Compiler Design and Implementation

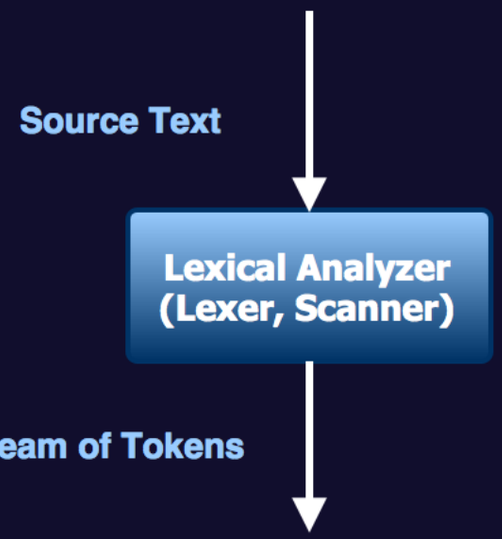
Árpád Goretity
@H2CO3_iOS

Budapest Swift Meetup 2015

Part 2:

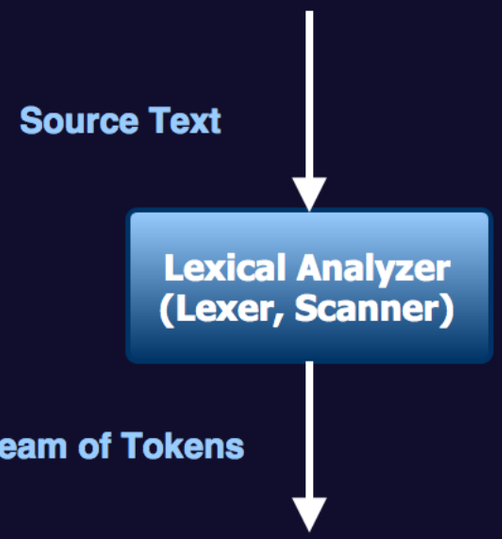
The Parser

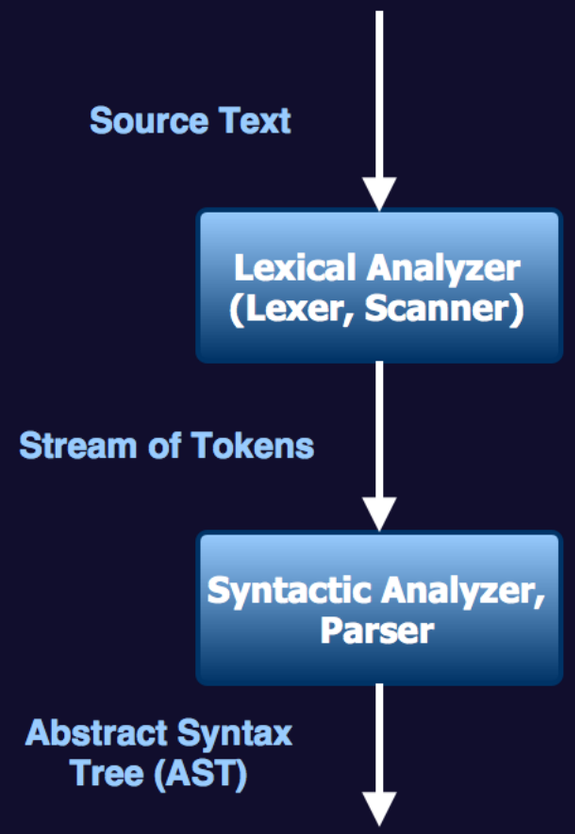
Reminder


Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens

```
let myNum = 1 - 2
```

```
(Keyword,    "let",    (1, 1))
(Identifier, "myNum",  (1, 5))
(Symbol,     "=",      (1, 11))
(Number,     "1",      (1, 13))
(Symbol,     "-",      (1, 15))
(Number,     "2",      (1, 17))
```

