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.

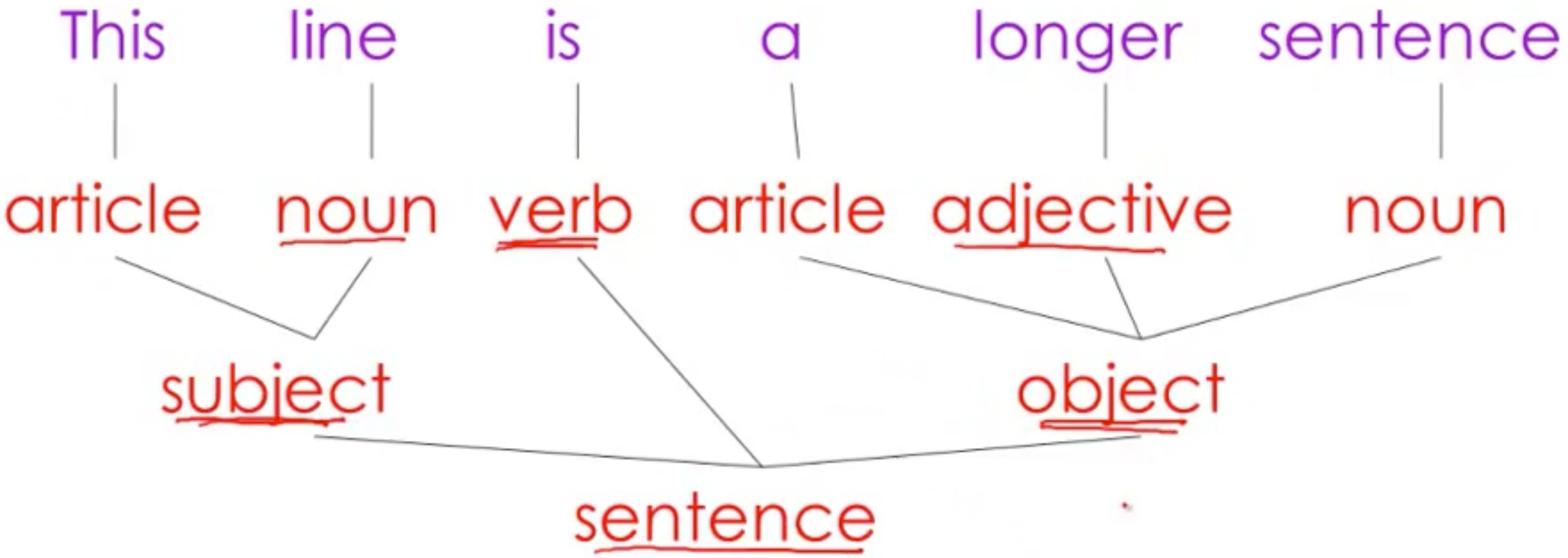
NOT EVERYONE CAN READ THIS

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer 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 noun verb article adjective noun

Parsing



This
line
is
a
longer
sentence
article
noun
verb
article
adjective
noun
subject
object
sentence

How humans read

- **Semantic Meaning**
  - Jack said John forgot <u>his</u> homework at home.
  - There wasn't a <u>single</u> 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.) $$(\div), \Sigma(+1)^p(\times, -) \text{ or } (1), \Sigma(+1)^p(2, 3),$$

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.) $$\Sigma(+1)^n\left\{(\div), \Sigma(+1)^p(\times, -)\right\}$$

