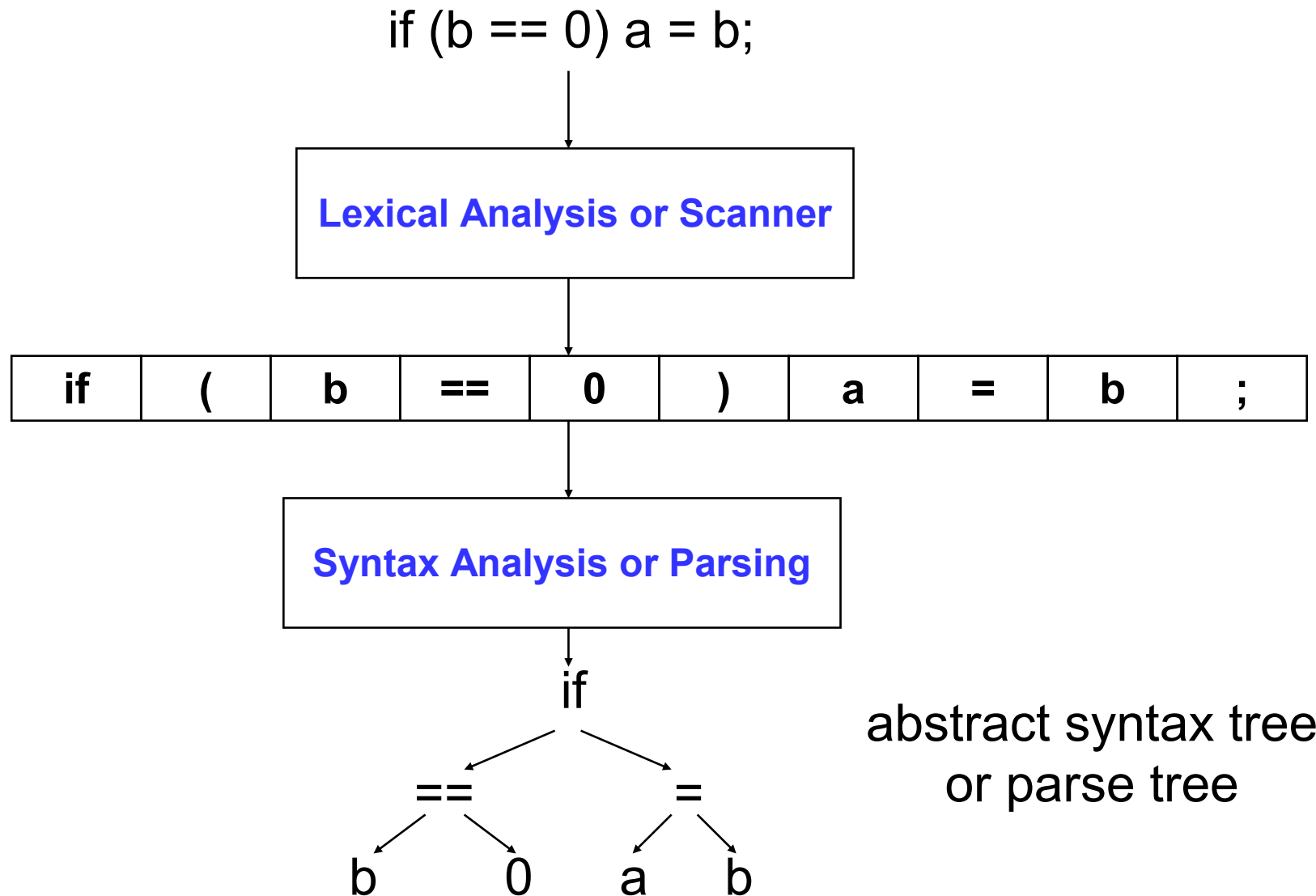


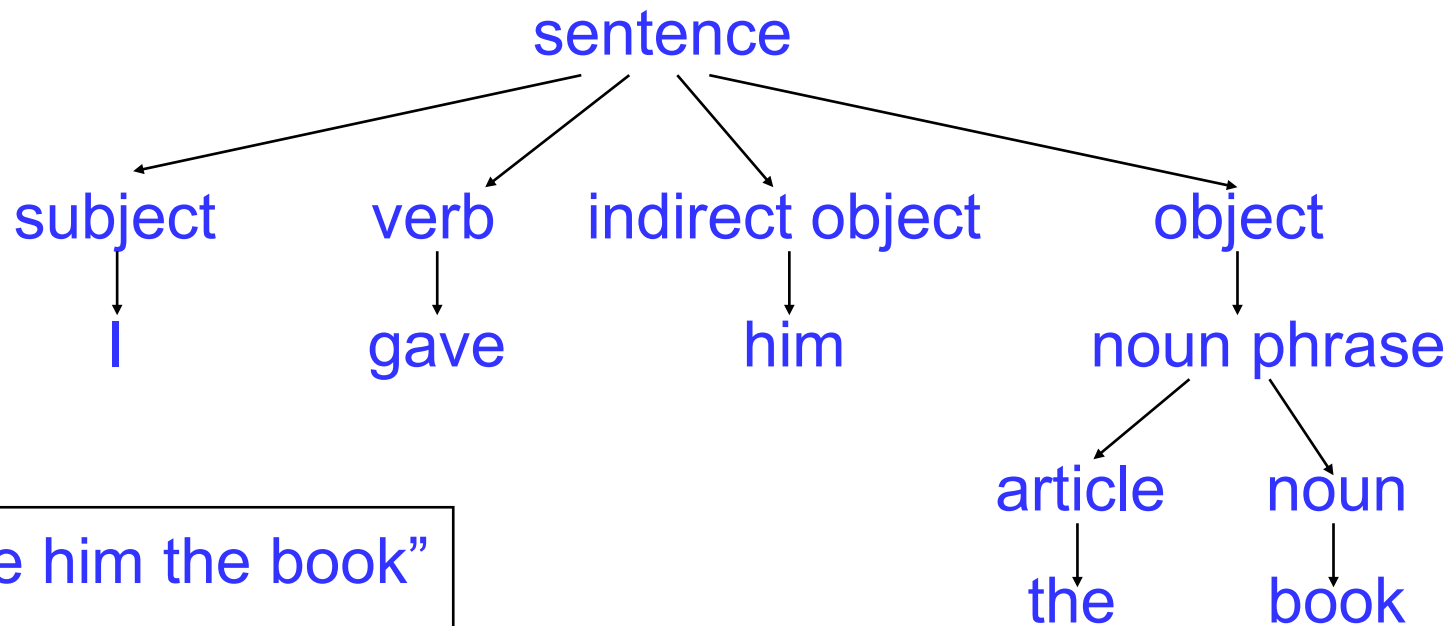
Syntax Analysis

Where is Syntax Analysis Performed?

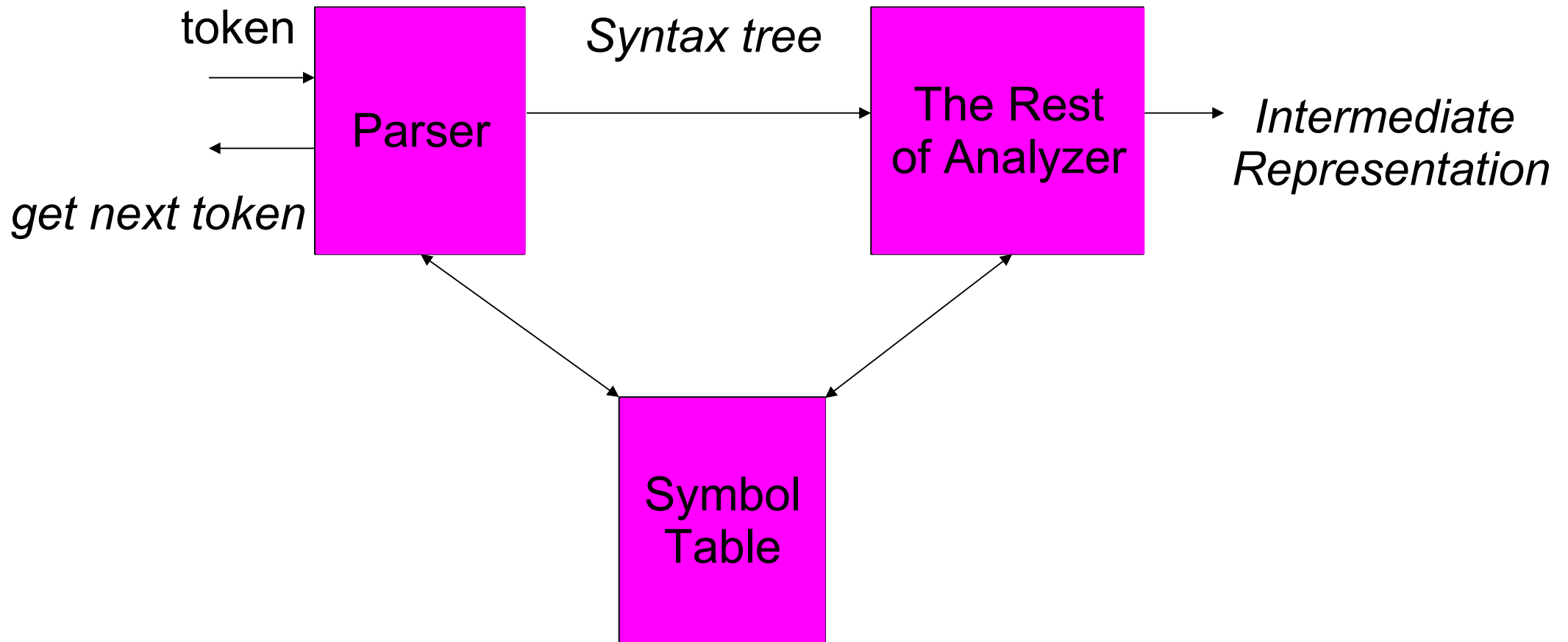


Parsing Analogy

- Syntax analysis for natural languages
- Recognize whether a sentence is grammatically correct
- Identify the function of each word



Place of A Parser in A Compiler



Syntax Analysis Overview

- Goal – Determine if the input token stream satisfies the syntax of the program
- What do we need to do this?
 - An expressive way to describe the syntax
 - A mechanism that determines if the input token stream satisfies the syntax description
- For lexical analysis
 - Regular expressions describe tokens
 - Finite automata = mechanisms to generate tokens from input stream