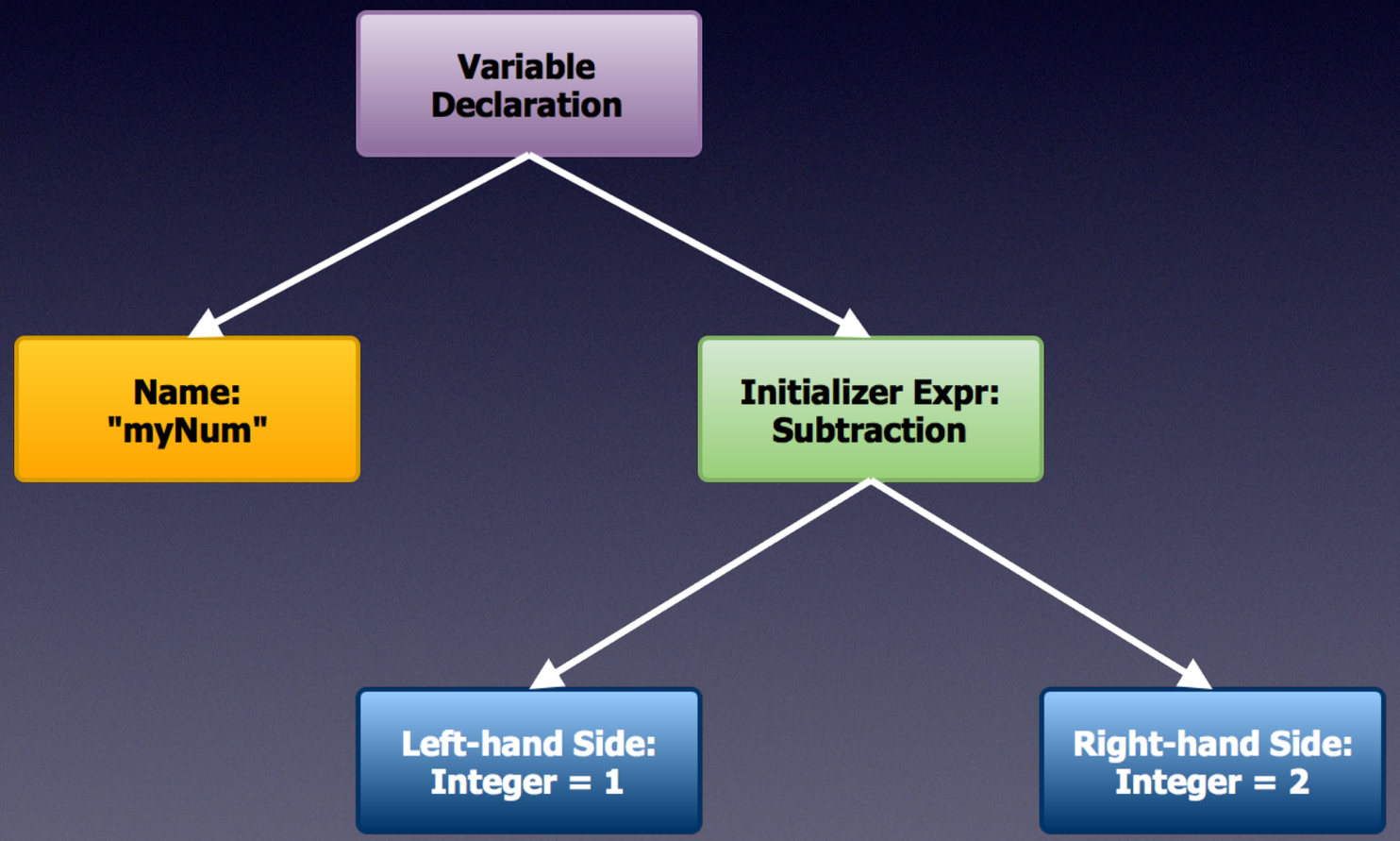

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens


Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer,
Parser
Abstract Syntax
Tree (AST)

