How i’ve built an Interpreter and JIT Compiler for The Brainfuck language

(Pre-AI 😉)

whoami

Amit Yahav

- Senior Software Engineer @ Cyolo
- Excited about how things work under the hood
    - Operating systems
    - Databases
    - Compilers
    - and more..

```
++++++++               Set Cell #0 to 8
[
    >++++               Add 4 to Cell #1; this will always set Cell #1 to 4
    [                   as the cell will be cleared by the loop
        >++             Add 2 to Cell #2
        >+++            Add 3 to Cell #3
        >+++            Add 3 to Cell #4
        >+              Add 1 to Cell #5
        <<<<-           Decrement the loop counter in Cell #1
    ]                   Loop till Cell #1 is zero; number of iterations is 4
    >+                  Add 1 to Cell #2
    >+                  Add 1 to Cell #3
    >-                  Subtract 1 from Cell #4
    >>+                 Add 1 to Cell #6
    [<]                 Move back to the first zero cell you find; this will
                        be Cell #1 which was cleared by the previous loop
    <-                  Decrement the loop Counter in Cell #0
]                       Loop till Cell #0 is zero; number of iterations is 8

>>.                     Cell #2 has value 72 which is 'H'
>---.                   Subtract 3 from Cell #3 to get 101 which is 'e'
+++++++..+++.           Likewise for 'llo' from Cell #3
>>.                     Cell #5 is 32 for the space
<-.                     Subtract 1 from Cell #4 for 87 to give a 'W'
<.                      Cell #3 was set to 'o' from the end of 'Hello'
+++.------.--------.    Cell #3 for 'rl' and 'd'
>>+.                    Add 1 to Cell #5 gives us an exclamation point
>++.                    And finally a newline from Cell #6
```

What is Brainfuck?

- Esoteric programming language created in 1993
- Minimalist design: Only 8 commands
- Turing-complete but intentionally difficult to use
- Perfect for learning about interpreters and compilers!

Brainfuck Commands

> Move pointer right

< Move pointer left

+ Increment current cel

- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

Example: > > < + - [> > > + +] < <

Brainfuck Commands

> Move pointer right

< Move pointer left

\+ Increment current cel

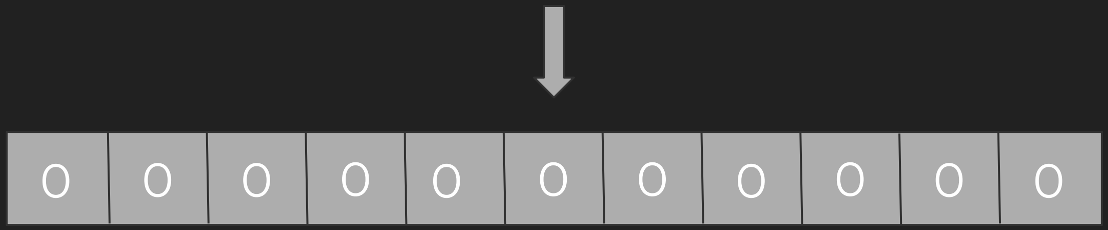
\- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0



Example: > > < + - [> > > + +] < <

Brainfuck Commands

> Move pointer right

< Move pointer left

+ Increment current cel

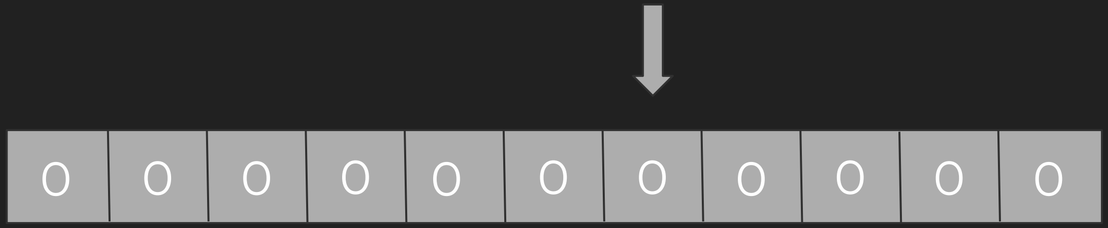
- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

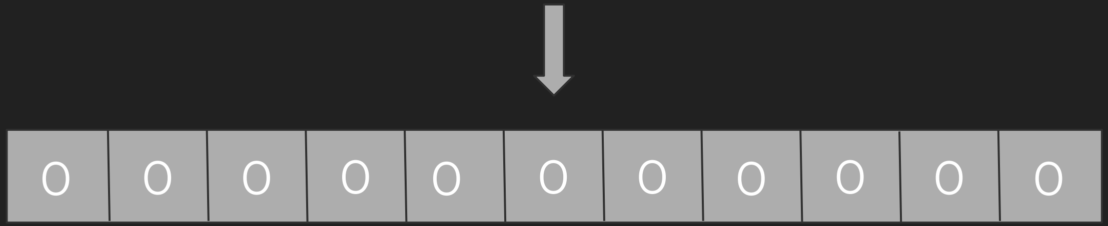
[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0