Context-Free Grammars

- Consist of 4 components:
 - Terminal symbols = token or ϵ
 - Non-terminal symbols = syntactic variables
 - Start symbol S = special non-terminal
 - Productions of the form $LHS \rightarrow RHS$
 - LHS = single non-terminal
 - RHS = string of terminals and non-terminals
 - Specify how non-terminals may be expanded
- Language generated by a grammar is the set of strings of terminals derived from the start symbol by repeatedly applying the productions
 - $L(G)$ = language generated by grammar G

S	\rightarrow	$a S a$
S	\rightarrow	T
T	\rightarrow	$b T b$
T	\rightarrow	ϵ

CFG - Example

- Grammar for balanced-parentheses language

- $S \rightarrow (S) S$

← Why is the final S required?

- $S \rightarrow \varepsilon$

- 1 non-terminal: S
 - 2 terminals: “(”, “)”
 - Start symbol: S
 - 2 productions

- If grammar accepts a string, there is a derivation of that string using the productions

- “(())”

- $S \Rightarrow (S)S \Rightarrow (S) \varepsilon \Rightarrow ((S) S) \varepsilon \Rightarrow ((S) \varepsilon) \varepsilon \Rightarrow ((\varepsilon) \varepsilon) \varepsilon \Rightarrow (())$

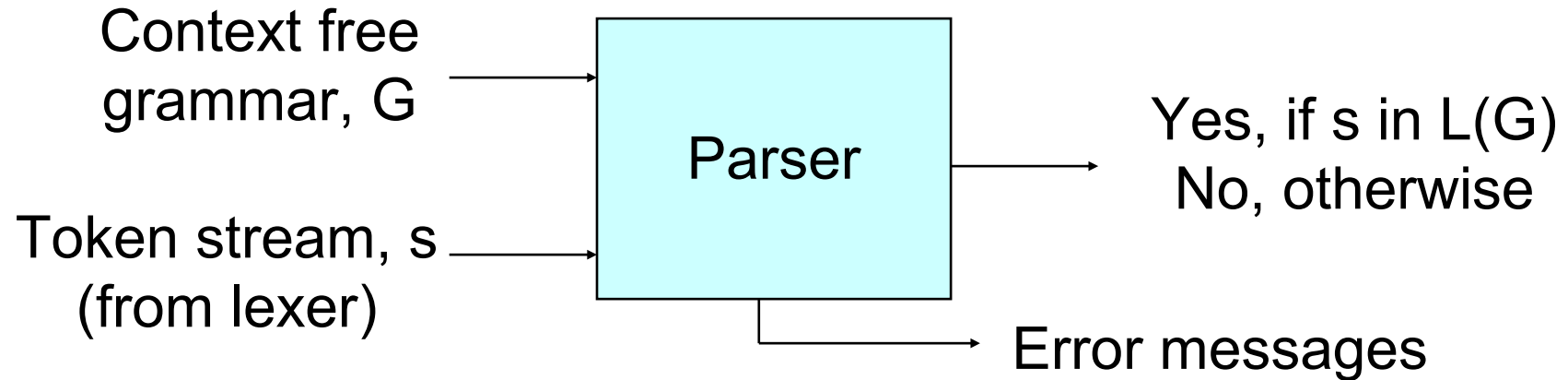
More on CFGs

- Shorthand notation – vertical bar for multiple productions
 - $S \rightarrow a S a \mid T$
 - $T \rightarrow b T b \mid \epsilon$
- CFGs powerful enough to expression the syntax in most programming languages
- Derivation = successive application of productions starting from S
- Acceptance? = Determine if there is a derivation for an input token stream

Constructs which Cannot Be Described by Context-Free Grammars

- Declarations of identifiers before their usage
- Function calls with the proper number of arguments

A Parser



Syntax analyzers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted

Various kinds: LL(k), LR(k), SLR, LALR

RE is a Subset of CFG

Can inductively build a grammar for each RE

ϵ	$S \rightarrow \epsilon$
a	$S \rightarrow a$
$R1 R2$	$S \rightarrow S1 S2$
$R1 R2$	$S \rightarrow S1 S2$
$R1^*$	$S \rightarrow S1 S \epsilon$

Where

$G1$ = grammar for $R1$, with start symbol $S1$

$G2$ = grammar for $R2$, with start symbol $S2$

Grammar for Sum Expression

- Grammar

- $S \rightarrow E + S \mid E$

- $E \rightarrow \text{number} \mid (S)$

- Expanded

- $S \rightarrow E + S$ 4 productions

- $S \rightarrow E$ 2 non-terminals (S,E)

- $E \rightarrow \text{number}$ 4 terminals: “(”, “)”, “+”, number

- $E \rightarrow (S)$ start symbol: S

- $E \rightarrow (S)$

Constructing a Derivation

- Start from S (the start symbol)
- Use productions to derive a sequence of tokens
- For arbitrary strings α , β , γ and for a production: $A \rightarrow \beta$
 - A single step of the derivation is
 - $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ (substitute β for A)
- **Example**
 - $S \rightarrow E + S$
 - $(\underline{S} + E) + E \Rightarrow (\underline{E + S} + E) + E$

