Andrea Bergia

Sr Staff Software Engineer, ServiceNow

LET'S WRITE AN INTERPRETER WITH A JIT

ABOUT ME

https://andreabergia.com

Interested in compilers and VMs for ages
Working as a compiler engineer in 2023

OVERVIEW

- Introduction
- Grammar, parser, and AST
- IR and optimizations
- Backend and JIT
- Q&A

INTRODUCTION

Let’s create an extremely simple programming language from scratch and a JIT compiler for it: **Emjay**

https://andreabergia.com/blog/2025/02/emjay-a-simple-jit-that-does-math/

https://github.com/andreabergia/emjay

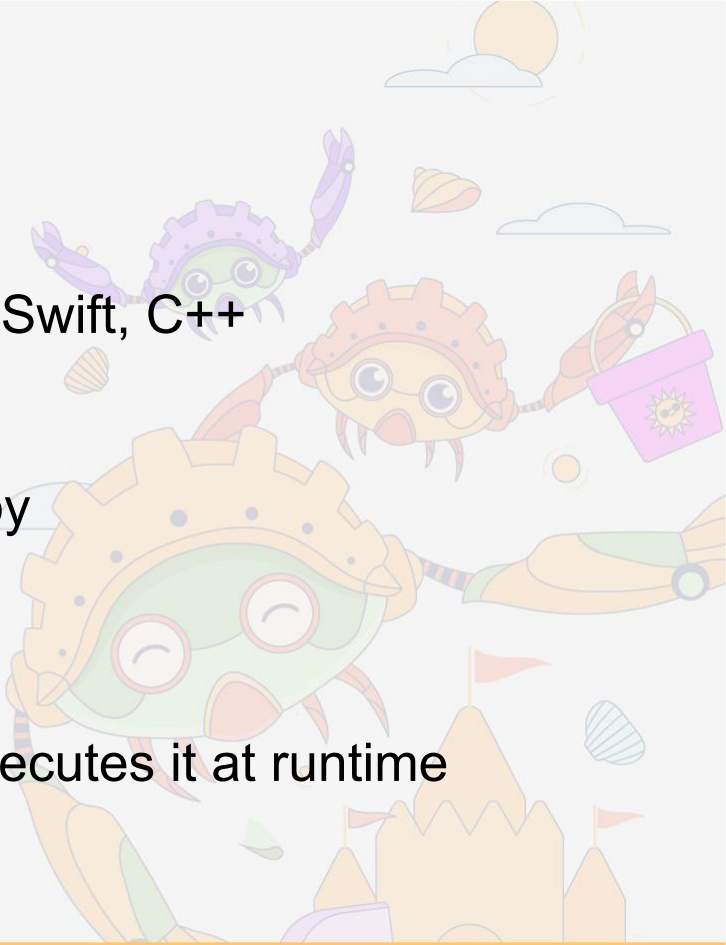
My objective: give you a high level overview of how a real programming language is implemented and works

CLASSIFICATION OF LANGUAGES

Compiled ahead-of-time: Rust, Go, Swift, C++
Generate *machine code*

Interpreted: JavaScript, Python, Ruby
VM executes source-code directly

Compiled to bytecode: Java, C#
Compiler generates bytecode, VM executes it at runtime



INTERPRETED LANGUAGES AND JIT

Interpreted languages have an **interpreter** that can evaluate code

Most non-toy interpreted languages include also one (or more) **JITs**: they can generate and execute efficient machine code for “hot” functions