Example: > > < + - [> > > + +] < <

Brainfuck Commands



\> Move pointer right

< Move pointer left

\+ Increment current cel

\- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

Example: > > < + - [> > > + +] < <

Brainfuck Commands

> Move pointer right

< Move pointer left

+ Increment current cel

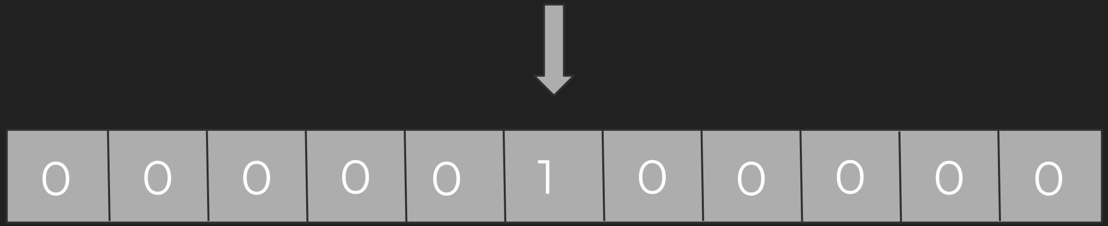
- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0



Example: > > < + - [> > > + +] < <

Brainfuck Commands

> Move pointer right

< Move pointer left

+ Increment current cel

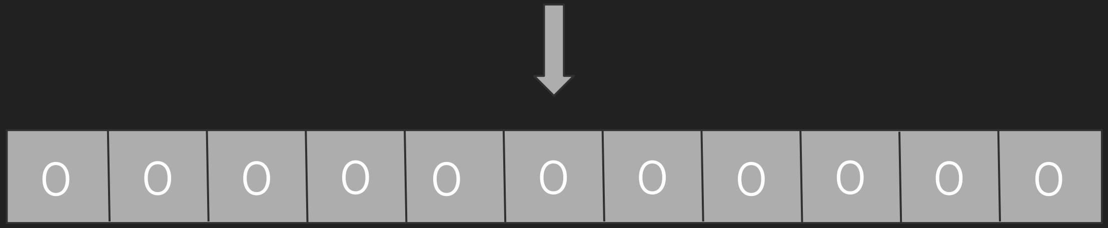
- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0



Example: > > < + - [> > > + +] < <

Brainfuck Commands

> Move pointer right

< Move pointer left

+ Increment current cel

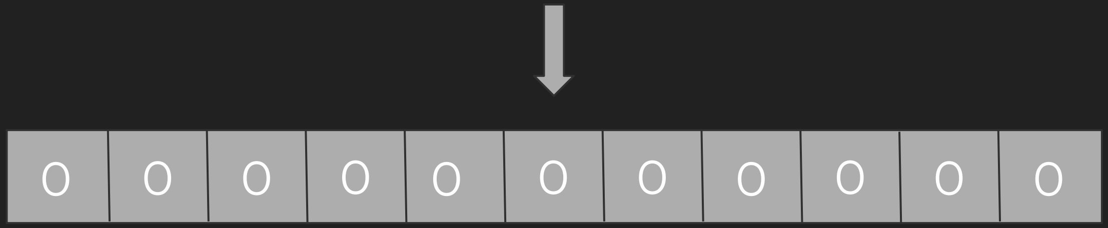
- Decrement current cell

. Output current cell (as ASCII)

, Input to current cell

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0



Example: > > < + - [> > > + +] < <

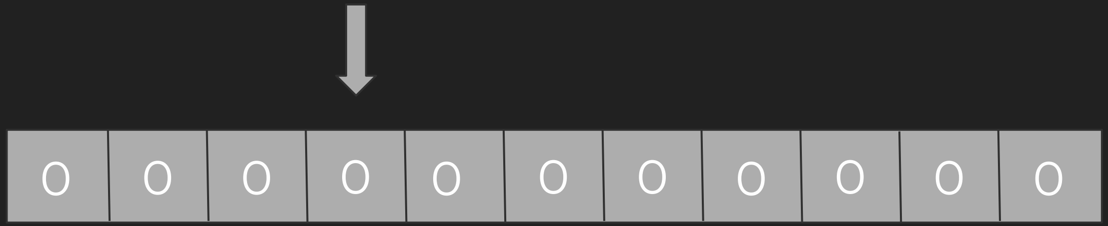
Brainfuck Commands

>	Move pointer right
<	Move pointer left
+	Increment current cel
-	Decrement current cell
.	Output current cell (as ASCII)
,	Input to current cell
[	Jump to the command after the matching] command if cell is 0
]	Jump back to the command after the matching [command if cell is not 0

Example: **> > < + - [> > > + +] < <**

Brainfuck Commands

>	Move pointer right
<	Move pointer left
+	Increment current cel
-	Decrement current cell
.	Output current cell (as ASCII)
,	Input to current cell
[	Jump to the command after the matching] command if cell is 0
]	Jump back to the command after the matching [command if cell is not 0



Example: **`> > < + - [ > > > + + ] < <`**

Interpreter

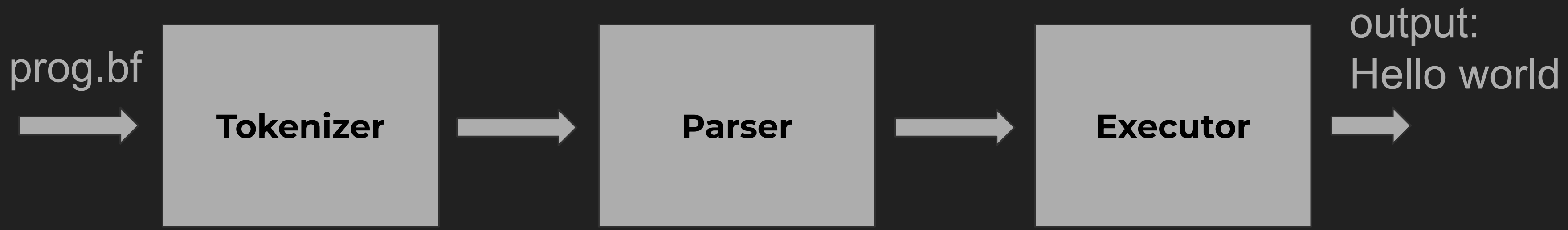
What is an Interpreter?