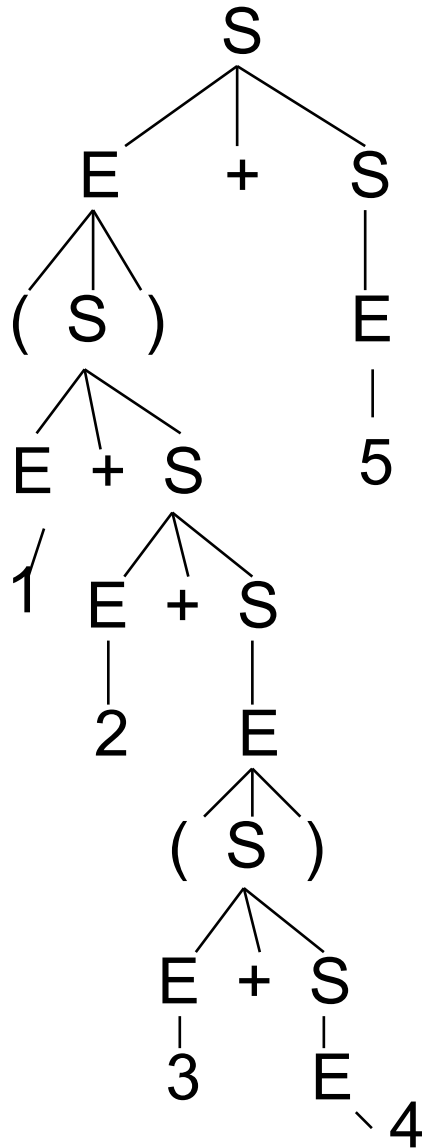
Class Problem

– $S \rightarrow E + S \mid E$

– $E \rightarrow \text{number} \mid (S)$

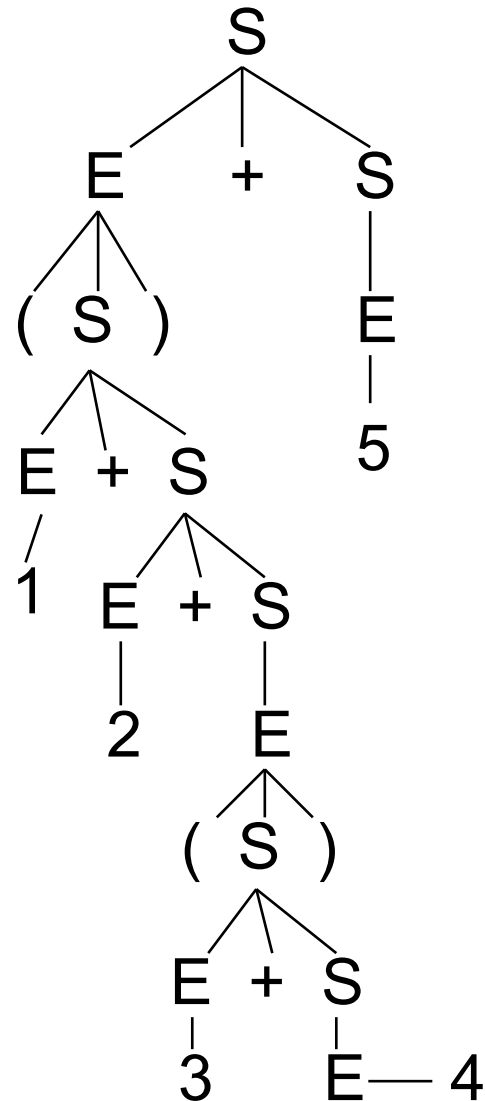
- Derive: $(1 + 2 + (3 + 4)) + 5$

Parse Tree

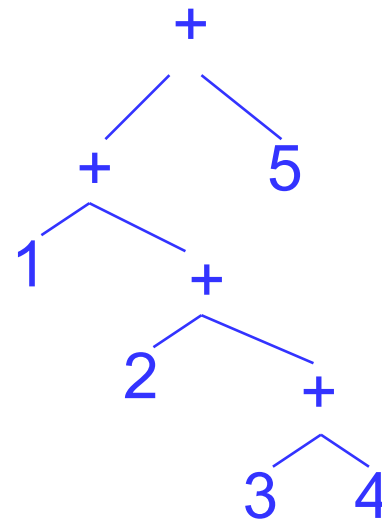
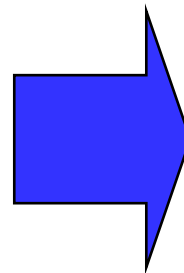


- Parse tree = tree representation of the derivation
- Leaves of the tree are terminals
- Internal nodes are non-terminals
- No information about the order of the derivation steps

Parse Tree vs Abstract Syntax Tree



Parse tree also called “concrete syntax”



