tree (AST)**
- **Backend (code generator):** translates the AST into the Bytecode / IR or Assembly

Most often, when we talk about compiler, we mean **Ahead-Of-Time (AOT)** compiler where the translation happens *before* execution. Another form of translation is **Just-In-Time (JIT)** compilation where translation happens right at the time of the execution.

From the diagram above, to distinguish between a program that translates for example, Python to Assembly vs. Python to Java, the former is called compiler and the latter

transpiler.

Relativity of low-level, high-level

Assembly is a *high-level* language compared to the Machine Language but is considered *low-level* when viewing it from C/C++/Rust. High-level and low-level are relative terms conveying the amount of *abstractions* involved.

Virtual Machine (VM)

Instructions are hardware and vendor specific. That is, an Intel CPU instructions are different from AMD CPU. A **VM** abstracts away details of the underlying hardware or operating system so that programs translated/compiled into the VM language becomes platform agnostic. A famous example is the **Java Virtual Machine (JVM)** which translates/compiles Java programs to JVM language aka Java **Bytecode**. Therefore, if you have a valid Java Bytecode and *Java Runtime Environment (JRE)* in your system, you can execute the Bytecode, regardless on what platform it was compiled on.

Bytecode

Another technique to translate a source code to Machine Code, is emulating the Instruction Set with a new (human friendly) encoding (perhaps easier than assembly). Bytecode is such an *intermediate language/representation* which is lower-level than the actual programming language that has been translated from and higher-level than Assembly language.

Stack Machine

Stack Machine is a simple model for a computing machine with two main components

- a memory (stack) array keeping the Bytecode instructions that supports `push`ing and `pop`ing instructions
- an instruction pointer (IP) and stack pointer (SP) guiding which instruction was executed and what is next.

Intermediate Representation (IR)

Any representation that's between source code and (usually) Assembly language is considered an intermediate representation. Mainstream languages usually have more than one such representations and going from one IR to another IR is called *lowering*.

Code Generation

Code generation for a compiler is when the compiler *converts an IR to some Machine Code*. But it has a wider semantic too for example, when using Rust declarative macro via `macro_rules!` to automate some repetitive implementations, you're essentially generating codes (as well as expanding the syntax).

Conclusion

In conclusion, we want to settle one of the most frequently asked questions

Is Python (or a language X) Compiled or Interpreted?

This is in fact the WRONG question to ask!

Being AOT compiled, JIT compiled or interpreted is **implementation-dependent**. For example, the standard Python *implementation* is **CPython** which compiles a Python source code (in CPython VM) to CPython Bytecode (contents of `.pyc`) and **interprets** the Bytecode. However, another implementation of Python is **PyPy** which (more or less) compiles a Python source code (in PyPy VM) to PyPy Bytecode and **JIT** compiles the PyPy Bytecode to the Machine Code (and is usually faster than CPython interpreter).

Calculator

Our first programming language is a simple calculator supporting addition and subtraction. This is perhaps the *simplest language* that helps us introducing the major topics from grammar to compilation and virtual machine.

If you haven't cloned the GitHub repo already, please do and navigate to the `calculator` subdirectory.

To start, we have `1 + 1;` in examples/simple.calc where you can compile with

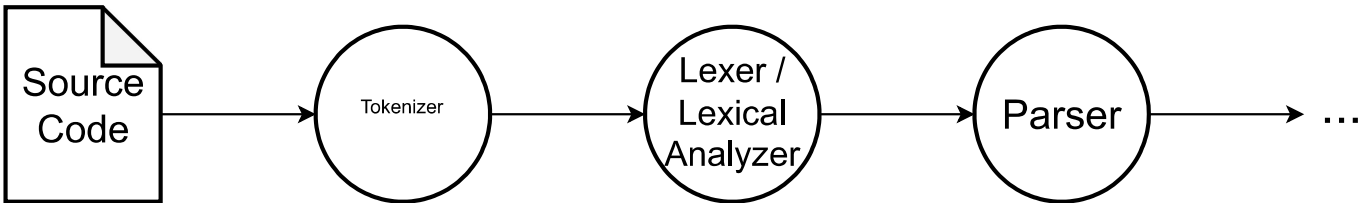
```
cargo build --bin main // create a simple executable for Calc
../target/debug/main examples/simple.calc
```

or simply

```
cargo run --bin main examples/simple.calc
```

Grammar-Lexer-Parser Pipeline

Here is a high-level view of a compiler *frontend* pipeline



Every language needs a (formal) grammar to describe its syntax and semantics. Once a program adheres to the rules of the grammar in *Source Code* (for example as input string or file format), it is *tokenized* and then *lexer* adds some metadata to each token for example, where each token starts and finishes in the original source code. Lastly, parsing (reshaping or restructuring) of the lexed outputs to Abstract Syntax Tree.

Grammar

While there are varieties of ways to define the grammar, in this book we will use the Parsing Expression Grammar (PEG).

