

Computer Languages

CS Saturday

CS Saturdays

- Context
- Q: Should I do it?

Agenda

- How humans read
- How computers read
- Why study compilers?
- History of computer languages
- How Compilers Work
- Grammars
- Top down parsing
- Writing our own parser

Key Takeaways

- Your program is also a piece of data
- Compilers have four main stages (lexing, parsing, optimization, code generation)
- Programming languages are defined by grammars
- There are two primary ways to parse a given language into a grammar, top-down and bottom-up

How We Understand Language

How humans read

This is a sentence.

word wordword word

Tihs is a steennce.

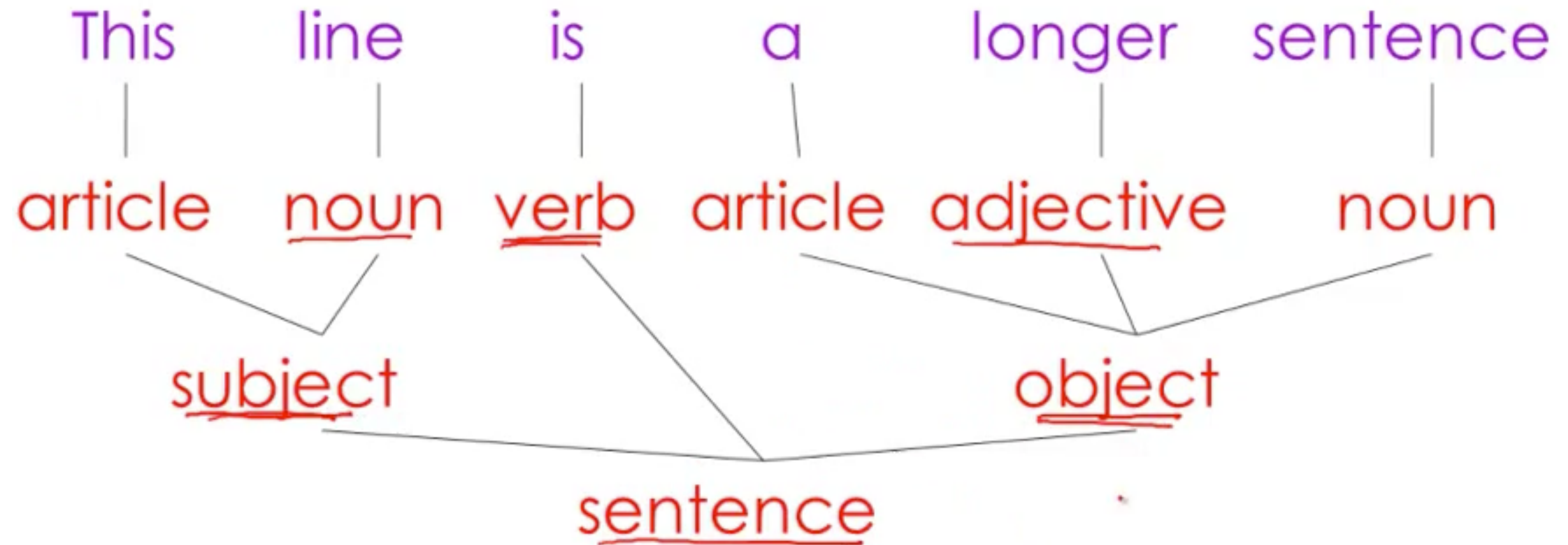
NOT EVERYONE CAN READ THIS

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttair in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid.....

Parsing

This	line	is	a	longer	sentence
article	<u>noun</u>	<u>verb</u>	article	adjective	noun

Parsing



How humans read

- Semantic Meaning

- Jack said John forgot his homework at home.
- There wasn't a single person at the party.

- Meaning is given by context, usually humans can disambiguate

How humans read

- Grammar very important
- "Eats, shoots and leaves" versus "Eats shoots and leaves"



Generation

- The words and grammar combine to generate meaning or action in our minds

Phases of Understanding

- **Four phases:**
 - Lexing (we do this so quickly we don't even notice)
 - Parsing (grammar)
 - Semantic Analysis
 - Generation

How computers read

- **Similar to us:**

- Convert a program (program.js) into a string of words (tokens)
- Parse those tokens into a grammar
- Semantically Analyze (is this program correct?)
- Generate a lower level of code (machine code or IR - intermediate representation)

- **Not similar to us:**

- Can't handle ambiguous semantics
- Definition of languages is strict and formal - grammar is very important
- Lots of focus on optimization

How computers read

- The systems that convert program code into another form (usually for execution) are called either compilers or interpreters
- They follow a 5 stage process:
 - Lexing
 - Parsing
 - Semantic Analysis
 - Optimization
 - Generation

Why study compilers

Why study compilers

- **Compiler theory is a third/fourth year CS course**
- **Compilers combine a lot of different CS fields**
 - Data structures
 - Parse trees
 - Symbol tables
 - Algorithms
 - Memory allocation
 - Register allocation
 - Stack management
 - Code optimization and minimization (tree-shaking, dead code elimination)

Why study compilers

- **Practical:**

- Implementing a programming language is a fun exercise and a pretty sure path to fame
- Compilers/parsers are embedded in a lot of tools we use

- **We already use a lot of compilers in our day to day work:**

- Angular: *\$parse* service (evaluate basic JavaScript like expressions)
- React: *JSX transformer* (convert HTML to JavaScript code)
- babel.js (convert ES6 to ES3, 4 5)
- ESLint (parses JavaScript to look for errors)
- SCSS (compiles SCSS to CSS)
- Uglify (minifies JS code for)
- ng-annotate (compiles DI to Inject statements)

History of Compilers

Ada Lovelace

1842

*Invention of the first computer
programming language for Babbage's
Analytical Engine*



Ada Lovelace

- Things Ada Lovelace foresaw in the Analytical Engine
 - Variables and Data Storage
 - Cycles (Loops, Nested Loops)
 - Sequences (Blocks)
 - Subroutines (Functions)

(6.) $(\div), \sum (+1)^p (\times, -)$ or $(1), \sum (+1)^p (2, 3)$,
where p stands for the variable ; $(+1)^p$ for the function of the variable,
that is, for ϕp ; and the limits are from 1 to p , or from 0 to $p - 1$,
each increment being equal to unity. Similarly, (4.) would be,—

(7.) $\sum (+1)^n \{ (\div), \sum (+1)^p (\times, -) \}$

the limits of n being from 1 to n , or from 0 to $n - 1$,

(8.) or $\sum (+1)^n \{ (1), \sum (+1)^p (2, 3) \}$.

<http://blog.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/>

Alan Turing

1936

Describes Turing Machine



Grace Hopper

1952

Creates the first compiler for a language called A-0. Later worked on COBOL.