AST discards (abstracts) unneeded information – more compact format

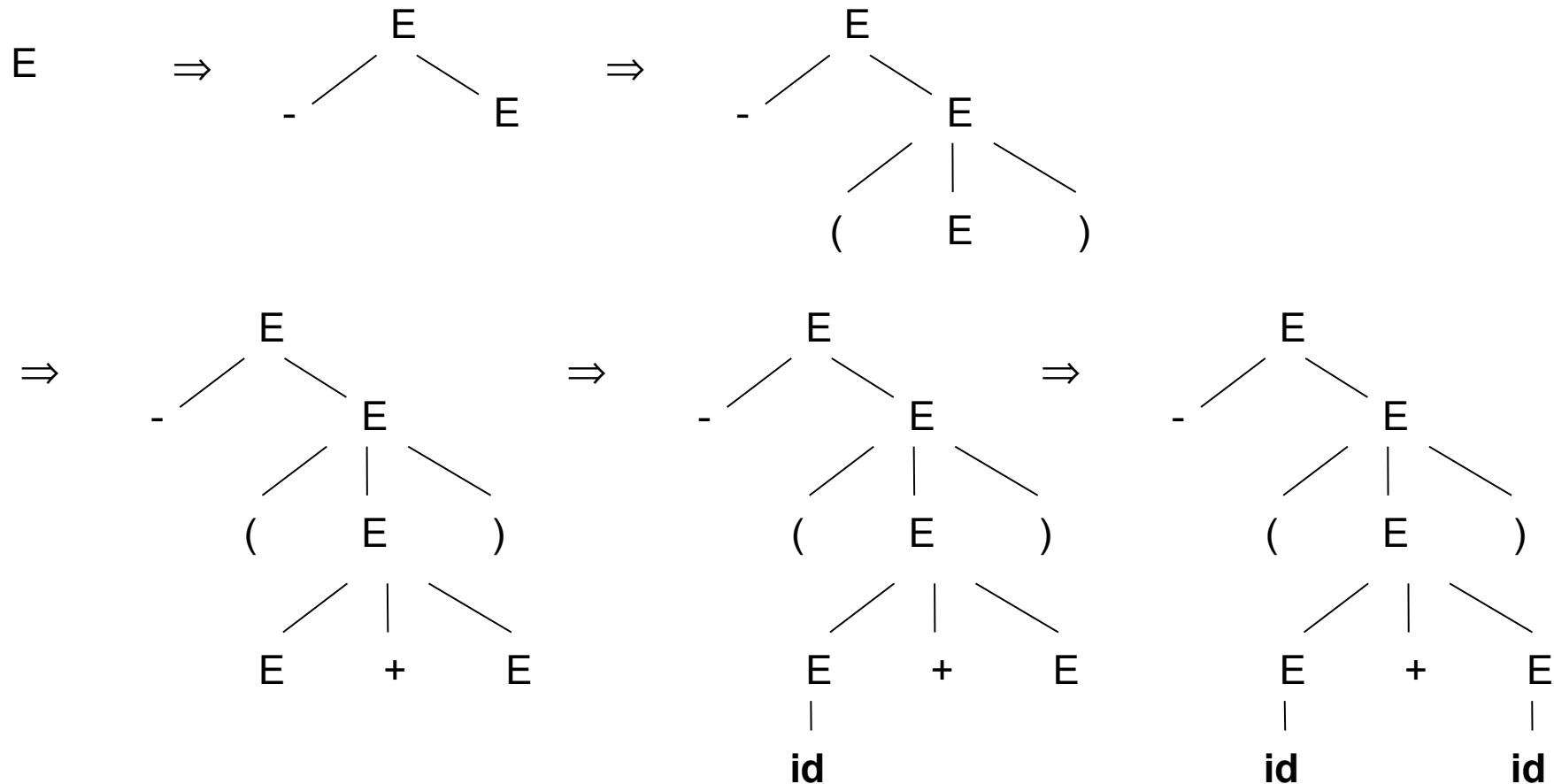
Derivation Order

- Can choose to apply productions in any order, select non-terminal and substitute RHS of production
- Two standard orders: left and right-most
- Leftmost derivation
 - In the string, find the leftmost non-terminal and apply a production to it
 - $E + S \xRightarrow{\text{lm}} 1 + S$
- Rightmost derivation
 - Same, but find rightmost non-terminal
 - $E + S \xRightarrow{\text{rm}} E + E + S$

Leftmost Derivation Example

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \text{id}$

$E \xRightarrow{\text{lm}} -E \xRightarrow{\text{lm}} -(E) \xRightarrow{\text{lm}} -(E+E) \xRightarrow{\text{lm}} -(\text{id}+E) \xRightarrow{\text{lm}} -(\text{id}+\text{id})$



Leftmost/Rightmost Derivation Examples

- $S \rightarrow E + S \mid E$
- $E \rightarrow \text{number} \mid (S)$
- Leftmost derive: $(1 + 2 + (3 + 4)) + 5$

$S \Rightarrow E + S \Rightarrow (S) + S \Rightarrow (E + S) + S \Rightarrow (1 + S) + S \Rightarrow (1 + E + S) + S \Rightarrow$
 $(1 + 2 + S) + S \Rightarrow (1 + 2 + E) + S \Rightarrow (1 + 2 + (S)) + S \Rightarrow (1 + 2 + (E + S)) + S \Rightarrow$
 $(1 + 2 + (3 + S)) + S \Rightarrow (1 + 2 + (3 + E)) + S \Rightarrow (1 + 2 + (3 + 4)) + S \Rightarrow$
 $(1 + 2 + (3 + 4)) + E \Rightarrow (1 + 2 + (3 + 4)) + 5$