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

(8.) $$\text{or } \Sigma(+1)^n\left\{(1), \Sigma(+1)^p(2, 3)\right\}.$$

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

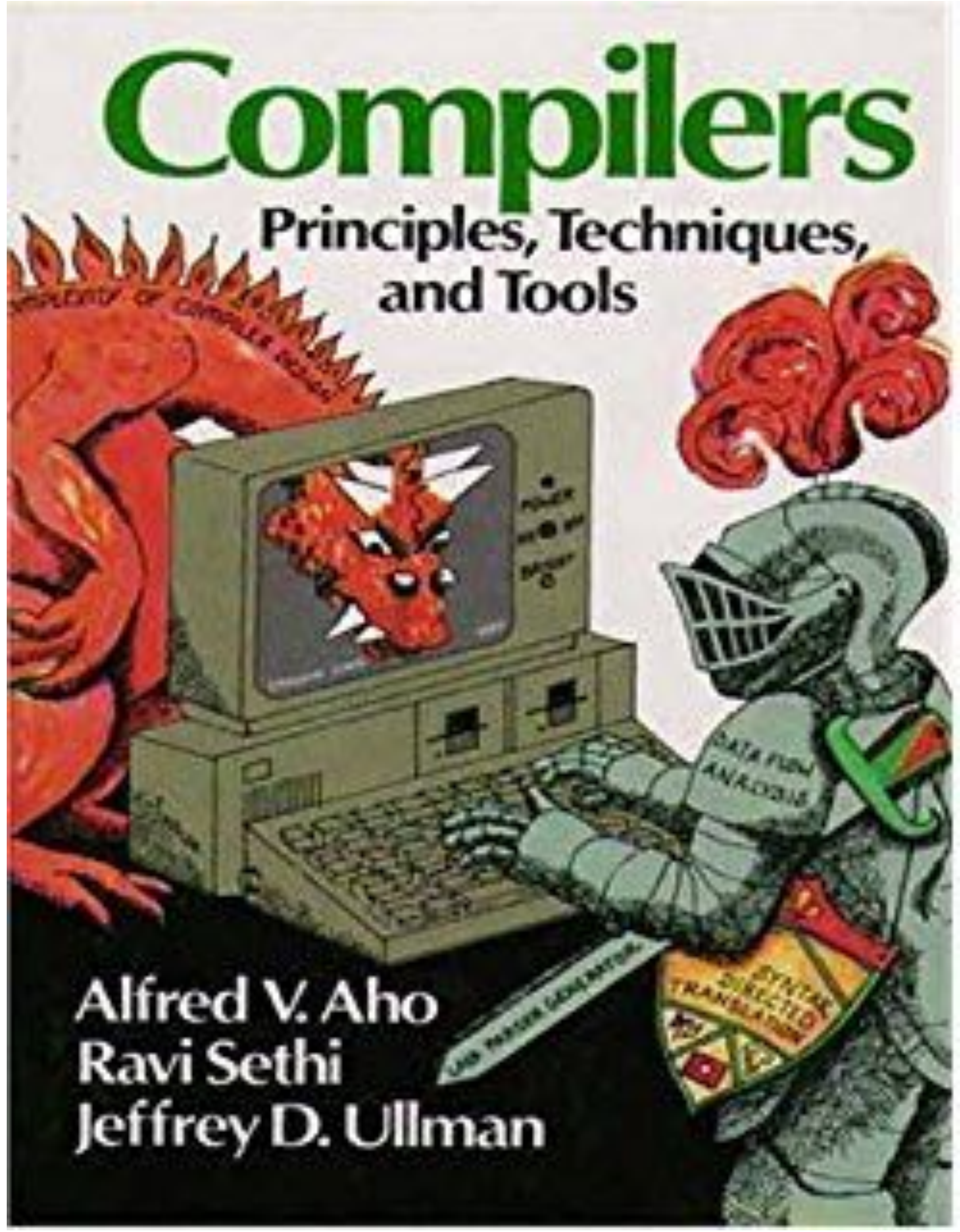
1953

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

Alfred Aho

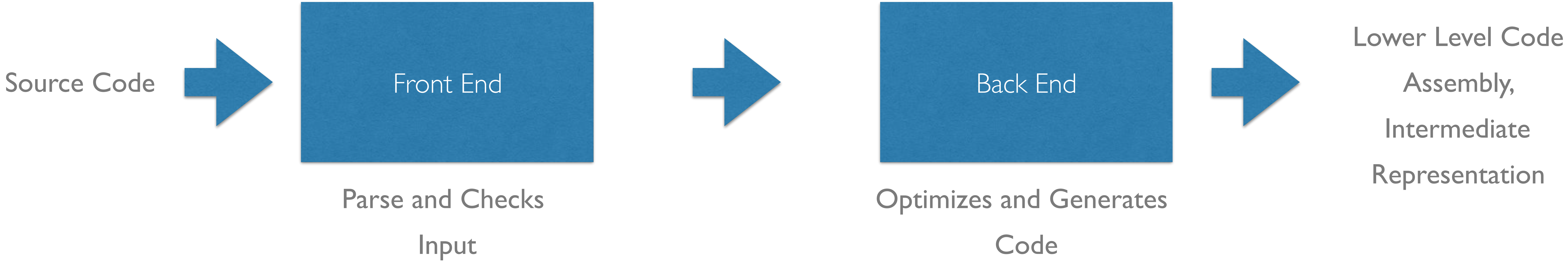
Present

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

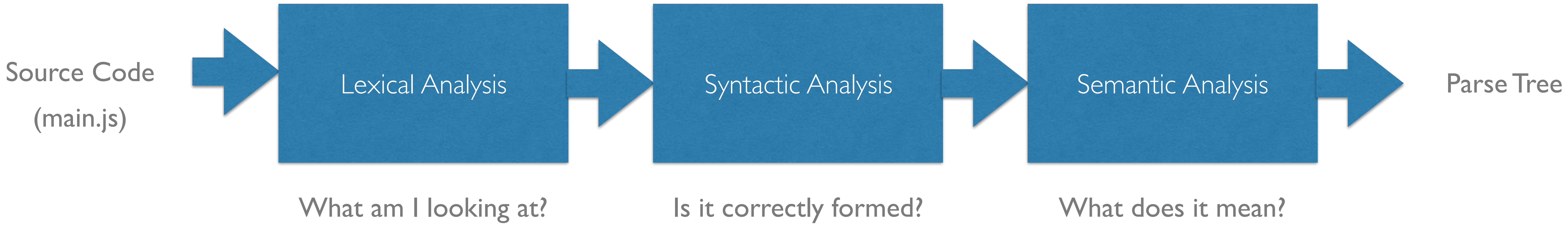


How Compilers Work

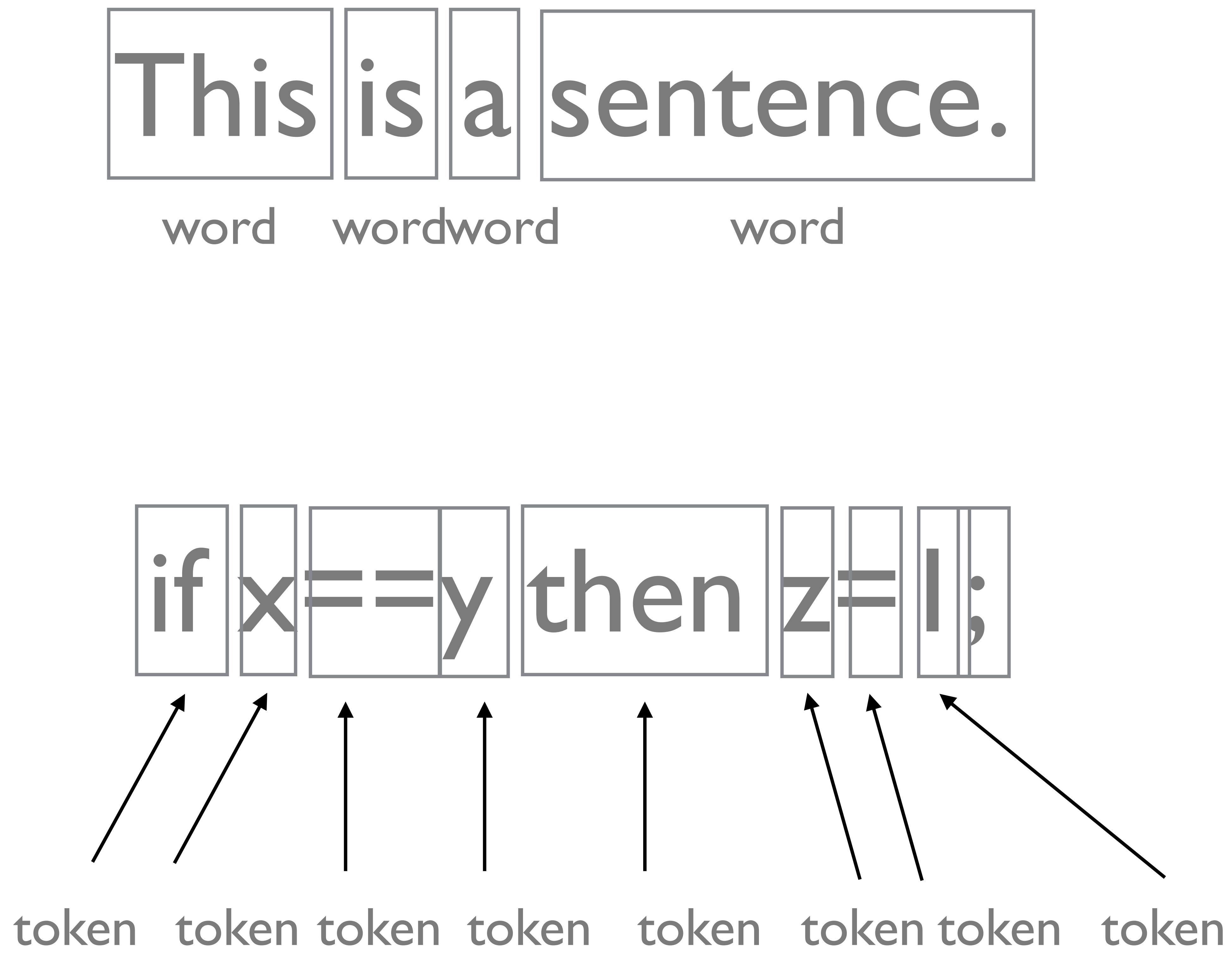
How Compilers Work



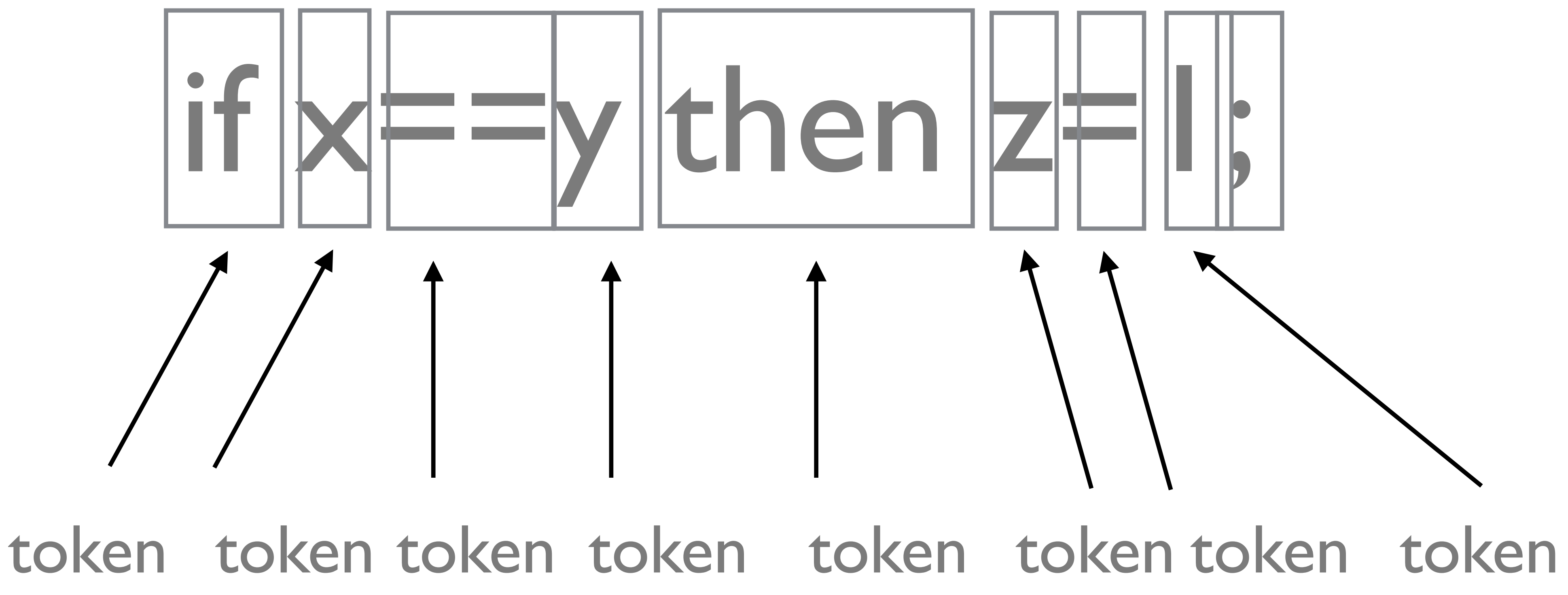