Type of computer program that reads and **executes code directly, line by line**

Interpreters in industry

- CPython (Python)
- V8 (Javascript)
- Zend Engine (PHP)

Interpreter Pipeline



Tokenizer

- Breaks a raw text input (source code) into **tokens** — small, meaningful pieces

- Each token represents something important: **keywords, symbols, numbers, operators, identifiers, etc**

- **In our case, groups consecutive characters of the same type:**

  for example: “<<<<” is a token of kind “<”

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

```go
type Token string // A Token is a group of consecutive characters of the same type

func (t *Tokenizer) Tokenize(content []byte) []Token {
    var tokens []Token

    currToken := nullToken()

    for _, c := range content {
        if !isNullToken(currToken) && currToken.Kind() != OpKind(c) {
            // When the character type changes, finalize the current token and add it to
            // the list of tokens
            tokens = append(tokens, currToken)
            currToken = nullToken()
        }

        switch OpKind(c) {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow:
            currToken += Token(c)
            .
            .
            .
        }
    }

    return tokens
```

content: <<<>>++--

tokens = [“<<<”, “>>”, “++”, “--”]

Parser

- Takes the list of tokens produced by the tokenizer

- Builds a representation like an **AST** (Abstract Syntax Tree) or similar structure where each node represents a construct in the language

- **In our case, we produce a list of operators**

```go
type Operator struct {
    Kind    OpKind
    Operand int
}


func (p *Parser) Parse(tokens []Token) []Operator {
    operators := make([]Operator, 0, len(tokens))

    for _, token := range tokens {
        var op Operator

        switch token.Kind() {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow, OpDot, OpComma:
            op = Operator{
                Kind:    token.Kind(),
                Operand: len(token),
            }

        .
        .
        .

        operators = append(operators, op)
    }

    return operators
}
```

```go
type Operator struct {
    Kind    OpKind
    Operand int
}


func (p *Parser) Parse(tokens []Token) []Operator {
    operators := make([]Operator, 0, len(tokens))

    for _, token := range tokens {
        var op Operator

        switch token.Kind() {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow, OpDot, OpComma:
            op = Operator{
                Kind:    token.Kind(),
                Operand: len(token),
            }


        .
        .
        .

        operators = append(operators, op)
    }

    return operators
}
```

```go
type Operator struct {
    Kind    OpKind
    Operand int
}


func (p *Parser) Parse(tokens []Token) []Operator {
    operators := make([]Operator, 0, len(tokens))

    for _, token := range tokens {
        var op Operator

        switch token.Kind() {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow, OpDot, OpComma:
            op = Operator{
                Kind:    token.Kind(),
                Operand: len(token),
            }

        .
        .
        .

        operators = append(operators, op)
    }

    return operators
}
```

```go
type Operator struct {
    Kind    OpKind
    Operand int
}


func (p *Parser) Parse(tokens []Token) []Operator {
    operators := make([]Operator, 0, len(tokens))

    for _, token := range tokens {
        var op Operator

        switch token.Kind() {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow, OpDot, OpComma:
            op = Operator{
                Kind:    token.Kind(),
                Operand: len(token),
            }

        .
        .
        .

        operators = append(operators, op)
    }

    return operators
}
```

```go
type Operator struct {
    Kind    OpKind
    Operand int
}


func (p *Parser) Parse(tokens []Token) []Operator {
    operators := make([]Operator, 0, len(tokens))

    for _, token := range tokens {
        var op Operator

        switch token.Kind() {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow, OpDot, OpComma:
            op = Operator{
                Kind:    token.Kind(),
                Operand: len(token),
            }


        .
        .
        .

        operators = append(operators, op)
    }

    return operators
}
```

```go
type Operator struct {
    Kind    OpKind
    Operand int
}


func (p *Parser) Parse(tokens []Token) []Operator {
    operators := make([]Operator, 0, len(tokens))

    for _, token := range tokens {
        var op Operator

        switch token.Kind() {
        case OpPlus, OpMinus, OpLeftArrow, OpRightArrow, OpDot, OpComma:
            op = Operator{
                Kind:    token.Kind(),
                Operand: len(token),
            }

        .
        .
        .

        operators = append(operators, op)
    }

    return operators
}
```

content: <<<>>++--

```go
operator[0] == Operator{
  Kind: OpLeftArrow,
  Operand: 3
}
```

Executor

- The component that actually runs the program

- Walks through the AST nodes and performs the operations they represent

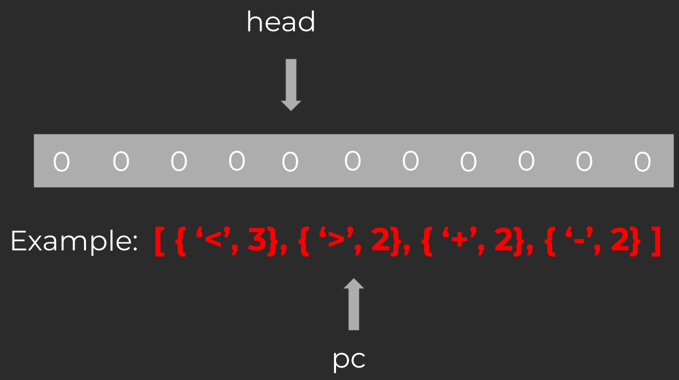
- Responsible for things like:
    - Updating memory
    - Running loops
    - Performing I/O

