Building Interpreters

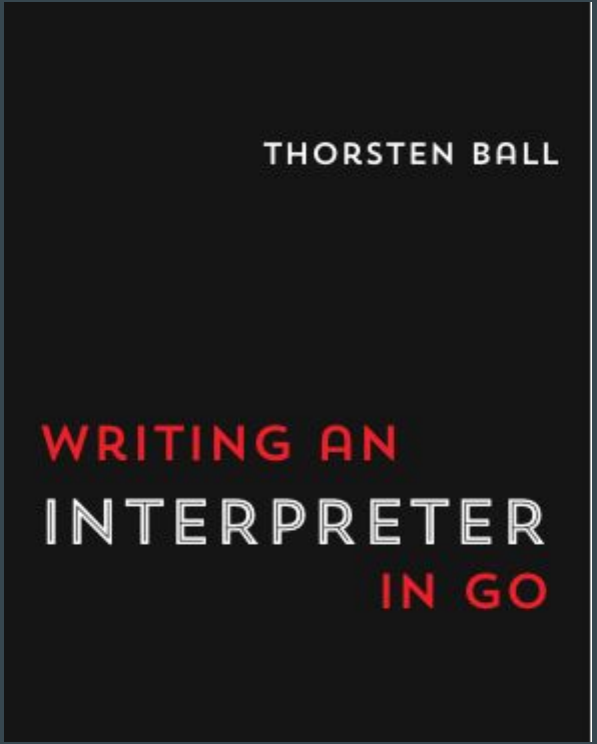
...

Anunay Inuganti

@ianunay

@I_Anunay

https://interpreterbook.com/



The official Monkey logo

What?

- Practical: implementation / no theory
- Different steps involved

Why?

- Design your own language & bring it to life
- Used in many day to day tools
- Deeper understanding of languages
- Build your own sandboxed environments
- Build your own DSL (Domain Specific Language)

```
// Bind values to names with let-statements
let version = 1;
let name = "Monkey programming language";
let myArray = [1, 2, 3, 4, 5];
let coolBooleanLiteral = true;

// Use expressions to produce values
let awesomeValue = (10 / 2) * 5 + 30;
let arrayWithValues = [1 + 1, 2 * 2, 3];
```

```
// Define a `fibonacci` function
let fibonacci = fn(x) {
  if (x == 0) {
    0                // Monkey supports implicit returning of values
  } else {
    if (x == 1) {
      return 1;      // ... and explicit return statements
    } else {
      fibonacci(x - 1) + fibonacci(x - 2); // Recursion! Yay!
    }
  }
};
```

```
// Here is an array containing two hashes, that use strings as keys and integers
// and strings as values
let people = [{"name": "Anna", "age": 24}, {"name": "Bob", "age": 99}];

// Getting elements out of the data types is also supported.
// Here is how we can access array elements by using index expressions:
fibonacci(myArray[4]);
// => 5

// We can also access hash elements with index expressions:
let getName = fn(person) { person["name"]; };

// And here we access array elements and call a function with the element as
// argument:
getName(people[0]); // => "Anna"
getName(people[1]); // => "Bob"
```

```
// Define the higher-order function `map`, that calls the given function `f`
// on each element in `arr` and returns an array of the produced values.
let map = fn(arr, f) {
  let iter = fn(arr, accumulated) {
    if (len(arr) == 0) {
      accumulated
    } else {
      iter(rest(arr), push(accumulated, f(first(arr))));
    }
  };

  iter(arr, []);
};

// Now let's take the `people` array and the `getName` function from above and
// use them with `map`.
map(people, getName); // => ["Anna", "Bob"]
```