Frontend


Source Code
(main.js)
Lexical Analysis
Syntactic Analysis
Semantic Analysis
Parse Tree
What am I looking at?
Is it correctly formed?
What does it mean?

Lexical Analysis



Lexical Analysis



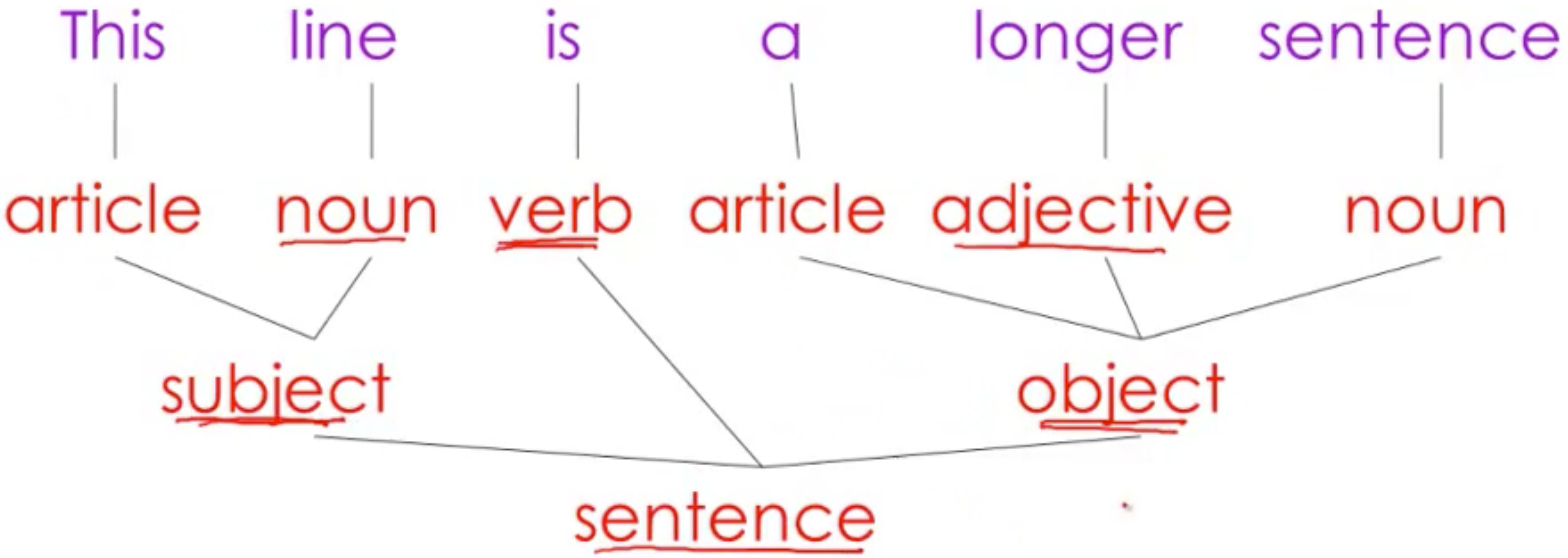
[IF] [ID X] [EQL] [ID Y] [THEN] [ID Z] [ASSIGN] [NUMBER 1] [SEMI]

Often referred to as the "token stream"

Parsing


This line is a longer sentence
article noun verb article adjective noun

Parsing



This
line
is
a
longer
sentence
article
noun
verb
article
adjective
noun
subject
object
sentence