- **In our case, walks through the list of operators and executes them**

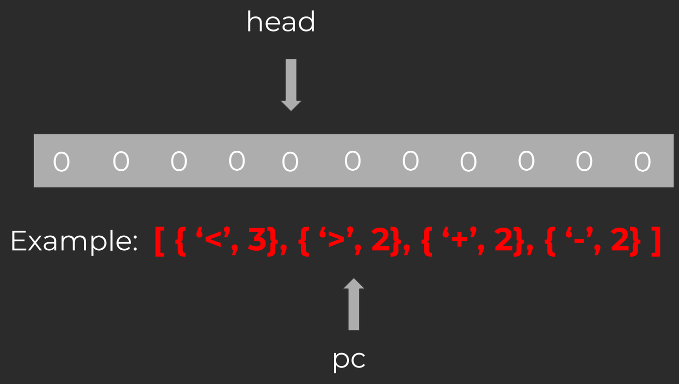
```go
func (i *Interpreter) Execute(ops []Operator) {
	var (
		head int
		pc   int
	)

	memory := make([]byte, i.memory)

	for pc < len(ops) {
		if head < 0 || head >= i.memory {
			log.Fatal("head points to invalid memory address")
		}

		op := ops[pc]

		switch op.Kind {
		case OpPlus:
			memory[head] += byte(op.Operand)
			pc++
		case OpRightArrow:
			head += op.Operand
			pc++
		case OpDot:
			fmt.Printf("%c", memory[head])
			pc++
		.
		.
		.

	}
}
```

```go
func (i *Interpreter) Execute(ops []Operator) {
    var (
        head int
        pc   int
    )

    memory := make([]byte, i.memory)

    for pc < len(ops) {
        if head < 0 || head >= i.memory {
            log.Fatal("head points to invalid memory address")
        }

        op := ops[pc]

        switch op.Kind {
        case OpPlus:
            memory[head] += byte(op.Operand)
            pc++
        case OpRightArrow:
            head += op.Operand
            pc++
        case OpDot:
            fmt.Printf("%c", memory[head])
            pc++
        .
        .
        .

    }
}
```

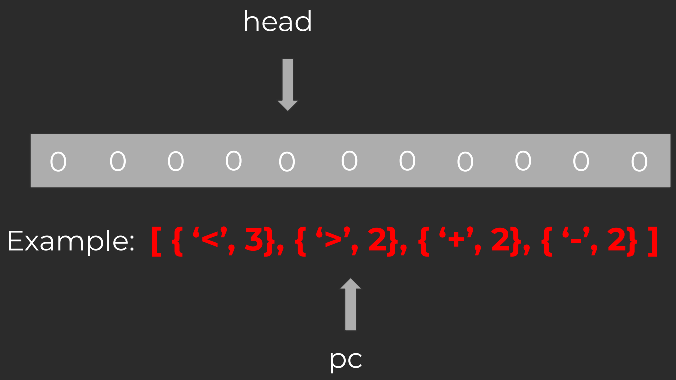
```
func (i *Interpreter) Execute(ops []Operator) {
    var (
        head int
        pc   int
    )

    memory := make([]byte, i.memory)

    for pc < len(ops) {
        if head < 0 || head >= i.memory {
            log.Fatal("head points to invalid memory address")
        }

        op := ops[pc]

        switch op.Kind {
        case OpPlus:
            memory[head] += byte(op.Operand)
            pc++
        case OpRightArrow:
            head += op.Operand
            pc++
        case OpDot:
            fmt.Printf("%c", memory[head])
            pc++
        .
        .
        .

    }
}
```



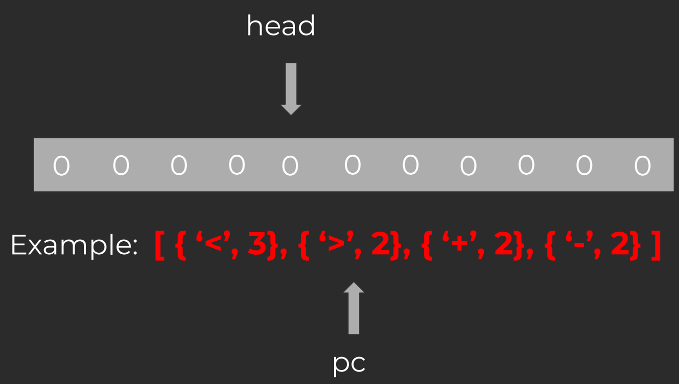
```go
func (i *Interpreter) Execute(ops []Operator) {
    var (
        head int
        pc   int
    )

    memory := make([]byte, i.memory)

    for pc < len(ops) {
        if head < 0 || head >= i.memory {
            log.Fatal("head points to invalid memory address")
        }

        op := ops[pc]

        switch op.Kind {
        case OpPlus:
            memory[head] += byte(op.Operand)
            pc++
        case OpRightArrow:
            head += op.Operand
            pc++
        case OpDot:
            fmt.Printf("%c", memory[head])
            pc++
        .
        .
        .

    }
}
```