The Parser

- Matches order of tokens with syntactic rules
- Arranges tokens in a tree called the AST
- Hierarchical structure
- Directly represents the source code
- However, more abstract
  – e.g. parentheses may be implicit

```
let myNum = 1 - 2
```


Variable Declaration
Name: "myNum"
Initializer Expr: Subtraction
Left-hand Side: Integer = 1
Right-hand Side: Integer = 2

Representing the AST:

Tree Nodes

```
class AST {
    let loc: SrcLoc
}
```

```
class IfStmt : AST {
    let cond: AST
    let thenBr: AST
    let elseBr: AST?
}
```

```
class VarDecl : AST {
    let name: String
    let initExpr: AST
}
```

```
class BinOp : AST {
    let op: String
    let lhs: AST
    let rhs: AST
}
```

Grammars

- Formal rules describing the syntactic structure of the language
- **Hierarchical** system of rules: productions
- "Template" / "Recipe" for deriving the AST from a token list

Grammars

- "Meta-syntactic" correctness:
  - Grammar recognizes **exactly** the constructs considered syntactically valid
  - Generative Grammars
- An entire field of maths: Formal Languages

Context-Free Grammars

- A.k.a. CFGs
- Syntax tree is derived **purely** from the textual value and the order of tokens
- No semantics or any kind of additional meaning is assigned to the tokens
  - E.g.: `int` is a type → who cares?

Context-Free Grammars

- CFGs are much easier to parse than Context-Sensitive Grammars (CSGs)
- Typically, CFGs are preferred in modern languages
- But lots of practical languages have CSGs!

CFGs and CSGs

- Context-sensitivity example (C): what if `a` is a type?

```
a * b;
```

- Context-free production (Swift):

```
var a: Int;
```

Elements of a Grammar

- **Production:** Head (LHS) and Body (RHS)
- **Head:** name of the production
- **Body:** actual pattern describing the rule
  - Consists of terminals and nonterminals

Elements of a Grammar

- **Terminal:** an "atomic" symbol, typically a token
  - Can be matched by just looking at its value, without any further analysis
- **Nonterminal:** "compound" element; consists of further terminals or nonterminals
  - Nonterminals can make a grammar recursive

Backus-Naur Form

- Family of notations used for specifying CFGs

```
Expr := Add
Add  := Add '+' Mul  | Mul
Mul  := Mul '*' Term | Term
Term := Number | '(' Expr ')'
```

Backus-Naur Form

```
Expr  := Add
Add   := Add '+' Mul  | Mul
Mul   := Mul '*' Term | Term
Term  := Number | '(' Expr ')'

Blue: Head, Yellow: Body
Lines: productions
Terminals: Number, '+', '*', '(', ')'
Non-terminals: Expr, Add, Mul, Term
```

Bottom-Up Parsing

- Match productions as early as possible; "eagerly"
- Builds AST by starting at leaves; joins nodes on the fly, while matching productions
- Typically used by parser generators