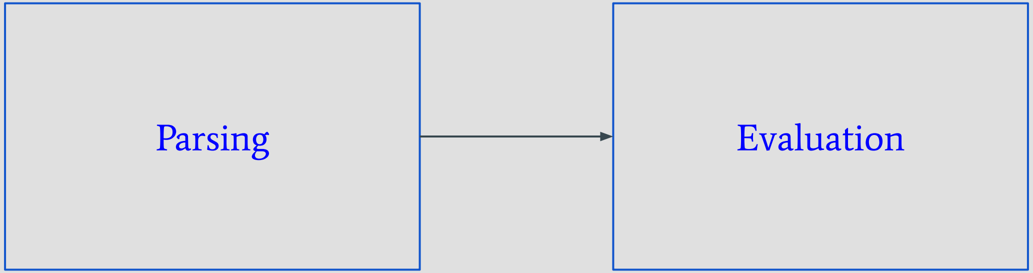
```
// newGreeter returns a new function, that greets a `name` with the given
// `greeting`.
let newGreeter = fn(greeting) {
  // `puts` is a built-in function we add to the interpreter
  return fn(name) { puts(greeting + " " + name); }
};

// `hello` is a greeter function that says "Hello"
let hello = newGreeter("Hello");

// Calling it outputs the greeting:
hello("dear, future Reader!"); // => Hello dear, future Reader!
```

Monkey has a C-like syntax, supports **variable bindings**, **prefix** and **infix** operators, has **first-class and higher-order functions**, can handle **closures** with ease and has **integers**, **booleans**, **arrays** and **hashes** built-in.

Interpretation Process



Interpretation Process

Tokenization

```
let version = 5;

let add = fn(x, y){
    x + y;
};
```

Tokenizer

```
{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "version" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.INT, literal: "5" },
{ type: tokens.SEMICOLON, literal: ";" },


{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "add" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.FUNCTION, literal: "fn" },
{ type: tokens.LPAREN, literal: "(" },
...
```

Approach

position

1 + 2 + 3

read position

```
{ type: tokens.INT, literal: "1" },
{ type: tokens.PLUS, literal: "+" },
{ type: tokens.INT, literal: "2" },
{ type: tokens.PLUS, literal: "+" },
{ type: tokens.INT, literal: "3" },
```

Tokens:

IDENT, INT, STRING ...

Operators:

ASSIGN: "=", PLUS: "+",
MINUS: "-", BANG: "!",
ASTERISK: "*", SLASH: "/" ...

Keywords:

FUNCTION, LET, TRUE,
FALSE, IF, ELSE, RETURN,

Approach

position

1 + 2 + 3

read position

```
{ type: tokens.INT, literal: "1" },
{ type: tokens.PLUS, literal: "+" },
{ type: tokens.INT, literal: "2" },
{ type: tokens.PLUS, literal: "+" },
{ type: tokens.INT, literal: "3" },
```

Tokens:

IDENT, INT, STRING ...

Operators:

ASSIGN: "=", PLUS: "+",
MINUS: "-", BANG: "!",
ASTERISK: "*", SLASH: "/" ...

Keywords:

FUNCTION, LET, TRUE,
FALSE, IF, ELSE, RETURN,

Approach

position

```
{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "version" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.INT, literal: "5" },
{ type: tokens.SEMICOLON, literal: ";" }
```

```
let version = 5;
```

read
position

Tokens:

IDENT, INT, STRING ...

Operators:

ASSIGN: "=", PLUS: "+",
MINUS: "-", BANG: "!",
ASTERISK: "*", SLASH: "/" ...

Keywords:

FUNCTION, LET, TRUE,
FALSE, IF, ELSE, RETURN,

```
export default class Lexer {
  input: string;
  position: number;
  readPosition: number;
  ch: string | null;

  constructor(input: string) {
    this.input = input;
    this.position = 0;
    this.readPosition = 0;
    this.ch = "";

    this.readChar();
  }

  readChar() {
    if (this.readPosition >= this.input.length) {
      this.ch = null;
    } else {
      this.ch = this.input[this.readPosition];
    }
    this.position = this.readPosition;
    this.readPosition += 1;
  }

  nextToken(): Token {
    let tok: Token;

    this.skipWhiteSpace();

    switch (this.ch) {
      case "=":
        if (this.peakChar() === "=") {
          this.readChar();
          tok = { type: tokens.EQ, literal: "==" };
        } else {
          tok = { type: tokens.ASSIGN, literal: this.ch };
        }
        break;
```

Pointers

Parser calls nextToken()

