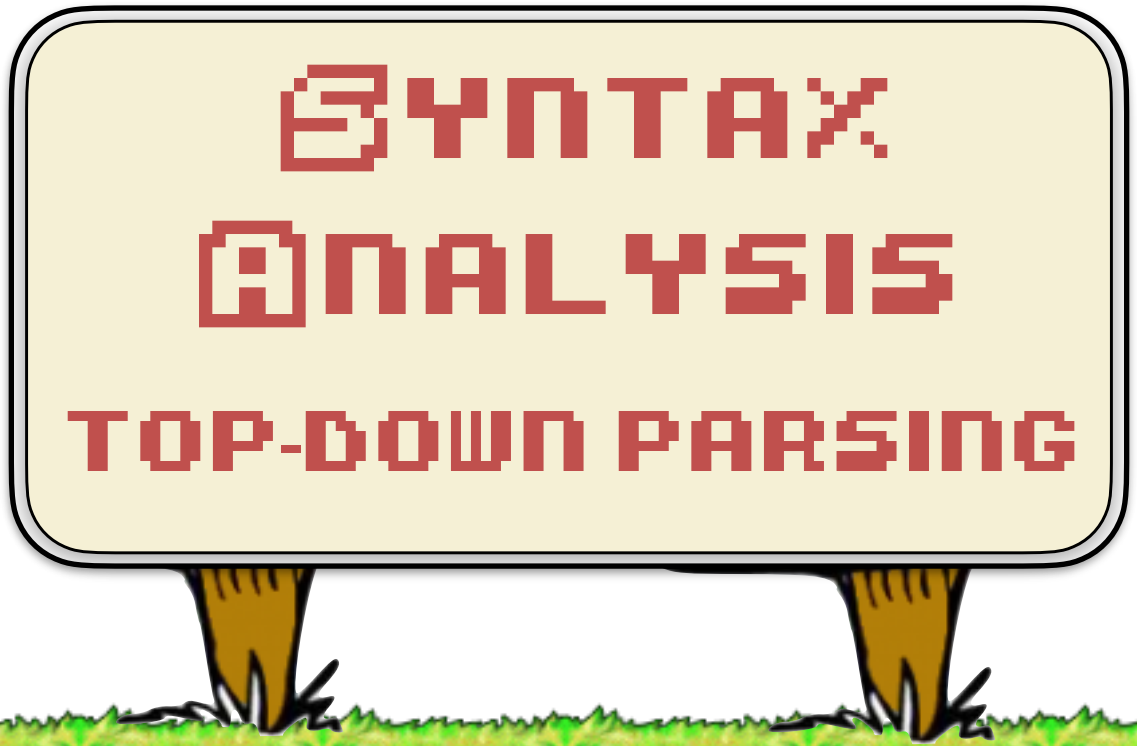
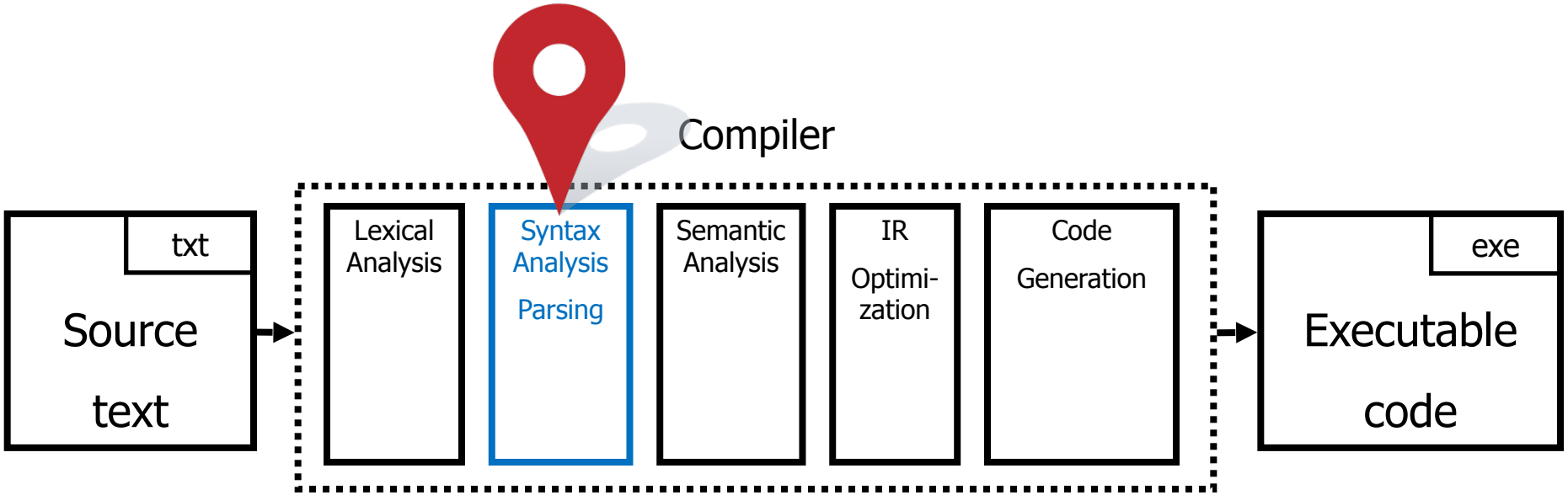
THEORY OF COMPILATION

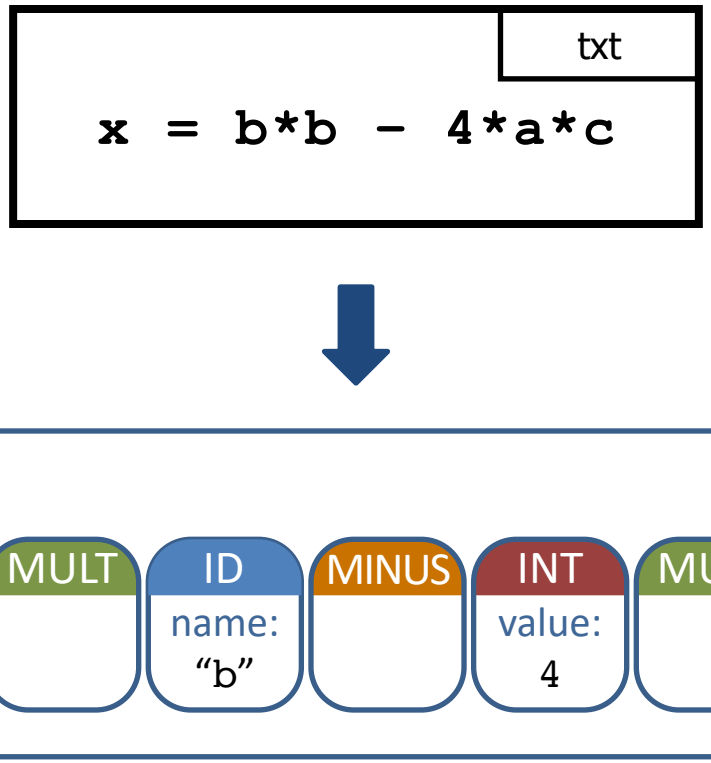
LECTURE 02



You are here

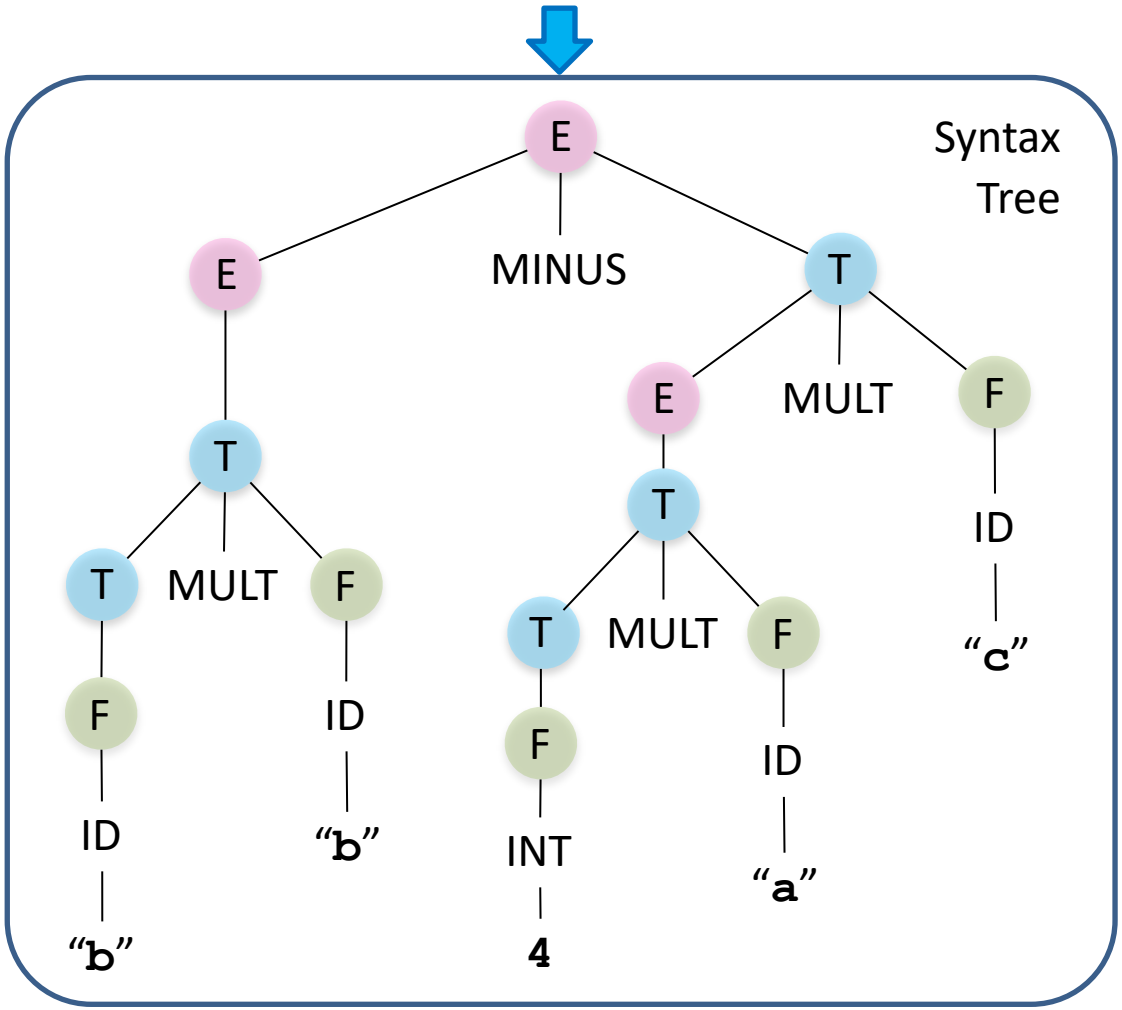

Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code
Generation
exe
Executable code

Previously: from characters to tokens



Today: from tokens to syntax tree

⟨ID,"b"⟩ ⟨MULT⟩ ⟨ID,"b"⟩ ⟨MINUS⟩ ⟨INT,4⟩ ⟨MULT⟩ ⟨ID,"a"⟩ ⟨MULT⟩ ⟨ID,"c"⟩



Parsing

- Goals
  - Decide whether the sequence of tokens a valid program in the source language
  - Construct a structured representation of the input
  - Error detection and reporting
- Challenges
  - How do you describe the programming language?
  - How do you check validity of an input?
  - Where do you report an error?