Here is how our simple calculator language `Calc` (supporting addition and subtraction) grammar looks like in PEG

```
Program = _{ SOI ~ Expr ~ EOF }

Expr = { UnaryExpr | BinaryExpr | Term }

Term = _{Int | "(" ~ Expr ~ ")" }

UnaryExpr = { Operator ~ Term }

BinaryExpr = { Term ~ (Operator ~ Term)+ }

Operator = { "+" | "-" }

Int = @{ Operator? ~ ASCII_DIGIT+ }

WHITESPACE = _{ " " | "\t" }

EOF = _{ EOI | ";" }
```

Filename: calculator/src/grammar.pest

This grammar basically defines the syntax and semantics where

- each `Program` consists of expressions (`Expr`)
- expressions are either unary (`-1`) or binary (`1 + 2`)
- unary or binary expressions are made of `Term` and `Operator` (`"+"` and `"-"`)
- the only *atom* is integer `Int`

Given our grammar, we will use pest which is a powerful *parser generator* of PEG grammars. (For more details on pest, checkout the pest book

`pest` *derives* the parser `CalcParser::parse` from our grammar

```
#[derive(pest_derive::Parser)]
#[grammar = "grammar.pest"]
struct CalcParser;
```

Filename: calculator/src/parser.rs

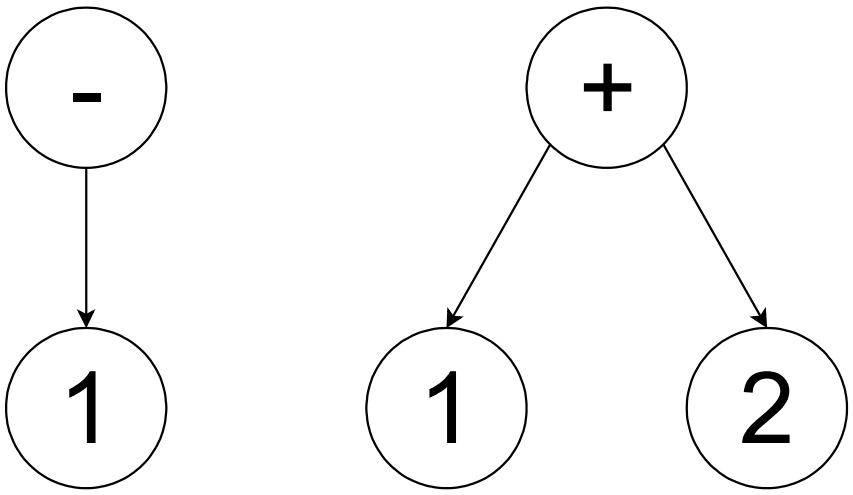
and does all the steps of the frontend pipeline that we mentioned so that we can start parsing any `Calc` source code (`source: &str`) via the `Rule` s of our grammar

```
CalcParser::parse(Rule::Program, source)
```

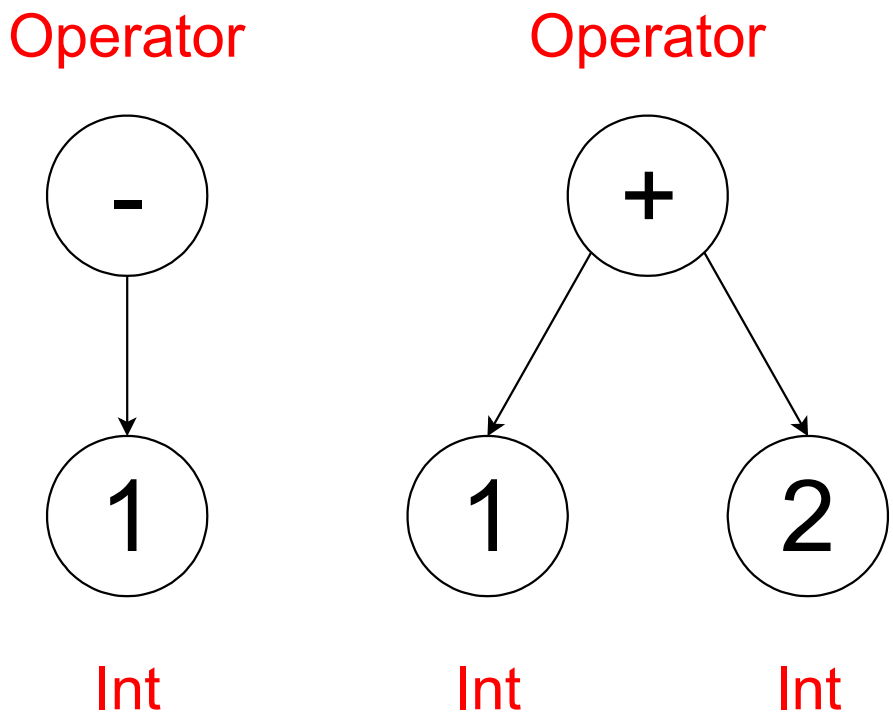
Before doing that, we need to define our Abstract Syntax Tree (AST) in the next section.

