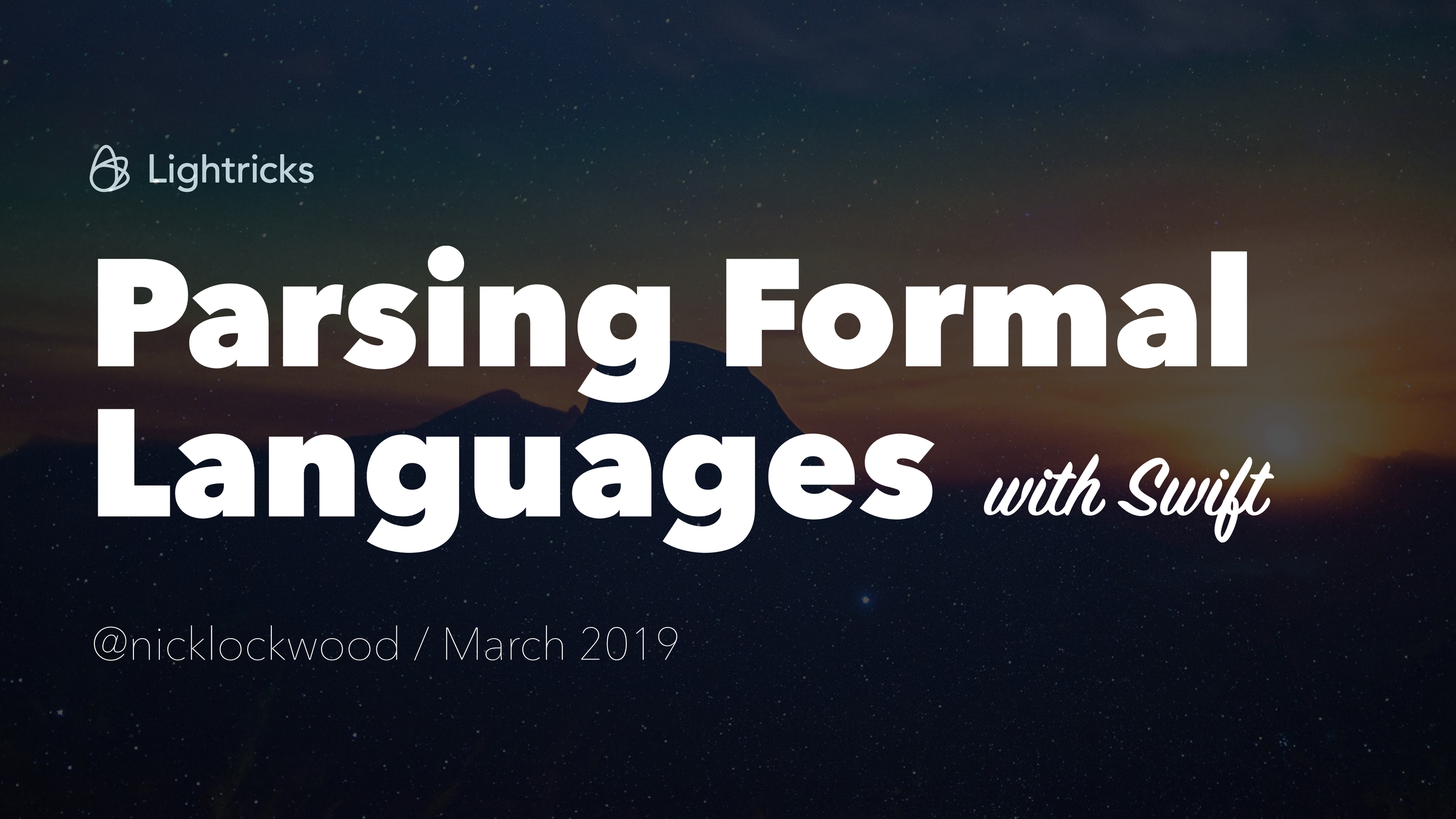

Lightricks
Parsing Formal Languages
with Swift
@nicklockwood / March 2019

What are Formal Languages?

- Text specifically designed to be read by a machine.
- The opposite of natural languages *(NLP)*.
- Markup, programming, or data serialization languages *(HTML, JSON, YAML, CSS, Java, C++, Swift, etc.)*

Why do we need to parse them?

- Linting / formatting.
- Documentation / localized strings extraction.
- Serialization & data interchange.
- DSLs *(for configuration, test harnesses, etc.)*
- Runtime scripting.