Some people, when confronted with a problem, think —

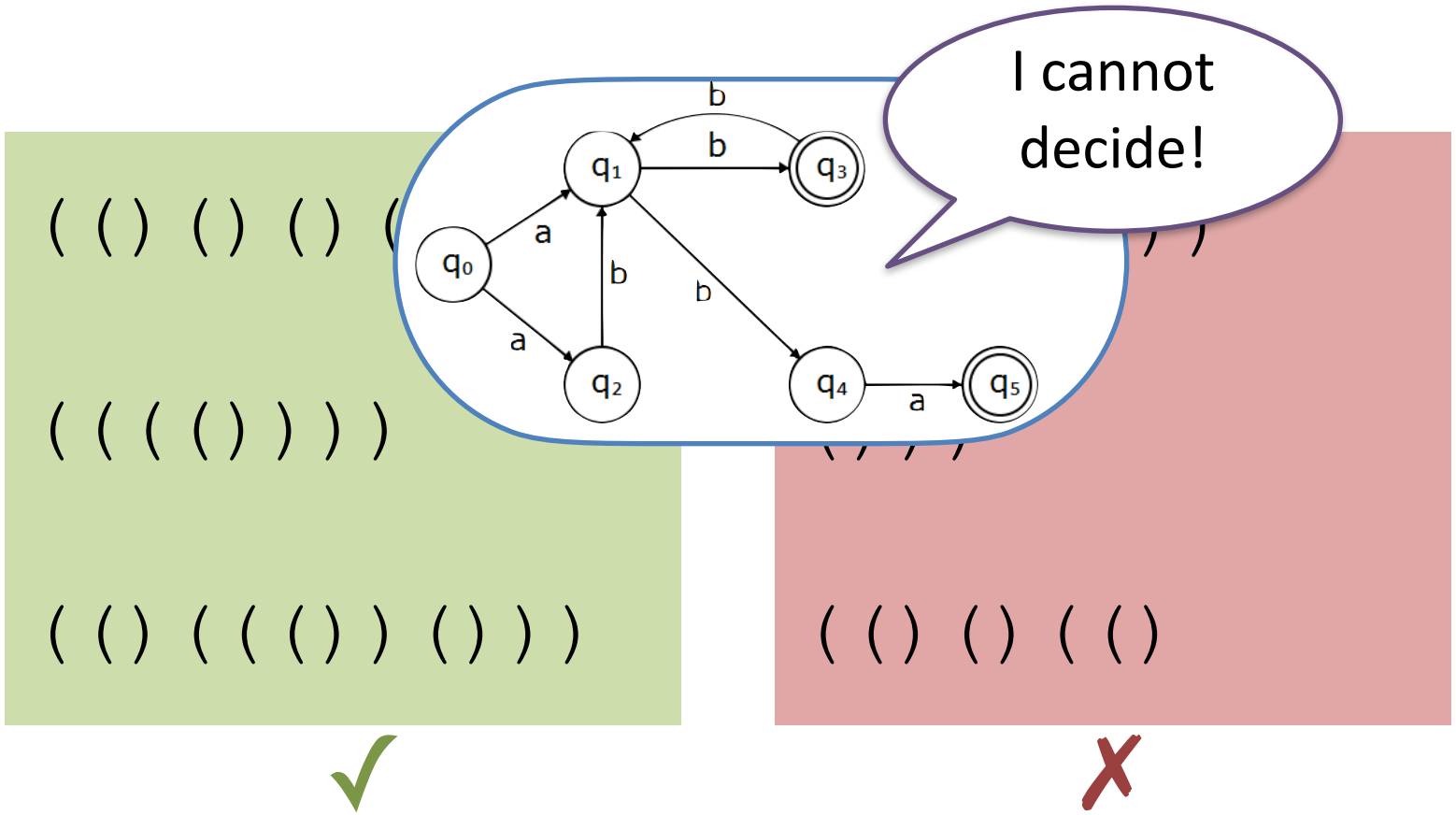
“I know, I'll use regular expressions.”

Now they have two problems.

The plural of *regex* is *regrets*.

Regular Expressions: Computability Bound

$L^{()}$ = language of balanced parentheses over Σ = {'(', ')'}



Context Free Grammars

$$G = \langle V, T, P, S \rangle$$

- V – non-terminals (syntactic variables)
- T – terminals (tokens)
- P – derivation rules
  - Each rule of the form

    $V \longrightarrow (T \cup V)^*$
- S – start symbol

ab^+c

$S \longrightarrow aKc$

$K \longrightarrow b$

$K \longrightarrow bK$

$S \Rightarrow aKc$

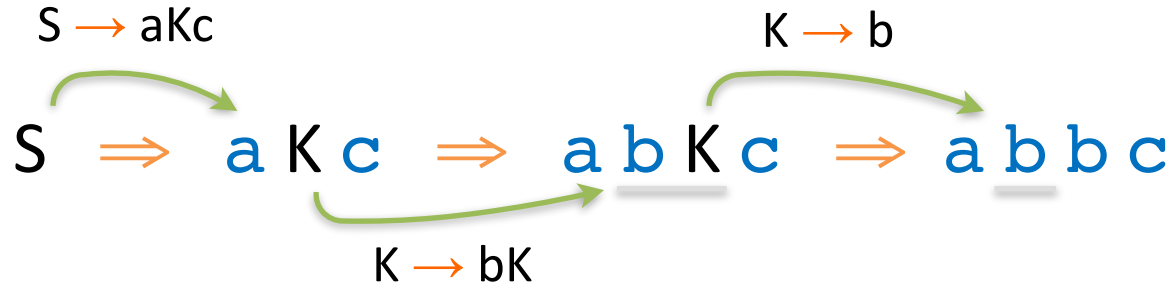
$\Rightarrow abKc$

$\Rightarrow abbKc$

$\Rightarrow abbbc$

Terminology

- **Derivation** — a sequence of replacements of non-terminals using the derivation rules



- **Language** — the set of strings of terminals ("words") derivable from the start symbol

- **Sentential form** — the result of a partial derivation, in which there may be non-terminals

Context Free Grammars > Regular Expressions

$L^{()}$ from before is accepted by the following CFG:

$$S \rightarrow \varepsilon$$

$$S \rightarrow (S)$$

$$S \rightarrow S S$$

$S \Rightarrow (S)$
$\Rightarrow (S S)$
$\Rightarrow ((S) S)$
$\Rightarrow (( ) S)$
$\Rightarrow (( ) (S))$
$\Rightarrow (( ) (S S))$
$\Rightarrow (( ) ((S) S))$
$\Rightarrow (( ) ((S) (S)))$
$\Rightarrow (( ) (((S)) (S)))$
$\Rightarrow (( ) ((( )) (S)))$
$\Rightarrow (( ) ((( )) ( )))$

Example

S → S ; S

S → id := E

E → id | num

| E + E | E * E | (E)