- Now, rightmost derive the same input string

$S \Rightarrow E + S \Rightarrow E + E \Rightarrow E + 5 \Rightarrow (S) + 5 \Rightarrow (E + S) + 5 \Rightarrow (E + E + S) + 5 \Rightarrow$
 $(E + E + E) + 5 \Rightarrow (E + E + (S)) + 5 \Rightarrow (E + E + (E + S)) + 5 \Rightarrow$
 $(E + E + (E + E)) + 5 \Rightarrow (E + E + (E + 4)) + 5 \Rightarrow (E + E + (3 + 4)) + 5 \Rightarrow$
 $(E + 2 + (3 + 4)) + 5 \Rightarrow (1 + 2 + (3 + 4)) + 5$

Result: Same parse tree: same productions chosen, but in different order

Class Problem

– $S \rightarrow E + S \mid E$

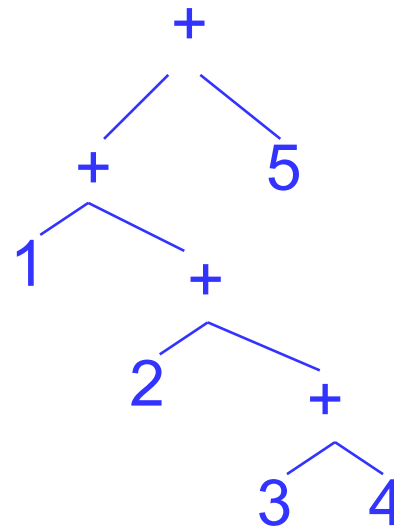
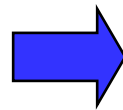
– $E \rightarrow \text{number} \mid (S) \mid -S$

- Do the rightmost derivation of : $1 + (2 + -(3 + 4)) + 5$

Ambiguous Grammars

- In the sum expression grammar, leftmost and rightmost derivations produced identical parse trees
- + operator associates to the right in parse tree regardless of derivation order

$(1+2+(3+4))+5$



Ambiguous Grammars

- + associates to the right because of the right-recursive production: $S \rightarrow E + S$
- Consider another grammar
 - $S \rightarrow S + S \mid S * S \mid \text{number}$
- Ambiguous grammar = different derivations produce different parse trees
 - More specifically, G is ambiguous if there are 2 distinct leftmost (rightmost) derivations for some sentence

Ambiguous Grammar - Example

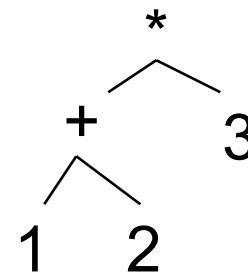
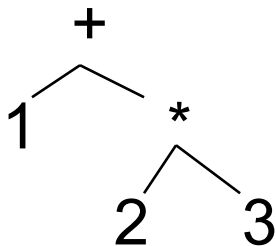
$$S \rightarrow S + S \mid S * S \mid \text{number}$$

Consider the expression: $1 + 2 * 3$

Derivation 1: $S \Rightarrow S+S \Rightarrow$
 $1+S \Rightarrow 1+S*S \Rightarrow 1+2*S \Rightarrow 1+2*3$

Derivation 2: $S \Rightarrow S*S \Rightarrow$
 $S+S*S \Rightarrow 1+S*S \Rightarrow 1+2*S \Rightarrow 1+2*3$

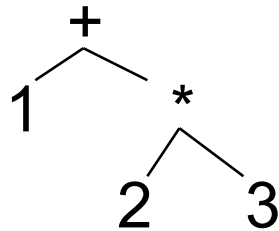
2 leftmost derivations



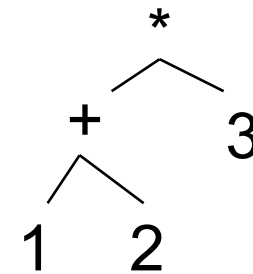
But, obviously not equal!

Impact of Ambiguity

- Different parse trees correspond to different evaluations!
- Thus, program meaning is not defined!!



= 7



= 9

Can We Get Rid of Ambiguity?

- Ambiguity is a function of the grammar, not the language!
- A context-free language L is inherently ambiguous if all grammars for L are ambiguous
- Every deterministic CFL has an unambiguous grammar
 - So, no deterministic CFL is inherently ambiguous
 - No inherently ambiguous programming languages have been invented
- To construct a useful parser, must devise an unambiguous grammar

Eliminating Ambiguity

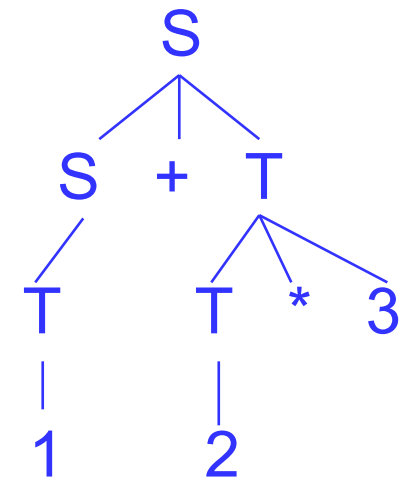
- Often can eliminate ambiguity by adding nonterminals and allowing recursion only on right or left