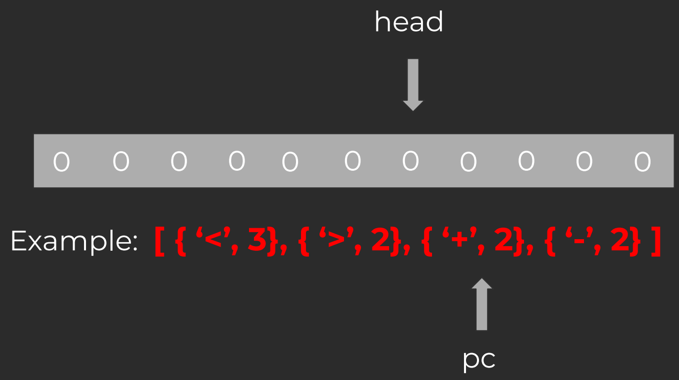
```go
func (i *Interpreter) Execute(ops []Operator) {
    var (
        head int
        pc   int
    )

    memory := make([]byte, i.memory)

    for pc < len(ops) {
        if head < 0 || head >= i.memory {
            log.Fatal("head points to invalid memory address")
        }

        op := ops[pc]

        switch op.Kind {
        case OpPlus:
            memory[head] += byte(op.Operand)
            pc++
        case OpRightArrow:
            head += op.Operand
            pc++
        case OpDot:
            fmt.Printf("%c", memory[head])
            pc++
        .
        .
        .

    }
}
```



```go
func (i *Interpreter) Execute(ops []Operator) {
    var (
        head int
        pc   int
    )

    memory := make([]byte, i.memory)

    for pc < len(ops) {
        if head < 0 || head >= i.memory {
            log.Fatal("head points to invalid memory address")
        }

        op := ops[pc]

        switch op.Kind {
        case OpPlus:
            memory[head] += byte(op.Operand)
            pc++
        case OpRightArrow:
            head += op.Operand
            pc++
        case OpDot:
            fmt.Printf("%c", memory[head])
            pc++
        .
        .
        .
    }
}
```



```go
func (i *Interpreter) Execute(ops []Operator) {
    var (
        head int
        pc   int
    )

    memory := make([]byte, i.memory)

    for pc < len(ops) {
        if head < 0 || head >= i.memory {
            log.Fatal("head points to invalid memory address")
        }

        op := ops[pc]

        switch op.Kind {
        case OpPlus:
            memory[head] += byte(op.Operand)
            pc++
        case OpRightArrow:
            head += op.Operand
            pc++
        case OpDot:
            fmt.Printf("%c", memory[head])
            pc++
        .
        .
        .

    }
}
```



Just-In-Time Compiler

What is a Compiler?

- A compiler is a program that takes your source code and turns it into machine code (an executable)

- That generated executable can be run anytime in the future

- This is also called **“ahead of time”** compilation because the program is fully compiled into machine code before you run it

What is a JIT Compiler?

- Converts code into machine code **while the program is already running**, not ahead of time
- It's usually embedded inside an interpreter
- The interpreter detects "hot paths" in the code and compiles only those to fast native machine code
- Executes the compiled parts **during interpretation**

What is a JIT Compiler?

- Converts code into machine code **while the program is already running**, not

- It's usually em

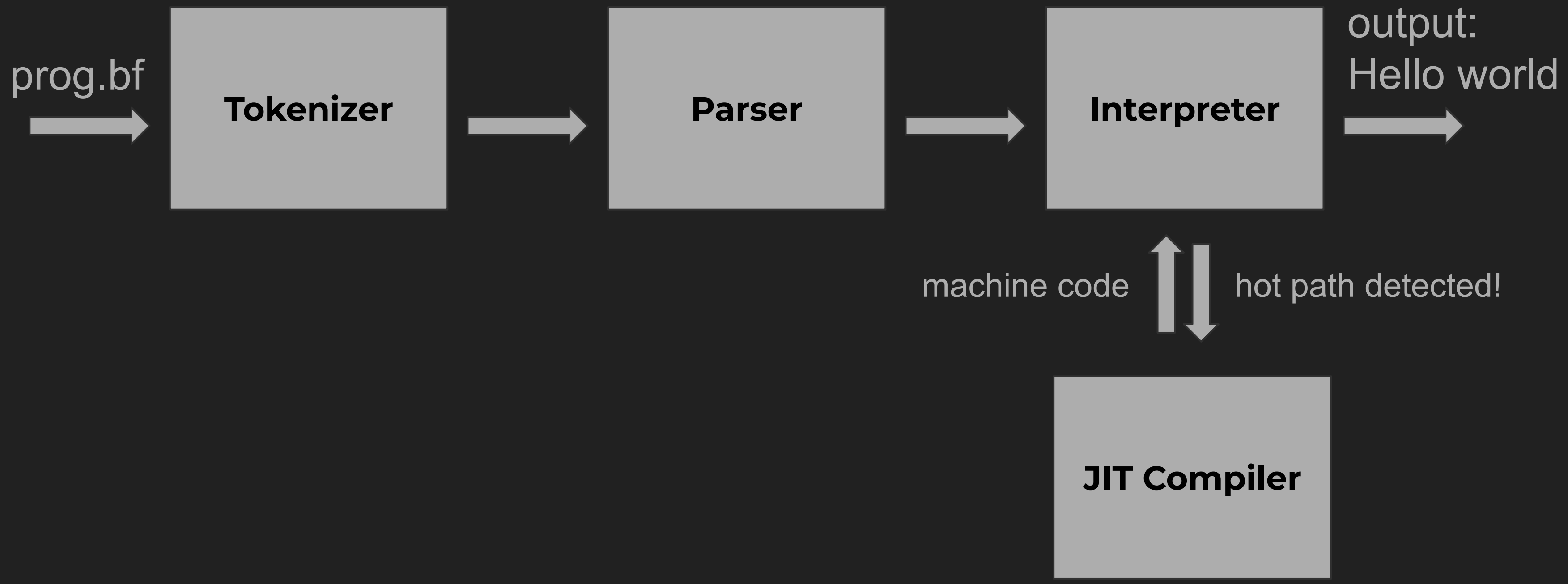
- The interprete mpiles only those to fast native machine code

- Executes the compiled parts **during interpretation**

Why is it better than pure interpretation?