John Backus

1953

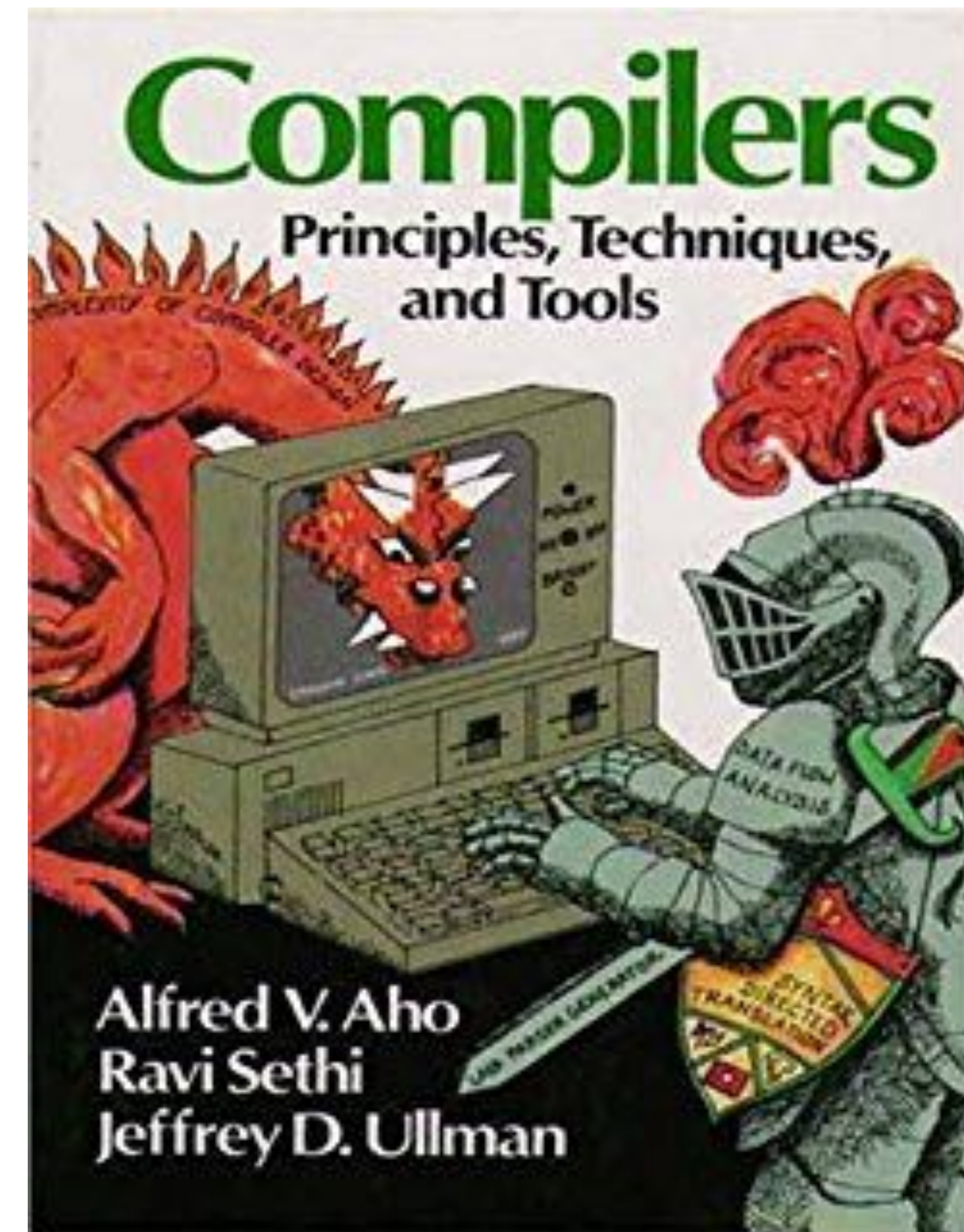
*Inventor of Backus-Naur Form - a
grammar for describing languages
and creator of Fortran*



Alfred Aho

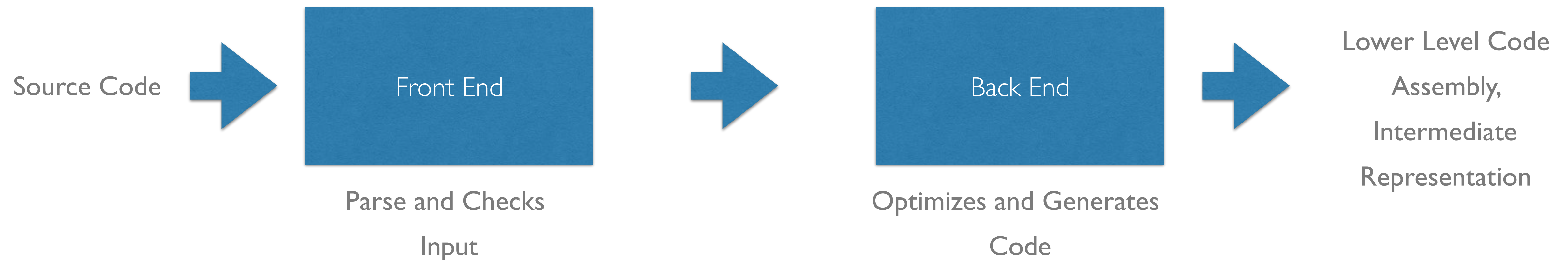
Present

*Author of the "Dragon book" and
professor at Columbia University.*

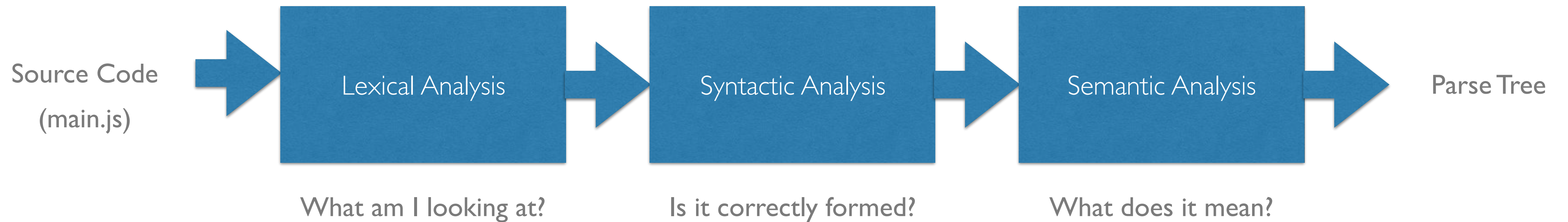


How Compilers Work

How Compilers Work



Frontend



Lexical Analysis

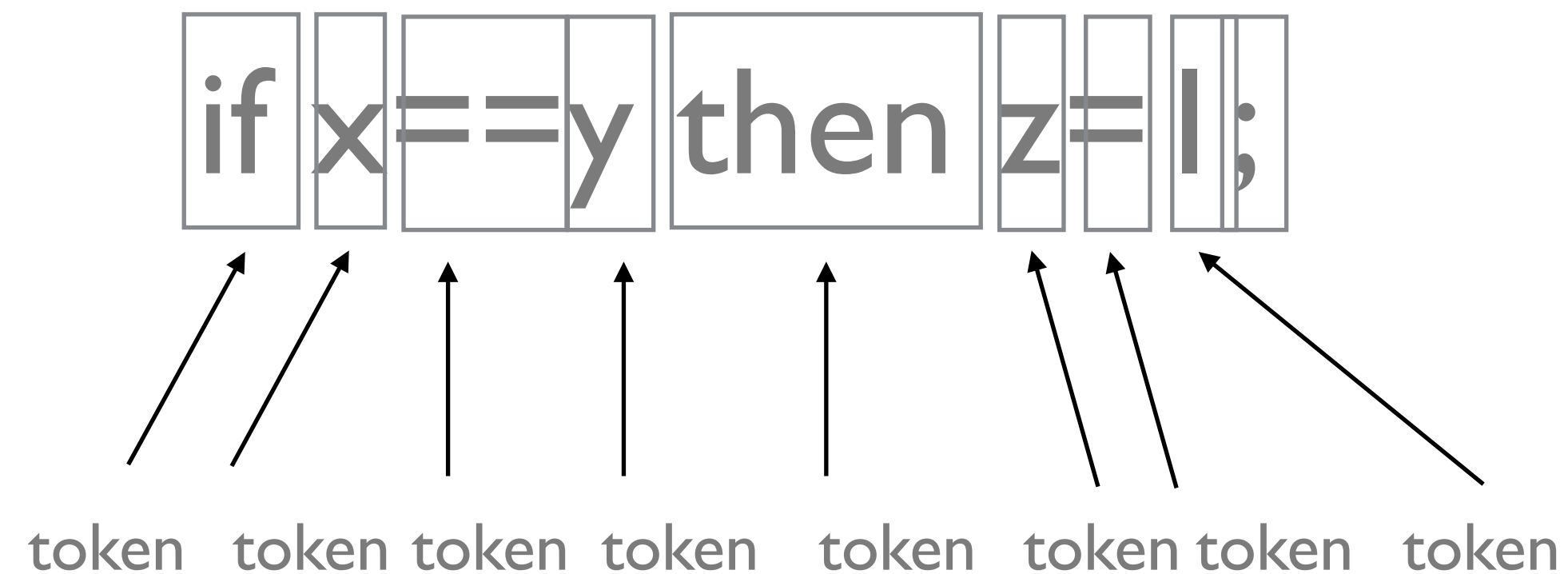
This is a sentence.