– $S \rightarrow S + T \mid T$

– $T \rightarrow T * \text{num} \mid \text{num}$

– T non-terminal enforces precedence

– Left-recursion; left associativity



A Closer Look at Eliminating Ambiguity

- Precedence enforced by
 - Introduce distinct non-terminals for each precedence level
 - Operators for a given precedence level are specified as RHS for the production
 - Higher precedence operators are accessed by referencing the next-higher precedence non-terminal

Associativity

- An operator is either left, right or non associative
 - Left: $a + b + c = (a + b) + c$
 - Right: $a \wedge b \wedge c = a \wedge (b \wedge c)$
 - Non: $a < b < c$ is illegal (thus undefined)
- Position of the recursion relative to the operator dictates the associativity
 - Left (right) recursion \rightarrow left (right) associativity
 - Non: Don't be recursive, simply reference next higher precedence non-terminal on both sides of operator

Class Problem

$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid -S \mid S \wedge S \mid \text{num}$

Enforce the standard arithmetic precedence rules and remove all ambiguity from the above grammar

Precedence (high to low)

$()$, unary $-$

\wedge

$*$, $/$

$+$, $-$

Associativity

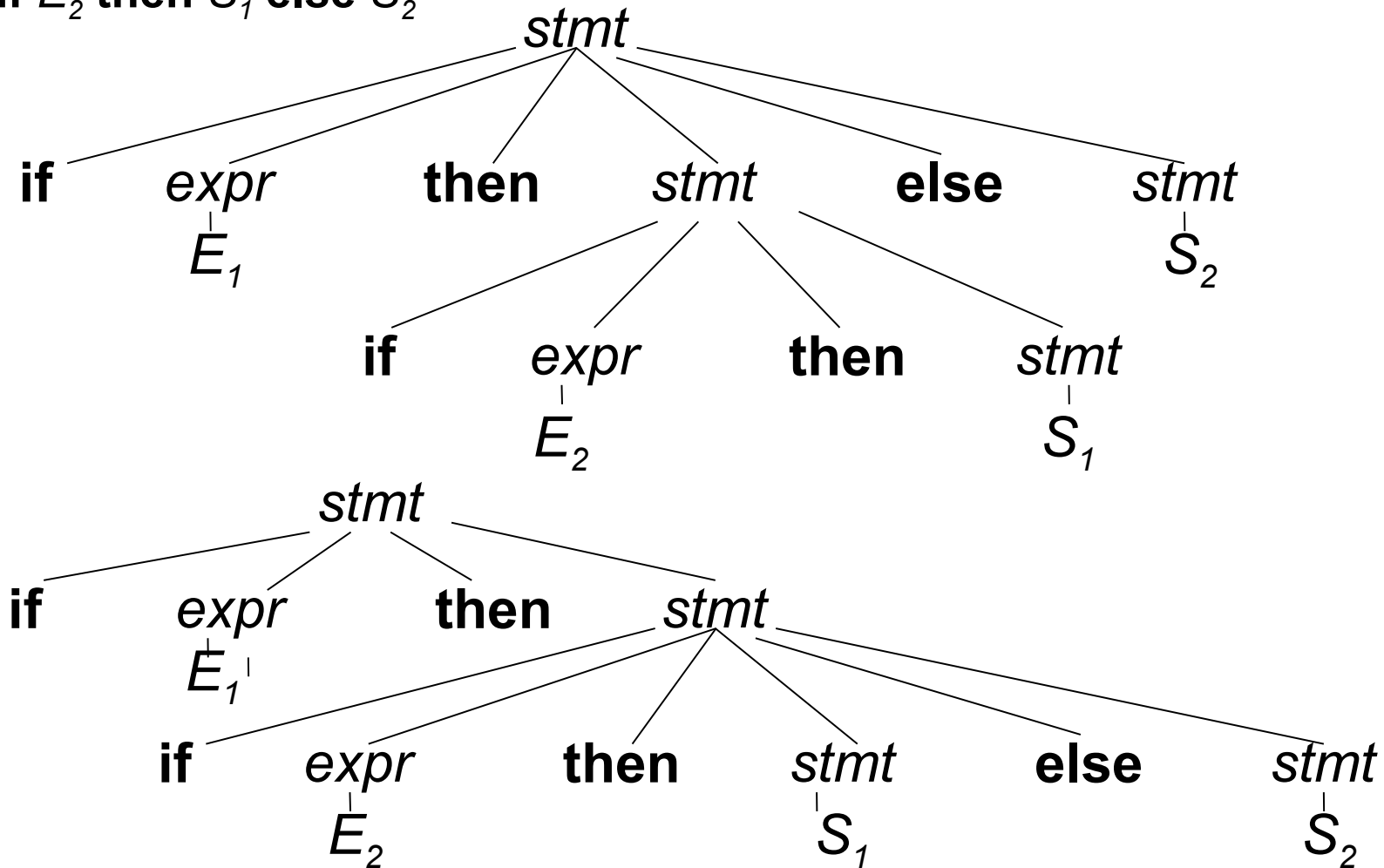
$\wedge =$ right

rest are left

“Dangling Else” Problem

stmt →
| **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

if E_1 then if E_2 then S_1 else S_2



Grammar for Closest-if Rule

- Want to rule out: if (E) if (E) S else S
- Impose that unmatched “if” statements occur only on the “else” clauses

```
stmt → matched_stmt  
      | unmatched_stmt  
matched_stmt → if expr then matched_stmt else matched_stmt  
              | other  
unmatched_stmt → if expr then stmt  
                 | if expr then matched_stmt else unmatched_stmt
```