Why do it ourselves? Why use Swift?

- Can't we just use a library?
- Isn't it easier to use a scripting language *(e.g. Python or Ruby?*
- Isn't Swift really horrible for string processing?
- What about a parser generator *(e.g. LEX and YACC, or Bison)?*

Slava Pestov
@slava_pestov

Parsing is weird because the theory is intimidating but also useless. A naive hand coded parser will be faster, easier to maintain and produce better error messages than using some fancy parser generator based on an academic formalism like LR(k) or whatever

11:13 PM - Aug 25, 2018

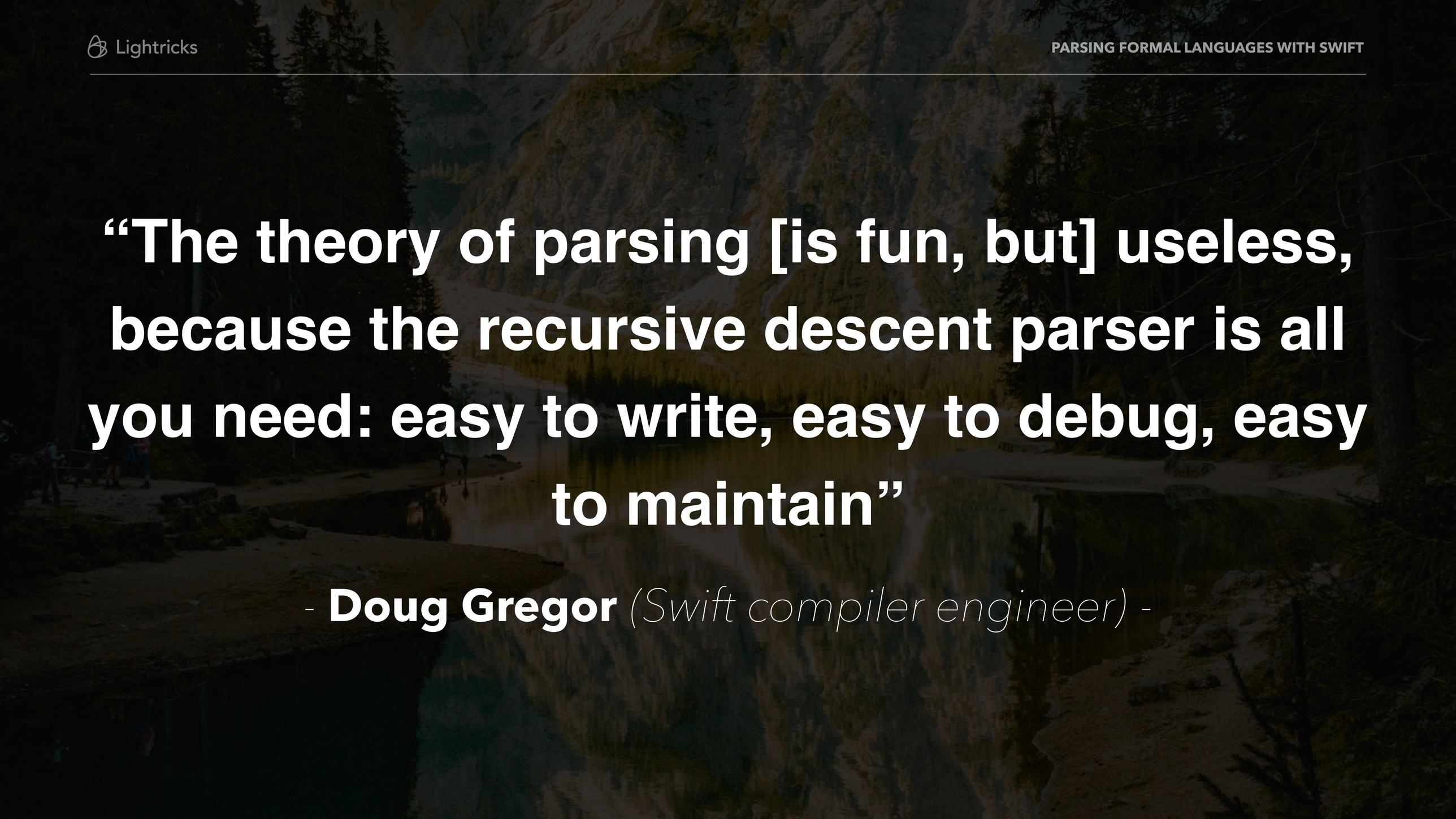
“A naive hand coded parser will be faster, easier to maintain and produce better error messages than using some fancy parser generator”