word wordword word

if x==y then z=l;

token token token token token token token token

Lexical Analysis



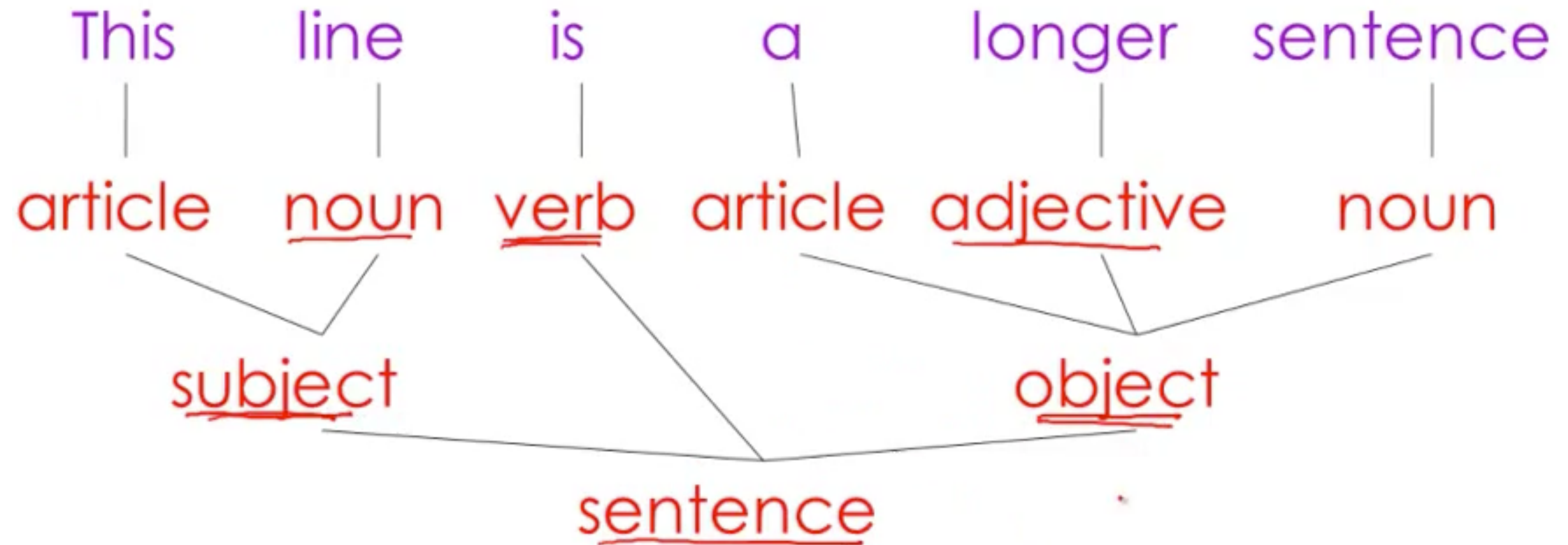
[IF] [ID X] [EQL] [ID Y] [THEN] [ID Z] [ASSIGN] [NUMBER 1] [SEMI]

Often referred to as the "token stream"

Parsing

This	line	is	a	longer	sentence
article	<u>noun</u>	<u>verb</u>	article	adjective	noun

Parsing



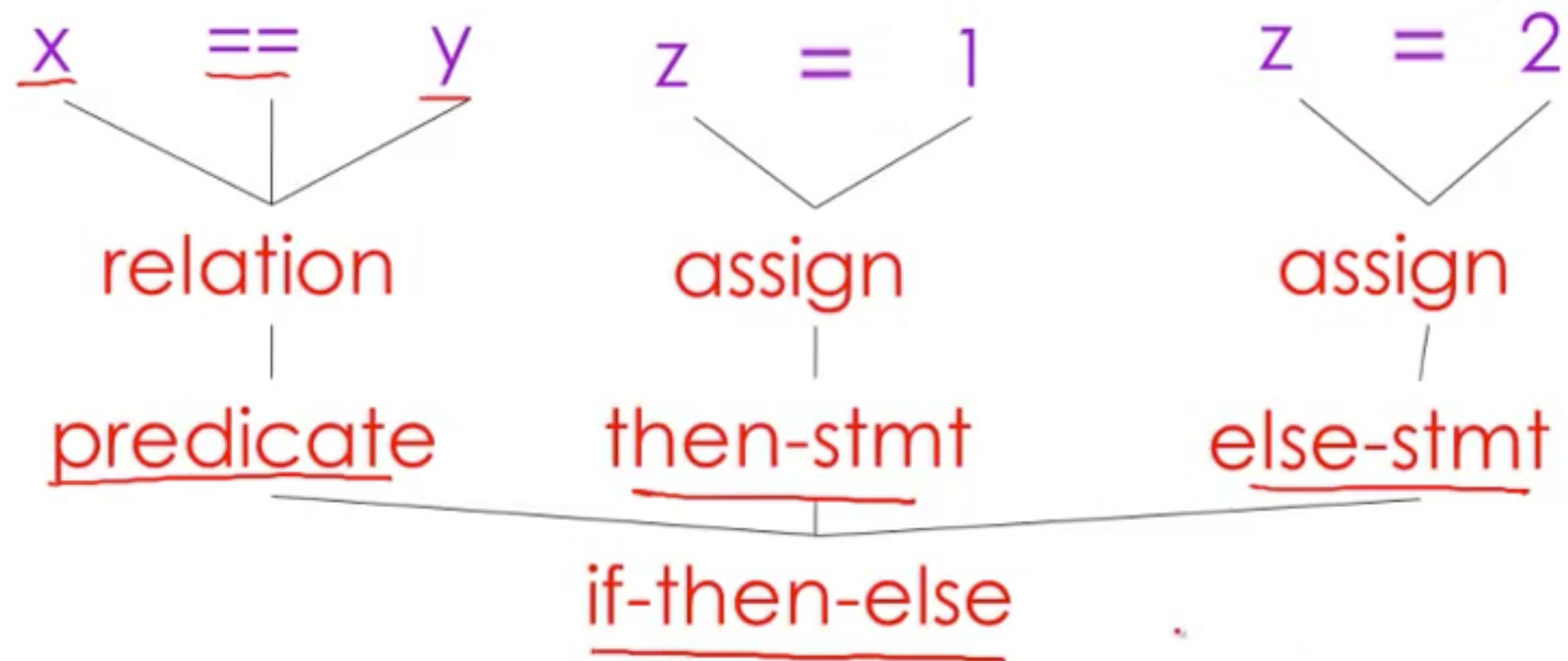
Parsing

if x == y then z = 1; else z = 2;

x == y z 1 z 2

Parsing

if x == y then z = 1; else z = 2;



Token Parsing Table

- What makes a valid token?
- List of all valid keywords (if, else, class, function)
- List of valid identifiers (variable names)
- List of valid literals (what numbers, strings look like)
- Defined using regular expressions
- <https://www.ecma-international.org/ecma-262/8.0/index.html#sec-identifier-names>

Grammars

Switch your thinking...

- Let's talk about simple math formulas
- If someone asked you - what are some basic rules you could create to generate simple math formulas...
- Start with some thing (E) that can turn into other things (including itself) while creating sentences... e.g. $E \rightarrow E + E$

Grammar for Mathematical Expressions

- E = Expression
- Start your sentence as an abstract idea of an expression: E
- $E \Rightarrow E + E$
- $E \Rightarrow E - E$
- $E \Rightarrow E * E$
- $E \Rightarrow -E$
- $E \Rightarrow (E)$
- $E \Rightarrow \text{num}$

token

Grammar for Mathematical Expressions

- $E = \text{Expression (start = } E)$

- $E \Rightarrow E + E$

- $E \Rightarrow E - E$

- $E \Rightarrow E * E$

- $E \Rightarrow -E$

- $E \Rightarrow (E)$

- $E \Rightarrow \text{num}$

production rule
left-hand side can produce right
hand side

Grammar for Mathematical Expressions

- $E = \text{Expression (start = E)}$

- $E \Rightarrow E + E$

- $E \Rightarrow E - E$

- $E \Rightarrow E * E$

- $E \Rightarrow -E$

- $E \Rightarrow (E)$

- $E \Rightarrow \text{num}$

Left hand side of a production rule must be a single symbol

These symbols are called "non-terminals"

This grammar only has one (E) but most grammars can have many non-terminals

Grammar for Mathematical Expressions

- $E = \text{Expression (start = E)}$

- $E \Rightarrow E + E$

Right hand side is a string of 0 or more symbols.