It eliminates the overhead of simulating every instruction one at a time

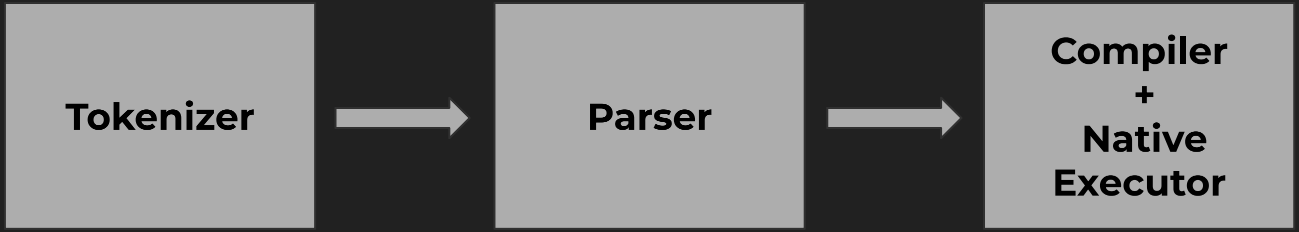
Interpreter + JIT Compiler Pipeline



JIT Compilers in industry

- PyPy (Python)
- Java (HotSpot JVM)
- EBPF

Our JIT Compiler Pipeline



Compiler

- Converts the list of operators to native machine instructions **amd64 x86-64 instruction set in our case**

- Handles other things as well, which we’ll cover later

```go
func (c *Compiler) CompileX86(ops []Operator) []byte {
    var code []byte

    // https://shell-storm.org/online/Online-Assembler-and-Disassembler/
    for _, op := range ops {
        switch op.Kind {
        case OpPlus:
            // ADD BYTE[rax], operand
            code = append(code, 0x80, 0x00, byte(op.Operand))
        case OpMinus:
            // SUB BYTE[rax], operand
            code = append(code, 0x80, 0x28, byte(op.Operand))
        case OpLeftArrow:
            // SUB rax, operand
            code = append(code, 0x48, 0x83, 0xE8, byte(op.Operand))
        case OpRightArrow:
            // ADD rax, operand
            code = append(code, 0x48, 0x83, 0xC0, byte(op.Operand))
        .
        .
        .
    }

    code = append(code, 0xC3) // RET

    return code
}
```

```go
func (c *Compiler) CompileX86(ops []Operator) []byte {
    var code []byte

    // https://shell-storm.org/online/Online-Assembler-and-Disassembler/
    for _, op := range ops {
        switch op.Kind {
        case OpPlus:
            // ADD BYTE[rax], operand
            code = append(code, 0x80, 0x00, byte(op.Operand))
        case OpMinus:
            // SUB BYTE[rax], operand
            code = append(code, 0x80, 0x28, byte(op.Operand))
        case OpLeftArrow:
            // SUB rax, operand
            code = append(code, 0x48, 0x83, 0xE8, byte(op.Operand))
        case OpRightArrow:
            // ADD rax, operand
            code = append(code, 0x48, 0x83, 0xC0, byte(op.Operand))
        .
        .
        .
    }

    code = append(code, 0xC3) // RET

    return code
}
```

```go
func (c *Compiler) CompileX86(ops []Operator) []byte {
    var code []byte

    // https://shell-storm.org/online/Online-Assembler-and-Disassembler/
    for _, op := range ops {
        switch op.Kind {
        case OpPlus:
            // ADD BYTE[rax], operand
            code = append(code, 0x80, 0x00, byte(op.Operand))
        case OpMinus:
            // SUB BYTE[rax], operand
            code = append(code, 0x80, 0x28, byte(op.Operand))
        case OpLeftArrow:
            // SUB rax, operand
            code = append(code, 0x48, 0x83, 0xE8, byte(op.Operand))
        case OpRightArrow:
            // ADD rax, operand
            code = append(code, 0x48, 0x83, 0xC0, byte(op.Operand))
        .
        .
        .
    }

    code = append(code, 0xC3) // RET

    return code
}
```

```
func (c *Compiler) CompileX86(ops []Operator) []byte {
    var code []byte

    // https://shell-storm.org/online/Online-Assembler-and-Disassembler/
    for _, op := range ops {
        switch op.Kind {
        case OpPlus:
            // ADD BYTE[rax], operand
            code = append(code, 0x80, 0x00, byte(op.Operand))
        case OpMinus:
            // SUB BYTE[rax], operand
            code = append(code, 0x80, 0x28, byte(op.Operand))
        case OpLeftArrow:
            // SUB rax, operand
            code = append(code, 0x48, 0x83, 0xE8, byte(op.Operand))
        case OpRightArrow:
            // ADD rax, operand
            code = append(code, 0x48, 0x83, 0xC0, byte(op.Operand))
        .
        .
        .
    }

    code = append(code, 0xC3) // RET

    return code
}
```

```go
func (c *Compiler) CompileX86(ops []Operator) []byte {
    var code []byte

    // https://shell-storm.org/online/Online-Assembler-and-Disassembler/
    for _, op := range ops {
        switch op.Kind {
        case OpPlus:
            // ADD BYTE[rax], operand
            code = append(code, 0x80, 0x00, byte(op.Operand))
        case OpMinus:
            // SUB BYTE[rax], operand
            code = append(code, 0x80, 0x28, byte(op.Operand))
        case OpLeftArrow:
            // SUB rax, operand
            code = append(code, 0x48, 0x83, 0xE8, byte(op.Operand))
        case OpRightArrow:
            // ADD rax, operand
            code = append(code, 0x48, 0x83, 0xC0, byte(op.Operand))
        .
        .
        .
    }

    code = append(code, 0xC3) // RET

    return code
}
```

```
func (c *Compiler) CompileX86(ops []Operator) []byte {
    var code []byte

    // https://shell-storm.org/online/Online-Assembler-and-Disassembler/
    for _, op := range ops {
        switch op.Kind {
        case O
            //
            co
        case O
            //
            co
        case O
            //
            co
        case O
            // ADD rax, operand
            code = append(code, 0x48, 0x83, 0xC0, byte(op.Operand))
        .
        .
        .
    }

    code = append(code, 0xC3) // RET

    return code
}
```