V = { S, E }

T = { id, num, +, *, (,), :=, ; }

Derivation

id := id ; id := id + id

lex

input

```
x := z;
y := x + z
```

grammar

S → S ; S

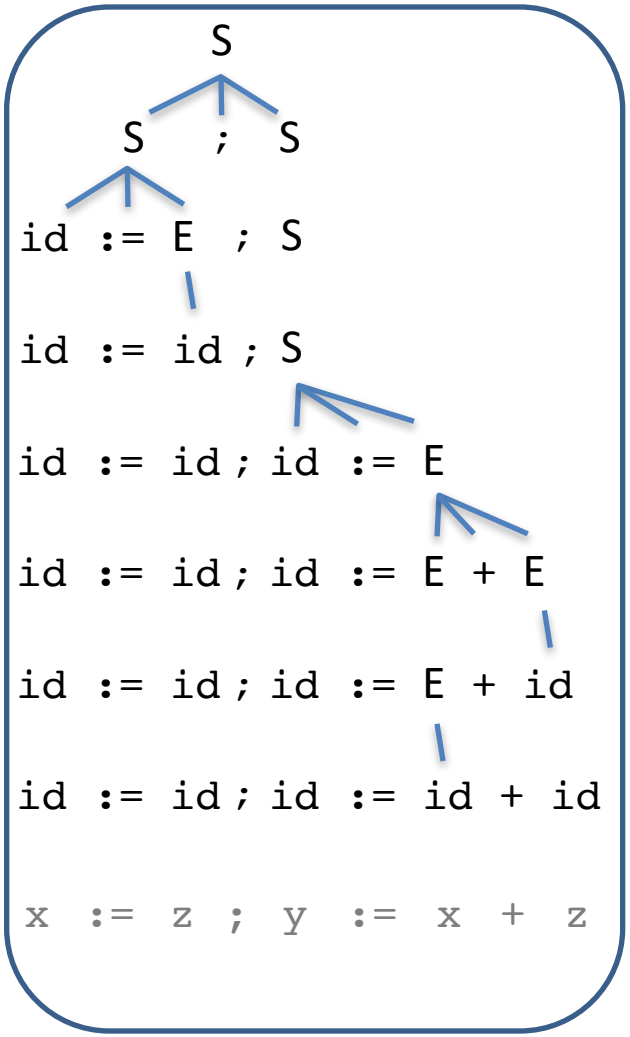
S → id := E

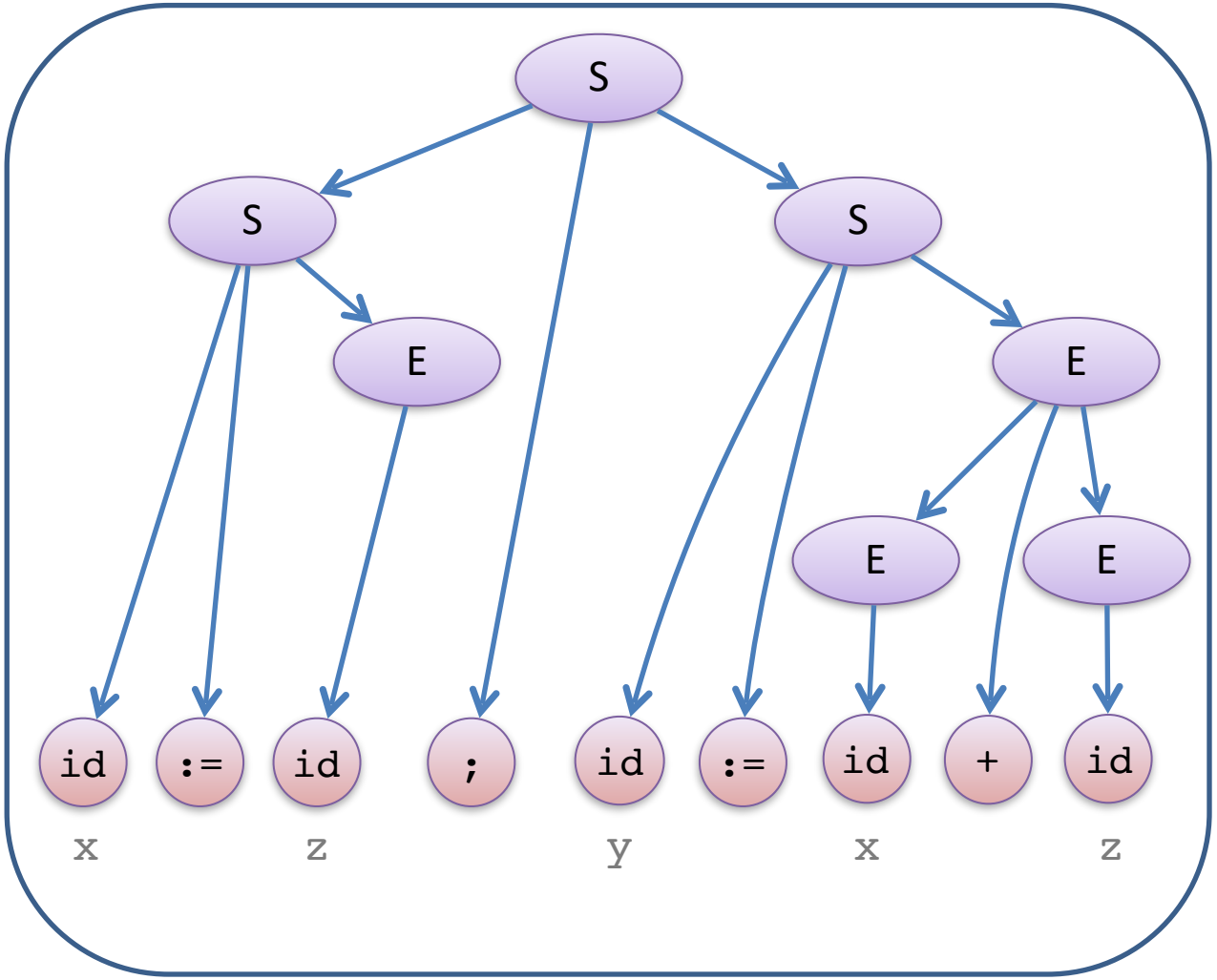
E → id | E + E | E * E | (E)

S	S → S ; S
S ; S	S → id := E
id := E ; S	S → id := E
id := E ; id := E	E → id
id := id ; id := E	E → E + E
id := id ; id := E + E	E → id
id := id ; id := E + id	E → id
id := id ; id := id + id	
x := y ; y := x + z	

Parse Tree

also: Syntax Tree, Derivation Tree





Questions

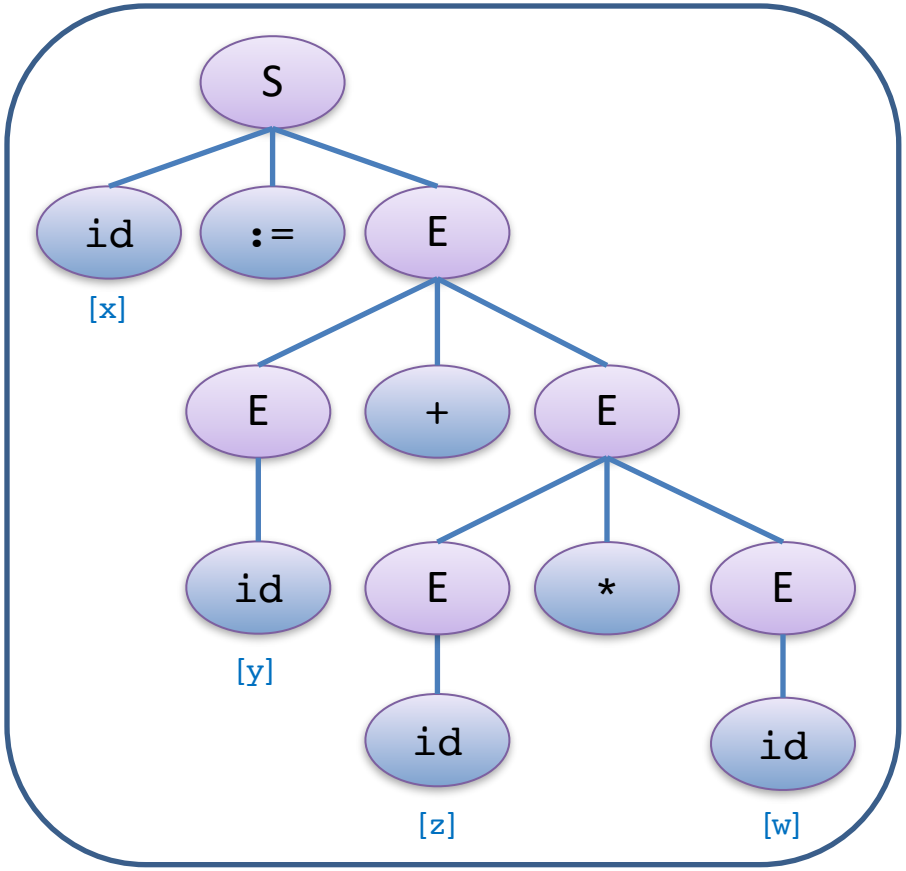
- How did we know which rule to apply on every step?

- Does it matter?
  - Can we get “stuck” if we make a wrong choice?
  - Would we always get the same parse tree?

Ambiguity

```
x := y + z * w
```

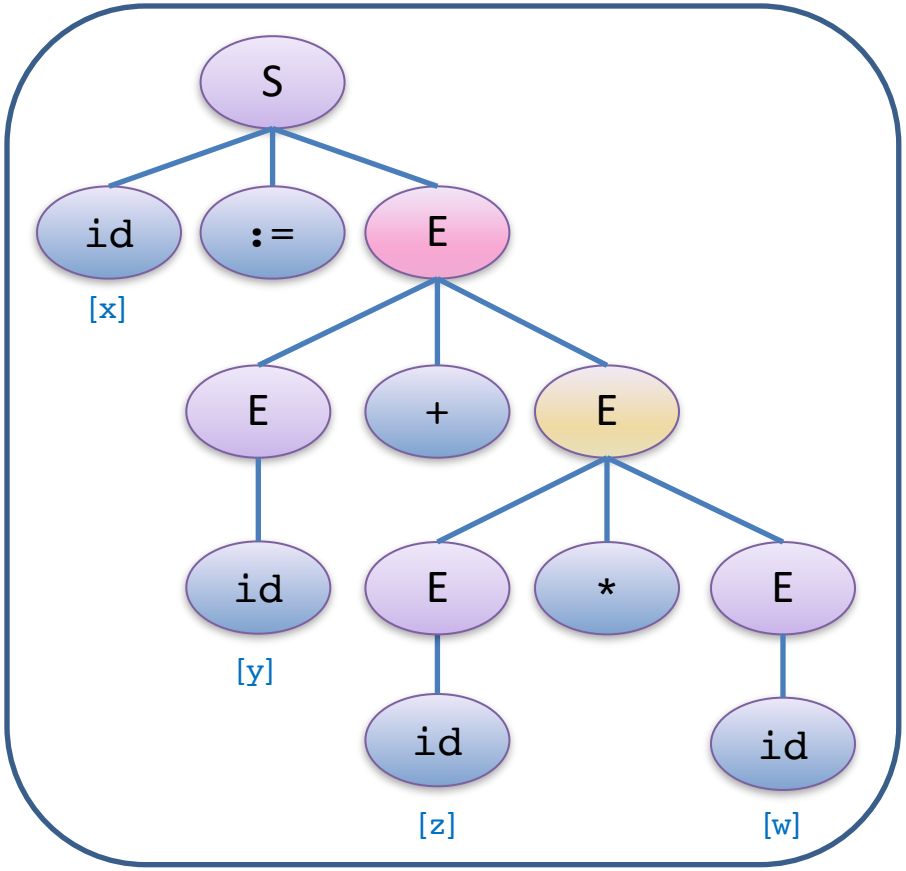
```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```

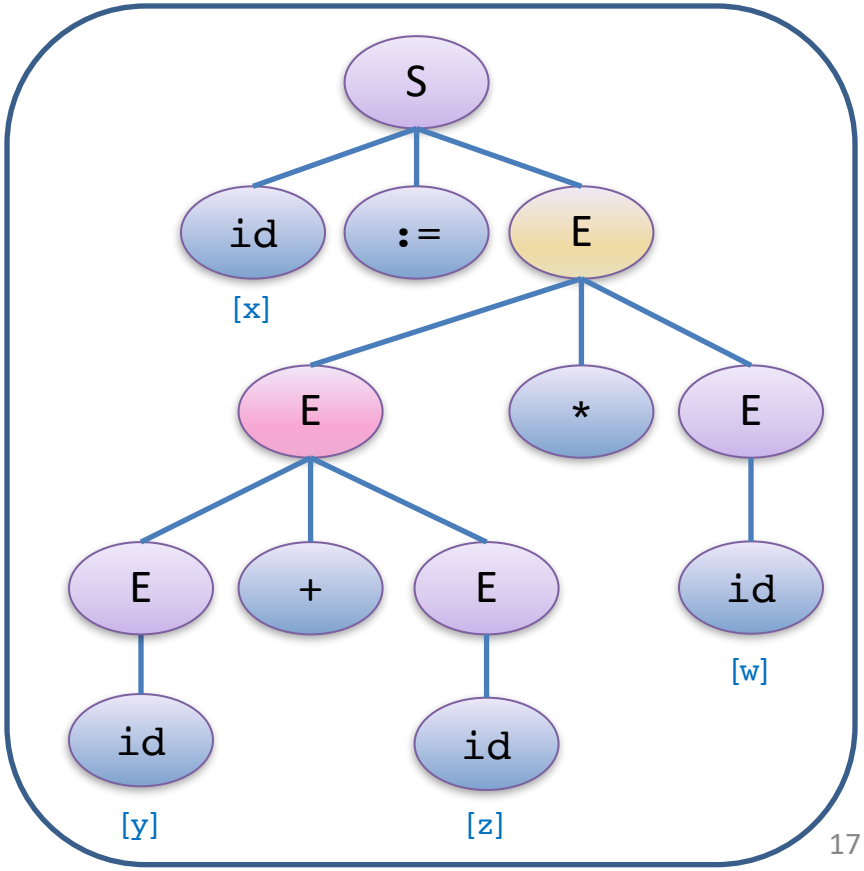


Ambiguity