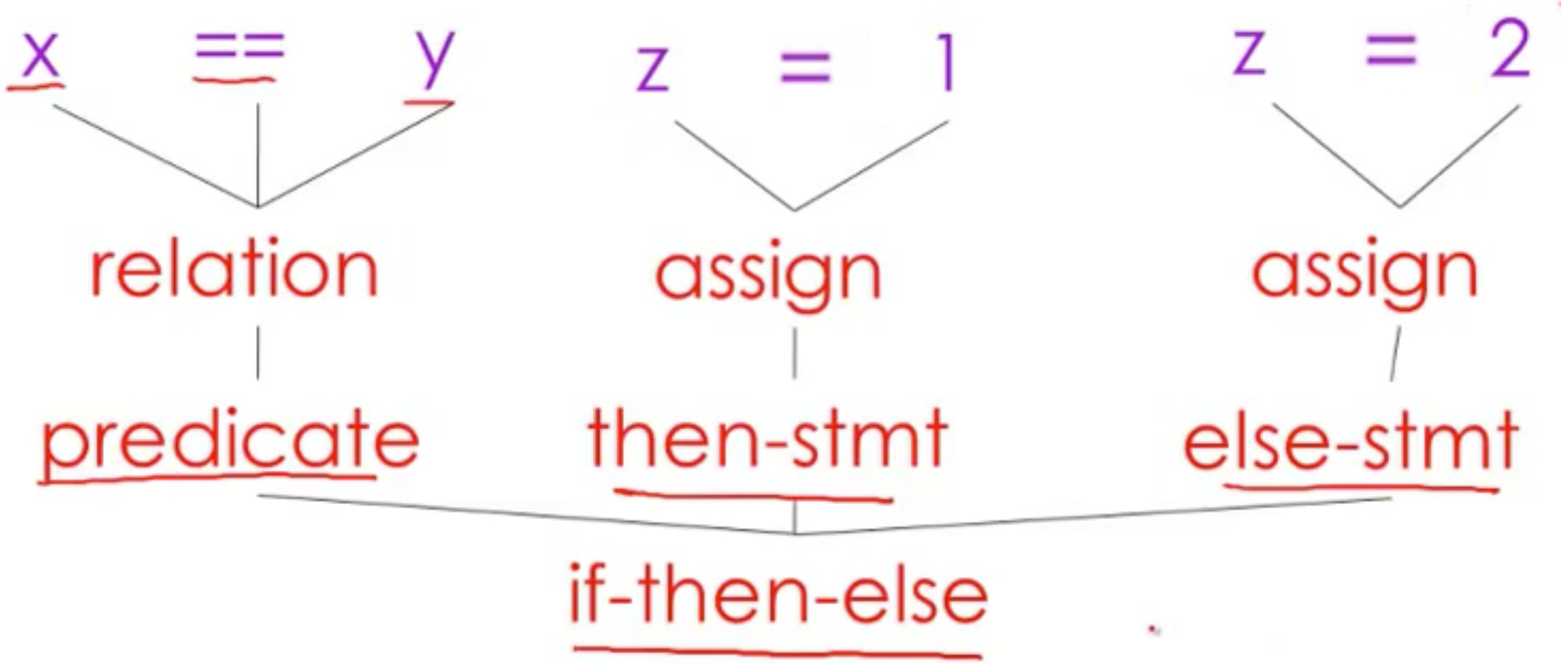
Parsing

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 --> E + E

Grammar for Mathematical Expressions

- E = Expression
- Start your sentence as an abstract idea of an expression: E

- E ==> E + E
- E ==> E - E
- E ==> E * E
- E ==> -E
- E ==> (E)
- E ==> num

token

Grammar for Mathematical Expressions

- E = Expression (start = E)

- E ==> E + E
- E ==> E - E
- E ==> E * E
- E ==> -E
- E ==> (E)
- E ==> num

production rule
left-hand side can produce right hand side

Grammar for Mathematical Expressions

- E = Expression (start = E)

- E ==> E + E
- E ==> E - E
- E ==> E * E
- E ==> -E
- E ==> (E)
- E ==> 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 = Expression (start = E)

- E ==> E + E
- E ==> E - E
- E ==> E * E
- E ==> -E
- E ==> (E)
- E ==> num

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

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, than 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

$1 + 2 \times 3$

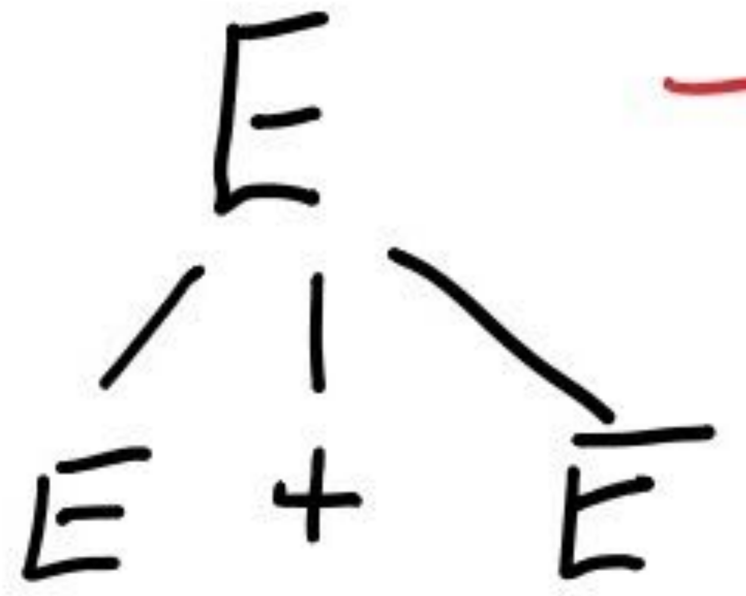
E ← sentence symbol

Grammar for Mathematical Expressions

$$E \to E + E$$
$$E \to E - E$$
$$E \to E \times E$$
$$E \to - E$$
$$E \to (E)$$
$$E \to \text{num}$$

Parse tree

$1 + 2 \times 3$



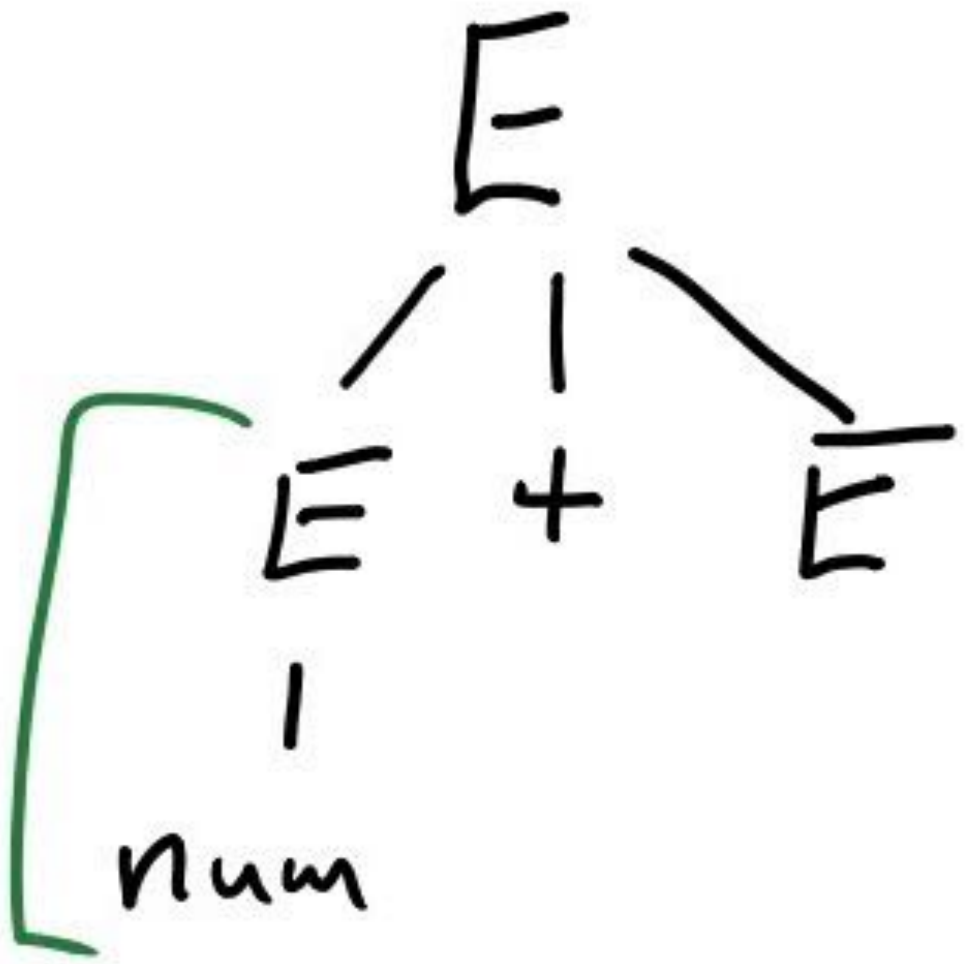
Parent = LHS
Children = RHS

Grammar for Mathematical Expressions

$$E \to E + E$$
$$E \to E - E$$
$$E \to E \times E$$
$$E \to -E$$
$$E \to (E)$$
$$E \to \text{num}$$