For now let's assume that **rax** holds our head pointer

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

```
        .
        .
        .
        case OpLeftBracket:
            // MOV bl, BYTE PTR[rax]
            code = append(code, 0x8A, 0x18)

            // cmp bl, 0
            code = append(code, 0x80, 0xFB, 0x00)

            // je <relative offset to the command after the corresponding closing bracket>
            code = append(code, 0x0F, 0x84, ?, ?, ?, ?)
        case OpRightBracket:
            // MOV bl, BYTE PTR[rax]
            code = append(code, 0x8A, 0x18)

            // cmp bl, 0
            code = append(code, 0x80, 0xFB, 0x00)

            // jne <relative offset to the command after the corresponding opening bracket>
            code = append(code, 0x0F, 0x85, ?, ?, ?, ?)
        .
        .
        .
```

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

```
.
.
case OpLeftBracket:
    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <relative offset to the command after the corresponding closing bracket>
    code = append(code, 0x0F, 0x84, ?, ?, ?, ?)
case OpRightBracket:
    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset to the command after the corresponding opening bracket>
    code = append(code, 0x0F, 0x85, ?, ?, ?, ?)
.
.
.
```

code

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

```
.
.
.
case OpLeftBracket:
    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <relative offset to the command after the corresponding closing bracket>
    code = append(code, 0x0F, 0x84, ?, ?, ?, ?)
case OpRightBracket:
    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset to the command after the corresponding opening bracket>
    code = append(code, 0x0F, 0x85, ?, ?, ?, ?)
.
.
.
```

code

MOV bl, BYTE PTR[rax]

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

```
.
.
.
case OpLeftBracket:
    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <relative offset to the command after the corresponding closing bracket>
    code = append(code, 0x0F, 0x84, ?, ?, ?, ?)
case OpRightBracket:
    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset to the command after the corresponding opening bracket>
    code = append(code, 0x0F, 0x85, ?, ?, ?, ?)
.
.
.
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

[Jump to the command after the matching] command if cell is 0

] Jump back to the command after the matching [command if cell is not 0

```
    .
    .
    .
    case OpLeftBracket:
        // MOV bl, BYTE PTR[rax]
        code = append(code, 0x8A, 0x18)

        // cmp bl, 0
        code = append(code, 0x80, 0xFB, 0x00)

        // je <relative offset to the command after the corresponding closing bracket>
        code = append(code, 0x0F, 0x84, ?, ?, ?, ?)
    case OpRightBracket:
        // MOV bl, BYTE PTR[rax]
        code = append(code, 0x8A, 0x18)

        // cmp bl, 0
        code = append(code, 0x80, 0xFB, 0x00)

        // jne <relative offset to the command after the corresponding opening bracket>
        code = append(code, 0x0F, 0x85, ?, ?, ?, ?)
    .
    .
    .
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je ????

How can we determine the **relative offset** before the target code has been emitted?

Backpatching

What is Backpatching?

- a technique used when you need to fill in jump offsets or addresses that aren’t known yet
- Steps involved:
    - Emit a jump instruction with a placeholder offset (e.g., 0)
    - Record the location of the placeholder
    - Later, when the target is known, go back and patch the jump’s offset with the correct value

Backpatching

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

Backpatching

```
type bp struct {
	// next instruction position
	np int32
	// placeholder position
	px int32
}
```

```go
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.p = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    p int32
}
```

code

```go
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.px = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    px int32
}
```

code

MOV bl, BYTE PTR[rax]

```go
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.px = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    px int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

```
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.px = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    px int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pX int32
}
```

```go
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.pX = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je _

```
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.pp = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

```go
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.pp = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

```
case OpLeftBracket:
    var backPatch bp

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // je <offset>
    code = append(code, 0x0F, 0x84)

    backPatch.pp = int32(len(code))

    code = append(code, 0x00, 0x00, 0x00, 0x00)

    backPatch.np = int32(len(code))

    c.stack.Push(backPatch)
```

```
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

```
MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

```
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pX)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pX int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne _ _ _ _

<next instruction>

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne ? ? ? ?

<next instruction>

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne ? ? ? ?

<next instruction>

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne ? ? ? ?

<next instruction>

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne ? ? ? ?

<next instruction>

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne <(-)relative>

<next instruction>

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne <(-)relative>

<next instruction>

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne <(-)relative>

<next instruction>

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne <(-)relative>

<next instruction>

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    pp int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.pp)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je 0000

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne <(-)relative>

<next instruction>

```go
type bp struct {
    // next instruction position
    np int32
    // placeholder position
    p int32
}
```

```go
case OpRightBracket:
    backPatch, ok := c.stack.Pop()
    if !ok {
        log.Fatal("compiler: unbalanced brackets")
    }

    // MOV bl, BYTE PTR[rax]
    code = append(code, 0x8A, 0x18)

    // cmp bl, 0
    code = append(code, 0x80, 0xFB, 0x00)

    // jne <relative offset>
    code = append(code, 0x0F, 0x85)

    np := int32(len(code) + 4)
    relative := int32toLittleEndian(backPatch.np - np)
    code = append(code, relative...)