- **Slava Pestov** *(Swift compiler engineer)* -

There are many types of parser:

- LL*(k)*
- LR
- LALR*(k)*
- Recursive descent *(learn this one)*


Doug Gregor
@dgregor79
I got really interested in the theory of parsing while studying CS. It's elegant and it's fun. It's also useless, because the recursive descent parser is all you need: easy to write, easy to debug, easy to maintain. No CS degree required and don't let any PhD tell you otherwise
6:19 AM - Aug 26, 2018



“The theory of parsing [is fun, but] useless, because the recursive descent parser is all you need: easy to write, easy to debug, easy to maintain”

- **Doug Gregor** *(Swift compiler engineer)* -

Recursive Descent Parsers

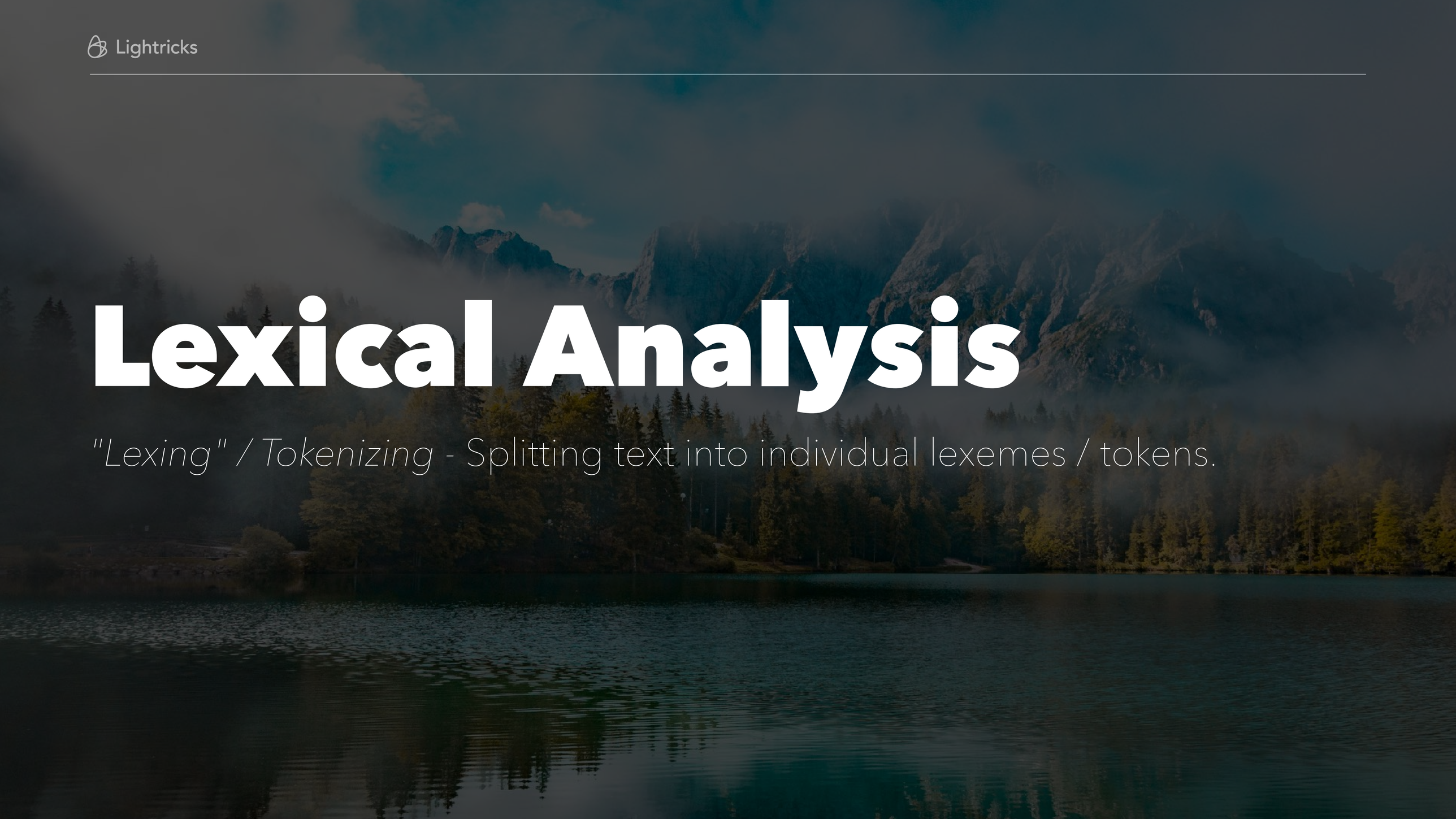
- Work from the top down.
- Try to match a pattern recursively.
- If it doesn't match, move on to the next pattern.
- Patterns are tested in a predictable order *(important)*.