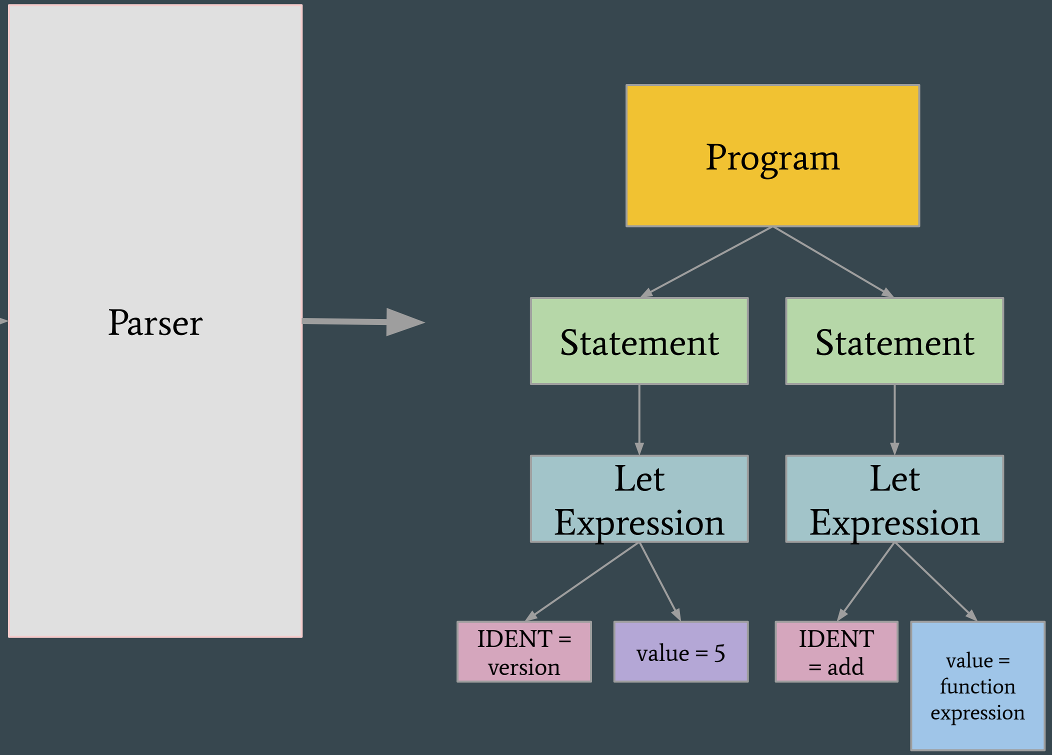
Skip white space

Parsing

Abstract
Syntax
Tree

```
{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "version" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.INT, literal: "5" },
{ type: tokens.SEMICOLON, literal: ";" },

{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "add" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.FUNCTION, literal: "fn" },
{ type: tokens.LPAREN, literal: "(" },
```



Parser Generators

```
statement
    : block
    | variableStatement
    | importStatement
    | exportStatement
    | emptyStatement_
    | classDeclaration
    | functionDeclaration
    | expressionStatement
    | ifStatement
    | iterationStatement
    | continueStatement
    | breakStatement
    | returnStatement
    | yieldStatement
    | withStatement
    | labelledStatement
    | switchStatement
    | throwStatement
    | tryStatement
    | debuggerStatement
    ;

block
    : '{' statementList? '}'
    ;

statementList
```

- generate a parser based on grammar
- yacc, bison, ANTLR, tree-sitter, jison

part of ANTLR Context Free Grammar for ECMAScript

Parser Generator in Skipper

```
% cat complicated_example.eskip
hostHeaderMatch:
         Host("^skipper.teapot.org$")
         -> setRequestHeader("Authorization", "Basic YWRtaW46YWRtaW5zcGFzc3dvcmQK")
         -> "https://target-to.auth-with.basic-auth.enterprise.com";
baiduPathMatch:
        Path("/baidu")
        -> setRequestHeader("Host", "www.baidu.com")
        -> setPath("/s")
        -> setQuery("wd", "godoc skipper")
        -> "http://www.baidu.com";
googleWildcardMatch:
        *
        -> setPath("/search")
        -> setQuery("q", "godoc skipper")
        -> "https://www.google.com";
yandexWildacardIfCookie:
        * && Cookie("yandex", "true")
        -> setPath("/search/")
        -> setQuery("text", "godoc skipper")
        -> tee("http://127.0.0.1:12345/")
        -> "https://yandex.ru";
```