Abstract Syntax Tree (AST)

AST comes into picture when we want to go from the string representation of our program like `"-1"` or `"1 + 2"` to something more manageable and easier to work with. Since our program is not a random string (the grammar is for), we can use the structure within the expressions `"-1"` and `"1 + 2"` to our own advantage and come up with a *new representation* like a tree



One thing to note here is that the *kinds* of the nodes in our tree are not the same i.e. `+` node is different from `1` node. In fact, `+` has an **Operator** type and `1` is an integer **Int** type



so we define our AST nodes as

```
pub enum Operator {
    Plus,
    Minus,
}

pub enum Node {
    Int(i32),
}
```

Referring back to our grammar, we actually have different kinds of *recursive* expressions;

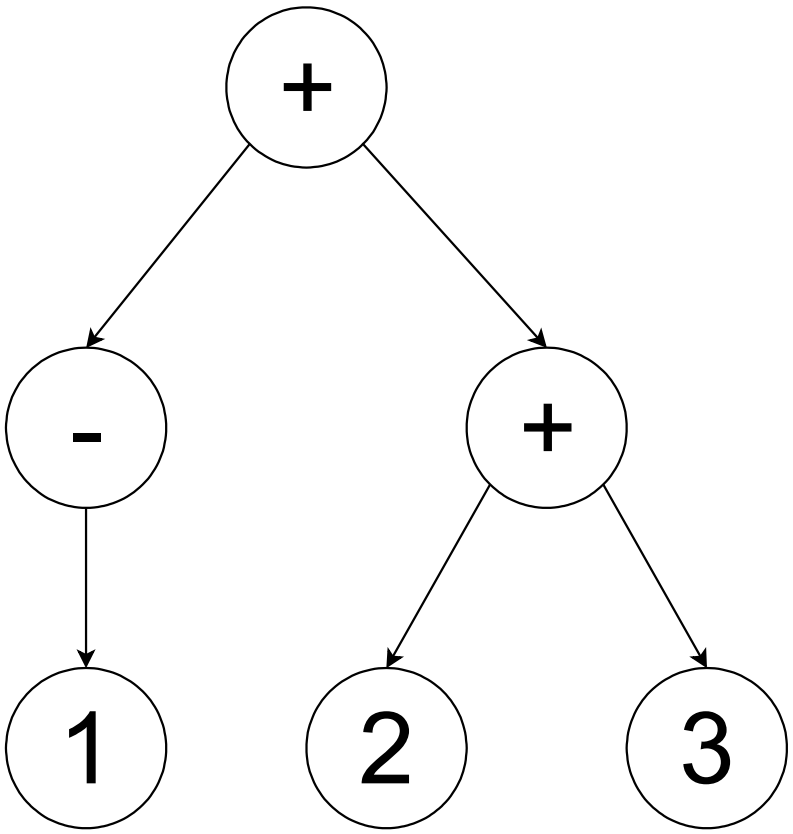
- **unary** grammar

  ```
  UnaryExpr = { Operator ~ Term }
  ```

- **binary** grammar

  ```
  BinaryExpr = { Term ~ (Operator ~ Term)* }
  ```

So for example, the expression `"-1 + (2 + 3)"` has this recursive structure



To include those into our AST to make it an actual tree data structure, we complete our AST definition as follows

```rust
pub enum Operator {
    Plus,
    Minus,
}

pub enum Node {
    Int(i32),
    UnaryExpr {
        op: Operator,
        child: Box<Node>,
    },
    BinaryExpr {
        op: Operator,
        lhs: Box<Node>,
        rhs: Box<Node>,
    },
}
```

Filename: calculator/src/ast.rs

Now, we can use the `pest` generated `CalcParser::parse` to map the Rules of our `Calc` language string to our AST.

```rust
pub fn parse(source: &str) -> std::result::Result<Vec<Node>,
pest::error::Error<Rule>> {
    let mut ast = vec![];
    let pairs = CalcParser::parse(Rule::Program, source)?;
    for pair in pairs {
        if let Rule::Expr = pair.as_rule() {
            ast.push(build_ast_from_expr(pair));
        }
    }
    Ok(ast)
}
```

Checkout calculator/src/parser.rs.

Note that `CalcParser::parse` takes care of the AST traversal and correctly maps it to `Vec<Node>` for easier access in later stages of compilation.