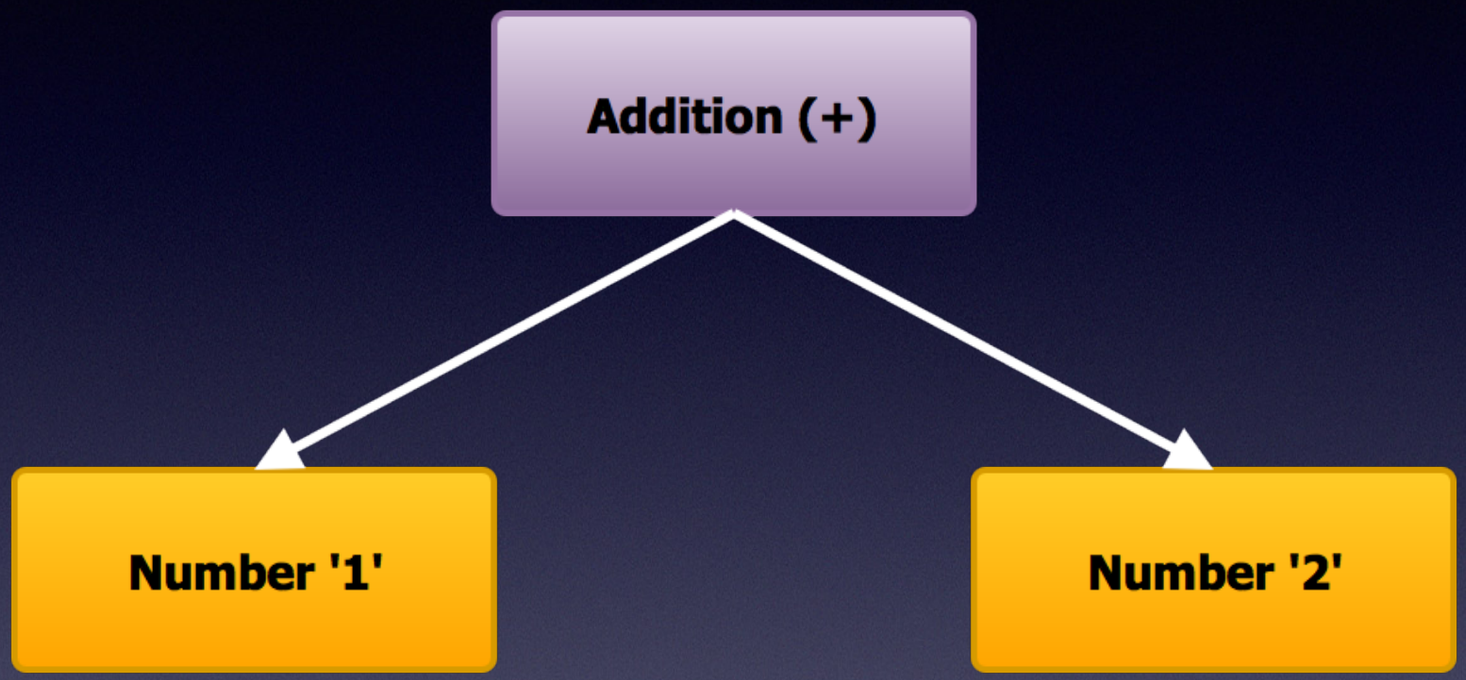
1 + 2 * 3

Number '1'