Emjay does not have an interpreter; just a basic JIT

EMJAY'S LANGUAGE

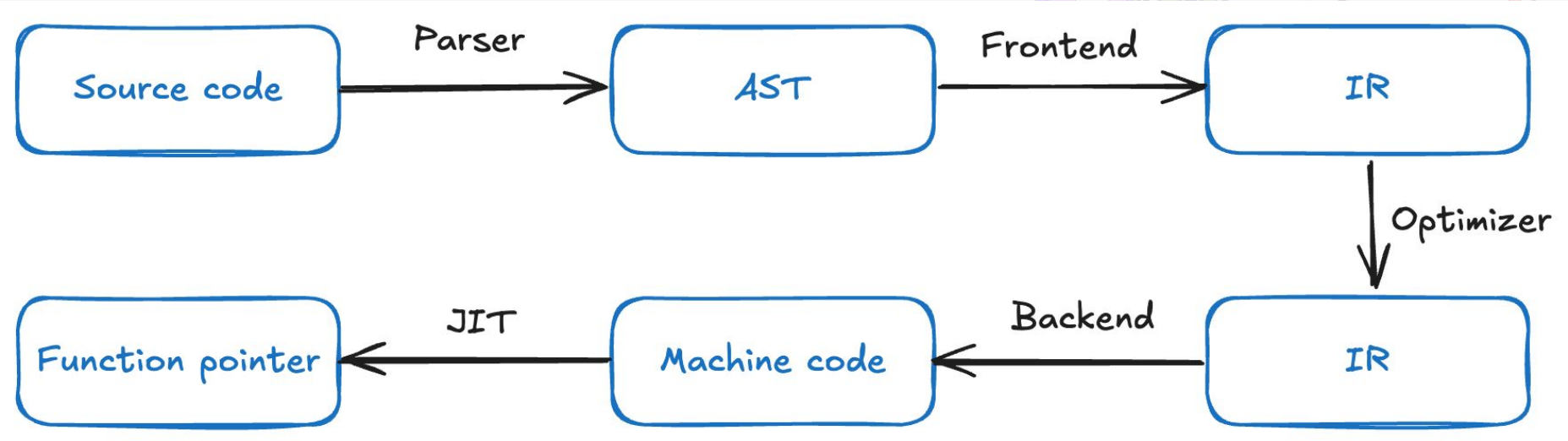
To make this simple, we'll set the following restrictions:

- only one type (i64)
- no control flow (if or loops)
- just expressions, variables, and function calls
- only aarch64 supported (Apple M1 and similar)

THE LANGUAGE SYNTAX

```
fn main() {
    let v = 1000;
    return v + f(3, 2, 1);
}

fn f(x, y, z) {
    return x * 100 + y * 10 + z;
}
```

HIGH LEVEL OVERVIEW



GRAMMARS

Many programming languages use a **grammar** to define the syntax

Grammars are generally defined in BNF form

We'll use the Pest library to handle the grammar for us

```
block = { "{" ~ (statement | block)* ~ "}" }
statement = _{ (
  letStatement |
  assignmentStatement |
  returnStatement
) ~ ";" }

letStatement =
  { "let" ~ identifier ~ "=" ~ expression }
assignmentStatement =
  { identifier ~ "=" ~ expression }
returnStatement =
  { "return" ~ expression }
```

AST

An **A**bstract **S**yntax **T**ree is a data structure that represents the parsed code

```
fn main(x) {
  let a = 2;
  return a + 3 + x;
}
```

```
Function { name: "main", args: ["x"], block:
  [
    LetStatement {
      name: "a",
      expression: Number(2)
    },
    ReturnStatement(
      Add(
        Add(
          Identifier("a"),
          Number(3)
        ),
        Identifier("x")
      )
    )
  ]
}
```

EMJAY'S AST

Here is the AST of Emjay for the following rule:

```
block = { "{" ~ (statement | block)* ~ "}" }
statement = _{ (
  letStatement |
  assignmentStatement |
  returnStatement
) ~ ";" }

letStatement =
  { "let" ~ identifier ~ "=" ~ expression }
assignmentStatement =
  { identifier ~ "=" ~ expression }
returnStatement =
  { "return" ~ expression }
```

```rust
pub type Block = Vec<BlockElement>;

#[derive(Debug, PartialEq)]
pub enum BlockElement {
    LetStatement {
        name: String,
        expression: Expression,
    },
    AssignmentStatement {
        name: String,
        expression: Expression,
    },
    ReturnStatement(Expression),
    NestedBlock(Block),
}
```

PARSERS

A **parser** recognizes the language, validates the syntax, and builds the AST

Many languages use a hand written parser, generally a recursive descent one

```
fn parse_block(rule: Pair<'_, Rule>) -> Block {
 rule.into_inner()
      .map(|statement| match statement.as_rule() {
        Rule::letStatement =>
          parse_statement_let(statement),
        Rule::assignmentStatement =>
          parse_statement_assignment(statement),
        Rule::returnStatement =>
          parse_statement_return(statement),
        Rule::block =>
          BlockElement::NestedBlock(
            parse_block(statement)),
        _ => unreachable!(),
      })
      .collect()
}
```

THE FRONTEND

The **frontend** of a compiler handles semantic analysis:

- does that variable exist?
- is the function being called with the correct arguments?
- is a value used after being moved?