```
x := y + z * w
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```



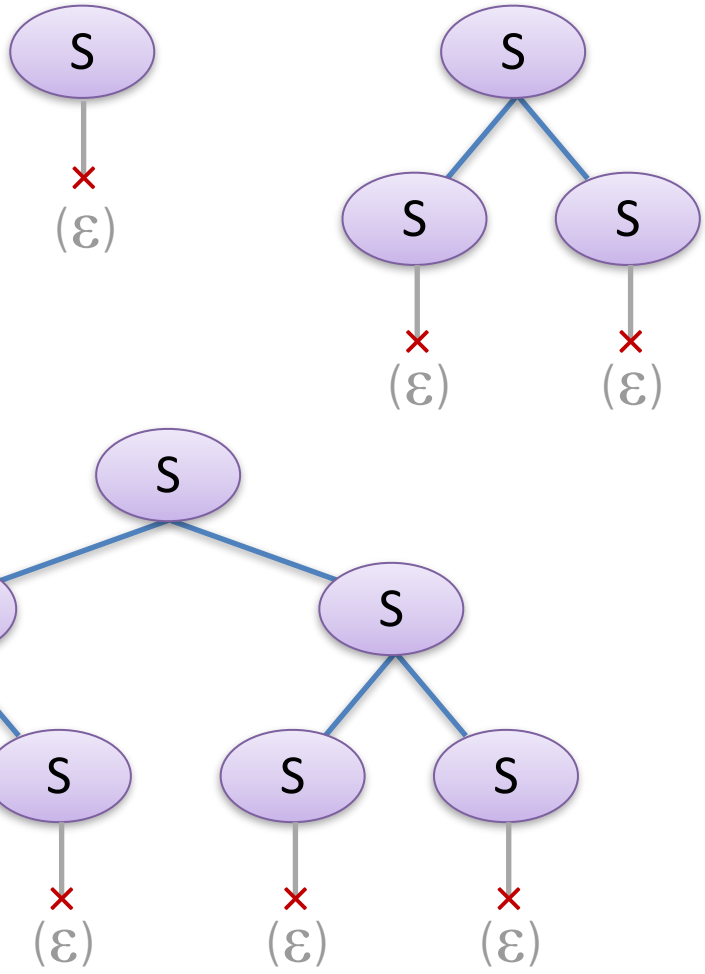


Ambiguity

$S \rightarrow \varepsilon$

$S \rightarrow (S)$

$S \rightarrow S S$



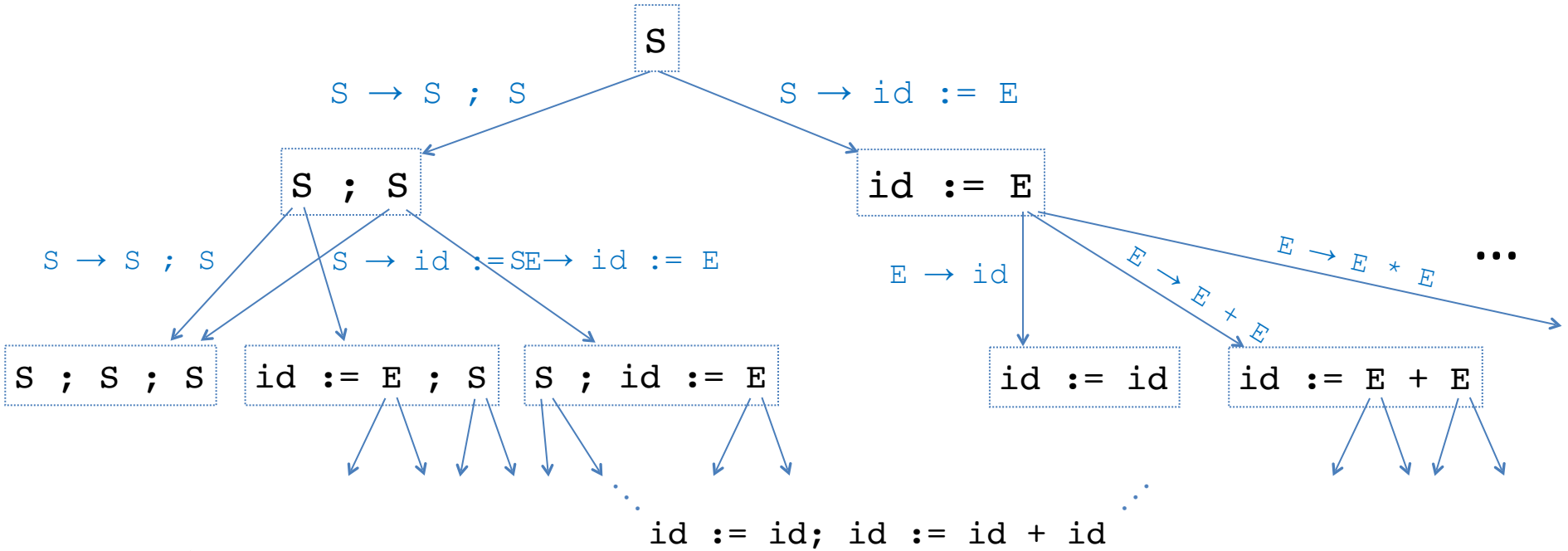
Complexity of Parsing

- Parsing can be seen as a search problem
  - Can you find a derivation from the start symbol to the input word?
  - Easy (but very expensive) to solve with backtracking