- $E \Rightarrow E - E$

- $E \Rightarrow E * E$

- $E \Rightarrow -E$

- $E \Rightarrow (E)$

- $E \Rightarrow \text{num}$

Symbols that are not non-terminals are called "terminals" Terminals do not appear on the left-hand side.

Here: + - * () num are terminals

How Grammar creates Languages

- The language created by a grammar is the **set of all terminal strings** that can be **generated** using the **production rules** of that grammar.
- **Corollary:**
 - If a valid parse tree can be generated from a terminal string using the production rules of the grammar, then that terminal string is in the language defined by the grammar.
 - e.g. the language of JavaScript is defined by the production rules of the grammar of JavaScript.
 - If a string of JavaScript code (terminal string) can be parsed using the production rules of JavaScript, that file is a valid JavaScript language file.

Parse Trees and Languages

Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree

$1 + 2 \times 3$

E

← sentence symbol

Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

$$E \rightarrow E \times E$$

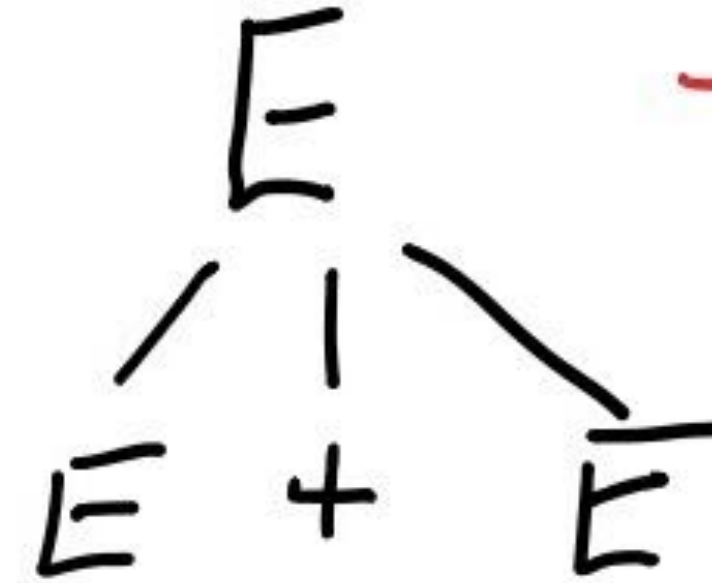
$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree

$1 + 2 \times 3$



Parent = LHS
Children = RHS

Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

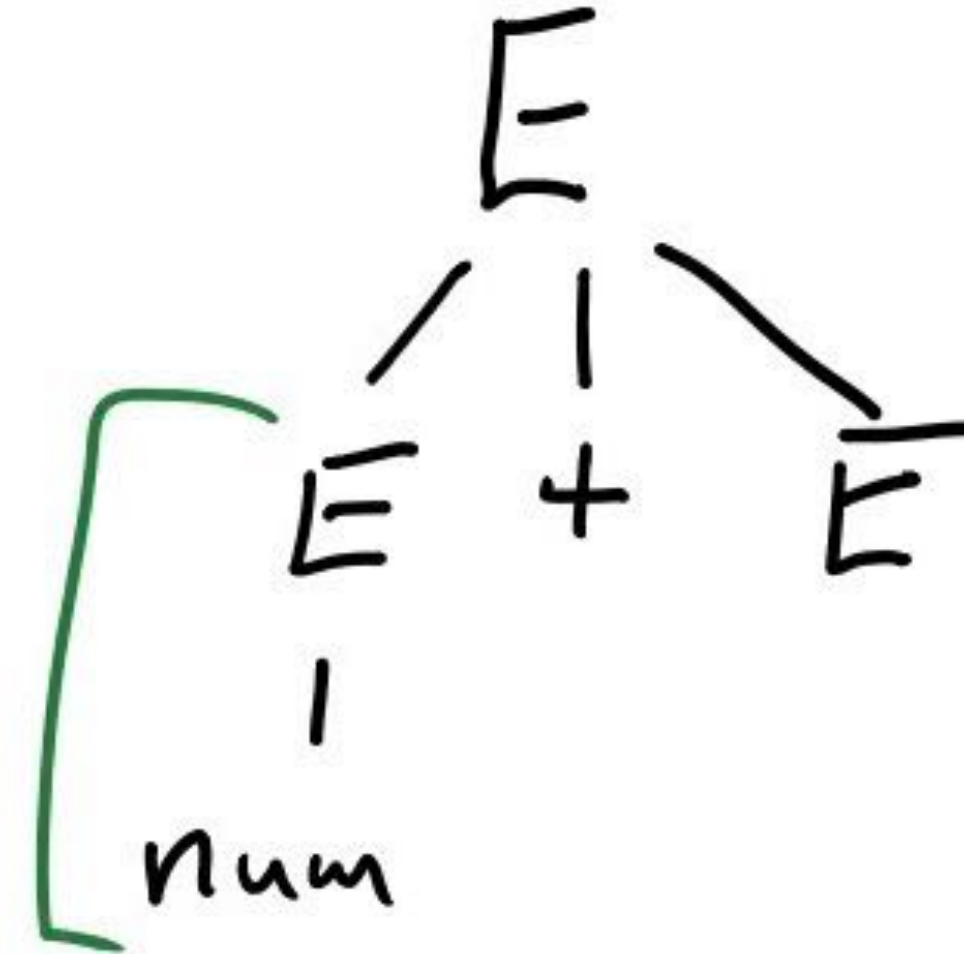
$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree
1 + 2 x 3



Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

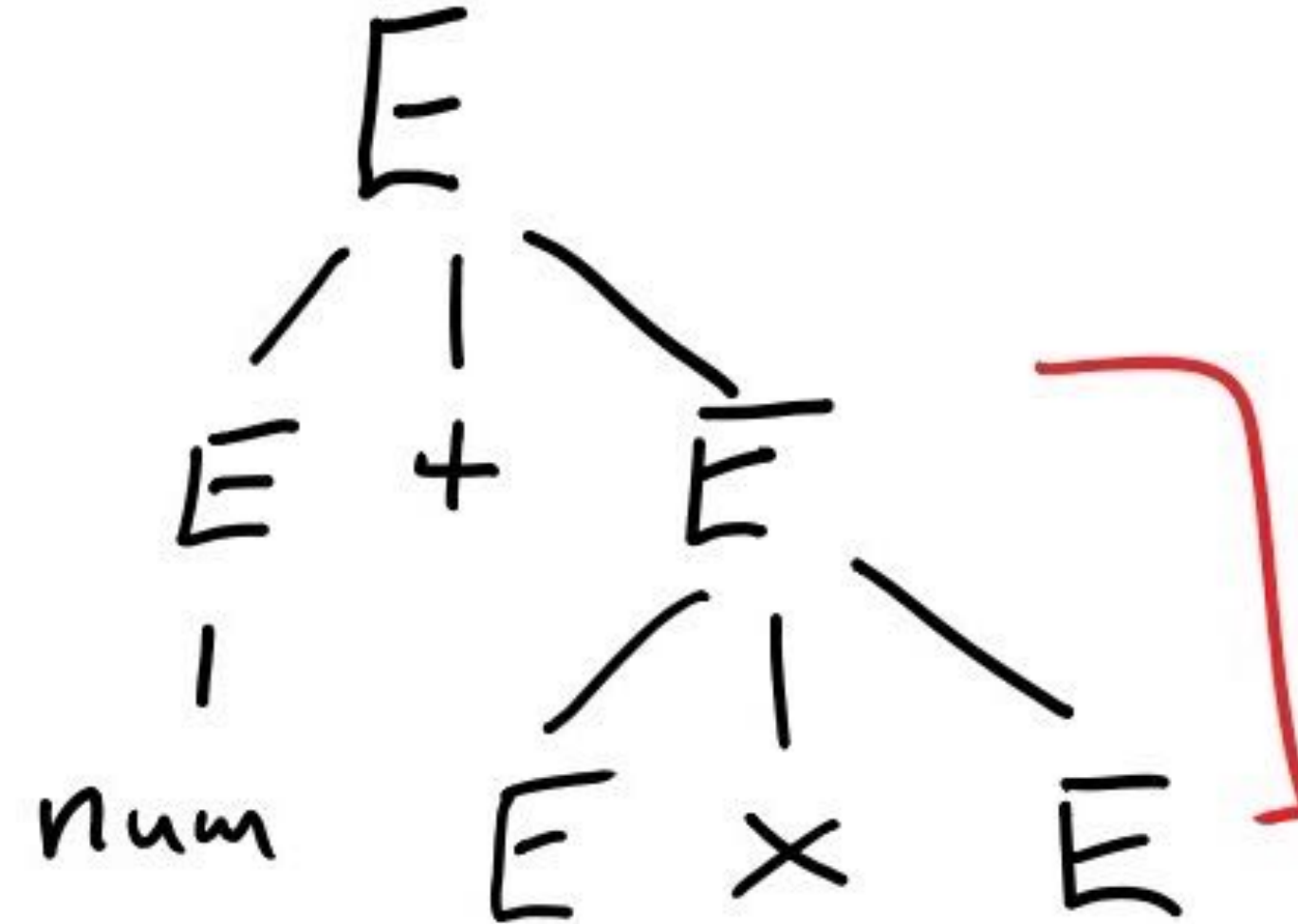
$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree
1 + 2 x 3



Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

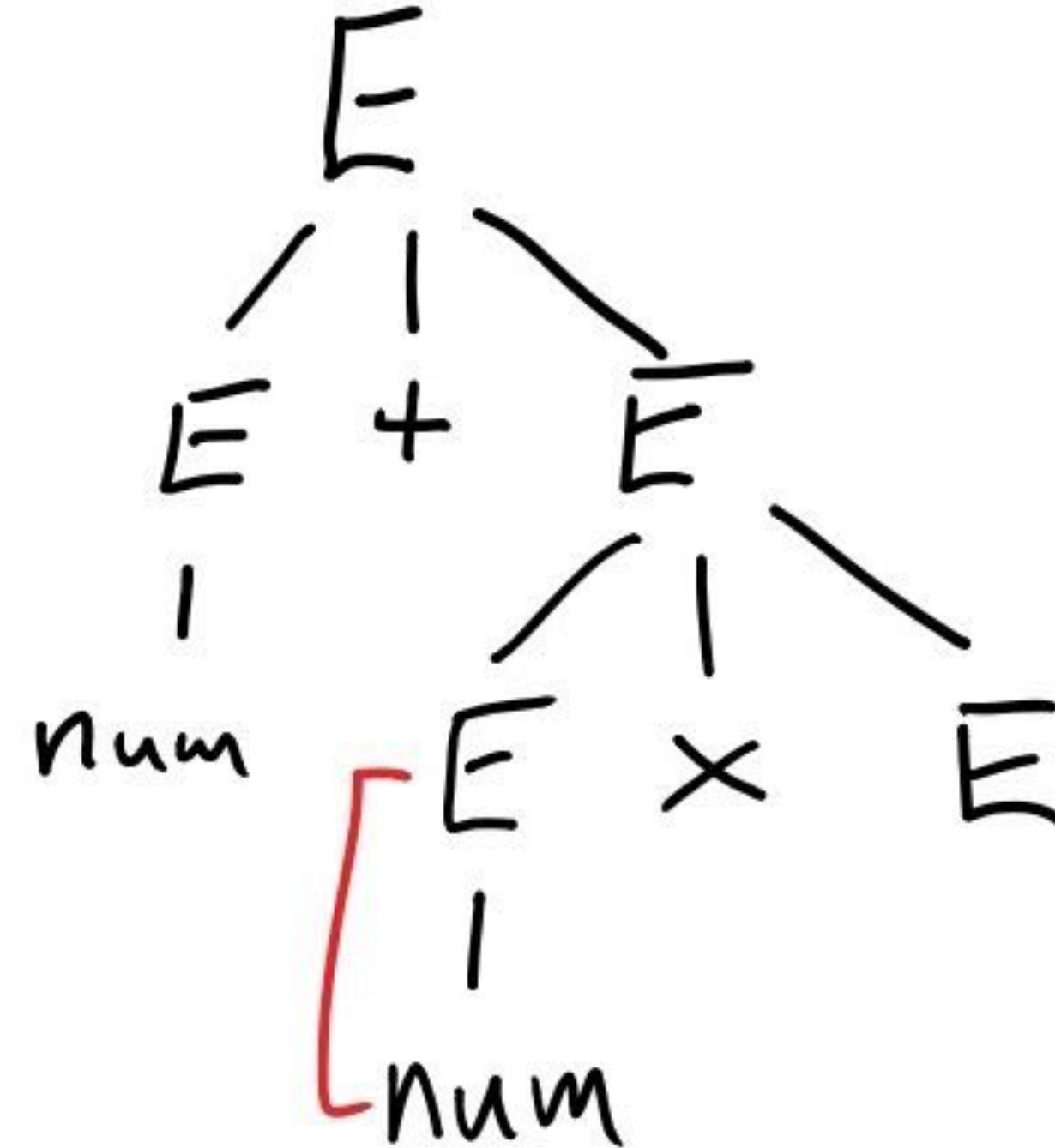
$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree
1 + 2 x 3



Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

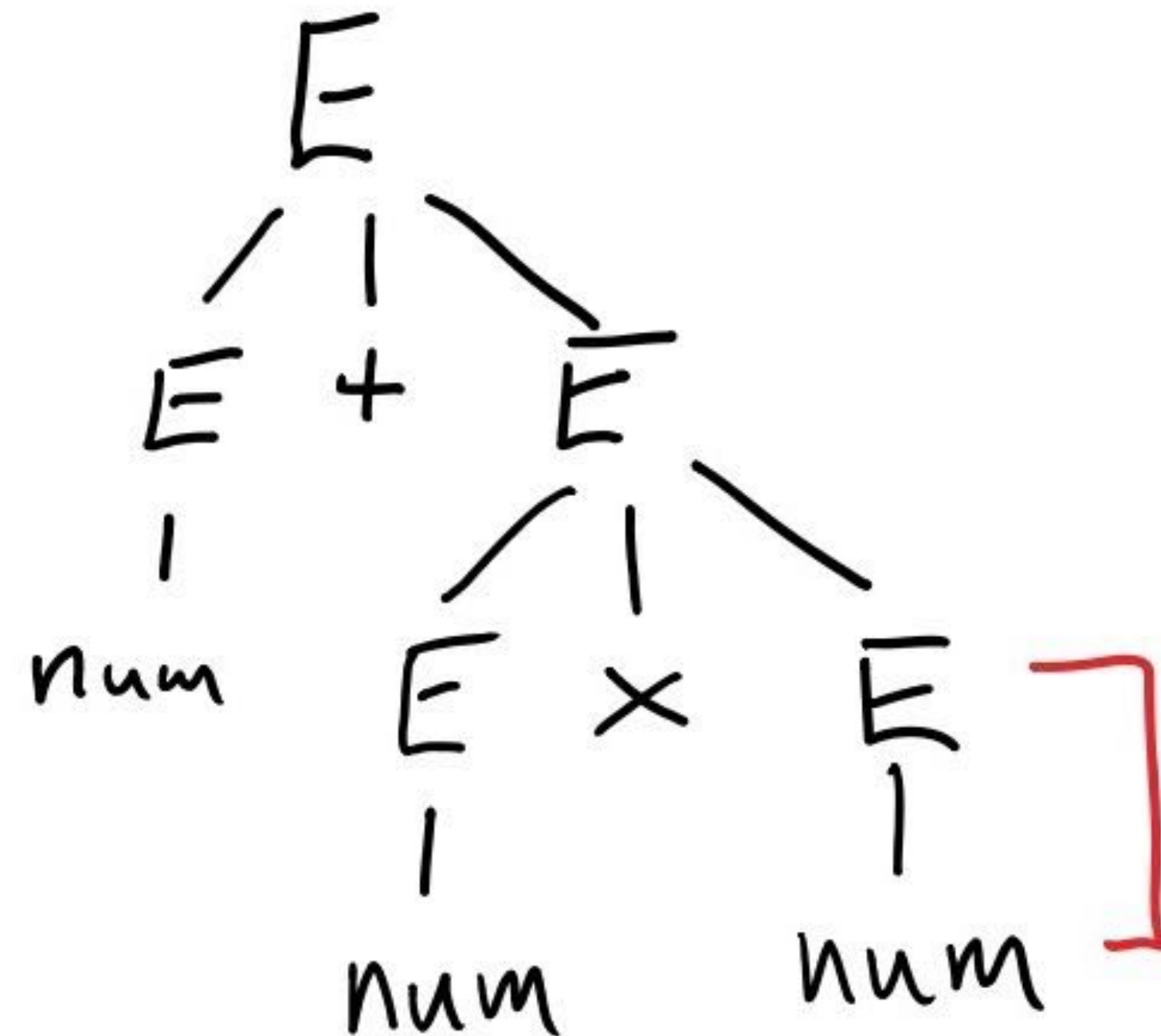
$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree
1 + 2 x 3



Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

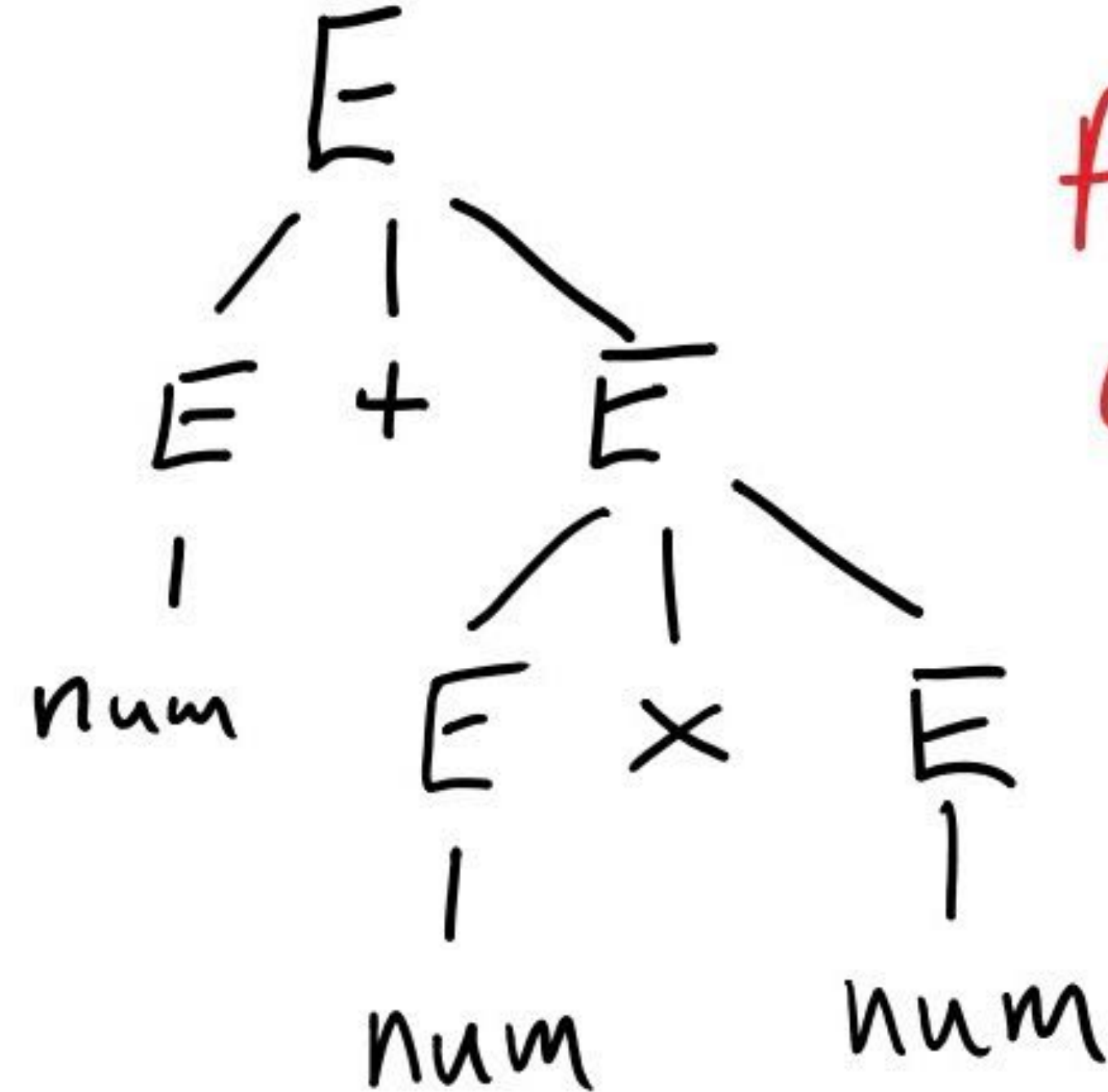
$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree
1 + 2 x 3