A simple language

```
let fruit = "apples"
let number = 5 + 5
print number + " " + fruit
```

The two stages of parsing:

1. Lexical analysis *("lexing" / tokenizing)*
   Breaking text into individual words *(lexemes / tokens)*.
2. Syntax analysis *("parsing")*
   Arranging tokens into meaningful statements.



Lexical Analysis

"Lexing" / Tokenizing - Splitting text into individual lexemes / tokens.

Tokens

Type	Pattern
keywords	`let|print`
operators	`[=+]`
identifiers	`[a-z][a-z0-9]*`
numbers	`[0-9.]+`
strings	`"[^"]*"`

Swift enums are perfect for tokens

```
enum Token {
  case assign // = operator
  case plus // + operator
  case identifier(String)
  case number(Double)
  case string(String)
  case `let` // let keyword
  case print // print keyword
}
```

Swift and Foundation's built-in tokenizing APIs

- `String.components(separatedBy: ...)`
  Good for delimited lists *(e.g. CSV files)*.
- `[NS]Scanner`
  Good for predictable patterns *(e.g. log files)*.
- `NSRegularExpression`
  …

“Some people, when confronted with a problem, think *‘I know, I'll use regular expressions’*. Now they have two problems.”

- **Jamie Zawinski** *(Internet troll)* -

4422

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a bre*ach between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of re**gex parsers for HTML will in**stantly transport a *programmer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes, ~~the pestilent s~~lithy regex-infection will **devour your H**TML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not f*ight h**e comes, 'his** unholy radiańcé *destr*oying *all enlightenment, HTML tags* ***leaking from your eyes' like liq*** *uid* pain, the song of regular expr~~ession parsing~~ will ext*inguish the voices of mor***tal man from the sp***here I can see it can you see it it is beautiful* the f `inal snuf` fing *of the lies of Man ALL IS LOST* **ALL IS** LOST the *pony he comes* he comes ~~he comes~~ **the ich**or permeates *all* MY FACE *MY FACE oh god no NO NOOOO* NO stop *the angles* `are` **not real ZALGO IS TONY THE PONY HE COMES**

Have you tried using an XML parser instead?

NSRegularExpression

```
let whitespace = try NSRegularExpression(pattern: "(\\s|\\n)+")
let assign = try NSRegularExpression(pattern: "=")
let plus = try NSRegularExpression(pattern: "\\+")
let identifier = try NSRegularExpression(pattern: "[A-Za-z][A-Za-z0-9]*")
let number = try NSRegularExpression(pattern: "[0-9.]+"
let string = try NSRegularExpression(pattern: "\"(\\\\\"|\\\\\\\\|[^\"\\\\])*\"")
```

Problem 1: Trying to match a string literal with escape sequences in regex is... not great 🤮

NSRange

```swift
var range = NSRange(location: 0, length: input.utf16.count)

func readToken(_ regex: NSRegularExpression, in range: inout NSRange) -> String? {
  guard let match = regex.firstMatch(in: input, options: .anchored, range: range) else {
    return nil
  }
  range.location += match.range.length
  range.length -= match.range.length
  return (input as NSString).substring(with: match.range)
}
```

Problem 2: NSRegularExpression's NSRange-based API doesn't interact nicely with Swift Strings.

Alternative?

Character-by-character processing.

