一个简单的解释器

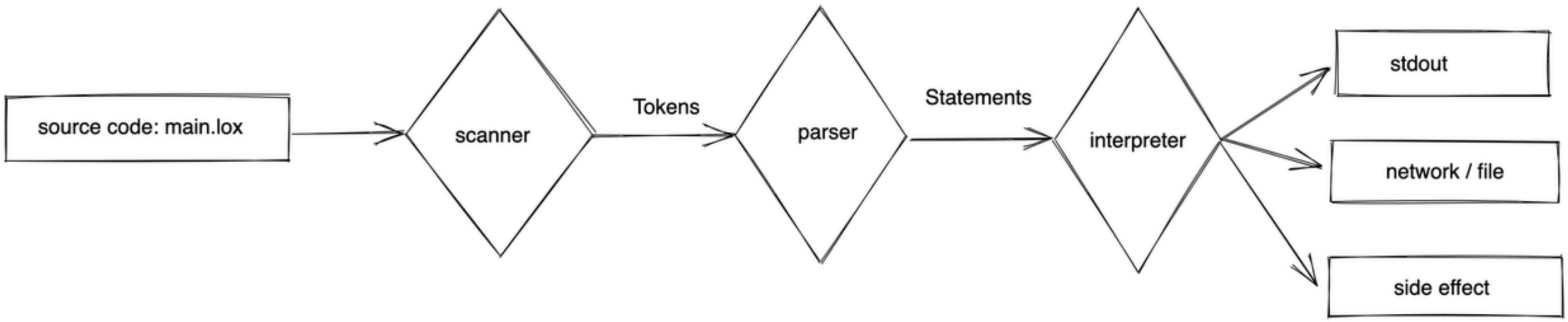
目录

code example

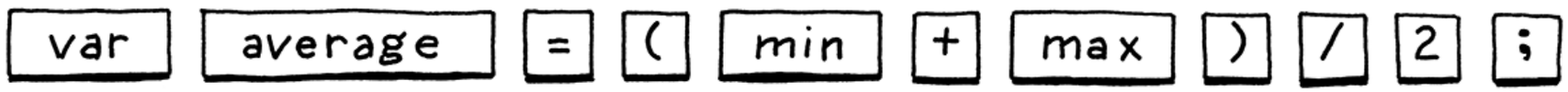
```
var a = 233;
var b = false;
var c = 1;
for (;c<1000;c = c+1) {
    if (a < 2000) {
        c = c + a;
    }
}
```

```
fun fc(arg) {
    {
        return "hello again!";
    }
}


println(c);
fc("hello");
println(fc(""));
```

数据流


source code: main.lox
scanner
Tokens
parser
Statements
interpreter
stdout
network / file
side effect