All leaves
are terminals

Grammar for Mathematical Expressions

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

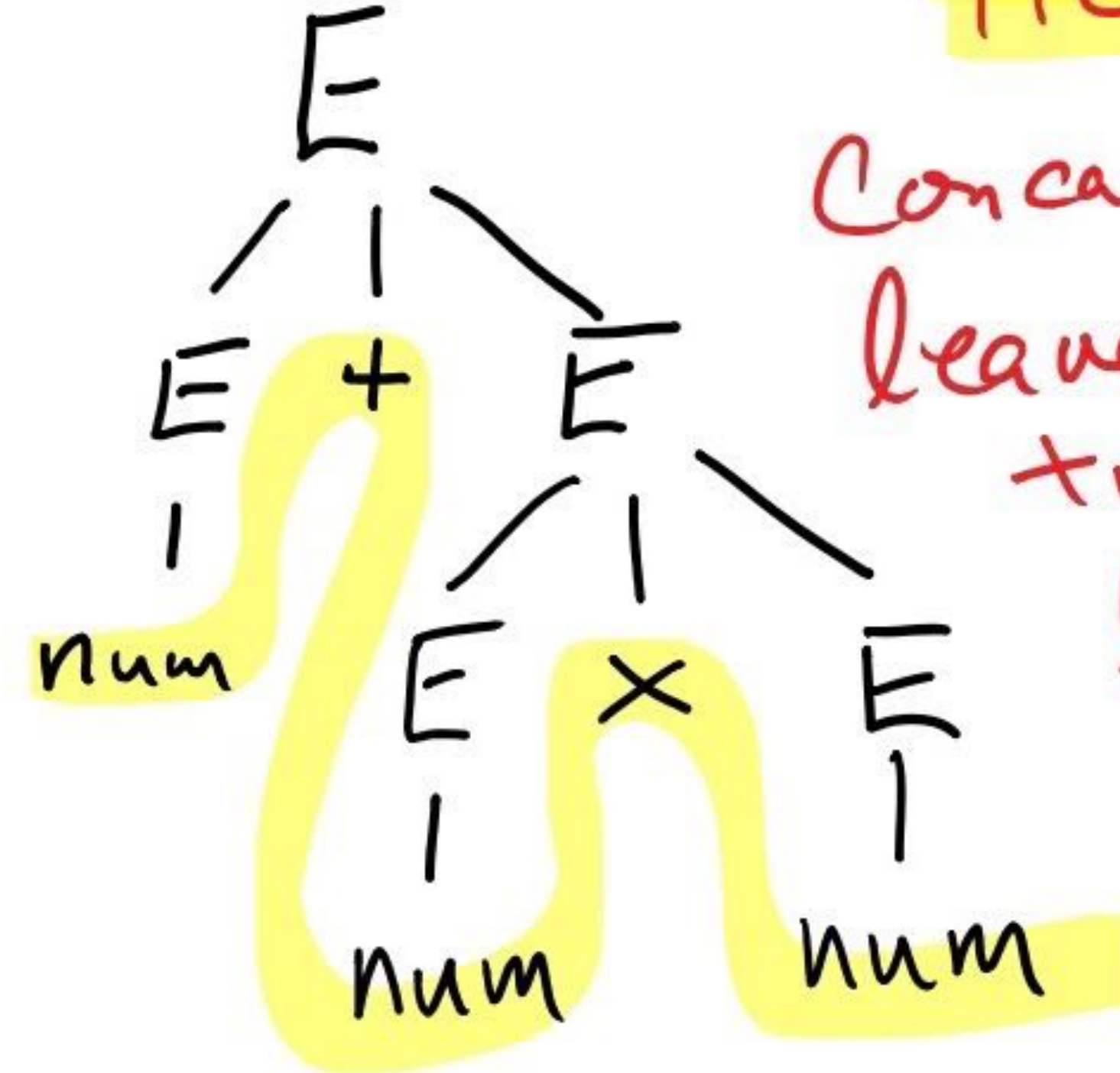
$$E \rightarrow E \times E$$

$$E \rightarrow -E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{num}$$

Parse tree
 $1 + 2 \times 3$



Yield

Concatenate
leaves of
tree from
left to
right.

$\text{num} + \text{num} \times \text{num}$ is in language of grammar.

Grammar Summary

- Grammar defines production rules
- Given various productions of a grammar you can create statements in a language
- Given statements language, can you parse it back into a tree of the production rules?

Parsing Approaches

- **Top Down**
 - Recursive Descent
 - Predictive parser
- **Bottom-Up**
 - Backtracking (recursion)
 - Shift Reduce (token table generation)

Stack Implementation of Shift-Reduce Parsing

Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \text{id}$

Find handles
to reduce

Stack	Input	Action
\$	id+id*id\$	shift
<u>\$id</u>	+id*id\$	reduce $E \rightarrow \text{id}$
\$E	+id*id\$	shift
\$E+	id*id\$	shift
<u>\$E+id</u>	*id\$	reduce $E \rightarrow \text{id}$
\$E+E	*id\$	shift (or reduce?)
\$E+E*	id\$	shift
<u>\$E+E*id</u>	\$	reduce $E \rightarrow \text{id}$
<u>\$E+E*E</u>	\$	reduce $E \rightarrow E * E$
<u>\$E+E</u>	\$	reduce $E \rightarrow E + E$
\$E	\$	accept

How to
resolve
conflicts?

Recursive Descent Parsing

- Manual Parser Creation
- Usually do it with a grammar that's defined as *LL(1)*
- LL = Left to right, Left-most derivation
- 1 = look one token ahead in the token array
- Some caveats:
 - The production rules have to be defined in a certain way
 - Have to be able to determine which production rule ($E \Rightarrow E + E$) to parse based only on next token
 - No left recursions: $E \Rightarrow E + E$ (recursion will never end)

Grammar for Mathematical Expressions

- $E = \text{Expression (start with } E)$

- $E \Rightarrow E + E$

NEED TO REMOVE LEFT
RECURSION

- $E \Rightarrow E - E$

- $E \Rightarrow E * E$

- $E \Rightarrow -E$

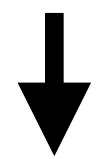
- $E \Rightarrow (E)$

- $E \Rightarrow \text{num}$

Removing Left Recursion

- ◉ $E \Rightarrow E + T$

- ◉ $E \Rightarrow T$



- ◉ $E \Rightarrow T E'$

- ◉ $E' \Rightarrow + T E'$

- ◉ $E' \Rightarrow \textit{epsilon} (\epsilon)$

- ◉ Read more: <http://www.csd.uwo.ca/~moreno/CS447/Lectures/Syntax.html/node8.html>

Grammar for Mathematical Expressions

- ◉ E = Expression, T = Term, F = Factor, {A, B} => placeholders

- ◉ E => T A

- ◉ A => + T A

- ◉ A => - T A

- ◉ A => *epsilon*

- ◉ T => F


- ◉ T => F * F

- ◉ T => F / F

- ◉ F => (E)

- ◉ F => -F

- ◉ F => number

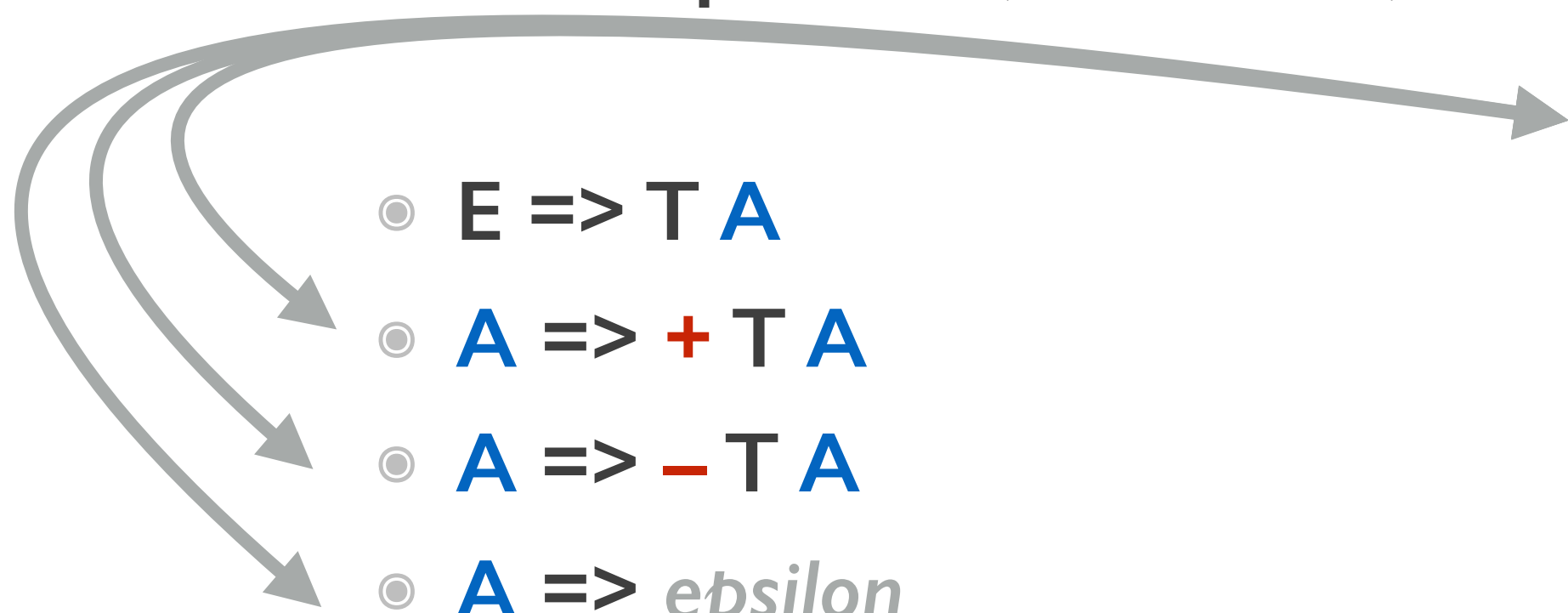


```
function parseExpression() {  
    var t = parseTerm();  
    var a = parseA();  
    return TreeNode('Expression', t, a);  
}
```

Grammar for Mathematical Expressions

- ◉ E = Expression, T = Term, F = Factor, {A, B} => placeholders

- ◉ E => T A
- ◉ A => + T A
- ◉ A => - T A
- ◉ A => *epsilon*
- ◉ T => F
- ◉ T => F * F
- ◉ T => F / F
- ◉ F => (E)
- ◉ F => -F
- ◉ F => number



```
function parseA() {  
    var nextToken = this.peek();  
    if (nextToken.name == "ADD") {  
        this.get();  
        var t = parseTerm();  
        var a = parseA();  
        return new TreeNode("A", "+", t, a);  
    } else if (...) { ...  
    } else {  
        return new TreeNode("A"); // no children  
    }  
}
```