“Brute-force” Parsing

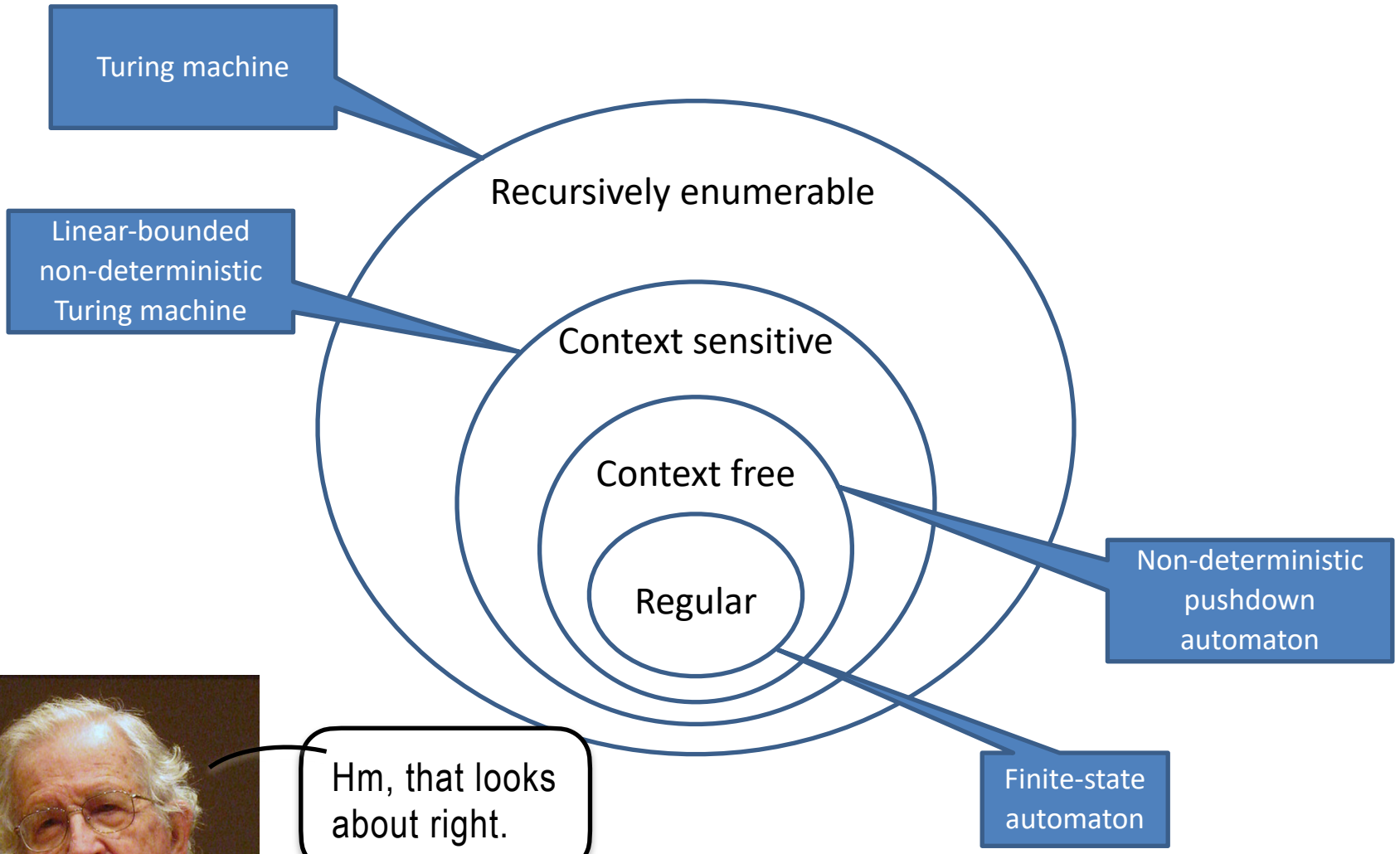
```
x := z;
y := x + z
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```



(not a parse tree… a **search** for a parse by exhaustively applying all rules)

Chomsky Hierarchy



Complexity of Parsing

- We already know: a regular language can be recognized by a deterministic finite automaton.
- A context free language can be recognized by a non-deterministic pushdown automaton.

Non-deterministic pushdown automaton?!

- We want ***efficient*** parsers that we can ***execute***
  - Deterministic computation model preferred

Parsing Algorithms

- *Earley* and *CYK* are parsing algorithms that can be used to parse any context-free language — with a (worst case) time complexity of $O(n^3)$

- We want *more* efficient parsers
  - Linear in input size

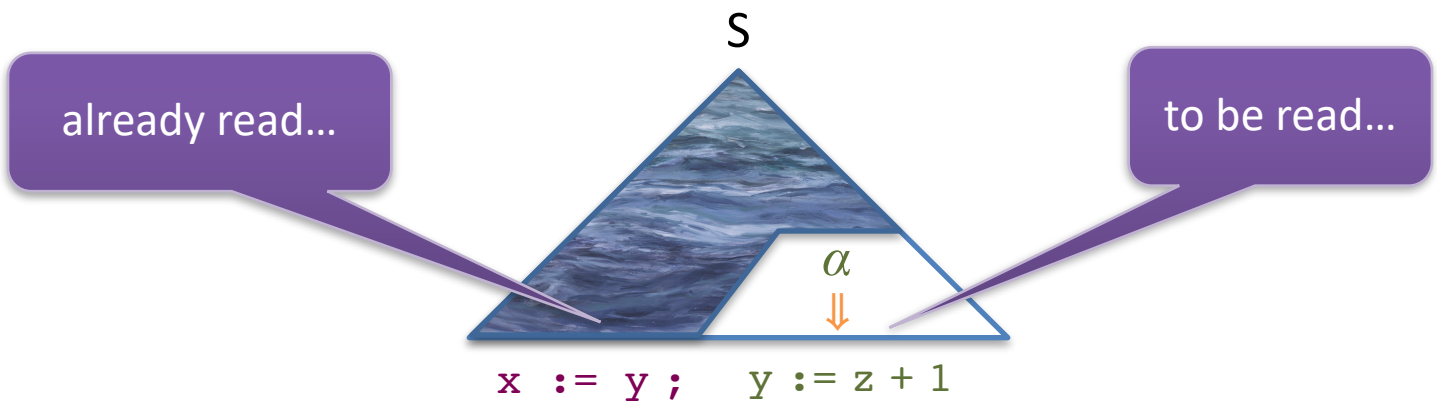
⇒ **We will sacrifice generality for efficiency**

Efficient Parsers

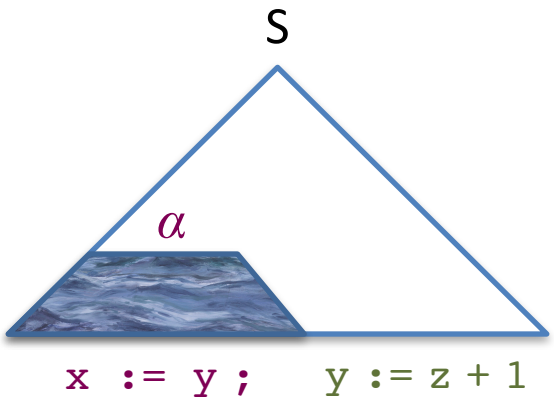
- Top-down (predictive)
  - Construct the leftmost derivation
  - Apply rules “from left to right”
  - Predict what rule to apply based on nonterminal and token
- Bottom up (shift-reduce)
  - Construct the rightmost derivation
  - Apply rules “from right to left”
  - Reduce a right-hand side of a production to its non-terminal

Efficient Parsers

- Top-down (predictive)



- Bottom-up (shift-reduce)



Parser Classes – Illustrated

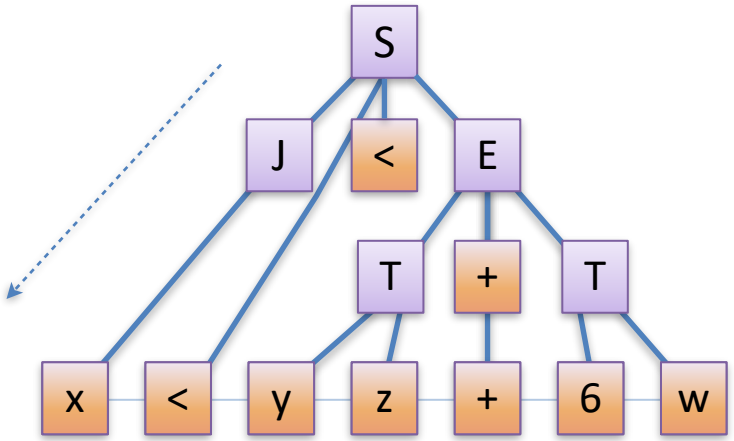
- Top-down (predictive)

S ⟶ J < E

J ⟶ ID

E ⟶ T + T

T ⟶ ID ID | NUM ID



Top-down Parsing

- Given a grammar G = ⟨V, T, P, S⟩ and a word *w*
- Goal: derive *w* using G
  - S $\Rightarrow^*$ w

- Idea
  - Apply production to leftmost nonterminal
  - Pick production rule based on next input token

Leftmost derivation

Lookahead

Leftmost Derivation

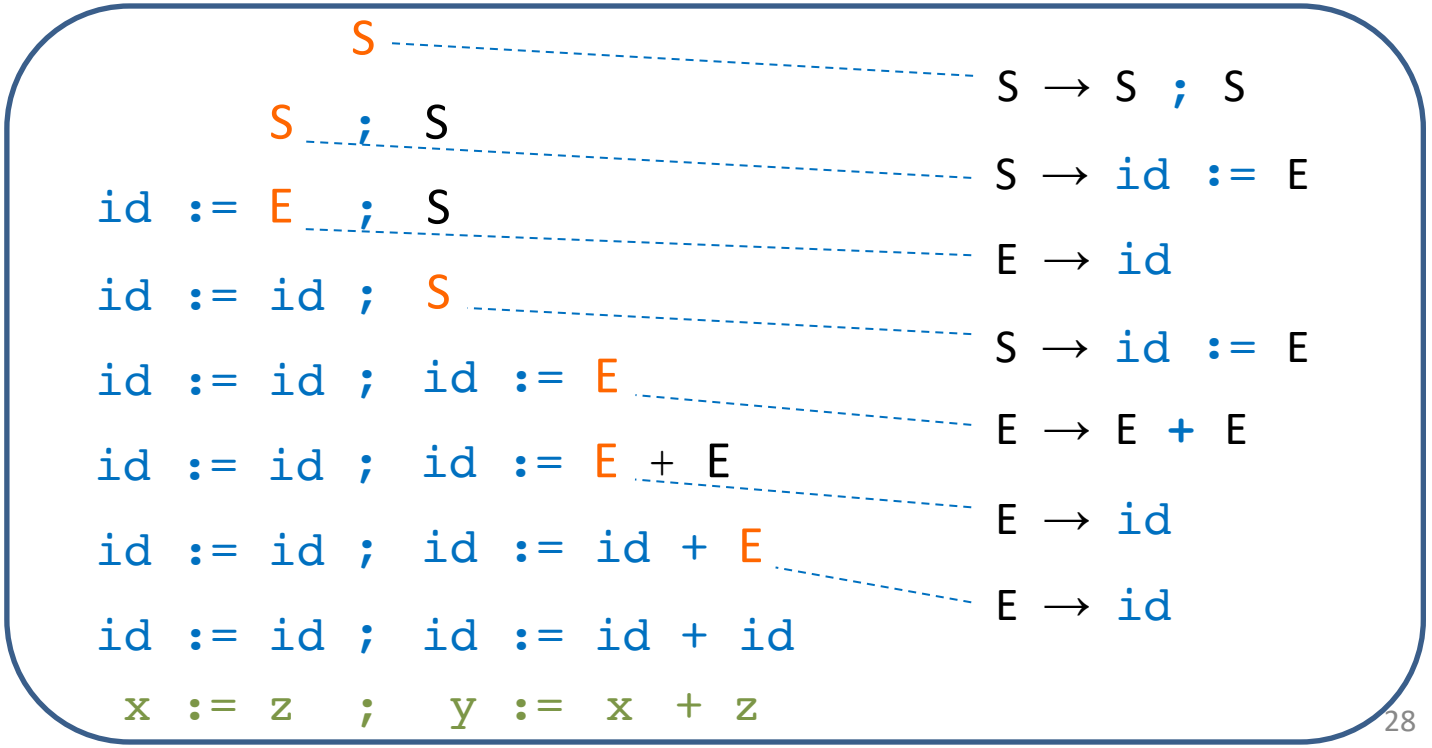
input

```
x := z;
y := x + z
```

grammar

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```

- always expand leftmost non-terminal

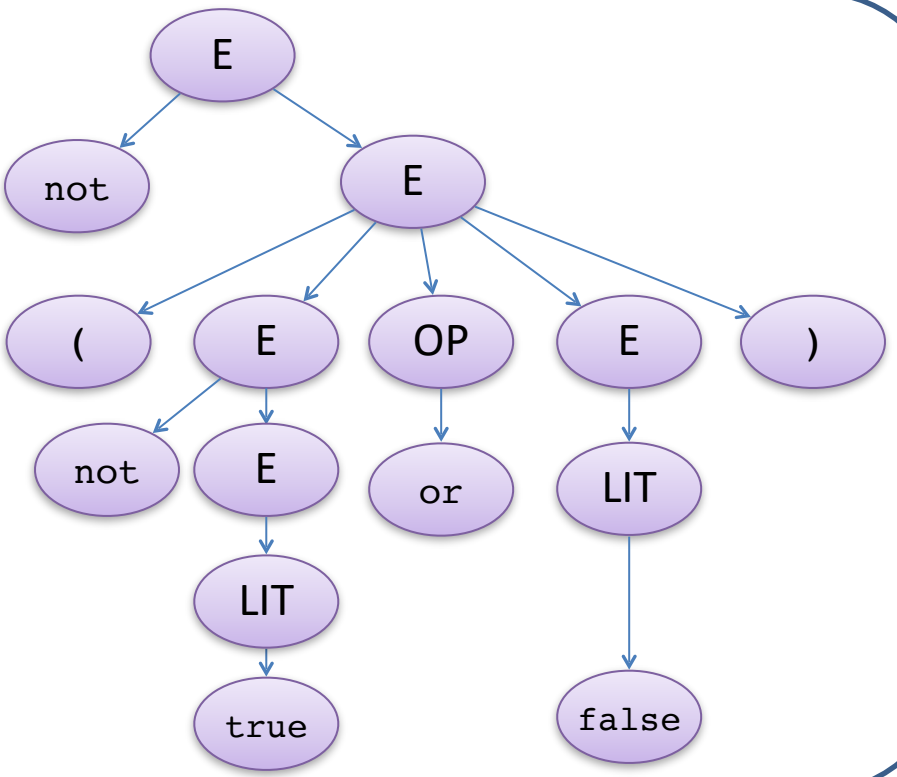


Boolean Expressions Example