It walks the AST and generates an **I**ntermediate **R**epresentation

Emjay has only one type of IR; real compilers often have multiple levels, each more “lower level”

EXAMPLE

```
fn main(x) {
  let a = 2;
  return a + 3 + x;
}
```

```
a => r0
2 => r1
and so on
```

```
fn main - #args: 1, #reg: 5 {
    0:  mvi  @r0, 2
    1:  mvi  @r1, 3
    2:  add  @r2, r0, r1
    3:  mva  @r3, a0
    4:  add  @r4, r2, r3
    5:  ret  r4
}
```

EMJAY'S IR

Static **S**ingle **A**ssignment form: each register is assigned just once, infinite number of registers

Emjay does not have control-flow, so SSA is trivial to build

```
pub struct IrRegister(pub usize);

pub enum IrInstruction {
    Mvi {
        dest: IrRegister,
        val: i64,
    },
    MvArg {
        dest: IrRegister,
        arg: ArgumentIndex,
    },
    BinOp {
        operator: BinOpOperator,
        dest: IrRegister,
        op1: IrRegister,
        op2: IrRegister,
    },
    ...
```

OPTIMIZATIONS

SSA is very helpful for optimizations.

Examples: constant propagation, strength reduction, common subexpression elimination

```
fn main - #args: 1, #reg: 5 {
    0:  mvi  @r0, 2
    1:  mvi  @r1, 3
    2:  add  @r2, r0, r1
    3:  mva  @r3, a0
    4:  add  @r4, r2, r3
    5:  ret  r4
}
```

After optimizations:

```
fn main - #args: 1, #reg: 3 {
    0:  mvi  @r0, 5
    1:  mva  @r1, a0
    2:  add  @r2, r0, r1
    3:  ret  r2
}
```

BACKEND

The **backend** of a compiler is what generates actual machine code from the IR

Multiple options for backends: LLVM and Cranelift are great!

For Emjay, I have chosen to hand-write a super simple backend

Supports only aarch64 (i.e. Mac's M1 and similar)

EMJAY'S BACKEND ARCHITECTURE

1. allocate registers, to map each virtual register to a physical location
2. process each IR instruction
3. generate one or more aarch64 instruction for each IR instruction
4. handle other details like prologue, epilogue

Note: I didn't really implement a separate assembler; I jumbled everything together and I'm generating machine code directly.

REGISTER ALLOCATIONS

We need to map the IRs' virtual, infinite register, onto the limited set of the CPU's real registers

In Emjay I have implemented linear scan allocation; real compilers have much more complex algorithms

```rust
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum AllocatedLocation<HardwareRegister>
{
    Register { register: HardwareRegister },
    Stack { offset: usize },
}

pub fn allocate<HardwareRegister>(
    function: &CompiledFunction,
    hw_registers: Vec<HardwareRegister>,
) -> Vec<AllocatedLocation<HardwareRegister>>
where
    HardwareRegister: Clone + fmt::Debug
...
```

EMJAY'S BACKEND ARCHITECTURE - CODE #1

```
enum Aarch64Instruction {
    Ret,
    MovImmToReg {
        register: Register,
        value: i64,
    },
    MovRegToReg {
        source: Register,
        destination: Register,
    },
    ...
```

```
enum Register {
    X0,
    X1,
    X2,
    ...,
    X29,
```

EMJAY'S BACKEND ARCHITECTURE - CODE #2

```rust
impl Aarch64Instruction {
  fn make_machine_code(&self) -> Vec<u8> {
    match self {
        Ret => vec![0xC0, 0x03, 0x5F, 0xD6],
        MovRegToReg { source, destination } => {
            let mut i: u32 = Self::MOV;
            i |= source.index() << 16;
            i |= destination.index();
            i.to_le_bytes().to_vec()
        },
        ...
```