scanner

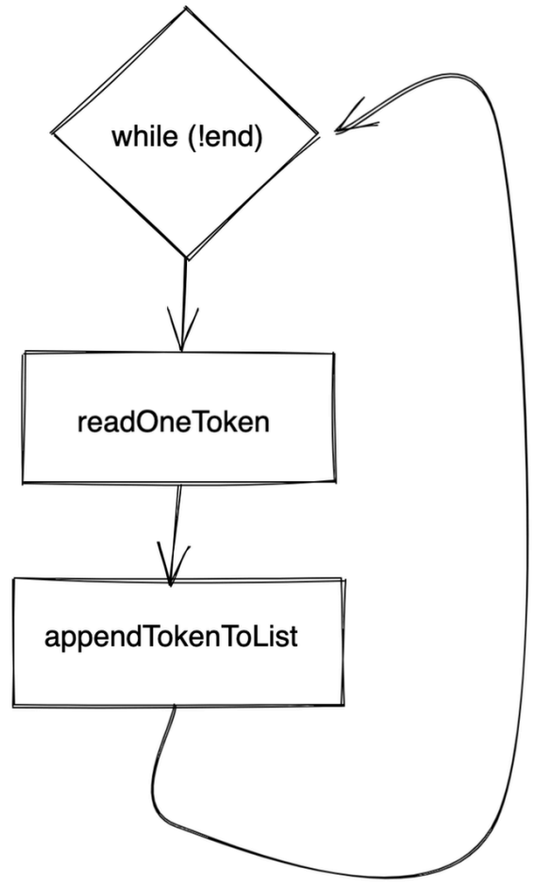



```
type Token struct {
    kind    TokenKind
    lexeme  string
    literal interface{}
    line    int
}
```

```
type TokenKind int

const (
    // single-character tokens
    LEFT_PAREN TokenKind = iota   (
    RIGHT_PAREN                   )
    LEFT_BRACE                    {
    RIGHT_BRACE                   }
    COMMA                         ,
    DOT                           .
    MINUS                         -
    PLUS                          +
    SEMICOLON                     ;
    SLASH                         /
    STAR                          *

    // one or two char tokens
    BANG                          !
    BANG_EQUAL                    !=
    EQUAL                         =
    EQUAL_EQUAL                   ==
    GREATER                       >
    GREATER_EQUAL                 >=
    LESS                          <
    LESS_EQUAL                    <=

    // literals
    IDENTIFIER                    a,b,abc
    STRING                        “hello”
    NUMBER                        123

    // keywords
    AND                           &&
    ELSE                          else
    FALSE                         false
    FUN                           func
    FOR                           for
    IF                            if
…
```

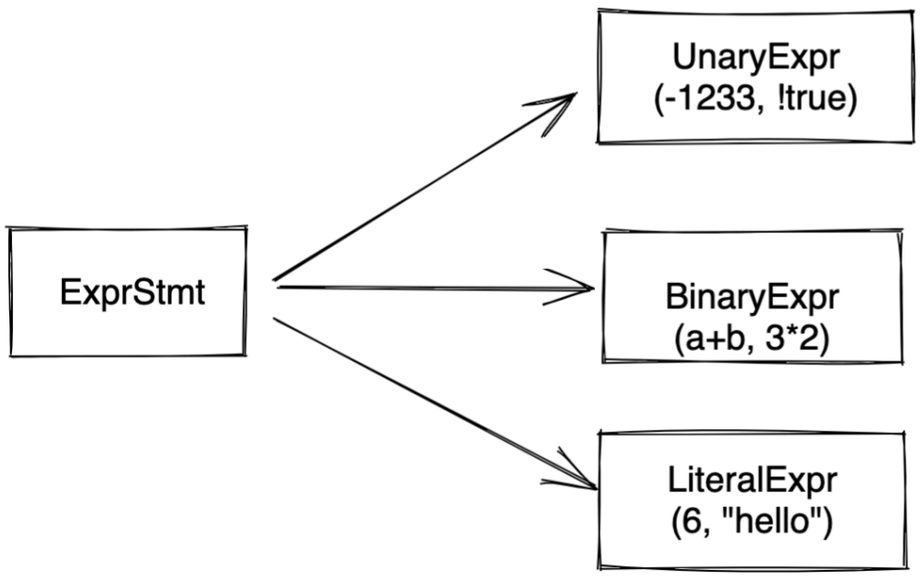
scanner

```
var a = 233;

Token[
   {
      kind: VAR,
      lexeme: "",
      literal: nil,
      line: 1
   },
   {
      kind: IDENTIFIER,
      lexeme: "a",
      literal: nil,
      line: 1
   },
   {
      kind: EQUAL,
      lexeme: "",
      literal: nil,
      line: 1
   },
   {
      kind: NUMBER,
      lexeme: "",
      literal: 233,
      line: 1
   },
   {
      kind: SEMICOLON,
      lexeme: "",
      literal: nil,
      line: 1
   }
]
```