```
not ( not true
      or false )
```

E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor

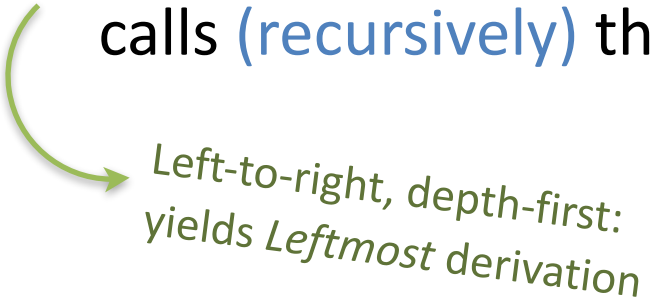
E ⟹
not E ⟹
not (E OP E) ⟹
not (not E OP E) ⟹
not (not LIT OP E) ⟹
not (not true OP E) ⟹
not (not true or E) ⟹
not (not true or LIT) ⟹
not (not true or false)



Production to apply is known from **next input token**

Recursive Descent Parsing

- Define a subroutine for every nonterminal
- Every subroutine works as follows:
  - Select applicable production rule
  - Iterate over symbols in rhs
    - For terminal symbols —
      check that it matches the next input token
    - For nonterminal symbols —
      calls (recursively) their corresponding functions



Matching tokens

```
void match(token t) {
  if (current == t)
    current = next_token();
  else
    error;
}
```

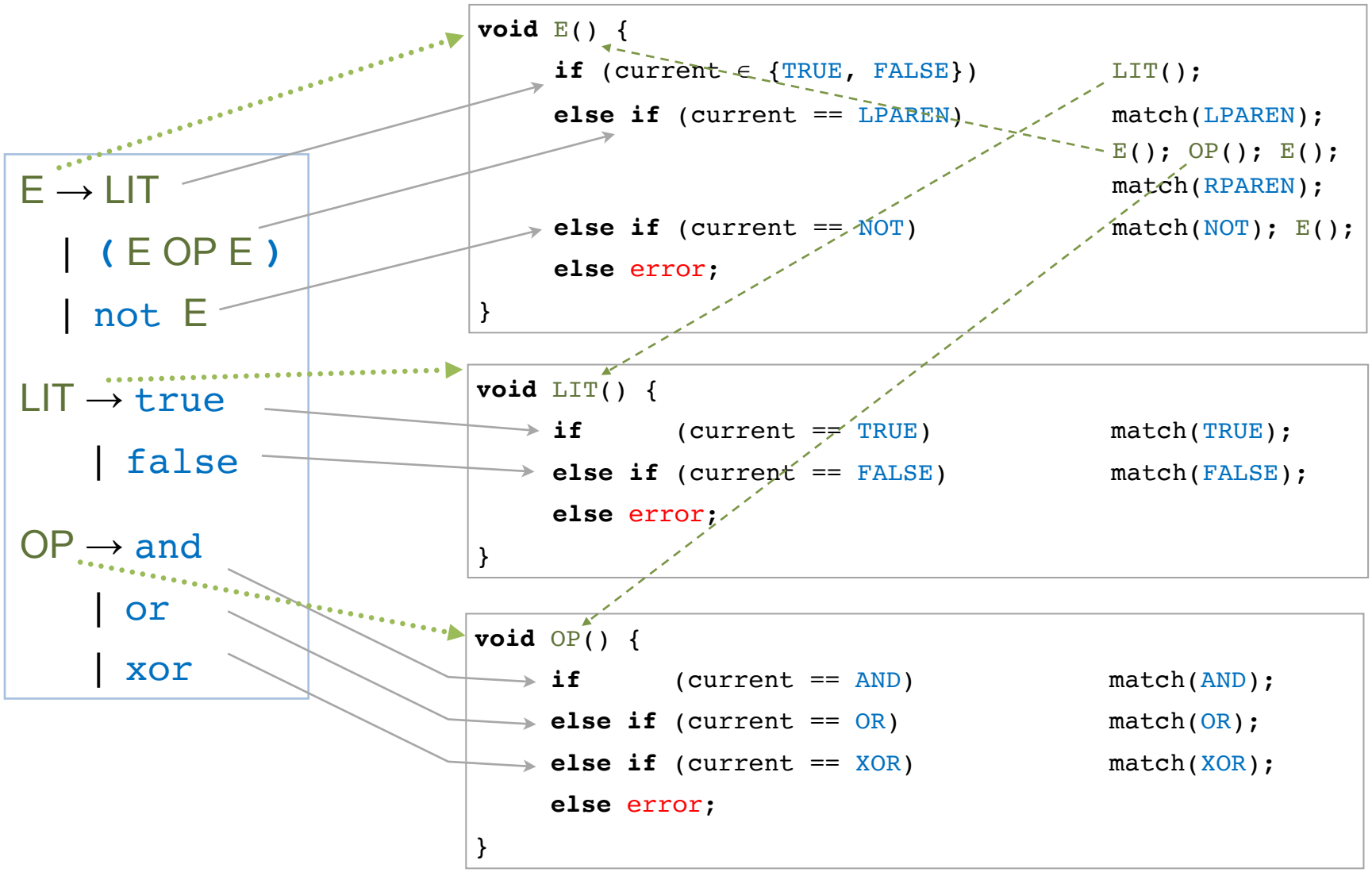
- Variable **`current`** holds the *Lookahead*

Functions for Nonterminals

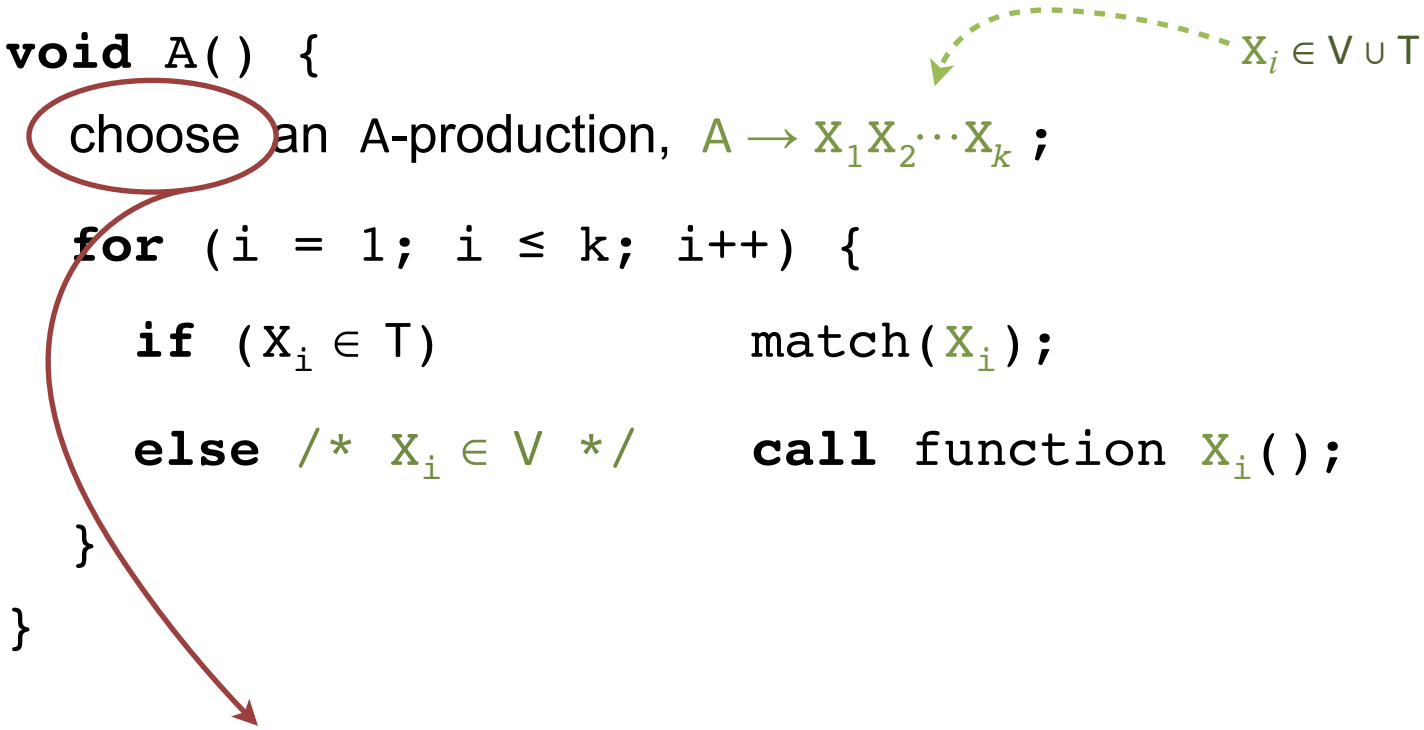
```
E  → LIT | ( E OP E ) | not E
LIT → true | false
OP → and | or | xor
```