Time for some *String Theory!*

- **`String`**
  A collection of `Character` *(grapheme clusters)*
- **`UnicodeScalarView`**
  A collection of `UnicodeScalar` *(UTF-32 code units)*
- **`UTF16View`**
  A collection of `UniChar` *(UTF-16 code units)*
- **`UTF8View`**
  A collection of `UTF8Char` *(UTF-8 code units)*

Parsing 1,000,000 characters

```
var input = Substring(source) // .unicodeScalars, .utf16, .utf8
while input.popFirst() != nil {
  count += 1
}
```

Method	Time (ASCII input)	Time (Unicode input)
String (Character)	62 ms	626 ms
UnicodeScalarView	4 ms	6 ms
UTF16View	34 ms	74 ms
UTF8View	25 ms	1278 ms

Slices & Substrings

Stepping through a `String` one character at a time requires us to keep track of two things:

- **A reference to the `String` data**
- **The current character index**

A `String` slice (`Substring`) encapsulates both a reference to the string data and a start/end index.

Using `Substring` instead of `String` means we don't have to keep track of the index separately.

Lexer Implementation *(tokenize)*

```
public func tokenize(_ input: String) throws -> [Token] {
  var scalars = Substring(input).unicodeScalars

  ...
  ...
  ...
  ...
  ...
  ...
  ...
}

extension Substring.UnicodeScalarView {

  ...
  ...
  ...
}
```

Lexer Implementation *(tokenize)*

```
public func tokenize(_ input: String) throws -> [Token] {
  var scalars = Substring(input).unicodeScalars
  var tokens: [Token] = []
  while let token = scalars.readToken() {
    tokens.append(token)
  }

  ...
  ...
  ...
}

extension Substring.UnicodeScalarView {

  mutating func readToken() -> Token? {
    ...
  }
}
```

Lexer Implementation *(tokenize)*

```
public func tokenize(_ input: String) throws -> [Token] {
  var scalars = Substring(input).unicodeScalars
  var tokens: [Token] = []
  while let token = scalars.readToken() {
    tokens.append(token)
  }
  if !scalars.isEmpty {
    throw LexerError.unrecognizedInput(String(scalars))
  }
  return tokens
}

extension Substring.UnicodeScalarView {

  mutating func readToken() -> Token? {
    ...
  }
}
```

Lexer Implementation *(readToken)*

```
extension Substring.UnicodeScalarView {

  mutating func readToken() -> Token? {
    self.skipWhitespace()

    ...
    ...
    ...
  }

  mutating func skipWhitespace() {
    let whitespace = CharacterSet.whitespacesAndNewlines
    while let scalar = self.first, whitespace.contains(scalar) {
      self.removeFirst()
    }
  }
}
```

Lexer Implementation *(readToken)*

```
extension Substring.UnicodeScalarView {

  mutating func readToken() -> Token? {
    self.skipWhitespace()
    return self.readOperator() ??
      self.readIdentifier() ??
      self.readNumber() ??
      self.readString()
  }

  mutating func skipWhitespace() { ... }

  mutating func readOperator() -> Token? { ... }
  mutating func readIdentifier() -> Token? { ... }
  mutating func readNumber() -> Token? { ... }
  mutating func readString() -> Token? { ... }
}
```

Lexer Implementation *(readOperator)*

```
extension Substring.UnicodeScalarView {

  mutating func readOperator() -> Token? {
    let start = self
    switch self.popFirst() {
    case "=":
      return Token.assign
    case "+":
      return Token.plus
    default:
      self = start
      return nil
    }
  }
}
```

Lexer Implementation *(readIdentifier)*

```
extension Substring.UnicodeScalarView {

  mutating func readIdentifier() -> Token? {
    guard let head = self.first, CharacterSet.letters.contains(head) else {
      return nil
    }
    var name = String(self.removeFirst())
    while let c = self.first, CharacterSet.alphanumerics.contains(c) {
      name.append(Character(self.removeFirst()))
    }

    ...
    ...
    ...
    ...
    ...
    ...
    ...
  }
}
```

Lexer Implementation *(readIdentifier)*

```
extension Substring.UnicodeScalarView {

  mutating func readIdentifier() -> Token? {
    guard let head = self.first, CharacterSet.letters.contains(head) else {
      return nil
    }
    var name = String(self.removeFirst())
    while let c = self.first, CharacterSet.alphanumerics.contains(c) {
      name.append(Character(self.removeFirst()))
    }
    switch name {
    case "let":
      return Token.let
    case "print":
      return Token.print
    default:
      return Token.identifier(name)
    }
  }
}
```

Lexer Implementation *(readString)*

```
extension Substring.UnicodeScalarView {

  mutating func readString() -> Token? {
    guard first == "\"" else { return nil }
    let start = self
    self.removeFirst()
    var string = "", escaped = false
    while let scalar = self.popFirst() {
      switch scalar {
      case "\"" where !escaped:
        return Token.string(string)
      case "\\" where !escaped:
        escaped = true
      default:
        string.append(Character(scalar))
        escaped = false
      }
    }
    self = start
    return nil
  }
}
```

Lexer Implementation *(usage)*

Usage:

```
let tokens = try tokenize("let foo = 5")
```

Output:

```
[
  Token.let,
  Token.identifier("foo"),
  Token.assign,
  Token.number(5)
]
```

Syntax Analysis

"Parsing" - Arranging tokens into meaningful statements.