NUMBER => float64
STRING => string

parser

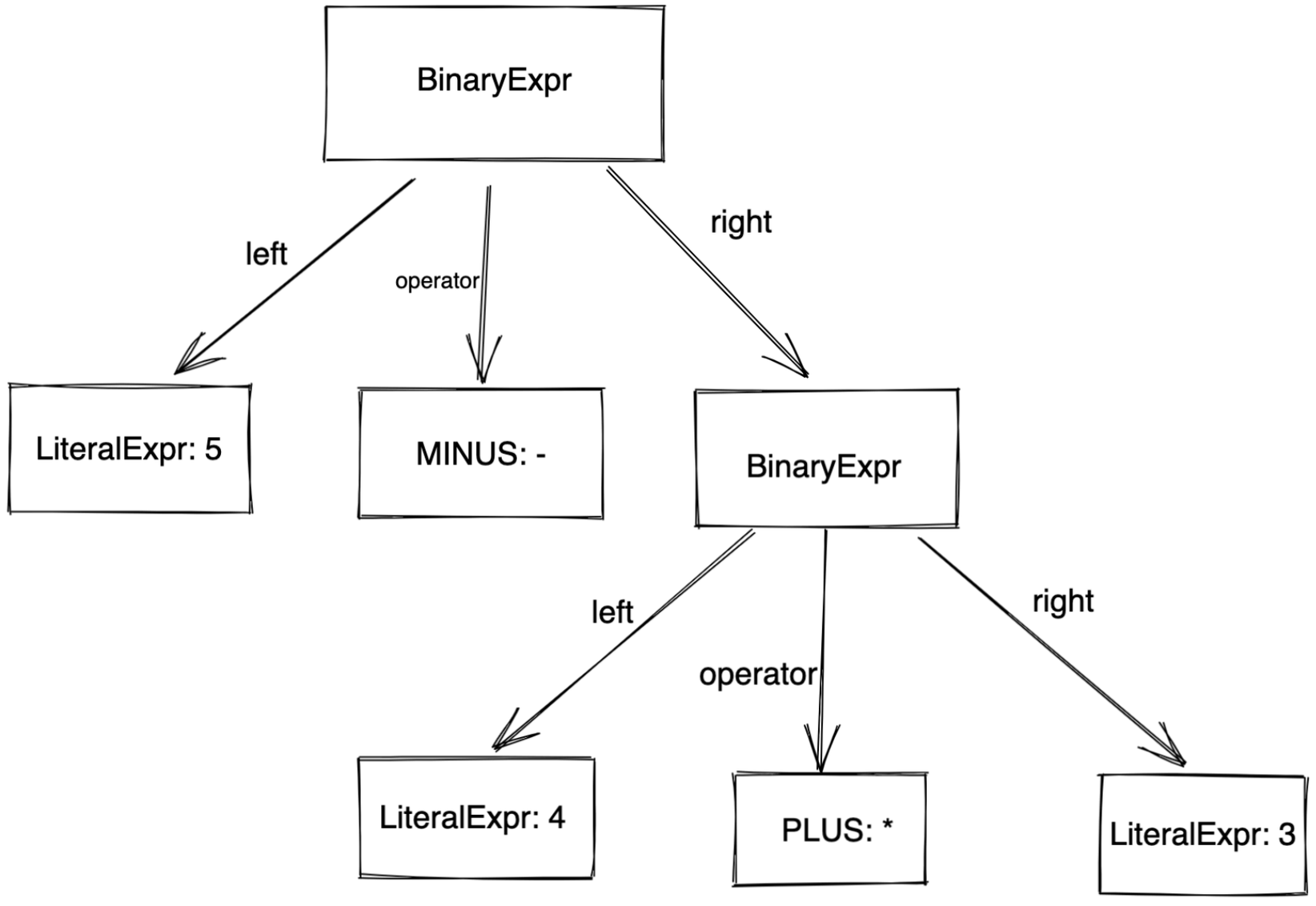



```
type Expr interface{}

type UnaryExpr struct {
    operator Token
    right    Expr
}

type BinaryExpr struct {
    left     Expr
    right    Expr
    operator Token
}

type LiteralExpr struct {
    obj interface{}
}
```