Interpreter

CPU is the *ultimate interpreter*. That is, it executes opcodes as it goes. To do that, after we have changed the representation (aka *lowered* the representation) of our source code `&str` to AST `Node`, a basic interpreter looks at each node of the AST (via any tree traversal methods) and simply **evaluates** it *recursively*

```
    pub fn eval(&self, node: &Node) -> i32 {
        match node {
            Node::Int(n) => *n,
            Node::UnaryExpr { op, child } => {
                let child = self.eval(child);
                match op {
                    Operator::Plus => child,
                    Operator::Minus => -child,
                }
            }
            Node::BinaryExpr { op, lhs, rhs } => {
                let lhs_ret = self.eval(lhs);
                let rhs_ret = self.eval(rhs);

                match op {
                    Operator::Plus => lhs_ret + rhs_ret,
                    Operator::Minus => lhs_ret - rhs_ret,
                }
            }
        }
    }
```

To sum up, we define a `Compile` trait that we will use throughout this chapter

```
pub trait Compile {
    type Output;

    fn from_ast(ast: Vec<Node>) -> Self::Output;

    fn from_source(source: &str) -> Self::Output {
        println!("Compiling the source: {}", source);
        let ast: Vec<Node> = parser::parse(source).unwrap();
        println!("{:?}", ast);
        Self::from_ast(ast)
    }
}
```

and we can now implement our interpreter

```
pub struct Interpreter;

impl Compile for Interpreter {
    type Output = Result<i32>;

    fn from_ast(ast: Vec<Node>) -> Self::Output {
        let mut ret = 0i32;
        let evaluator = Eval::new();
        for node in ast {
            ret += evaluator.eval(&node);
        }
        Ok(ret)
    }
}
```

Filename: calculator/src/compiler/interpreter.rs

and test

```
assert_eq!(Interpreter::from_source("1 + 2").unwrap(), 3);
```

Run such tests locally with

```
cargo test interpreter --tests
```

Just-In-Time (JIT) Compiler with LLVM

JIT compilation is a combination of Ahead-Of-Time (AOT) compilation and interpretation. As we saw previously, our `Calc` interpreter evaluates AST to values (actual integer `i32` values) but a JIT compiler differs from an interpreter in what it outputs. Intuitively, JIT outputs are like AOT outputs but generated at runtime when traversing the AST.

LLVM

LLVM (which is *not* an acronym) is a mature compiler backend (code generator) infrastructure powering many languages such as Clang, Rust, Swift, etc. It has its own IR and Virtual Machine Bytecode abstracting away the underlying platform-specific differences.

We will use inkwell which provides a safe Rust wrapper around LLVM.

Alternatives

Other code generators that you can use (see the exercises) in this book (not at mature as LLVM) are cratelift-simpljit and gcc-jit.

Addition

Setup

The code is available in `calculator/examples/llvm/src/main.rs`. Because my `llvm-config --version` shows `14.0.6` so I'm using `features = ["llvm14-0"]` in inkwell

```
inkwell = { git = "https://github.com/TheDan64/inkwell",
branch = "master", features = ["llvm14-0"] }
```

Go to `calculator/examples/llvm` sub-crate and `cargo run`.

Add Function

We want to define an add function like

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

but using the **LLVM language** and JIT it. For that, we need to define *every* bit of what makes up a function through LLVM basic constructs such as context, module, function signature setups, argument types, basic block, etc.

Here is how to *stitch* our add function in LLVM

1. We start by creating a `context`, adding the `addition` module and setting up the data type we want to use `i32_type` of type `IntType`

```
    let context = Context::create();
    let module = context.create_module("addition");
    let i32_type = context.i32_type();
```

2. We define the signature of `add(i32, i32) -> i32`, add the function to our module, create a basic block entry point and a builder to add later parts

```
    let fn_type = i32_type.fn_type(&[i32_type.into(),
i32_type.into()], false);
    let fn_val = module.add_function("add", fn_type, None);
    let entry_basic_block =
context.append_basic_block(fn_val, "entry");

    let builder = context.create_builder();
    builder.position_at_end(entry_basic_block);
```

3. We create the arguments `x` and `y` and add them to the `builder` to make up the return instruction

```
    let x =
fn_val.get_nth_param(0).unwrap().into_int_value();
    let y =
fn_val.get_nth_param(1).unwrap().into_int_value();

    let ret = builder.build_int_add(x, y, "add").unwrap();
    let return_instruction =
builder.build_return(Some(&ret)).unwrap();
```

4. Finally, we create a JIT execution engine (with no optimization for now) and let LLVM handle rest of the work for us

```
    let execution_engine = module
        .create_jit_execution_engine(OptimizationLevel::None)
        .unwrap();
    unsafe {
        type Addition = unsafe extern "C" fn(i32, i32) ->
i32;
        let add: JitFunction<Addition> =
execution_engine.get_function("add").unwrap();
        let x = 1;
        let y = 2;
        assert_eq!(add.call(x, y), x + y);
    }
```

Yes! all of this just to add two integers.