Parse tree

$1 + 2 \times 3$



Grammar for Mathematical Expressions

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E \times E$

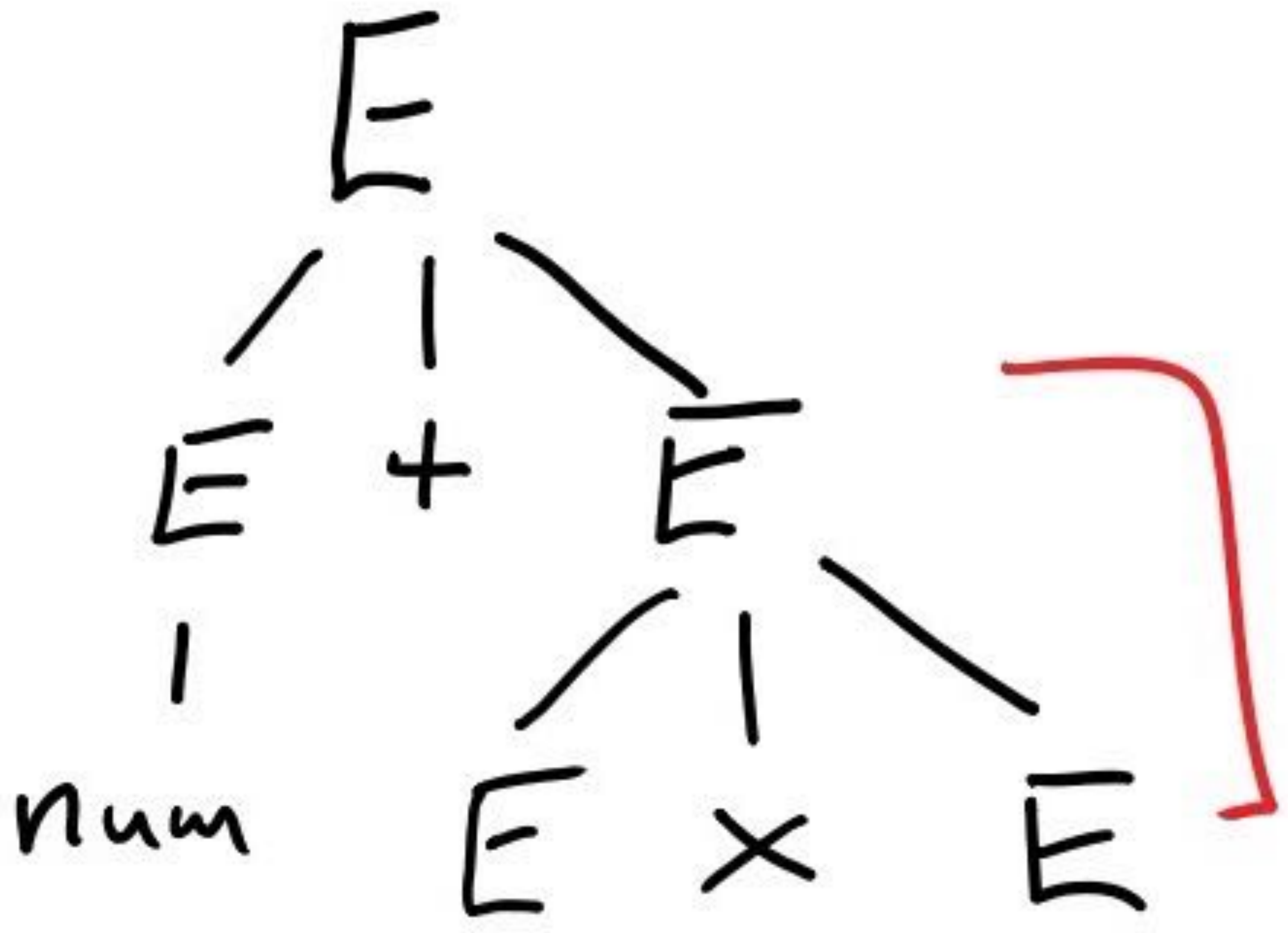
$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow \text{num}$

Parse tree

$1 + 2 \times 3$

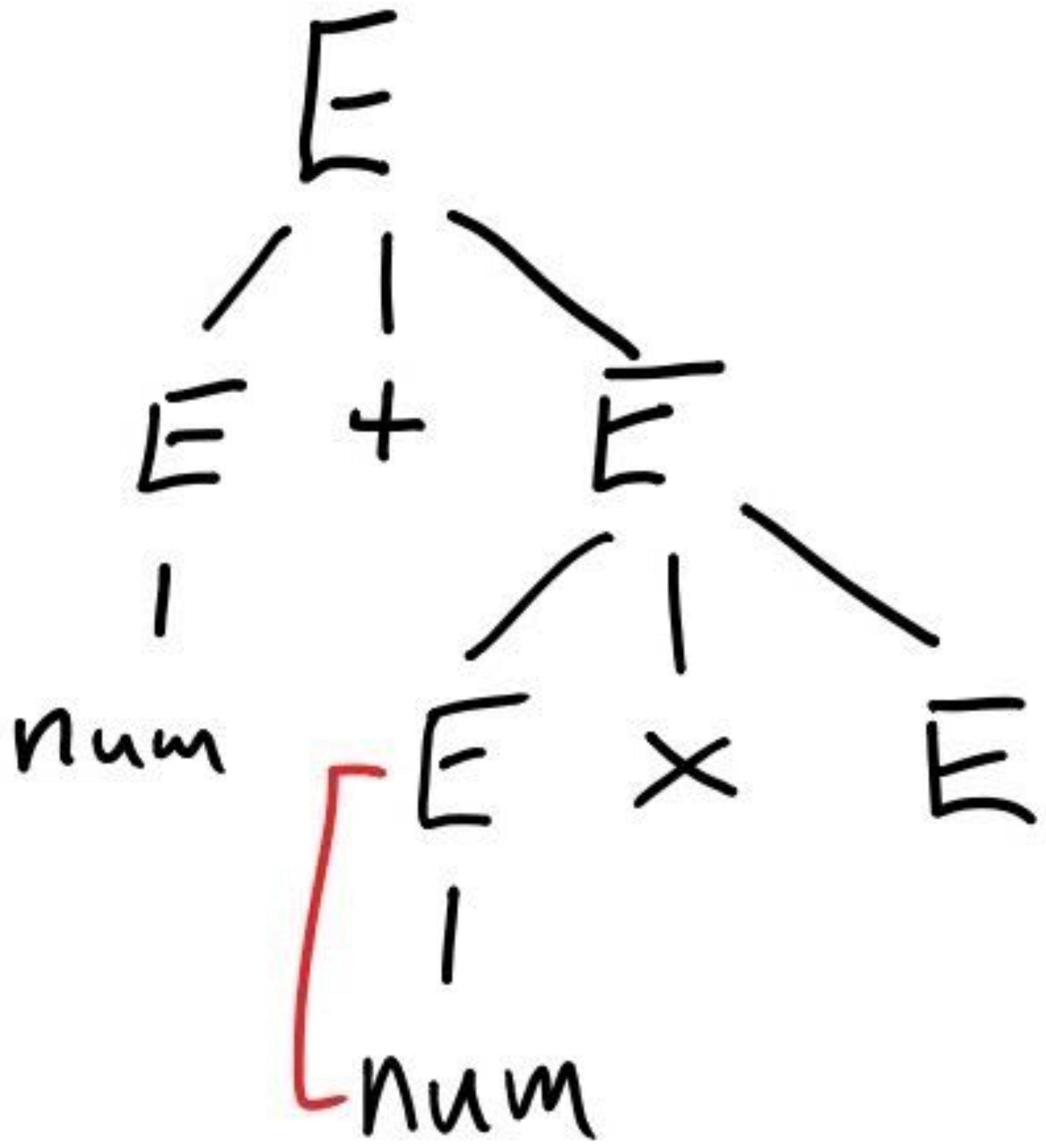


Grammar for Mathematical Expressions

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E \times E$$
$$E \rightarrow -E$$
$$E \rightarrow (E)$$
$$E \rightarrow \text{num}$$