5-4*3

parser

5-4*3


BinaryExpr
left
operator
right
LiteralExpr: 5
MINUS: -
BinaryExpr
left
operator
right
LiteralExpr: 4
PLUS: *
LiteralExpr: 3

parser

巴科斯范式

```
菜     -> 做法? 材料
做法  -> "红烧" | "清蒸" | "水煮"
材料  -> "非常"+"新鲜的鸡块" | "茄子" | "鲈鱼"
主食  -> "米饭" | "面条" | "馒头"
晚饭  -> 菜 ("、" 晚饭)? | 主食
```

```
米饭
红烧茄子、馒头
清蒸鲈鱼、红烧非常非常非常新鲜的鸡块、面条
```

parser

```
expression     → literal
               | unary
               | binary
               | grouping ;
literal        → NUMBER | STRING | "true" | "false" | "nil" ;
grouping       → "(" expression ")" ;
unary          → ( "-" | "!" ) expression ;
binary         → expression operator expression ;

operator       → "==" | "!=" | "<" | "<=" | ">" | ">="
               | "+"  | "-"  | "*" | "/" ;
```

```
expression     → equality ;
equality       → comparison ( ( "!=" | "==" ) comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           → factor ( ( "-" | "+" ) factor )* ;
factor         → unary ( ( "/" | "*" ) unary )* ;
unary          → ( "!" | "-" ) unary
               | primary ;
primary        → NUMBER | STRING | "true" | "false" | "nil"
               | "(" expression ")" ;
```

parser

```
func expression() {
    return equality()
}

func equality() {
    ex = comparison()
    for match("!=" | "==") {
        right = comparison()
        ex = BinaryExpr {
            left: ex,
            right: right,
            op: "!=" | "==",
        }
    }
    return ex
}

....
```

```
func unary() {
    if match("!" | "-") {
        op = "!" | "-"
        ex = primary()
        return UnaryExpr{
            op: op,
            right: ex,
        }
    }
    return primary()
}

func primary() {
    if match(TRUE) {
        return LiteralExpr{obj: true}
    }
    if match(False) {
        ...
    }
    if match(NUMBER, STRING) {
        return LiteralExpr{obj: ex.literal}
    }
}
```

interpreter

```
func evaluate(e) {
    switch v = e.type {
    case UnaryExpr:
        return evaluateUnary(e)
    case BinaryExpr:
        return evaluateBinary(e)
    case LiteralExpr:
        return evaluateLiteral(e)
    }
    return nil
}
```

```
func evaluateLiteral(l) {
    return l.obj
}
```

```
func evaluateUnary(u) {
    right = evaluate(u.right)
    switch u.op{
    case "!":
        return !bool(right)
    case "-":
        return -1*float(right)
    }
    return nil
}
```

```
func evaluateBinary(b) {
    left := evaluate(b.left)
    right := evaluate(b.right)

    switch b.op {
    case "-":
        return float(left) - float(right)
    case "*":
        return float(left) * float(right)
    ....
    case ">=":
        return float(left) >= float(right)
    }
}
```

stmt

```
type Stmt interface{}

type ExprStmt struct {
    expr Expr
}

type VarStmt struct {
    name        Token
    initializer Expr
}
```

var decl

parser

```
func varDecl() {
    name = consume(IDENTIFIER)

    if match(EQUAL) {
        initializer = expression()
    }
    return VarStmt{
        name, initializer
    }
}
```

interpreter

```
type Env struct {
    values map[string]interface{}
}
```

```
func evaluateVarStmt(v VarStmt) {
    var obj interface{}
    if v.initializer != nil {
        obj = evaluate(v.initializer)
    }
    env[v.name.lexeme] = obj
    return nil
}
```

var a = 123;

var assign

parser

```
func assignment() {
    token := consume(VarExpr)
    consume(EQUAL)
    value := equality()
    return AssignExpr{
      name:  token.name,
      value: value,
   }
}
```

interpreter

```
func evaluateAssignExpr(a AssignExpr) {
    value := evaluate(a.value)
    env[a.name] = value
    return value
}
```