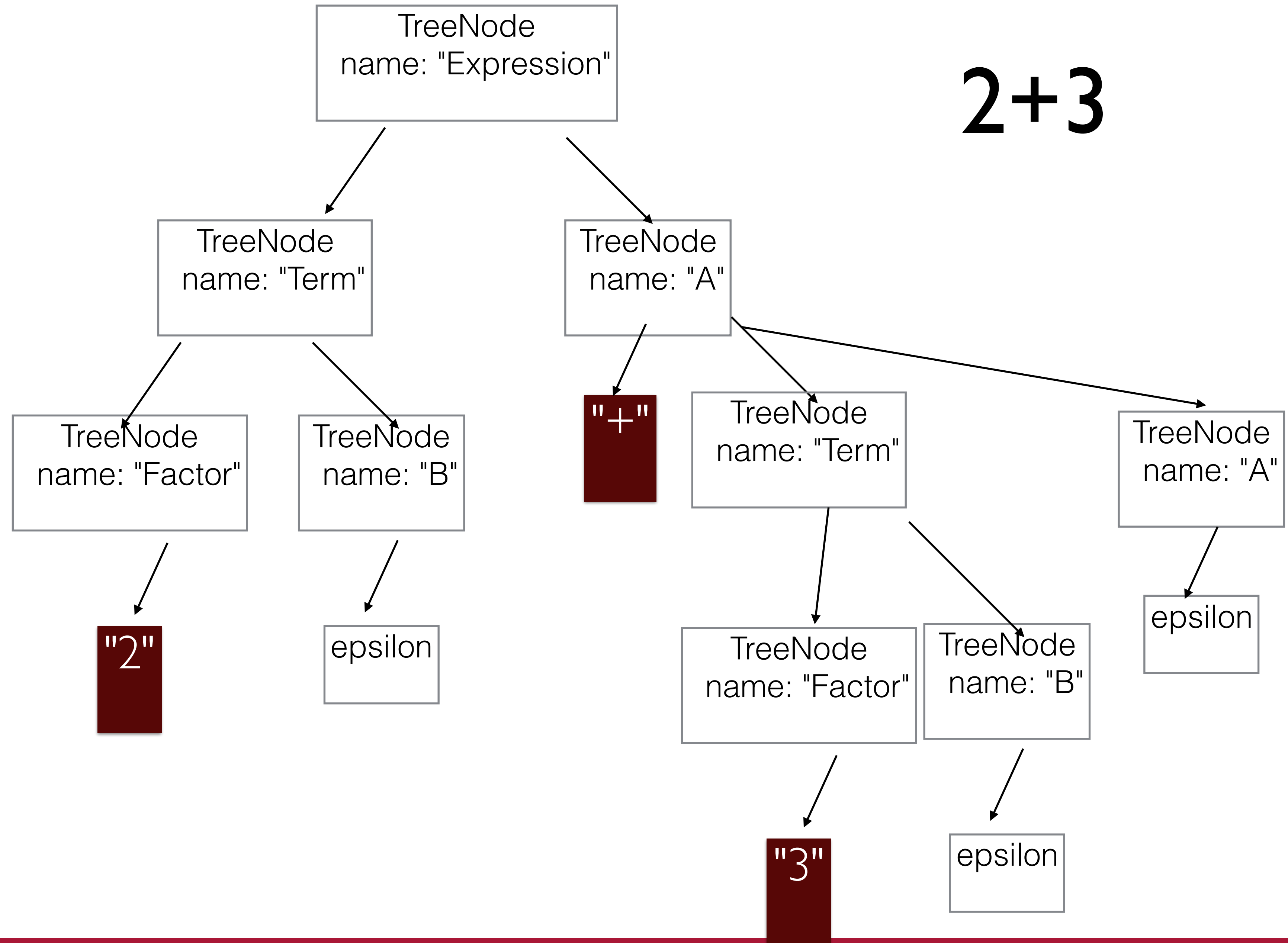
Grammar for Mathematical Expressions

- ◉ E = Expression, T = Term, F = Factor, {A, B} => placeholders

- ◉ $E \Rightarrow T A$
- ◉ $A \Rightarrow + T A$
- ◉ $A \Rightarrow - T A$
- ◉ $A \Rightarrow \textit{epsilon}$
- ◉ $T \Rightarrow F B$
- ◉ $B \Rightarrow * F B$
- ◉ $B \Rightarrow / F B$
- ◉ $B \Rightarrow \textit{epsilon}$
- ◉ $F \Rightarrow (E)$
- ◉ $F \Rightarrow -F$
- ◉ $F \Rightarrow \textit{number}$

```
function parseA() {  
    var nextToken = this.peek();  
    if (nextToken.name == "ADD") {  
        this.get();  
        var t = parseTerm();  
        var a = parseA();  
        return new TreeNode("A", "+", t, a);  
    } else if (...) { ...  
    } else {  
        return new TreeNode("A"); // no children  
    }  
}
```

2+3



Grammar

- ◎ **Examples of Grammars**

- <http://zaach.github.io/jison/demos/calc/>
- <https://github.com/zaach/jison/blob/master/examples/jscore.jison>

- ◎ **Parser Generators**

- Lex/Yacc
- Flex/Bison
- Jison - jison.org

- ◎ **Parsers**

- Esprima - <http://esprima.org/>

```

1 // Life, Universe, and Everything
2 var answer = 6 * 7;
3

```

No error

Syntax node location info (start, end):

- ☐ Index-based range
- ☐ Line and column-based
- ☐ Attach comments

Syntax

Tree

Tokens

```

{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "answer"
          },
          "init": {
            "type": "BinaryExpression",
            "operator": "*",
            "left": {
              "type": "Literal",
              "value": 6,
              "raw": "6"
            },
            "right": {
              "type": "Literal",
              "value": 7,
              "raw": "7"
            }
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "script"
}

```



```
1 // Life, Universe, and Everything
2 var answer = 6 * 7;
3
```

No error

Syntax node location info (start, end):

- ☐ Index-based range
- ☐ Line and column-based
- ☐ Attach comments

Syntax

Tree

Tokens

Expand All

Collapse All

- ▼ Program body [1]
 - ▼ VariableDeclaration
 - ▼ declarations [1]
 - ▼ VariableDeclarator
 - ▼ id
 - ▼ Identifier
 - name: answer
 - ▼ init
 - ▼ BinaryExpression
 - operator: *
 - ▼ left
 - ▼ Literal
 - value: 6
 - raw: 6
 - ▼ right
 - ▼ Literal
 - value: 7
 - raw: 7

kind: var

```

1 // Life, Universe, and Everything
2 var answer = 6 * 7;
3

```

No error

Syntax node location info (start, end):

- ☐ Index-based range
- ☐ Line and column-based
- ☐ Attach comments

Syntax

Tree

Tokens

```

[
  {
    "type": "Keyword",
    "value": "var"
  },
  {
    "type": "Identifier",
    "value": "answer"
  },
  {
    "type": "Punctuator",
    "value": "="
  },
  {
    "type": "Numeric",
    "value": "6"
  },
  {
    "type": "Punctuator",
    "value": "*"
  },
  {
    "type": "Numeric",
    "value": "7"
  },
  {
    "type": "Punctuator",
    "value": ";"
  }
]

```

Key Takeaways

Key Takeaways

- Your program is itself a piece of data
- Compilers first two main tasks are lexing and parsing
- Programming languages are defined by grammars
 - Backus-Naur Form (BNF)
- There are two primary ways to parse a given language into a grammar, top-down and bottom-up
- Top-down parsing can be done using recursive-descent if the grammar is LL(1) (Left to right, left recursive, 1 token lookahead)

Workshop 1

- **Implement a recursive-descent parser for mathematical expressions**
 - Verify that your tree is correct
 - Output your tree
 - Convert your tree into Reverse Polish Notation