Parse tree

$1 + 2 \times 3$

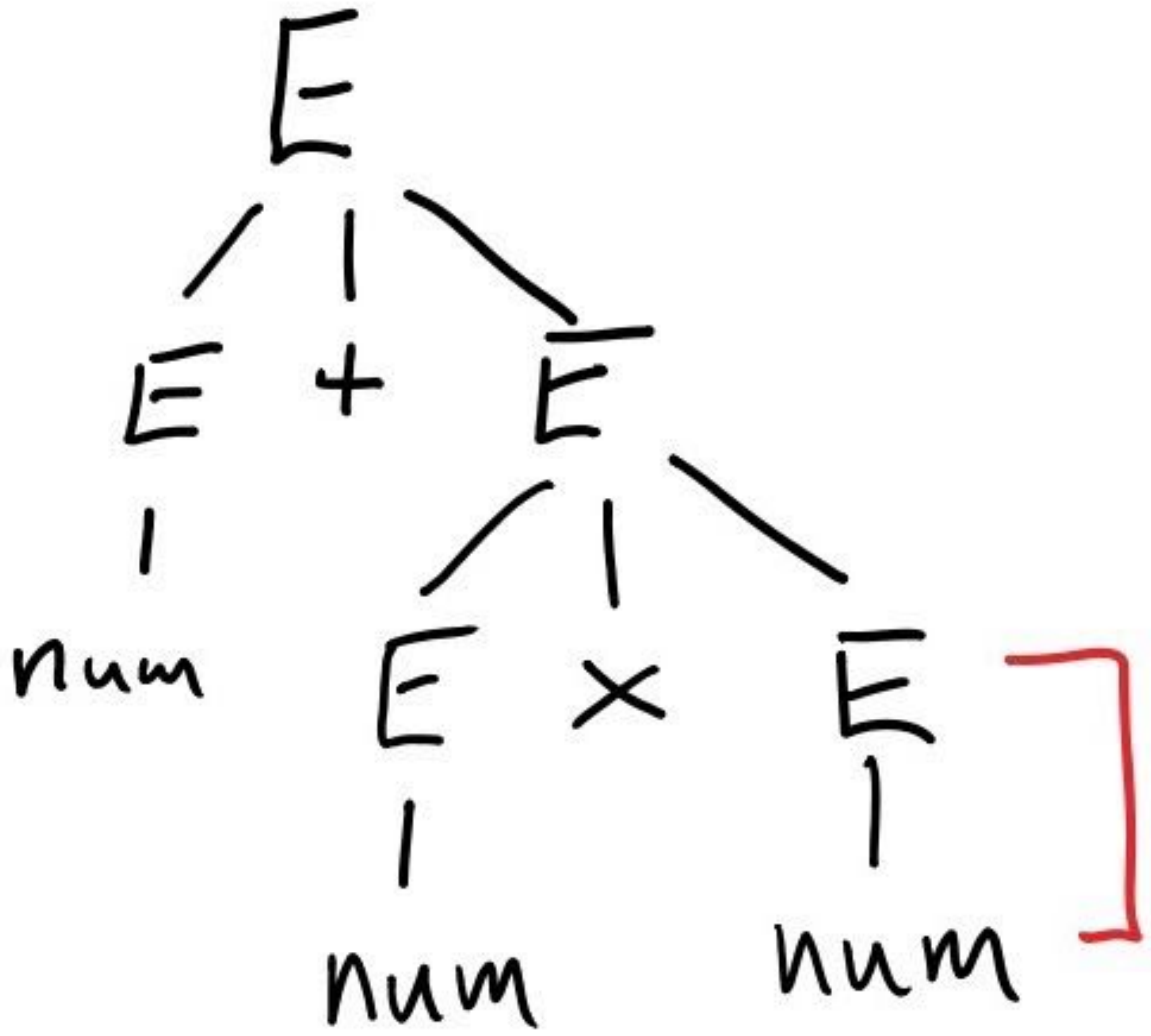


Grammar for Mathematical Expressions

$$E \to E + E$$
$$E \to E - E$$
$$E \to E \times E$$
$$E \to -E$$
$$E \to (E)$$
$$E \to \text{num}$$