AST Traversal Patterns

Recall from the previous section that JITing our add function was very detailed and cumbersome to write. Fortunately, there are some useful patterns for traversing complicated ASTs (and IRs)

- **Builder pattern**
- **Visitor pattern** (Will be introduced in chapter 4)

Builder Pattern

Recall how we have interpreted our AST by traversing recursively and evaluating the nodes

```
struct Eval;

impl Eval {
    pub fn new() -> Self {
        Self
    }
    pub fn eval(&self, node: &Node) -> i32 {
        match node {
            Node::Int(n) => *n,
            Node::UnaryExpr { op, child } => {
                let child = self.eval(child);
                match op {
                    Operator::Plus => child,
                    Operator::Minus => -child,
                }
            }
            Node::BinaryExpr { op, lhs, rhs } => {
                let lhs_ret = self.eval(lhs);
                let rhs_ret = self.eval(rhs);

                match op {
                    Operator::Plus => lhs_ret + rhs_ret,
                    Operator::Minus => lhs_ret - rhs_ret,
                }
            }
        }
    }
}
```

Filename: calculator/src/compiler/interpreter.rs

but instead, we can take advantage of the inkwell Builder and recursively traverse our `Calc` AST as follows

```
struct RecursiveBuilder<'a> {
    i32_type: IntType<'a>,
    builder: &'a Builder<'a>,
}

impl<'a> RecursiveBuilder<'a> {
    pub fn new(i32_type: IntType<'a>, builder: &'a Builder) -
> Self {
        Self { i32_type, builder }
    }
    pub fn build(&self, ast: &Node) -> IntValue {
        match ast {
            Node::Int(n) => self.i32_type.const_int(*n as
u64, true),
            Node::UnaryExpr { op, child } => {
                let child = self.build(child);
                match op {
                    Operator::Minus => child.const_neg(),
                    Operator::Plus => child,
                }
            }
            Node::BinaryExpr { op, lhs, rhs } => {
                let left = self.build(lhs);
                let right = self.build(rhs);

                match op {
                    Operator::Plus => self
                        .builder
                        .build_int_add(left, right,
"plus_temp")
                        .unwrap(),
                    Operator::Minus => self
                        .builder
                        .build_int_sub(left, right,
"minus_temp")
                        .unwrap(),
                }
            }
        }
    }
}
```

and similar to our addition example, we can JIT the builder output

```
pub struct Jit;

impl Compile for Jit {
    type Output = Result<i32>;

    fn from_ast(ast: Vec<Node>) -> Self::Output {
        let context = Context::create();
        let module = context.create_module("calculator");

        let builder = context.create_builder();

        let execution_engine = module

.create_jit_execution_engine(OptimizationLevel::None)
            .unwrap();

        let i32_type = context.i32_type();
        let fn_type = i32_type.fn_type(&[], false);

        let function = module.add_function("jit", fn_type,
None);
        let basic_block =
context.append_basic_block(function, "entry");

        builder.position_at_end(basic_block);

        for node in ast {
            let recursive_builder =
RecursiveBuilder::new(i32_type, &builder);
            let return_value =
recursive_builder.build(&node);
            let _ =
builder.build_return(Some(&return_value));
        }
        println!(
            "Generated LLVM IR: {}",
            function.print_to_string().to_string()
        );

        unsafe {
            let jit_function: JitFunction<JitFunc> =
execution_engine.get_function("jit").unwrap();

            Ok(jit_function.call())
        }
```

```
    }
}
```

Filename: calculator/src/compiler/jit.rs

Finally, we can test it

```
assert_eq!(Jit::from_source("1 + 2").unwrap(), 3)
```

Run such tests locally with

```
cargo test jit --tests
```

Exercise

To get most out of this chapter, it is recommended to at least try the first exercise below

1. Add support for multiplication and division to the calculator and allow computations on floating numbers `f32`. Can you include standard operator precedence?
2. JIT with cranelift-simplejit
3. JIT with gcc-jit

Virtual Machine

Recall from the crash course that a (process) VM abstracts away hardware specific instructions so that its Bytecodes (abstract instructions) can be executed in any environment that has the Bytecode runtime support. So to create our VM and its runtime, we need to define our

- Opcodes (new encoding atoms)
- Bytecode representation and
- **Runtime model** as Stack Machine

Opcode

Since our expression based `Calc` language is made up of

- *constant* integers
- *unary* (plus, minus sign) operators and
- *binary* (addition, subtraction) operators

we can define our new opcode encodings like

```
pub enum OpCode {
    OpConstant(u16), // pointer to constant table
    OpPop,           // pop is needed for execution
    OpAdd,
    OpSub,
    OpPlus,
    OpMinus,
}
```

Filename: calculator/src/compiler/vm/opcode.rs

We choose the simplest form of encoding i.e. encoding the ops as bytes `u8` (in hex format). That is,

```
        OpCode::OpConstant(arg) => make_three_byte_op(0x01,
arg),
        OpCode::OpPop => vec![0x02],  // decimal repr is 2
        OpCode::OpAdd => vec![0x03],  // decimal repr is 3
        OpCode::OpSub => vec![0x04],  // decimal repr is 4
        OpCode::OpPlus => vec![0x0A], // decimal repr is 10
        OpCode::OpMinus => vec![0x0B], // decimal repr is 11
```

For easy of access, we store constant `Node::Int(i32)` nodes in a separate memory and `OpConstant(arg)` tracks these values. In a unary expression like `"1"`, we encode `Node::Int(1)` as the opcode `[1, 0, 0]` as the first constant. (**0x01** in decimal is **1**).