    // back patch
    cp := int32(len(code))
    relative = int32toLittleEndian(cp - backPatch.np)
    for i := 0; i < 4; i++ {
        code[int(backPatch.p)+i] = relative[i]
    }
```

code

MOV bl, BYTE PTR[rax]

cmp bl, 0

je <(+)relative>

<next instruction>

...

...

...

MOV bl, BYTE PTR[rax]

cmp bl, 0

jne <(-)relative>

<next instruction>

Native Executor

- Maps the emitted instructions into an executable memory region
- Sets up additional program context (in our case, allocates memory for the memory tape)
- Runs the program

How do we create an executable memory region?

mmap syscall

mmap

- System call in Unix-like operating systems
- Maps a file, device, or anonymous region directly into your program's memory space
- Once mapped, you can read/write the memory as if it were a normal array
- The mapped memory can be marked as executable, allowing you to run code stored there

```go
func createProgram(instructions []byte) func(pointer *byte) {
    code, err := syscall.Mmap(-1, 0, len(instructions),
        syscall.PROT_EXEC|syscall.PROT_WRITE|syscall.PROT_READ,
        syscall.MAP_PRIVATE|syscall.MAP_ANON)
    if err != nil {
        panic(err)
    }

    copy(code, instructions)

    codePtr := &code

    return *(*func(pointer *byte))(unsafe.Pointer(&codePtr))
}
```

```go
func createProgram(instructions []byte) func(pointer *byte) {
    code, err := syscall.Mmap(-1, 0, len(instructions),
        syscall.PROT_EXEC|syscall.PROT_WRITE|syscall.PROT_READ,
        syscall.MAP_PRIVATE|syscall.MAP_ANON)
    if err != nil {
        panic(err)
    }

    copy(code, instructions)

    codePtr := &code

    return *(*func(pointer *byte))(unsafe.Pointer(&codePtr))
}
```

```go
func createProgram(instructions []byte) func(pointer *byte) {
    code, err := syscall.Mmap(-1, 0, len(instructions),
        syscall.PROT_EXEC|syscall.PROT_WRITE|syscall.PROT_READ,
        syscall.MAP_PRIVATE|syscall.MAP_ANON)
    if err != nil {
        panic(err)
    }

    copy(code, instructions)

    codePtr := &code

    return *(*func(pointer *byte))(unsafe.Pointer(&codePtr))
}
```

```go
func createProgram(instructions []byte) func(pointer *byte) {
    code, err := syscall.Mmap(-1, 0, len(instructions),
        syscall.PROT_EXEC|syscall.PROT_WRITE|syscall.PROT_READ,
        syscall.MAP_PRIVATE|syscall.MAP_ANON)
    if err != nil {
        panic(err)
    }

    copy(code, instructions)

    codePtr := &code

    return *(*func(pointer *byte))(unsafe.Pointer(&codePtr))
}
```

```go
func createProgram(instructions []byte) func(pointer *byte) {
    code, err := syscall.Mmap(-1, 0, len(instructions),
        syscall.PROT_EXEC|syscall.PROT_WRITE|syscall.PROT_READ,
        syscall.MAP_PRIVATE|syscall.MAP_ANON)
    if err != nil {
        panic(err)
    }

    copy(code, instructions)

    codePtr := &code

    return *(*func(pointer *byte))(unsafe.Pointer(&codePtr))
}
```

```go
func (c *Compiler) Execute(instructions []byte) {
    program := createProgram(instructions)

    memory := make([]byte, c.memory)

    program(&memory[0])
}
```

```
func (c *Compiler) Execute(instructions []byte) {
    program := createProgram(instructions)

    memory := make([]byte, c.memory)

    program(&memory[0])
}
```

```go
func (c *Compiler) Execute(instructions []byte) {
	program := createProgram(instructions)

	memory := make([]byte, c.memory)

	program(&memory[0])
}
```

```go
func (c *Compiler) Execute(instructions []byte) {
    program := createProgram(instructions)

    memory := make([]byte, c.memory)

    program(&memory[0])
}
```

remember **rax**?

```
case OpPlus:
    // ADD BYTE[rax], operand
    code = append(code, 0x80, 0x00, byte(op.Operand))
```

How did I know the memory tape pointer would be in the **rax** register?

```
case OpPlus:
    // ADD BYTE[rax], operand
    code = append(code, 0x80, 0x00, byte(op.Operand))
```

I guessed!

I guessed! Well, partially

Go internal ABI specification

- defines the layout of data in memory and the **conventions for calling between Go functions**

- Function calls pass arguments and results using a **combination of the stack and machine registers**

- Arguments and results are passed in registers whenever possible, because **registers are faster to access than memory**

- Go’s amd64 ABI uses the following sequence of 9 registers for integer arguments and results: **RAX**, RBX, RCX, RDI, RSI, R8, R9, R10, R11

What We've Learned

- The existence of the bizarre Brainfuck language
- How basic Interpreters and JIT Compilers work under the hood
- How to create an executable memory region using the mmap syscall
- Go's internal ABI

Special thanks to
youtube.com/@TsodingDaily

source code:

Thank you! Questions?