eskip - **D**omain **S**pecific **L**anguage

Parser Generator in Skipper

```
predicate:
        any {
                $$.predicate = &Predicate{"*", nil}
        }
        |
        symbol openparen args closeparen {
                $$.predicate = &Predicate{$1.token, $3.args}
                $3.args = nil
        }

filters:
        filter {
                $$.filters = []*Filter{$1.filter}
        }
        |
        filters arrow filter {
                $$.filters = $1.filters
                $$.filters = append($$.filters, $3.filter)
        }

filter:
        symbol openparen args closeparen {
                $$.filter = &Filter{
                        Name: $1.token,
                        Args: $3.args}
                $3.args = nil
        }

args:
        |
        arg {
                $$.args = []interface{}{$1.arg}
        }
        |
```

https://github.com/zalando/skipper/blob/master/eskip/parser.y

Parser Approach

Current Token

Peek Token

```
{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "version" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.INT, literal: "5" },
{ type: tokens.SEMICOLON, literal: ";" },

{ type: tokens.LET, literal: "let" },
{ type: tokens.IDENT, literal: "add" },
{ type: tokens.ASSIGN, literal: "=" },
{ type: tokens.FUNCTION, literal: "fn" },
{ type: tokens.LPAREN, literal: "(" },
```

Statements:

LetStatement,
ReturnStatement,
ExpressionStatement ...

Literals:

IntegerLiteral,
StringLiteral,
ArrayLiteral ...

Expressions:

PrefixExpression,
InfixExpression,
IfExpression ...

Expression Parsing

Expression

```
let version = 3 + 5 * 4;
```

❌

(3 + 5) * 4

✅

3 + (5 * 4)

Pratt parsing

Top Down Operator
Precedence

Vaughan R. Pratt
Massachusetts Institute of Technology

1. <u>Survey of the Problem Domain</u>.

There is little agreement on the extent to which syntax should be a consideration in the design and implementation of programming languages. At one extreme, it is considered vital, and one may go to any lengths [Van Wijngaarden 1969, McKeeman 1970] to provide adequate syntactic capabilities. The other extreme is the spartan denial of a need for a rich syntax [Minsky 1970]. In between, we find some language implementers willing to incorporate as much syntax as possible provided they do not have to work hard at it [Wirth 1971].

In this paper we present what should

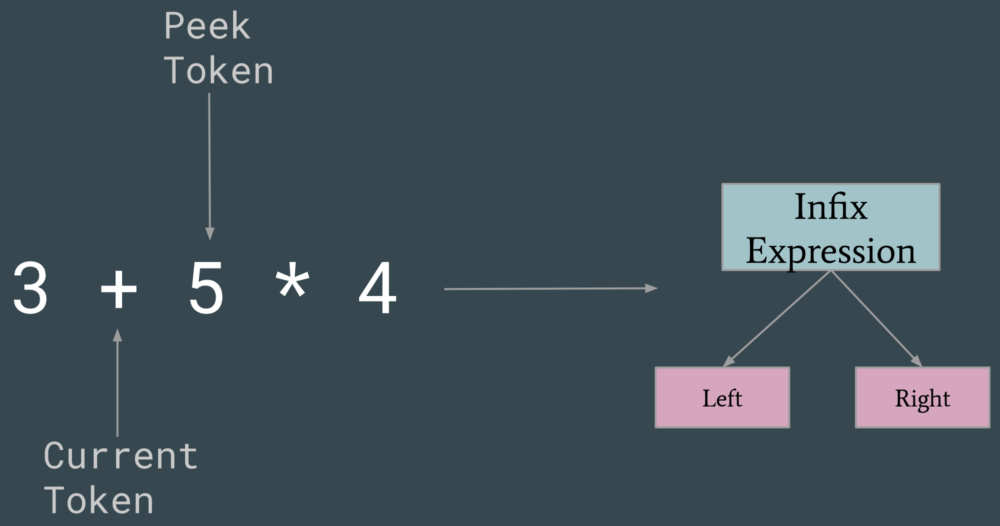
of this kind of oversight is our universal preoccupation with BNF grammars and their various offspring: type 1 [Chomsky 1959], indexed [Aho 1968], macro [Fischer 1968], LR(k) [Knuth 1965], and LL(k) [Lewis 1968] grammars, to name a few of the more prominent ones, together with their related automata and a large body of theorems. I am personally enamored of automata theory per se, but I am not impressed with the extent to which it has so far been successfully applied to the writing of compilers or interpreters. Nor do I see a particularly promising future in this direction. Rather, I see automata theory as holding back the development of ideas valuable to language design that are not visibly in the domain