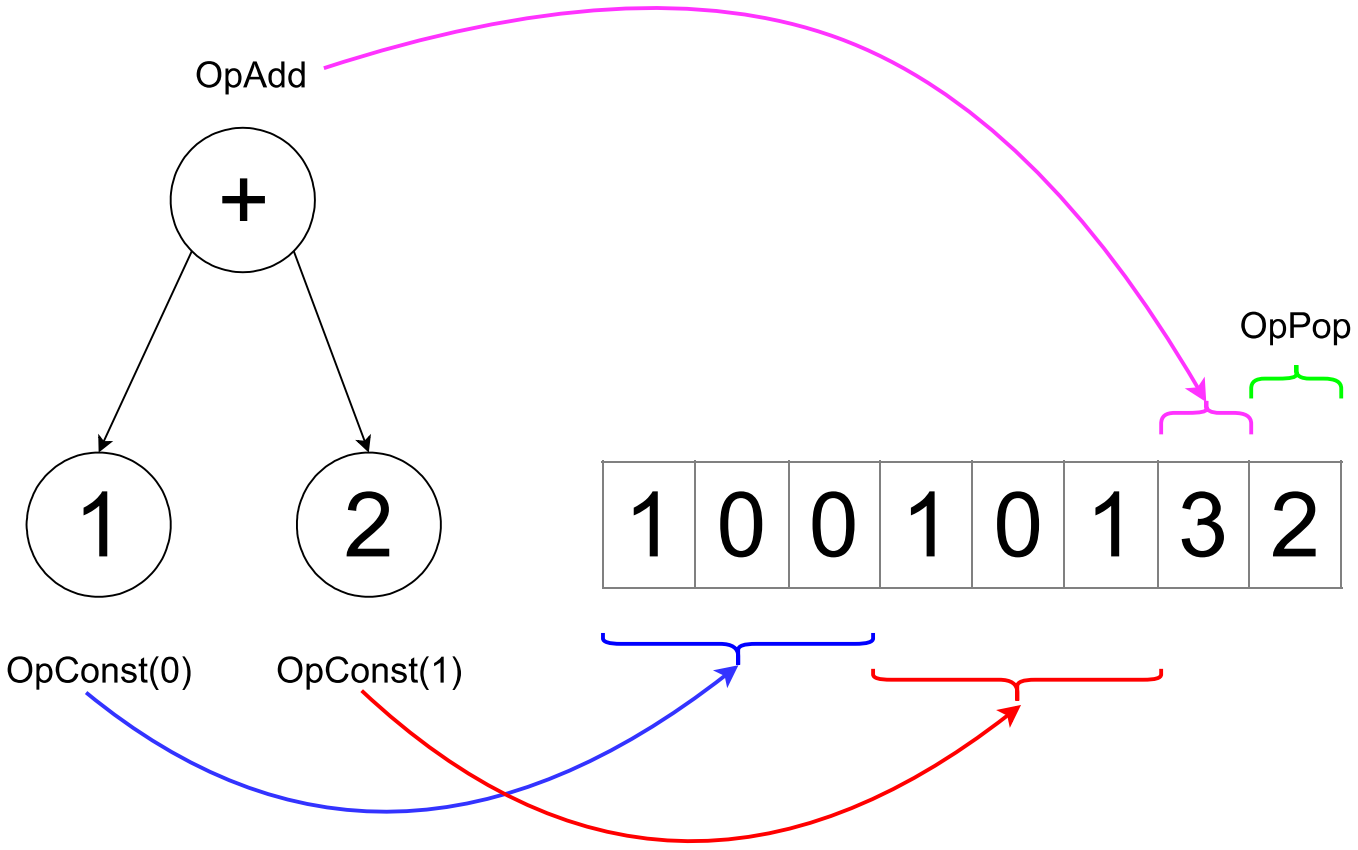
Bytecode

We define

```
pub struct Bytecode {
    pub instructions: Vec<u8>,
    pub constants: Vec<Node>,
}
```

Filename: calculator/src/compiler/vm/bytecode.rs

and pictorially, here is how `"1 + 2"` AST to Bytecode conversion would look like



and in Rust

```
ByteCode {
    instructions: [1, 0, 0, 1, 0, 1, 3, 2],
    constants: [Int(1), Int(2)]
}
```

Now, we can implement our Bytecode interpreter

```rust
pub struct Interpreter {
    bytecode: Bytecode,
}

impl Compile for Interpreter {
    type Output = Bytecode;

    fn from_ast(ast: Vec<Node>) -> Self::Output {
        let mut interpreter = Interpreter {
            bytecode: Bytecode::new(),
        };
        for node in ast {
            println!("compiling node {:?}", node);
            interpreter.interpret_node(node);
            // pop one element from the stack after
            // each expression statement to clean up
            interpreter.add_instruction(OpCode::OpPop);
        }
        interpreter.bytecode
    }
}
```

Filename: calculator/src/compiler/vm/bytecode.rs

Runtime

From previous example, our interpreter goes through the bytecode instructions and executes them.