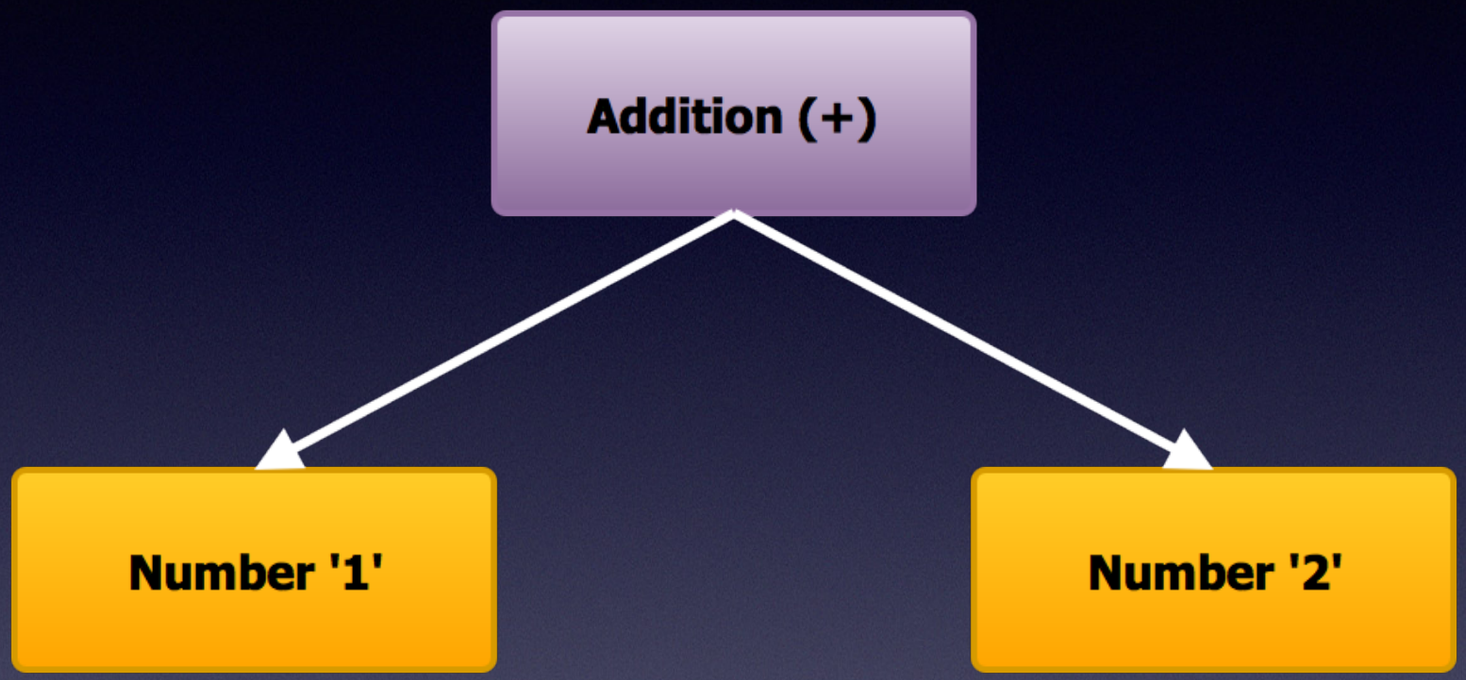
1 + 2 * 3

Number '1'