https://dl.acm.org/doi/pdf/10.1145/512927.512931

Precedence:

```
LOWEST = 1,
EQUALS  = 2, // ==
LESSGREATER = 3, // > or <
SUM = 4, // +
PRODUCT = 5, // *
PREFIX = 6, // -X or !X
CALL = 7, // myFunction(X)
INDEX = 8, // array[index]
```

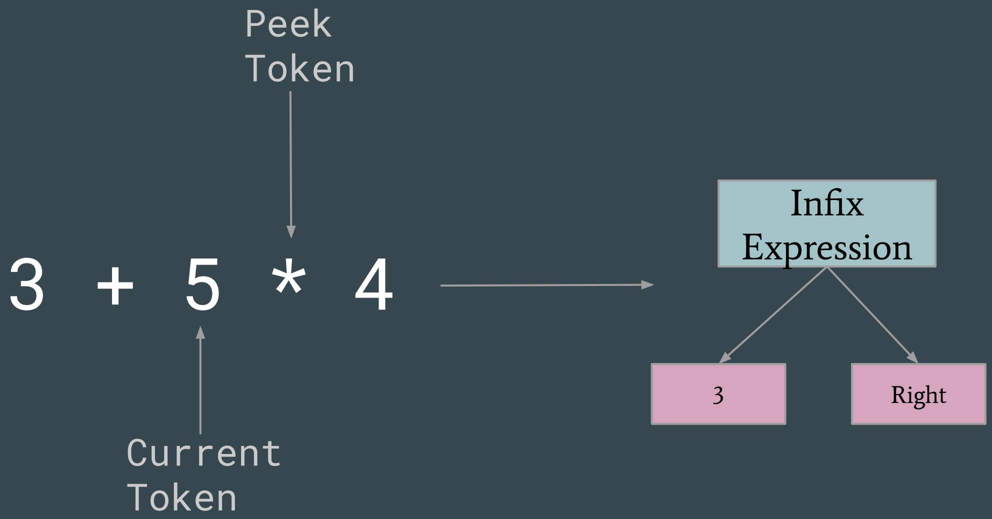
Recursive Descent Parsing



Recursive Descent Parsing

Precedence:

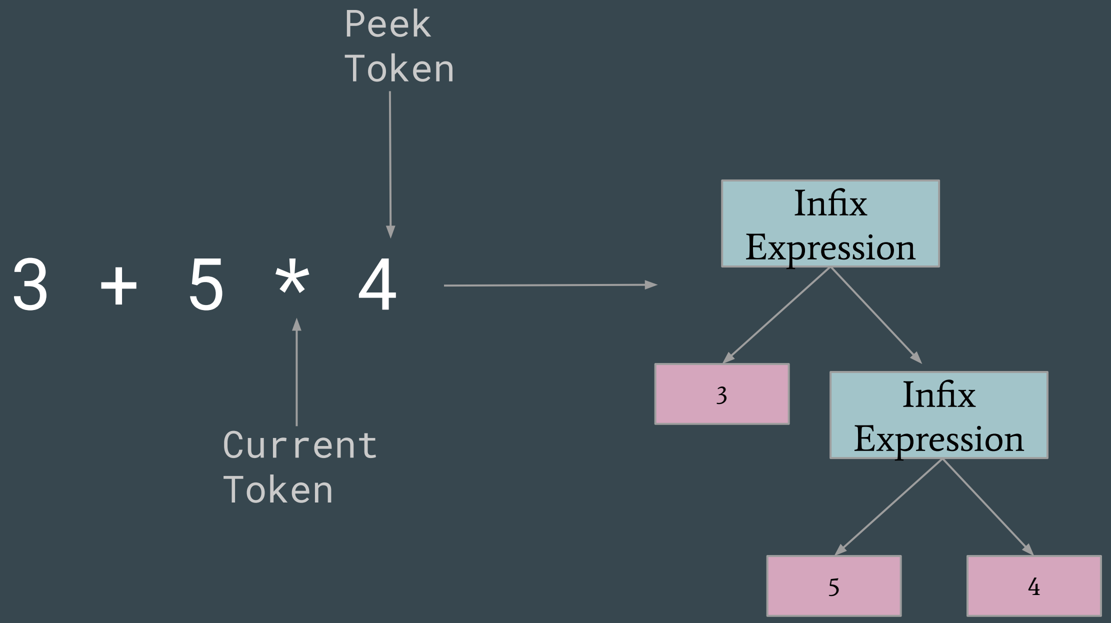
```
LOWEST = 1,
EQUALS  = 2, // ==
LESSGREATER = 3, // > or <
SUM = 4, // +
PRODUCT = 5, // *
PREFIX = 6, // -X or !X
CALL = 7, // myFunction(X)
INDEX = 8, // array[index]
```