if else

parser

```
func ifStmt(){
    consume(IF)
    condition = expression()

    thenBranch = statement()
    var elseBranch Stmt
    if match(ELSE) {
        elseBranch = statement()
    }
    return IfStmt{
        condition:  condition,
        thenBranch: thenBranch,
        elseBranch: elseBranch,
    }
}
```

interpreter

```
func evaluateIfStmt(v IfStmt){
    if evaluate(v.condition).(bool) {
        evaluate(v.thenBranch)
    } else if v.elseBranch != nil {
        evaluate(v.elseBranch)
    }
    return nil
}
```

while

parser

```
func whileStmt() {
    consume(While)
    condition = expression()
    body = statement()

    return WhileStmt{
        condition: condition,
        body: body
    }
}
```

interpreter

```
func evaluateWhileStmt(v WhileStmt){
    for evaluate(v.condition).(bool) {
        evaluate(v.body)
    }
    return nil
}
```

func decl

```
type FuncStmt struct {
    name   Token
    params []Token
    body   []Stmt
}
```

parser

```
func function(kind string) {
    name = consume(IDENTIFIER)
    consume(LEFT_PAREN)
    var params []Token
    if check(RIGHT_PAREN) {
        params = append(params, consume(IDENTIFIER))
        for match(COMMA) {
            params = append(params, consume(IDENTIFIER))
        }
    }
    consume(RIGHT_PAREN)
    consume(LEFT_BRACE)

    body = blockStmt()
    return FuncStmt{
        name:   name,
        params: params,
        body:   body,
    }
}
```

interpreter

```
func evaluateFuncStmt(v FuncStmt) {
    fn = v
    i.env.define(fn.name.lexeme, fn)
    return nil
}
```

func decl

```
type Callalble interface {
    arity() int
    call(i *Interpreter, args []interface{}) interface{}
}
```

```
func (f FuncStmt) arity() int {
    return len(f.params)
}
```

```
func (f FuncStmt) call(args []interface{}) interface{} {
    env := NewEnv(Env)
    for idx, _ := range f.params {
        env.define(f.params[idx].lexeme, args[idx])
    }

    evaluateBlockStmt(BlockStmt{f.body}, env)
    return nil
}
```

func call

var t = a;

var t =a();

parser

```
func finishCall(){
    callee = consume(INDENTIFIER)

    var args []Expr
    consume(LEFT_PAREN)
    if !check(RIGHT_PAREN) {
        args = append(args, expression())
        for match(COMMA) {
            args = append(args, expression())
        }
    }
    consume(RIGHT_PAREN)
    return CallExpr{
        callee: callee,
        paren:  previous(),
        args:   args,
    }
}
```

interpreter

```
func evaluateCallExpr(v CallExpr) {
    callee = evaluate(v.callee)

    var args []interface{}
    for _, a := range v.args {
        args = append(args, evaluate(a))
    }
    if fn, ok := callee.(Callalble); ok {
        if fn.arity() != len(args) {
            panic(fmt.Sprintf("args num not match, require %d,
        }
        return fn.call(i, args)
    } else {
        panic(fmt.Sprintf("invalid fun call at line %d", v.pare
    }
}
```

return

parser

```
func returnStmt(){
    token = previous()
    var value Expr
    if !check(SEMICOLON) {
        value = expression()
    }
    consume(SEMICOLON)
    return ReturnStmt{
        keyword: token,
        value:   value,
    }
}
```

interpreter

```
func evaluateReturnStmt(v ReturnStmt){
    if v.value != nil {
        return ReturnErr{value: evaluate(v.value)}
    }
    return ReturnErr{value: nil}
}

func evaluateBlockStmt(v BlockStmt, e *Env){
    previousEnv = env
    env = e
    var (
        err      ReturnErr
        returnOK bool
    )
    for _, stmt := range v.stmts {
        t = evaluate(stmt)
        if t != nil {
            if err, returnOK = t.(ReturnErr); returnOK {
                break
            }
        }
    }
    env = previousEnv
    if returnOK {
        return err
    }
    return nil
}
```

QA

参考资料

- https://github.com/munificent/craftinginterpreters
- http://craftinginterpreters.com