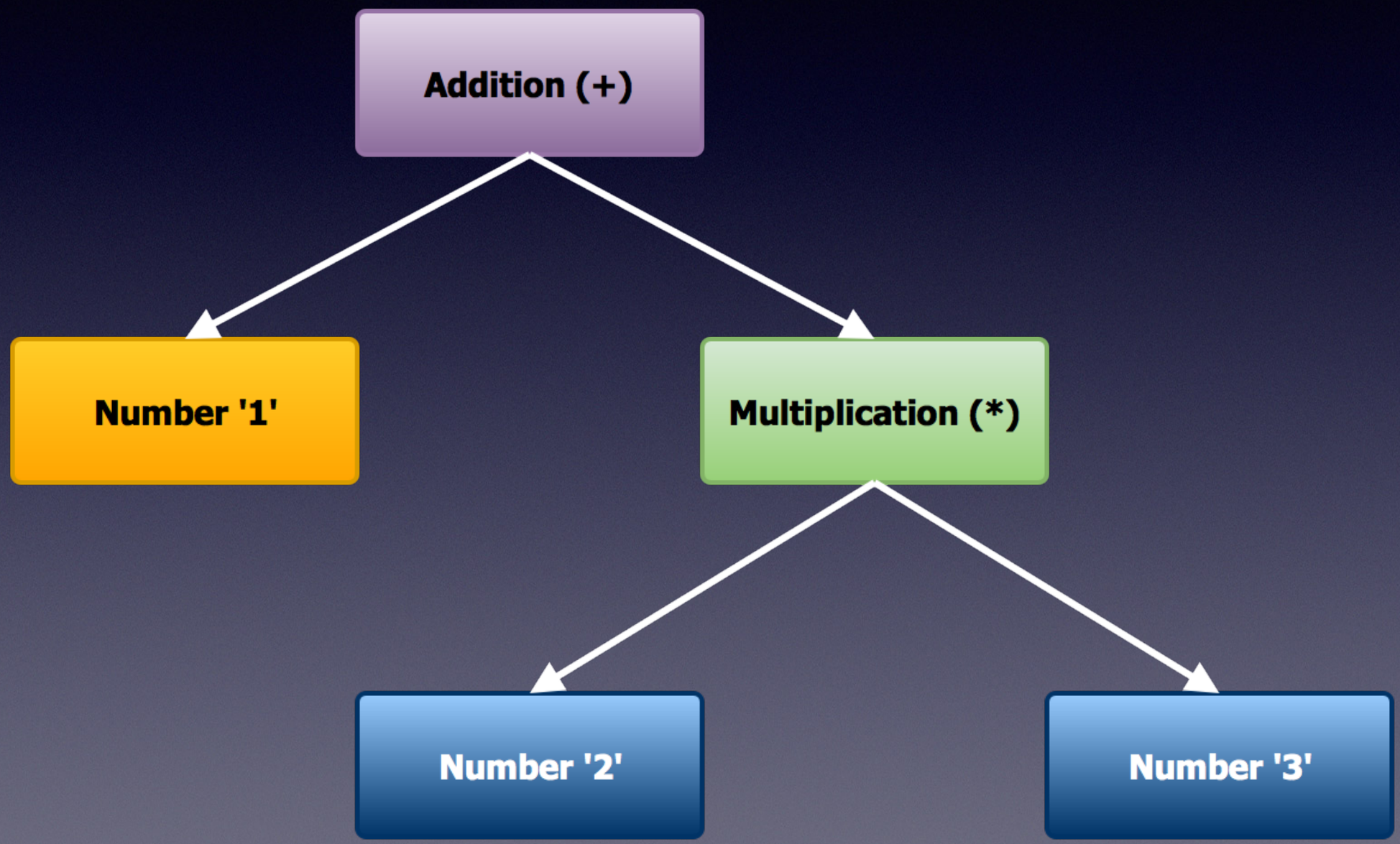
1 + 2 * 3



1 + 2 * 3



1 + 2 * 3



Bottom-Up Parsing

- Google: Shunting Yard Algorithm, LR Parser, Shift-Reduce, Operator Precedence Parser

+ : Can be faster than top-down parsing

– : Less intuitive, harder to understand

Top-Down Parsing

- Matches productions as if by applying the productions as-is, from the starting non-terminal
- Builds AST by starting at the root
- Typically used when hand-crafting a parser

Top-Down Parsing

- Top-Down:
  - Google: LL Parser

+ : More intuitive, easier to understand, algorithmize, modify and maintain

– : Can be slower than Bottom-Up Parsing

Recursive Descent

- Very popular top-down parsing algorithm
- Parser's code *directly follows* CFG productions

Recursive Descent

- Starting at the top-level production:
  - Using the lookahead token(s) from the lexer, decide what the next sub-production is
- Call corresponding parser function
  - That, in turn, calls the lexer, parsers of its own sub-productions, etc.

Recursive Descent

```
// Term := Number | '(' Expr ')'

func parseTerm() {
    if isAtToken(Number) {
        return NumberAST(accept())
    } else {
        expect("(")
        let ast = parseExpr()
        expect(")")
        return ast
    }
}
```

Recursive Descent

```
// Mul := Mul '*' Term | Term
// (production is left-associative)

func parseMul() {
    // Infinite Recursion!
    let lhs = parseMul()
    if accept('*') {
        let rhs = parseTerm()
        return Mul(lhs, rhs)
    }
    return lhs
}
```

Recursive Descent

```
// Mul := Mul '*' Term | Term
// (production is left-associative)

func parseMul() {
    var lhs = parseTerm()
    while accept('*') {
        let rhs = parseTerm()
        let tmp = MulAST(lhs, rhs)
        lhs = tmp
    }
    return lhs
}
```

Recursive Descent

```
// Assignment := Add | Add "=" Assignment
// (production is right-associative)

func parseAssignment() {
    let lhs = parseAdd() // no problem…
    if accept("=") {
        let rhs = parseAssignment()
        return AssignmentAST(lhs, rhs)
    }
    return lhs // …we do have a base case
}
```

Let's Get to Work!