```
void E() {
  if (current ∈ {TRUE, FALSE}) // E → LIT
    LIT();
  else if (current == LPAREN)  // E → ( E OP E )
    match(LPAREN); E(); OP(); E(); match(RPAREN);
  else if (current == NOT)     // E → not E
    match(NOT); E();
  else
    error;
}
void LIT() {
  if      (current == TRUE)  match(TRUE);
  else if (current == FALSE) match(FALSE);
  else error;
}
```

Functions for Nonterminals



Recursive Descent — General Structure



- How do you pick the right A-production?
  - Can we generalize the rationale behind all those `if` statements we saw in the example?

LL(*k*) Grammars

- A grammar is in the class LL(*k*) when it can be parsed via:
  - Top-down analysis
  - Scanning the input from left to right (L)
  - Producing the leftmost derivation (L)
  - With lookahead of *k* tokens (*k*)
- A language is said to be LL(*k*) if it has an LL(*k*) grammar
- The simplest case is LL(1), which we discuss next

FIRST Sets

- To formalize the property (of a grammar) that we can determine a rule using a single lookahead token, we define the FIRST sets.
- For a sentential form $\alpha \in (V \cup T)^*$
  - FIRST(α) = all terminals that α can start with
  - *i.e.*, $\sigma \in \text{FIRST}(\alpha) \Leftrightarrow$ there exists some derivation sequence $\alpha \Rightarrow^* \sigma w$
  - Special case: $\varepsilon \in \text{FIRST}(\alpha) \Leftrightarrow \alpha \Rightarrow^* \varepsilon$ (α is *nullable*)

(not really a terminal)

FIRST Sets

- For every terminal σ, nonterminal N:
  - FIRST(ε) = $\{\varepsilon\}$
  - FIRST(σ) = $\{\sigma\}$ — σ terminal
  - FIRST(N) = $\bigcup \{\text{FIRST}(\alpha) \mid (N \longrightarrow \alpha) \in P\}$ — N nonterminal
  - FIRST($x\alpha$) = $\begin{cases} \text{FIRST}(x) & \varepsilon \notin \text{FIRST}(x) \\ \text{FIRST}(x) \setminus \{\varepsilon\} \cup \text{FIRST}(\alpha) & \varepsilon \in \text{FIRST}(x) \end{cases}$

FIRST Sets

```
E   → LIT | ( E OP E ) | not E
LIT → true | false
OP → and | or | xor
```

- In our Boolean expressions example:
  - FIRST(LIT) = { **true**, **false** }
  - FIRST((E OP E)) = { '(' }
  - FIRST(**not** E) = { **not** }

These are all the alternatives for E → α

Recursive Descent (revised)

```
void A() {
   find an A-production, A → X1X2…Xk ,
                                  α
         such that current ∈ FIRST(α)
   for (i = 1; i ≤ k; i++) {
     if (Xi ∈ T)              match(Xi);
     else /* Xi ∈ V */        call function Xi();
   }
}
```

FIRST Sets

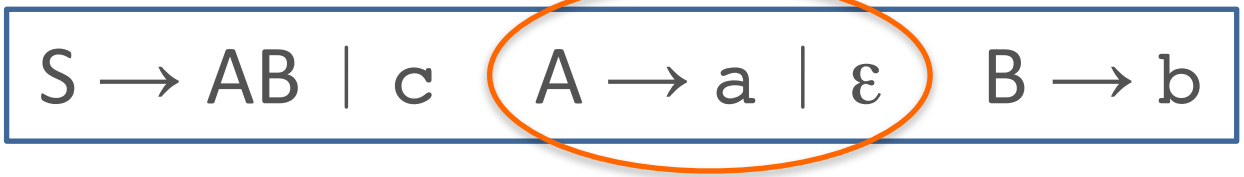
- No intersection between FIRST sets $\Rightarrow$ can *always** pick a single rule

- If the FIRST sets intersect, may need longer lookahead
  - LL(1) $\subset$ LL(2) $\subset$ LL(3) $\subset \cdots$
  - But the size of FIRST sets grow exponentially with *k*

* except when null productions exist — see next slide

FOLLOW Sets

- What do we do with nullable alternatives?
- Example:



  - Need to know what may come after A in the language to select the right production
- For *every* nonterminal A
  - FOLLOW(A) = set of tokens that can immediately follow A (in some derivation)

FOLLOW Sets

- (Fixpoint computation, similar to FIRST)
  - FOLLOW(S) = {$} S = initial symbol, $ = end-of-input marker
  - FOLLOW(N) = N nonterminal (N ≠ S)

$$\bigcup \{\text{FIRST}(\omega) \setminus \varepsilon \mid (A \longrightarrow \alpha N \omega) \in P\} \quad \cup$$

$$\bigcup \{\text{FOLLOW}(A) \mid (A \longrightarrow \alpha N \omega) \in P, \varepsilon \in \text{FIRST}(\omega)\}$$

Recursive Descent (revised again)

```
void A() {
```

find an A-production, $A \rightarrow \overbrace{X_1 X_2 \ldots X_k}^{\alpha}$,

such that `current` $\in$ FIRST(α)

or $\varepsilon \in$ FIRST(α) and `current` $\in$ FOLLOW(A)

```
  for (i = 1; i ≤ k; i++) {
    if (Xi ∈ T)              match(Xi);
    else /* Xi ∈ V */        call function Xi();
  }
}
```

This is the basis for an LL(1) parser.

(but it is still recursive — stay tuned)

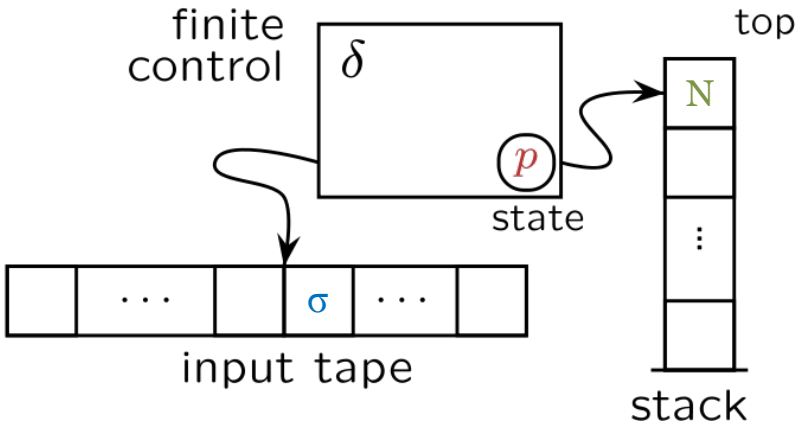
LL(*k*) Parsers

- Recursive Descent
  - Manual construction
  - Uses recursion

- Wanted
  - A parser that can be generated automatically
  - Iterative — does not use recursion
  - Uses a deterministic computational model

LL(1) Parsing with Pushdown Automata

- A PDA uses
  - Prediction stack
  - Input stream
  - Transition table



- δ : nonterminals × tokens → production alternative
  (right-hand side)

- Entry $\delta[N,\sigma]$ for nonterminal N and terminal σ contains the alternative of N that would be predicted when current input starts with σ

Example Transition Table

(1) E → LIT

(2) E → (E OP E)

(3) E → not E

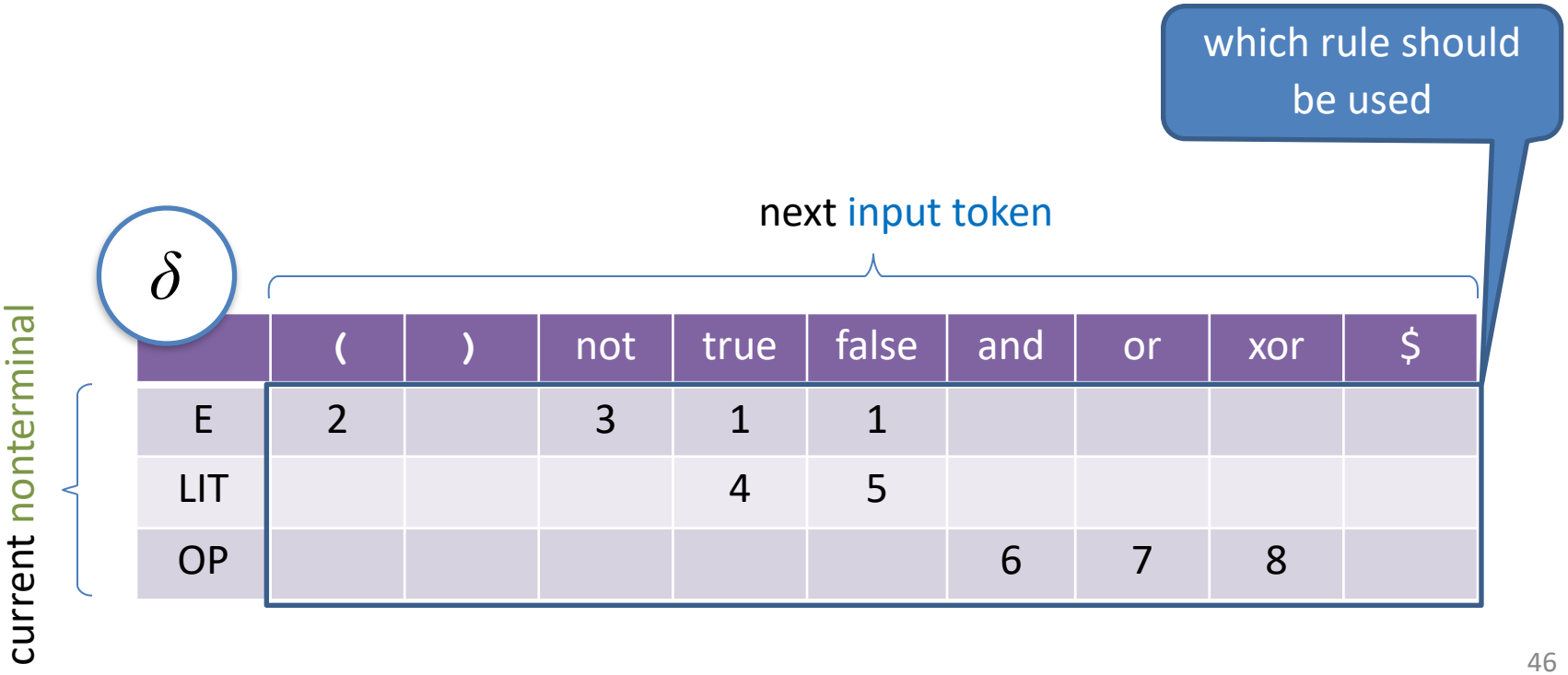
(4) LIT → true

(5) LIT → false

(6) OP → and

(7) OP → or

(8) OP → xor



	(	)	not	true	false	and	or	xor	$
E	2		3	1	1				
LIT				4	5				
OP						6	7	8	

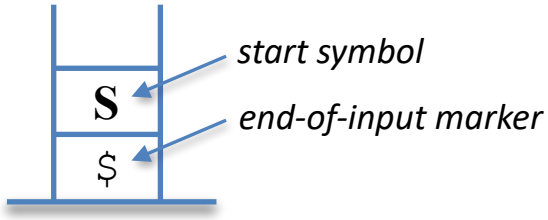
The Transition Table

- Constructing the transition table is easy
  - It relies on FIRST and FOLLOW
  - Based on the concept of our recursive descent earlier

- You will construct FIRST, FOLLOW, and the table in the tutorials

LL(k) parsing with pushdown automata

- Initial state —



- Two possible moves (σ = next token)
  - Predict

    When top of stack is nonterminal N, pop N, lookup δ[N,σ].

    If δ[N,σ] is defined, push δ[N,σ] on prediction stack.
    Otherwise – syntax error.
  - Match

    When top of stack is a terminal t, it must be equal to next input token.
    If t = σ, pop t and consume σ.
    If t ≠ σ – syntax error.

- Parsing terminates when prediction stack is empty.
  - At this point the input *must be finished* ⇒ success.

Simple Example

aacbb$

A → aAb | c

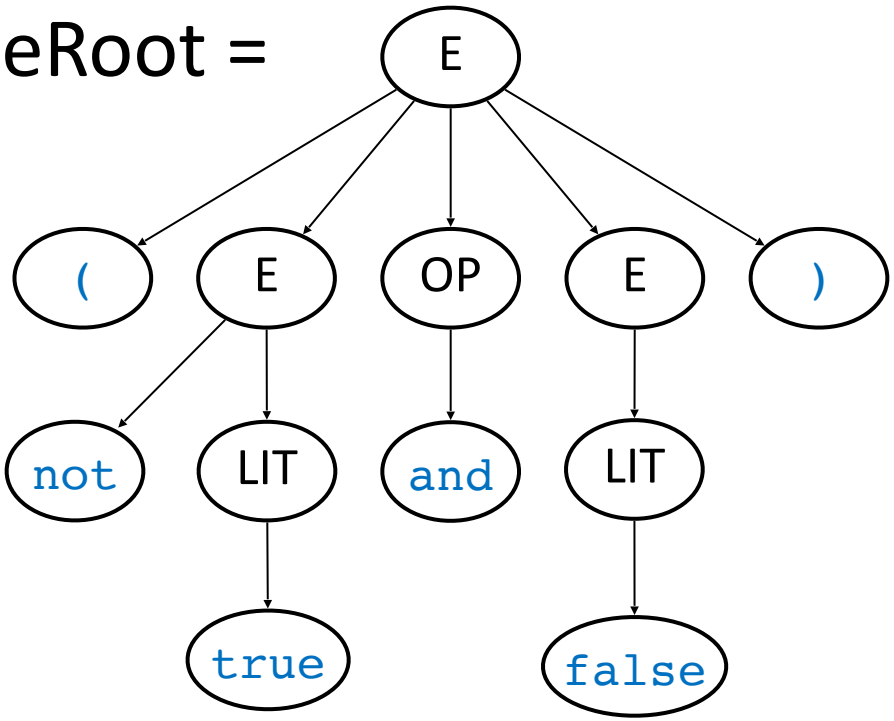
Stack content (top is left)	Remaining input	Action
A$	**a**acbb$	predict(A,a) = aAb
aAb$	**a**acbb$	match(a,a)
Ab$	**a**cbb$	predict(A,a) = aAb
aAbb$	**a**cbb$	match(a,a)
Abb$	**c**bb$	predict(A,c) = c
cbb$	**c**bb$	match(c,c)
bb$	**b**b$	match(b,b)
b$	**b**$	match(b,b)
$	**$**	match($,$) accept

	a	b	c
A	aAb		c

Building the Syntax Tree

- Input = "`( not true and false )`";

- `Node treeRoot =`



Building the Syntax Tree

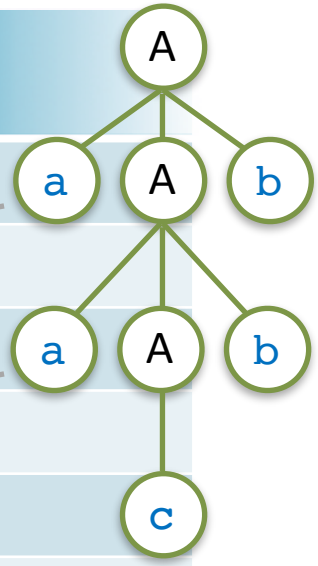
- Can add an action to perform on each production — in the LL(*k*) case, at predict
- Every symbol on the parser stack (except $) represents a node in the syntax tree.
  - The initial entry for S is the root.
  - When the top of the stack is a nonterminal N and parser performs predict(N, $X_1X_2…X_k$):
    - Create nodes for X_i's as child nodes of N

Simple Example

aacbb$

A → aAb | c

Stack content (top is left)	Remaining input	Action
A$	**a**acbb$	predict(A,a) = aAb
aAb$	**a**acbb$	match(a,a)
Ab$	**a**cbb$	predict(A,a) = aAb
aAbb$	**a**cbb$	match(a,a)
Abb$	**c**bb$	predict(A,c) = c
cbb$	**c**bb$	match(c,c)
bb$	**b**b$	match(b,b)
b$	**b**$	match(b,b)
$	**$**	match($,$) accept



	a	b	c
A	aAb		c

LL(1) Conflicts

term → `ID` | indexed

indexed → `ID` `[` expr `]`

- FIRST(`ID`) = { `ID` }
- FIRST(indexed) = { `ID` }

Both are alternatives for term → α

FIRST/FIRST conflict

⇒ This grammar is not in LL(1). Can we "fix" it?

Left Factoring

- Rewrite the grammar to be in LL(1)

term → `ID` | indexed

indexed → `ID` `[` expr `]`

term → `ID` after_ID

after_ID → `[` expr `]` | ε

Intuition: just like factoring $x \cdot y + x \cdot z$ into $x \cdot (y + z)$

Left Factoring

another example