Recursive Descent Parsing

Precedence:

```
LOWEST = 1,
EQUALS  = 2, // ==
LESSGREATER = 3, // > or <
SUM = 4, // +
PRODUCT = 5, // *
PREFIX = 6, // -X or !X
CALL = 7, // myFunction(X)
INDEX = 8, // array[index]
```



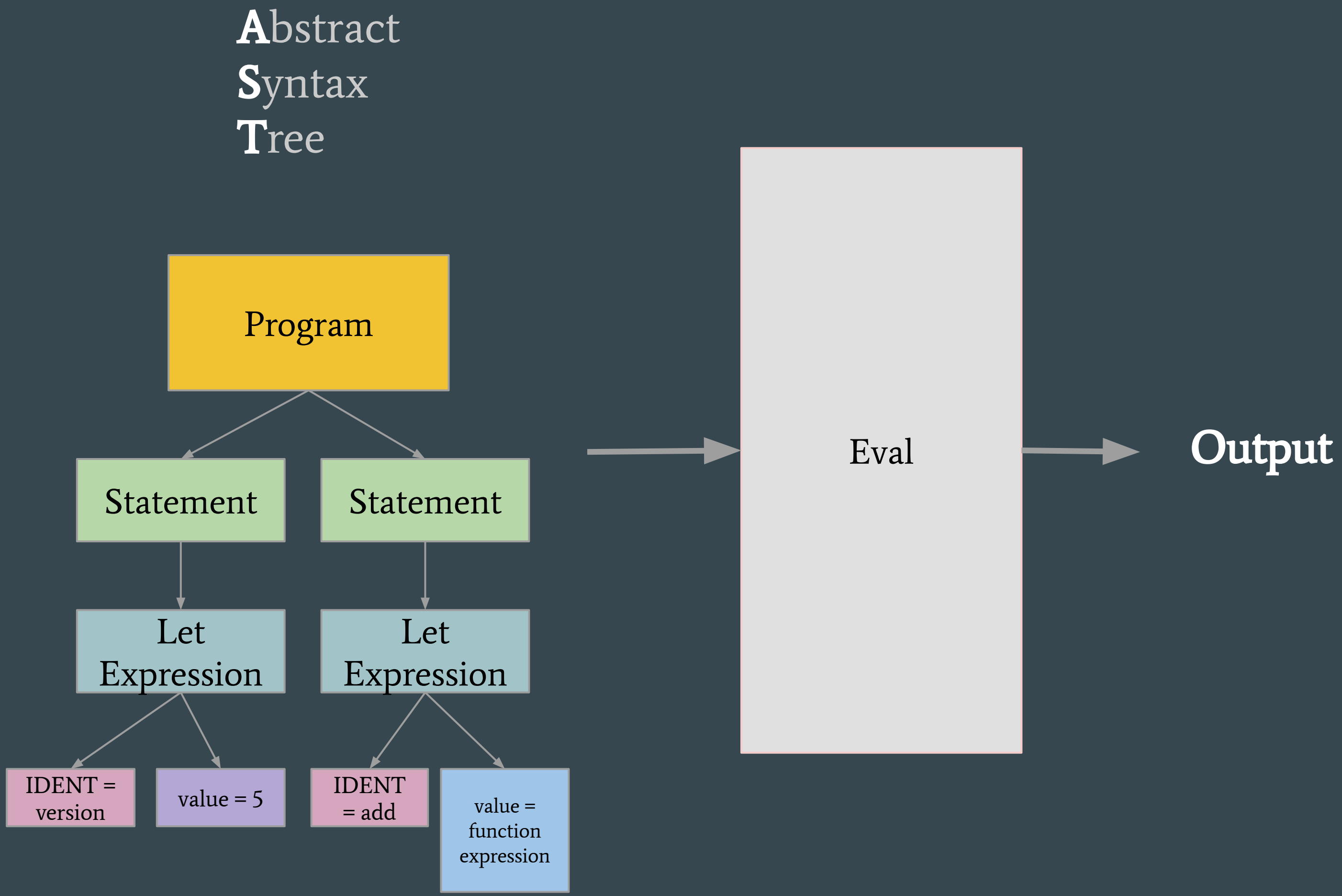
Notes

- AST Explorer https://astexplorer.net/
- Prettier: Pretty prints an AST

Evaluation

- Interpreter / Compiler ??
- JIT compilation
- Tree walking interpreter

Evaluation



Abstract
Syntax
Tree
Program
Statement
Statement
Let Expression
Let Expression
IDENT = version
value = 5
IDENT = add
value = function expression
Eval
Output