Abstract Syntax Trees *(ASTs)*

Lexical Analysis produces a flat array of tokens.

Syntax Analysis produces a tree structure.

This structure is called an Abstract Syntax Tree *(AST)*.

AST Nodes

Node	Pattern
variable declaration	`let identifier = <expression>`
print statement	`print <expression>`
operand	`number | string | identifier`
expression	`<operand>` `<operand> + <expression>`

Like tokens, enums are great for AST nodes

```
enum Statement {
  case declaration(name: String, value: Expression)
  case print(Expression)
}

indirect enum Expression {
  case number(Double)
  case string(String)
  case variable(String)
  case addition(lhs: Expression, rhs: Expression)
}
```

Parser Implementation *(parse)*

```
public func parse(_ input: String) throws -> [Statement] {
  var tokens = try ArraySlice(tokenize(input))
  var statements: [Statement] = []
  while let statement = tokens.readStatement() {
    statements.append(statement)
  }

  ...
    ...
  ...
}

extension ArraySlice where Element == Token {

  mutating func readStatement() -> Statement? {
    ...
  }
}
```

Parser Implementation *(parse)*

```
public func parse(_ input: String) throws -> [Statement] {
  var tokens = try ArraySlice(tokenize(input))
  var statements: [Statement] = []
  while let statement = tokens.readStatement() {
    statements.append(statement)
  }
  if let token = tokens.first {
    throw ParserError.unexpectedToken(token)
  }
  return statements
}

extension ArraySlice where Element == Token {

  mutating func readStatement() -> Statement? {
    ...
  }
}
```

Parser Implementation *(readStatement)*

```
extension ArraySlice where Element == Token {

  mutating func readStatement() -> Statement? {
    return self.readDeclaration() ?? self.readPrintStatement()
  }

  mutating func readDeclaration() -> Statement? { ... }

  mutating func readPrintStatement() -> Statement? {
    let start = self
    guard self.popFirst() == .print, let value = self.readExpression() else {
      self = start
      return nil
    }
    return Statement.print(value)
  }

  mutating func readExpression() -> Expression? { ... }
}
```

Parser Implementation *(readExpression)*

```
extension ArraySlice where Element == Token {

  mutating func readExpression() -> Expression? {
    guard let lhs = readOperand() else {
      return nil
    }

    let start = self
    guard self.popFirst() == .plus, let rhs = readExpression() else {
      self = start
      return lhs
    }
    return Expression.addition(lhs: lhs, rhs: rhs)
  }
}
```

Parser Implementation *(readOperand)*

```
extension ArraySlice where Element == Token {

  mutating func readOperand() -> Expression? {
    let start = self
    switch self.popFirst() {
    case Token.identifier(let variable)?:
      return Expression.variable(variable)
    case Token.number(let double)?:
      return Expression.number(double)
    case Token.string(let string)?:
      return Expression.string(string)
    default:
      self = start
      return nil
    }
  }
}
```

Parser Implementation *(usage)*

Usage:

```
let program = try parse("let foo = 5 \n print foo")
```

Output:

```
[
  Statement.declaration(
    name: "foo",
    value: Expression.number(5)
  ),
  Statement.print(
    Expression.identifier("foo")
  )
]
```

What can we do with an AST?

- Build a formatter / *"pretty-printer"*.
- Build a static code analyzer / linter.
- Build an interpreter to execute the code.
- Build a compiler or transpiler.