```
S → if E then { S } else { S }
  | if E then { S }
  | …
```

```
S → if E then { S } S'
  | …
S' → else { S } | ε
```

Are We Done?

S → A a b

A → a | ε

- Select a rule for A with a in the look-ahead:
  - Should we pick (1) A → a or (2) A → ε ?

- FIRST(a) = { 'a' }
- FIRST(ε) = { ε } FOLLOW(A) = { 'a' }

} alternatives for A → α

FIRST/FOLLOW conflict

⇒ The grammar is not in LL(1). Can we fix *that*?

Grammatical Substitution

S → A a b

A → a | ε

Substitute A in S

S → a a b | a b

Left factoring

S → a after_a

after_a → a b | b

So Far

- Can determine if a grammar is in LL(1)
  - The FIRST and FOLLOW sets
  - Algorithms for computing these – in tutorials
- Have some techniques for modifying a grammar to find an equivalent grammar in LL(1)
  - Left factoring
  - Assignment

- Now let's look at a third example and present one more such technique

Left Recursion

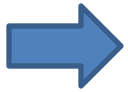
$$E \rightarrow E + \text{term} \mid \text{term}$$

- Left recursion cannot be handled with a bounded lookahead
- What can we do?
- Theorem: Any grammar with left recursion has an equivalent grammar without left recursion

Left Recursion Removal

$N \rightarrow N\alpha \mid \beta$

G_1



$N \rightarrow \beta N'$

$N' \rightarrow \alpha N' \mid \varepsilon$

G_2

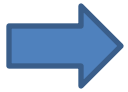
$L(G_1) = \{\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots\}$

$N' \Rightarrow^* \varepsilon, \alpha, \alpha\alpha, \alpha\alpha\alpha, \ldots$

$L(G_2) = \{\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots\}$

- For our example:

$E \rightarrow E + term \mid term$



$E \rightarrow term\ E'$

$E' \rightarrow +\ term\ E' \mid \varepsilon$

Error Handling

- Types of errors
  - Lexical errors
  - Syntax errors
  - Semantic errors (e.g., type mismatch)
  - Runtime errors (e.g., division by zero)

  - Requirements
    - Report the error clearly
    - Recover and continue so more errors can be discovered
    - Be efficient

Simple Example, Bad Word

abcbb$

A → aAb | c

Stack content	Remaining input	Action
A$	**a**bcbb$	predict(A,a) = aAb
aAb$	**a**bcbb$	match(a,a)
Ab$	**b**cbb$	predict(A,b) = error

	a	b	c
A	aAb		c

Simple Example, Bad Word

abcbb$

A → aAb | c

Stack content	Remaining input	Action
A$	**a**bcbb$	predict(A,a) = aAb
aAb$	**a**bcbb$	match(a,a)
Ab$	**b**cbb$	predict(A,b) = ✗ skip
Ab$	**c**bb$	predict(A,a) = A → c
cb$	**c**bb$	match(c,c)
b$	**b**b$	match(b,b)
$	**b**$	match($,b) = ✗ skip
$	**$**	match($,$)

	a	b	c
A	aAb		c

Error Handling in LL Parsers

- The **acceptable-set method**
  - Step 1: construct a set A of *acceptable* terminals based on the current state of the parser.
  - Step 2: discard input tokens until a token $t_A \in A$ is encountered.
  - Step 3: advance the parser to the next state in which it can consume t_A.

A = FIRST(stack content)

don't need to do anything

panic mode

Error Handling in LL Parsers

- The **acceptable-set method**
  - Step 1: construct a set A of *acceptable* terminals based on the current state of the parser.
  - Step 2: discard input tokens until a token $t_A \in A$ is encountered.
  - Step 3: advance the parser to the next state in which it can consume t_A.

A = FOLLOW(stack top)

pop the topmost nonterminal

FOLLOW-set error recovery

Error Handling

- Different compilers adopt different approaches.
- Some examples
  - Panic mode (or another acceptable-set method): drop tokens until reaching a synchronizing token, like a semicolon, a right parenthesis, end of file, etc.
  - Phrase-level recovery: attempting local changes: replace “,” with “;”, eliminate or add a “;”, etc.
  - Error production: anticipate errors and automatically handle them by adding special rules to the grammar
  - Global correction: find the minimum modification to the program that will make it derivable in the grammar
    - Not a practical solution, except with very small grammars

A Note About 

- ANTLR = ANother Tool for Language Recognition
- LL(*) algorithm
  - Like LL(*k*) on steroids
  - Notable extensions:
    - Repeat operators — like in regex
    - Lookahead predicates — allow for unbounded lookahead at the cost of backtracking

✶ There's a nice demo at lab.antlr.org

* *LL(*): The Foundation of the ANTLR Parser Generator*, Parr and Fischer, PLDI 2011

Summary

- Parsing
  - ✓ Top-down or bottom-up
- Top-down parsing
  - ✓ Recursive descent
  - ✓ LL(k) grammars
  - ✓ LL(k) parsing with pushdown automata
- LL(k) parsers limitations
  - ✓ Cannot deal with common prefixes and left recursion
  - ✓ Left-recursion removal might result in complicated grammar

Coming Up