Evaluation

Program

```
let number = 5;

let add2 = fn(x){
    x + 2;
};

add2(number)
```

Environment

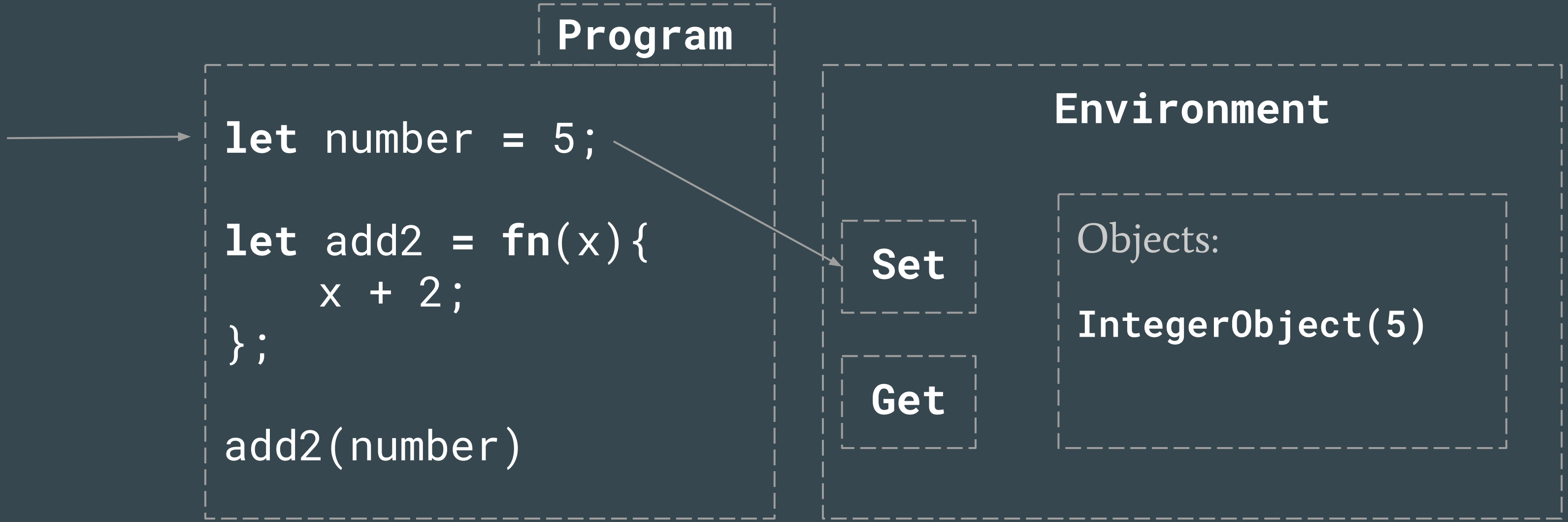
Set

Get

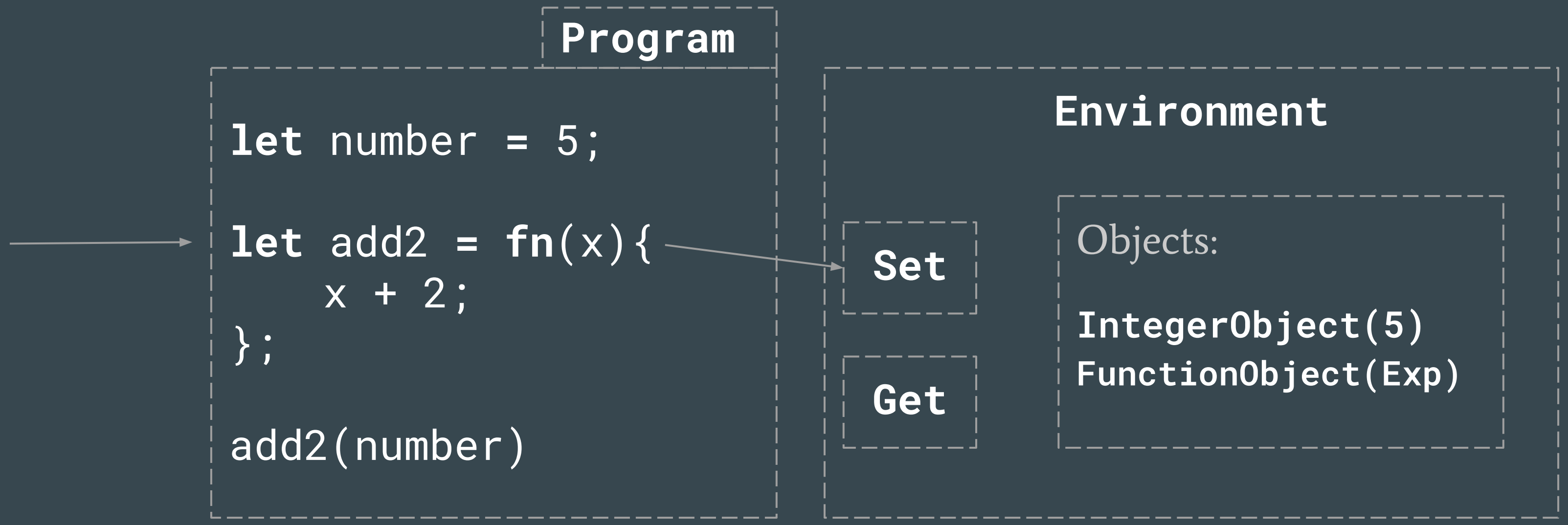
Objects:

`IntegerObject,`
`BooleanObject,`
`FunctionObject ...`

Evaluation


Program
let number = 5;
let add2 = fn(x){
x + 2;
};
add2(number)
Environment
Set
Get
Objects:
IntegerObject(5)

Evaluation


Program
let number = 5;
let add2 = fn(x){
x + 2;
};
add2(number)
Environment
Set
Get
Objects:
IntegerObject(5)
FunctionObject(Exp)

Evaluation

Program

```
let number = 5;

let add2 = fn(x){
    x + 2;
};

add2(number)
```

Environment

Set

Get

Objects:

IntegerObject(5)
FunctionObject(Exp)

Evaluation

Program

```
let init = fn(){
 let name = "Hello ";
 let sayName = fn(){
    name + x;
 };
 sayName()
}

init()
```

Environment

Set

Get

Objects:

IntegerObject(5)
FunctionObject(Exp)

Parent

```
If "IDENT defined in ENV":
 use IDENT

else "IDENT defined in Parent":
 use Parent value

else:
 undefined
```

Code - https://github.com/ianunay/monkey-lang-interpreter-ts

Thank You!

Experience

- Off by one errors