Parse tree

$1 + 2 \times 3$



Grammar for Mathematical Expressions

$E \to E + E$

$E \to E - E$

$E \to E \times E$

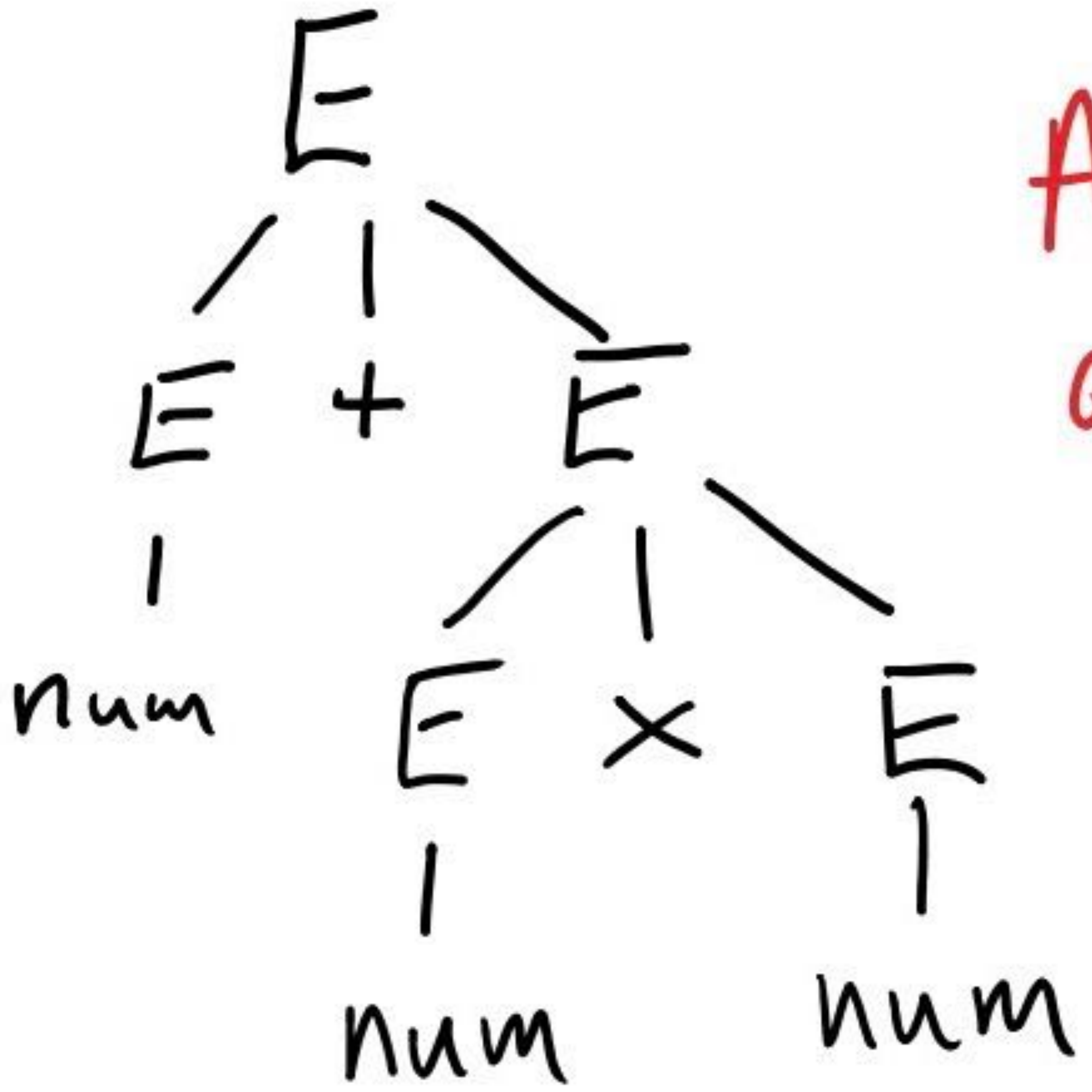
$E \to - E$

$E \to (E)$

$E \to \text{num}$

Parse tree

$1 + 2 \times 3$



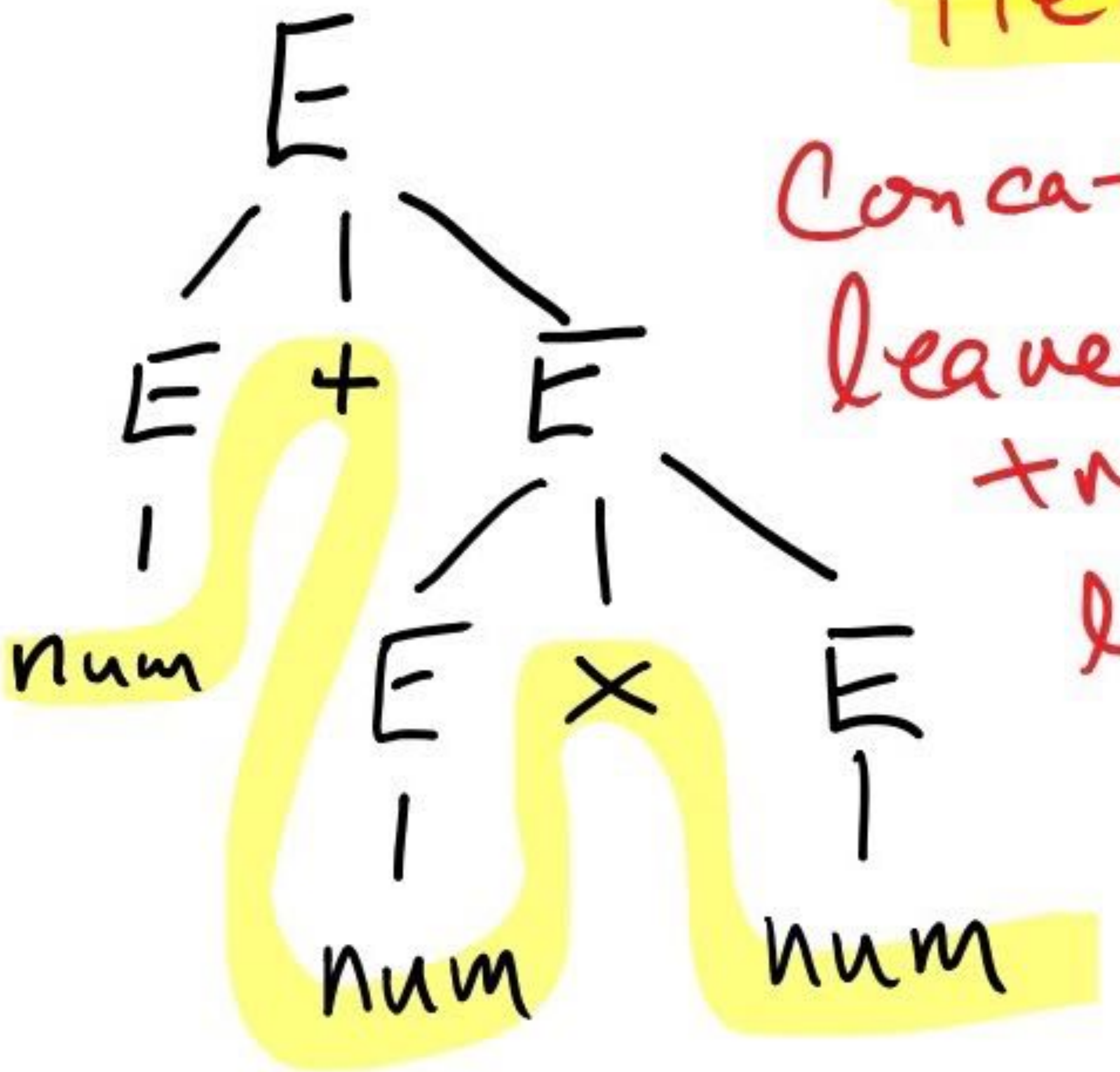
All leaves are terminals

Grammar for Mathematical Expressions

$$E \to E + E$$
$$E \to E - E$$
$$E \to E \times E$$
$$E \to -E$$
$$E \to (E)$$
$$E \to \text{num}$$

Parse tree

$1 + 2 \times 3$



Yield

Concatenate leaves of tree from left to right.

num + num × 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:

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow \mathbf{id}$

Stack	Input	Action
$	**id+id*id$**	shift
$\underline{\mathbf{id}}$	**+id*id$**	reduce $E \rightarrow \mathbf{id}$
$E	**+id*id$**	shift
$E+	**id*id$**	shift
$E+\underline{\mathbf{id}}$	***id$**	reduce $E \rightarrow \mathbf{id}$
$E+E	***id$**	shift (or reduce?)
$E+E*	**id$**	shift
$E+E*\underline{\mathbf{id}}$	**$**	reduce $E \rightarrow \mathbf{id}$
$E+\underline{E*E}$	**$**	reduce $E \rightarrow E * E$
$\underline{E+E}$	**$**	reduce $E \rightarrow E + E$
$E	**$**	accept

Find handles to reduce

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 => E + E) to parse based only on next token
  - No left recursions: E => E + E (recursion will never end)

Grammar for Mathematical Expressions

- E = Expression (start with E)

- E ==> E + E
- E ==> E - E
- E ==> E * E
- E ==> -E
- E ==> (E)
- E ==> num

NEED TO REMOVE LEFT RECURSION

Removing Left Recursion

- E => E + T
- E => T

↓

- E => T E'
- E' => + T E'
- E' => *epsilon* (ε)