Example Formatter

```swift
public func format(_ program: [Statement]) -> String {
  var output = ""
  for statement in programs {
    output.append(statement.description + "\n")
  }
  return output
}

extension Statement: CustomStringConvertible {

  var description: String {
    switch self {
    case .declaration(name: let name, value: let expression):
      return "let \(name) = \(expression)"
    case .print(let expression):
      return "print \(expression)"
    }
  }
}

extension Expression: CustomStringConvertible {

  var description: String {
    switch self {
    case .number(let double):
      return String(format: "%g", double)
    case .string(let string):
      let escapedString = string
        .replacingOccurrences(of: "\\", with: "\\\\")
        .replacingOccurrences(of: "\"", with: "\\\"")
      return "\"\(escapedString)\""
    case .variable(let name):
      return name
    case .addition(lhs: let lhs, rhs: let rhs):
      return "\(lhs) + \(rhs)"
    }
  }
}
```

Example Interpreter

```
public func evaluate(_ program: [Statement]) throws -> String {
  let environment = Environment()
  for statement in program {
    try statement.evaluate(in: environment)
  }
  return environment.output
}

public enum RuntimeError: Error, Equatable {
  case undefinedVariable(String)
}

class Environment {
  var variables: [String: Value] = [:]
  var output = ""
}

enum Value: CustomStringConvertible, Equatable {
  case number(Double)
  case string(String)

  var description: String {
    switch self {
    case .number(let double):
      return String(format: "%g", double)
    case .string(let string):
      return string
    }
  }
}
```

```
extension Statement {

  func evaluate(in environment: Environment) throws {
    switch self {
    case .declaration(name: let name, value: let expression):
      let value = try expression.evaluate(in: environment)
      environment.variables[name] = value
    case .print(let expression):
      let value = try expression.evaluate(in: environment)
      environment.output.append("\(value)\n")
    }
  }
}

extension Expression {

  func evaluate(in environment: Environment) throws -> Value {
    switch self {
    case .number(let double):
      return .number(double)
    case .string(let string):
      return .string(string)
    case .variable(let name):
      guard let value = environment.variables[name] else {
        throw RuntimeError.undefinedVariable(name)
      }
      return value
    case .addition(lhs: let expression1, rhs: let expression2):
      let value1 = try expression1.evaluate(in: environment)
      let value2 = try expression2.evaluate(in: environment)
      switch (value1, value2) {
      case (.number(let lhs), .number(let rhs)):
        return .number(lhs + rhs)
      case (.string, _), (_, .string):
        return .string("\(value1)\(value2)")
      }
    }
  }
}
```

Example Transpiler

```swift
public func transpile(_ program: [Statement]) throws -> String {
  let context = Context()
  for statement in program {
    try statement.transpile(in: context)
  }
  return context.output
}

enum Type {
  case number
  case string
}

extension Statement {

  func transpile(in context: Context) throws {
    switch self {
    case .declaration(name: let name, value: let expression):
      let (type, value) = try expression.transpile(in: context)
      context.variables[name] = type
      // TODO: escape reserved names
      context.output.append("let \(name) = \(value)\n")
    case .print(let expression):
      let (_, value) = try expression.transpile(in: context)
        context.output.append("print(\(value))\n")
      }
  }
}
```

```swift
class Context {
  var variables: [String: Type] = [:]
  var output = ""
}

public enum TranspilerError: Error, Equatable {
  case undefinedVariable(String)
}

extension Expression {

  func transpile(in context: Context) throws -> (Type, String) {
    switch self {
    case .number(let double):
      return (.number, String(double))
    case .string(let string):
      let escapedString = string
        .replacingOccurrences(of: "\\", with: "\\\\")
        .replacingOccurrences(of: "\"", with: "\\\"")
      return (.string, "\"\(escapedString)\"")
    case .variable(let name):
      guard let type = context.variables[name] else {
        throw TranspilerError.undefinedVariable(name)
      }
      // TODO: escape reserved names
      return (type, name)
    case .addition(lhs: let expression1, rhs: let expression2):
      let (type1, lhs) = try expression1.transpile(in: context)
      let (type2, rhs) = try expression2.transpile(in: context)
      switch (type1, type2) {
      case (.number, .number):
        return (.number, "\(lhs) + \(rhs)")
      case (.string, _), (_, .string):
        return (.string, "\"\\(\(lhs))\\(\(rhs))\"")
      }
    }
  }
}
```

Enhancements

Extending the parser with other operators and better error reporting.

Adding a Multiply (*) operator

Our language currently only supports one type of operator (`+`) and one type of expression (`a + b`).

To add a new operator we will need to extend both the Lexer and Parser.