Continuing our `"1 + 2"` Bytecode example,

```
instructions: [1, 0, 0, 1, 0, 1,   3,   2],
               -------- --------    -    -
                  |        |       |    |
constants: [  Int(1),   Int(2)]  OpAdd  OpPop

[1, 0, 0] points to the first element in constants table i.e.
Int(1)
[1, 0, 1] points to Int(2)
[3] (or [0x03]) corresponding to the Opcode *OpAdd*, performs
the addition operation Int(1 + 2)
[2] (or [0x02]) corresponding to the Opcode *OpPop* pops out
the computed Bytecodes
```

and since we want to model our runtime as a Stack Machine so we define our VM as struct with Bytecode, stack memory (in Stack Machine) and a stack pointer to the next free space

```
const STACK_SIZE: usize = 512;

pub struct VM {
    bytecode: Bytecode,
    stack: [Node; STACK_SIZE],
    stack_ptr: usize, // points to the next free space
}
```

and with the help of *instruction pointer (IP)*, we execute the Bytecodes as follows

```
    pub fn run(&mut self) {
        let mut ip = 0; // instruction pointer
        while ip < self.bytecode.instructions.len() {
            let inst_addr = ip;
            ip += 1;

            match self.bytecode.instructions[inst_addr] {
                0x01 => {
                    //OpConst
                    let const_idx = convert_two_u8s_to_usize(
                        self.bytecode.instructions[ip],
                        self.bytecode.instructions[ip + 1],
                    );
                    ip += 2;

self.push(self.bytecode.constants[const_idx].clone());
                }
                0x02 => {
                    //OpPop
                    self.pop();
                }
                0x03 => {
                    // OpAdd
                    match (self.pop(), self.pop()) {
                        (Node::Int(rhs), Node::Int(lhs)) =>
self.push(Node::Int(lhs + rhs)),
                        _ => panic!("Unknown types to
OpAdd"),
                    }
                }
                0x04 => {
                    // OpSub
                    match (self.pop(), self.pop()) {
                        (Node::Int(rhs), Node::Int(lhs)) =>
self.push(Node::Int(lhs - rhs)),
                        _ => panic!("Unknown types to
OpSub"),
                    }
                }
                0x0A => {
                    // OpPlus
                    match self.pop() {
                        Node::Int(num) =>
self.push(Node::Int(num)),
                        _ => panic!("Unknown arg type to
```

```
OpPlus"),
                    }
                }
                0x0B => {
                    // OpMinus
                    match self.pop() {
                        Node::Int(num) =>
self.push(Node::Int(-num)),
                        _ => panic!("Unknown arg type to
OpMinus"),
                    }
                }
                _ => panic!("Unknown instruction"),
            }
        }
    }

    pub fn push(&mut self, node: Node) {
        self.stack[self.stack_ptr] = node;
        self.stack_ptr += 1; // ignoring the potential stack
overflow
    }

    pub fn pop(&mut self) -> Node {
        // ignoring the potential of stack underflow
        // cloning rather than mem::replace for easier
testing
        let node = self.stack[self.stack_ptr - 1].clone();
        self.stack_ptr -= 1;
        node
    }
```

Filename: calculator/src/compiler/vm/vm.rs

To examine the generated Bytecodes and run our VM, we can do

```
let byte_code = Interpreter::from_source(source);
println!("byte code: {:?}", byte_code);
let mut vm = VM::new(byte_code);
vm.run();
println!("{}", vm.pop_last());
```

Run tests locally for our VM with

```
cargo test vm --tests
```

Checkout the next section on how to create a REPL for our `Calc` to compare different compilation paths.