- 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 => T A
- A => + T A
- A => – T A
- A => *epsilon*
- T => F B
- B => * F B
- B => / F B
- B => *epsilon*
- 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
  }
}
```

Parsing (just) a Term

Production Rules

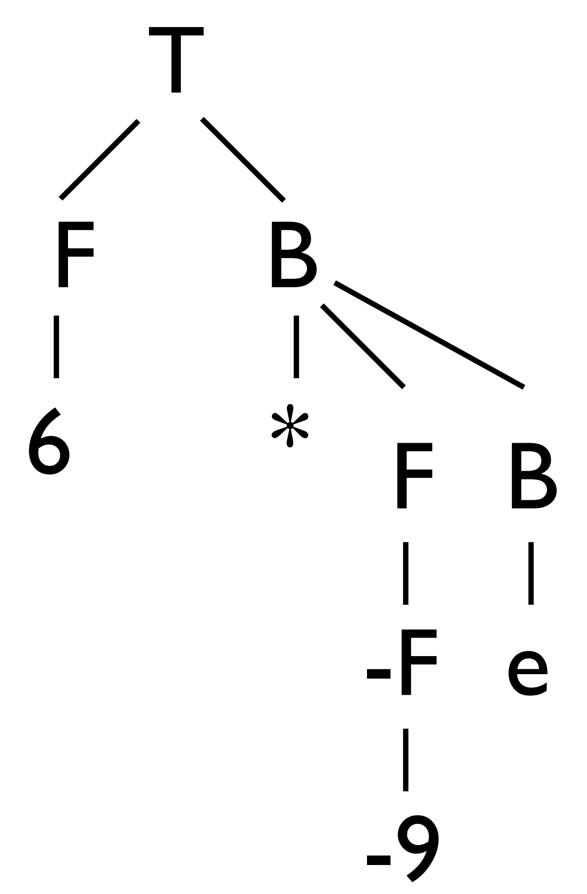
- E => T A
- A => + T A
- A => – T A
- A => *epsilon*
- T => F B
- B => * F B
- B => / F B
- B => *epsilon*
- F => (E)
- F => -F
- F => <number>

Input String	Lexed Token Stream
"6 * -9"	[NUMBER 6] -> [OP *] -> [SYMBOL -] -> [NUMBER 9]

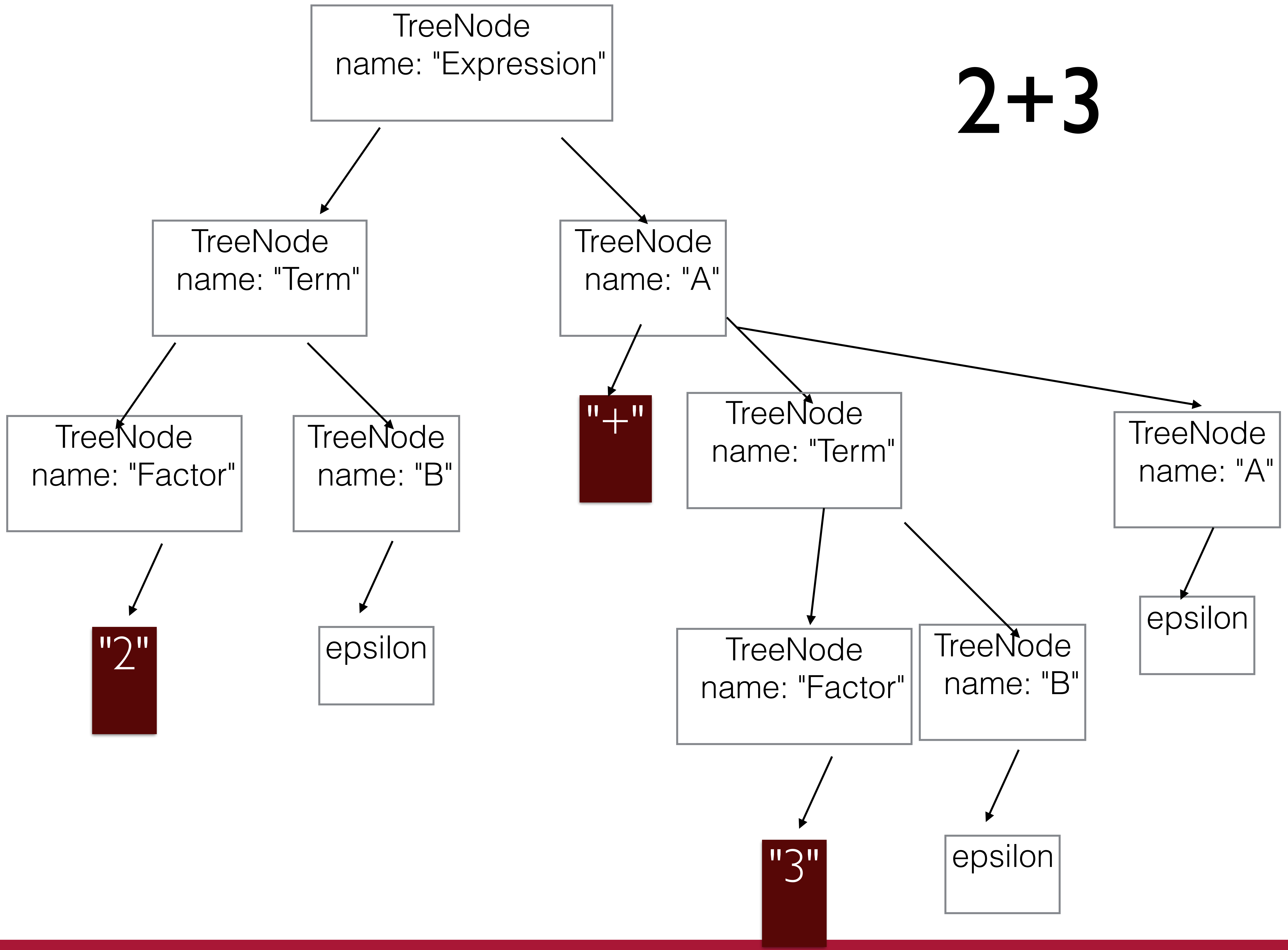
Derivation Steps

T	[NUMBER 6] -> [OP *] -> [SYMBOL -] -> [NUMBER 9]
F B	[NUMBER 6] -> [OP *] -> [SYMBOL -] -> [NUMBER 9]
6 B	[OP *] -> [SYMBOL -] -> [NUMBER 9]
6 * F B	[SYMBOL -] -> [NUMBER 9]
6 * -F B	[NUMBER 9]
6 * -9 B	
6 * -9	

Parsed Tree



2+3



TreeNode
name: "Expression"
TreeNode
name: "Term"
TreeNode
name: "A"
TreeNode
name: "Factor"
TreeNode
name: "B"
"+"
TreeNode
name: "Term"
TreeNode
name: "A"
"2"
epsilon
TreeNode
name: "Factor"
TreeNode
name: "B"
epsilon
"3"
epsilon

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/

```javascript
// Life, Universe, and Everything
var answer = 6 * 7;
```

No error

Syntax node location info (start, end):

- [ ] Index-based range
- [ ] Line and column-based

- [ ] Attach comments

Syntax | Tree | Tokens


```json
{
    "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"
}
```

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

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

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

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": ";"
    }
]
```


No error

Syntax node location info (start, end):

- [ ] Index-based range
- [ ] Line and column-based

- [ ] Attach comments

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