Extending the Token type and Lexer

```
enum Token {
  case assign
  case plus
  case times /* new */
  case number(Double)
  case `let`
}

mutating func readOperator() -> Token? {
  let start = self
  switch self.popFirst() {
  case "=": return Token.assign
  case "+": return Token.plus
  case "*": return Token.times /* new */
  default:
    self = start
    return nil
  }
}
```

Extending the AST

```
indirect enum Expression {
  case number(Double)
  case string(String)
  case variable(String)
  case addition(lhs: Expression, rhs: Expression)
  case multiplication(lhs: Expression, rhs: Expression) /* new */
}
```

Extending the Parser *(naive solution)*

```
mutating func readExpression() -> Expression? {
  guard let lhs = readOperand() else {
    return nil
  }
  let start = self
  guard let op = self.popFirst, let rhs = readExpression() else { /* new */
    self = start
    return lhs
  }
  switch op { /* new */
  case Token.plus:
    return Expression.addition(lhs: lhs, rhs: rhs)
  case Token.times:
    return Expression.multiplication(lhs: lhs, rhs: rhs)
  default:
    self = start
    return lhs
  }
}
```

Problem: Operator Precedence

Expression parsing is naturally right-associative, so

```
let a = b + c + d + ...
```

is interpreted as

```
let a = b + (c + (d + (...)))
```

(as expected)

Problem: Operator Precedence

But that means

```
let a = b * c + d
```

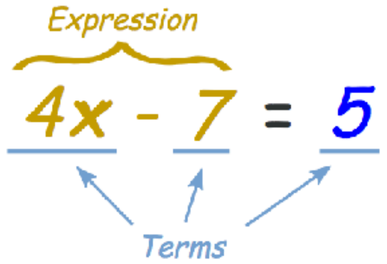
is interpreted as

```
let a = b * (c + d)
```

when it should be

```
let a = (b * c) + d
```

Solution: Introduce a "term" node



In Algebra a term is either a single number or variable, or numbers and variables multiplied together.

Terms are separated by + or − signs

https://www.mathsisfun.com/definitions/term.html

AST Nodes

Node	Pattern
operand	number | string | identifier
term	<operand> <operand> + ***<term>***
expression	***<term>*** ***<term>*** + <expression>

Extending the Parser *(correct solution)*

```
mutating func readExpression() -> Expression? {
  guard let lhs = readTerm() else { /* new */
    return nil
  }
  let start = self
  guard self.popFirst = .plus, let rhs = readExpression() else {
    self = start
    return lhs
  }
  return Expression.addition(lhs: lhs, rhs: rhs)
}

mutating func readTerm() -> Expression? {
  guard let lhs = readOperand() else {
    return nil
  }
  let start = self
  guard self.popFirst = .times, let rhs = readTerm() else {
    self = start
    return lhs
  }
  return Expression.multiplication(lhs: lhs, rhs: rhs)
}
```

Supporting user-defined operators

Using separate parser rules for each expression type only works if operators are known at parsing time.

For languages like Swift that support user-defined operators, a different solution is needed.

Swift puts all operators and operands into a flat array, then collapses the subexpressions later once precedence is known

Better error reporting

If an error is encountered in the parser, we know which token caused the problem, but we don't know:

1. Where that token was in the original source code
2. What the problem actually was

Let's fix that!

Step 1: Store source offsets in tokens

```
enum TokenType {
  case assign
  case plus
  case identifier(String)
  case number(Double)
  case string(String)
  case `let`
  case print
}

struct Token {
  let type: TokenType
  let range: Range<String.Index>
}
```

Step 2: Report errors at the source

```
// good
func readToken() throws -> Token? {
  ...
}

// better
enum TokenType {
  case identifier(String)
  case number(Double)
  ...
  case error(LexerError)
}
```

Conclusions

- Parsers are useful and fun!
- Swift is great for writing parsers.
- Recursive descent is a simple, powerful technique.
- Hand-written parsers are easy to extend, provide good error feedback.

Thank You!

@nicklockwood / March 2019

Links

- Me *(Nick Lockwood)*
  `https://twitter.com/nicklockwood`
- This presentation *(Code and slides)*
  `https://github.com/nicklockwood/Parsing`
- SwiftFormat *(Making of a Swift tokenizer & formatter)*
  `http://bytes.schibsted.com/swift-format-part-1`
- Expression *(Parser with custom operator support)*
  `https://github.com/nicklockwood/Expression`