Read-Eval-Print Loop (REPL)

REPL (as its name implies) loops through every line of the input and compiles it. We use rustyline crate to create our REPL. For each line of input, we can optionally choose to

- directly interpret the AST
- JIT the AST
- compile to our bytecode VM and interpret it

```
fn main() -> Result<()> {
    let mut rl = DefaultEditor::new()?;
    println!("Calculator prompt. Expressions are line
evaluated.");
    loop {
        let readline = rl.readline(">> ");
        match readline {
            Ok(line) => {
                cfg_if! {
                    if #[cfg(any(feature = "jit", feature =
"interpreter"))] {
                        match Engine::from_source(&line) {
                            Ok(result) => println!("{}",
result),
                            Err(e) => eprintln!("{}", e),
                        };
                    }
                    else if #[cfg(feature = "vm")] {
                        let byte_code =
Engine::from_source(&line);
                        println!("byte code: {:?}",
byte_code);
                        let mut vm = VM::new(byte_code);
                        vm.run();
                        println!("{}", vm.pop_last());
                    }
                }
            }
            Err(ReadlineError::Interrupted) => {
                println!("CTRL-C");
                break;
            }
            Err(ReadlineError::Eof) => {
                println!("CTRL-D");
                break;
            }
            Err(err) => {
                println!("Error: {:?}", err);
                break;
            }
        }
    }
    Ok(())
```

Filename: calculator/src/bin/repl.rs

Now, we can use run the REPL and choose different compilation path

```
cargo run --bin repl --features jit
// OR
cargo run --bin repl --features interpreter
// OR
cargo run --bin repl --features vm
```

In any of them, you should see the prompt like

```
Calculator prompt. Expressions are line evaluated.
>>>
```

waiting for your inputs. Here are some sample outputs of different compilation paths in debug mode.

- with `--features jit`

```
Calculator prompt. Expressions are line evaluated.
>> 1 + 2
Compiling the source: 1 + 2
[BinaryExpr { op: Plus, lhs: Int(1), rhs: Int(2) }]
Generated LLVM IR: define i32 @jit() {
entry:
  ret i32 3
}

3
>> (1 + 2) - (8 - 10)
Compiling the source: (1 + 2) - (8 - 10)
[BinaryExpr { op: Minus, lhs: BinaryExpr { op: Plus, lhs:
Int(1), rhs: Int(2) }, rhs: BinaryExpr { op: Minus, lhs:
Int(8), rhs: Int(10) } }]
Generated LLVM IR: define i32 @jit() {
entry:
  ret i32 5
}

5
>>
CTRL-C
```

- with `--features vm`

```
Calculator prompt. Expressions are line evaluated.
>> 1 + 2
Compiling the source: 1 + 2
[BinaryExpr { op: Plus, lhs: Int(1), rhs: Int(2) }]
compiling node BinaryExpr { op: Plus, lhs: Int(1), rhs:
Int(2) }
added instructions [1, 0, 0] from opcode OpConstant(0)
added instructions [1, 0, 0, 1, 0, 1] from opcode
OpConstant(1)
added instructions [1, 0, 0, 1, 0, 1, 3] from opcode OpAdd
added instructions [1, 0, 0, 1, 0, 1, 3, 2] from opcode OpPop
byte code: Bytecode { instructions: [1, 0, 0, 1, 0, 1, 3, 2],
constants: [Int(1), Int(2)] }
3
>> (1 + 2) - (8 - 10)
Compiling the source: (1 + 2) - (8 - 10)
[BinaryExpr { op: Minus, lhs: BinaryExpr { op: Plus, lhs:
Int(1), rhs: Int(2) }, rhs: BinaryExpr { op: Minus, lhs:
Int(8), rhs: Int(10) } }]
compiling node BinaryExpr { op: Minus, lhs: BinaryExpr { op:
Plus, lhs: Int(1), rhs: Int(2) }, rhs: BinaryExpr { op:
Minus, lhs: Int(8), rhs: Int(10) } }
added instructions [1, 0, 0] from opcode OpConstant(0)
added instructions [1, 0, 0, 1, 0, 1] from opcode
OpConstant(1)
added instructions [1, 0, 0, 1, 0, 1, 3] from opcode OpAdd
added instructions [1, 0, 0, 1, 0, 1, 3, 1, 0, 2] from opcode
OpConstant(2)
added instructions [1, 0, 0, 1, 0, 1, 3, 1, 0, 2, 1, 0, 3]
from opcode OpConstant(3)
added instructions [1, 0, 0, 1, 0, 1, 3, 1, 0, 2, 1, 0, 3, 4]
from opcode OpSub
added instructions [1, 0, 0, 1, 0, 1, 3, 1, 0, 2, 1, 0, 3, 4,
4] from opcode OpSub
added instructions [1, 0, 0, 1, 0, 1, 3, 1, 0, 2, 1, 0, 3, 4,
4, 2] from opcode OpPop
byte code: Bytecode { instructions: [1, 0, 0, 1, 0, 1, 3, 1,
0, 2, 1, 0, 3, 4, 4, 2], constants: [Int(1), Int(2), Int(8),
Int(10)] }
5
>>>
CTRL-C
```

Conclusion

This concludes our Calculator chapter. We took advantage of the simplicity of our `Calc` language to touch on a lot of topics.

Thanks for following along and reading up this far!

Stay tuned for the next chapter where we gradually work our way up to create a statically typed language named creatively as **Firstlang** :D