EMJAY'S BACKEND ARCHITECTURE - CODE #3

```
impl MachineCodeGenerator for Aarch64Generator {
    fn generate_machine_code(
        &mut self,
        function: &CompiledFunction,
    ) -> Result<GeneratedMachineCode, BackendError> {
        let mut instructions = Vec::new();
        for instruction in function.body.iter() {
            match instruction {
                continues on next slide
```

EMJAY'S BACKEND ARCHITECTURE - CODE #4

```
IrInstruction::Mvi { dest, val } => {
    let AllocatedLocation::Register { register } = self.locations[dest.0] else {
        return Err(BackendError::NotImplemented(
            "move immediate to stack".to_string(),
        ));
    };

    instructions.push(MovImmToReg {
        register,
        value: *val,
    })
}
```

JIT

We now have a sequence of instructions, that we can turn into bytes - i.e. the machine code!

How do we run it though?

We need to get a *function pointer* to invoke and execute it!

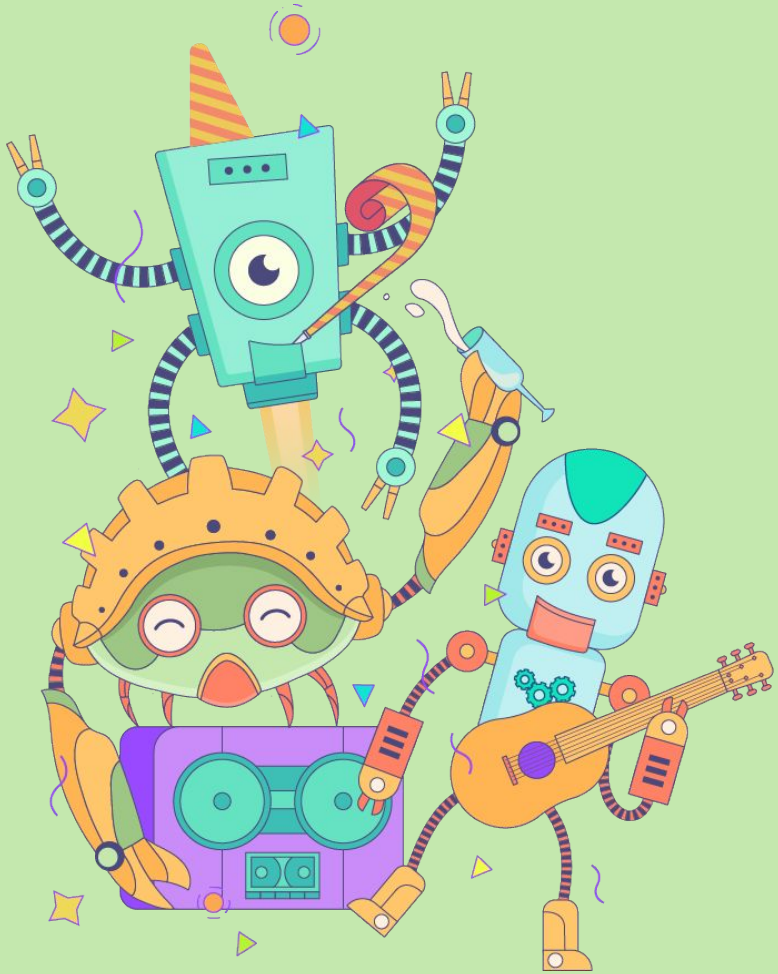
EMJAY'S JIT

```rust
unsafe fn to_function_pointer(bytes: &[u8]) -> Result<JitFn, MmapError> {
    let size = bytes.len();

    let map = mmap_anonymous(
        std::ptr::null_mut(), size, ProtFlags::WRITE, MapFlags::PRIVATE,
    )?;
    std::ptr::copy_nonoverlapping(bytes.as_ptr(), map as *mut u8, size);
    mprotect(map, size, MprotectFlags::EXEC)?;

    let f: JitFn = std::mem::transmute(map);
    Ok(f)
}
```

END-TO-END TESTS

```rust
#[test]
fn can_generate_valid_basic_function() {
  let source = "fn test() { let a = 2; return -a + 1; }";
  let program = jit_compile_program(source, "test")
      .expect("function should compile");
  let res = program.main_function(); // Call it!
  assert_eq!(res, -1);
}
```

Q&A



ANDREA BERGIA

andreabergia@gmail.com

https://andreabergia.com

Highly recommended book:

https://craftinginterpreters.com/