Parsing Top-Down

Goal: construct a leftmost derivation of string while reading in sequential token stream

$$S \rightarrow E + S \mid E$$
$$E \rightarrow \text{num} \mid (S)$$

Partly-derived String	Lookahead	parsed part unparsed part
$\rightarrow E + S$	((1+2+(3+4)) +5
$\rightarrow (S) + S$	1	(1+2+(3+4)) +5
$\rightarrow (E+S)+S$	1	(1+2+(3+4)) +5
$\rightarrow (1+S)+S$	2	(1+2+(3+4)) +5
$\rightarrow (1+E+S)+S$	2	(1+2+(3+4)) +5
$\rightarrow (1+2+S)+S$	2	(1+2+(3+4)) +5
$\rightarrow (1+2+E)+S$	((1+2+(3+4)) +5
$\rightarrow (1+2+(S))+S$	3	(1+2+(3+4)) +5
$\rightarrow (1+2+(E+S))+S$	3	(1+2+(3+4)) +5
$\rightarrow \dots$		

Problem with Top-Down Parsing

Want to decide which production to apply based on next symbol

$$\begin{aligned} S &\rightarrow E + S \mid E \\ E &\rightarrow \text{num} \mid (S) \end{aligned}$$

Ex1: "(1)"

$$S \Rightarrow E \Rightarrow (S) \Rightarrow (E) \Rightarrow (1)$$

Ex2: "(1)+2"

$$S \Rightarrow \underline{E+S} \Rightarrow (S)+S \Rightarrow (E)+S \Rightarrow (1)+E \Rightarrow (1)+2$$

How did you know to pick E+S in Ex2, if you picked E followed by (S), you couldn't parse it?

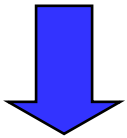
Grammar is Problem

$$\begin{aligned} S &\rightarrow E + S \mid E \\ E &\rightarrow \text{num} \mid (S) \end{aligned}$$

- This grammar cannot be parsed top-down with only a single look-ahead symbol!
- Not LL(1) = Left-to-right scanning, Left-most derivation, 1 look-ahead symbol
- Is it LL(k) for some k?
- If yes, then can rewrite grammar to allow top-down parsing: create LL(1) grammar for same language

Making a Grammar LL(1)

$S \rightarrow E + S$
 $S \rightarrow E$
 $E \rightarrow \text{num}$
 $E \rightarrow (S)$



$S \rightarrow ES'$
 $S' \rightarrow \epsilon$
 $S' \rightarrow +S$
 $E \rightarrow \text{num}$
 $E \rightarrow (S)$

- Problem: Can't decide which S production to apply until we see the symbol after the first expression
- Left-factoring: Factor common S prefix, add new non-terminal S' at decision point. S' derives $(+S)^*$
- Also: Convert left recursion to right recursion

Parsing with New Grammar

$$S \rightarrow ES'$$

$$S' \rightarrow \varepsilon \mid +S$$

$$E \rightarrow \text{num} \mid (S)$$

Partly-derived String	Lookahead	parsed part	unparsed part
$\rightarrow ES'$	((1+2+(3+4))+5	
$\rightarrow (S)S'$	1	(1+2+(3+4))+5	
$\rightarrow (ES')S'$	1	(1+2+(3+4))+5	
$\rightarrow (1S')S'$	+	(1+2+(3+4))+5	
$\rightarrow (1+ES')S'$	2	(1+2+(3+4))+5	
$\rightarrow (1+2S')S'$	+	(1+2+(3+4))+5	
$\rightarrow (1+2+S)S'$	((1+2+(3+4))+5	
$\rightarrow (1+2+ES')S'$	((1+2+(3+4))+5	
$\rightarrow (1+2+(S)S')S'$	3	(1+2+(3+4))+5	
$\rightarrow (1+2+(ES')S')S'$	3	(1+2+(3+4))+5	
$\rightarrow (1+2+(3S')S')S'$	+	(1+2+(3+4))+5	
$\rightarrow (1+2+(3+E)S')S'$	4	(1+2+(3+4))+5	
$\rightarrow \dots$			

Predictive Parsing

- LL(1) grammar:
 - For a given non-terminal, the lookahead symbol uniquely determines the production to apply
 - Top-down parsing = predictive parsing
 - Driven by predictive parsing table of
 - non-terminals x terminals \rightarrow productions

Adaptation for Predictive Parsing

- Elimination of left recursion

$expr \rightarrow expr + term \mid term$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \epsilon$

- Left factoring

$stmt \rightarrow \mathbf{if\ expr\ then\ stmt}$

$\mid \mathbf{if\ expr\ then\ stmt\ else\ stmt}$

$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

Transformation for Arithmetic Expression Grammar

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \mathbf{id}$$

$$E \rightarrow TE'$$

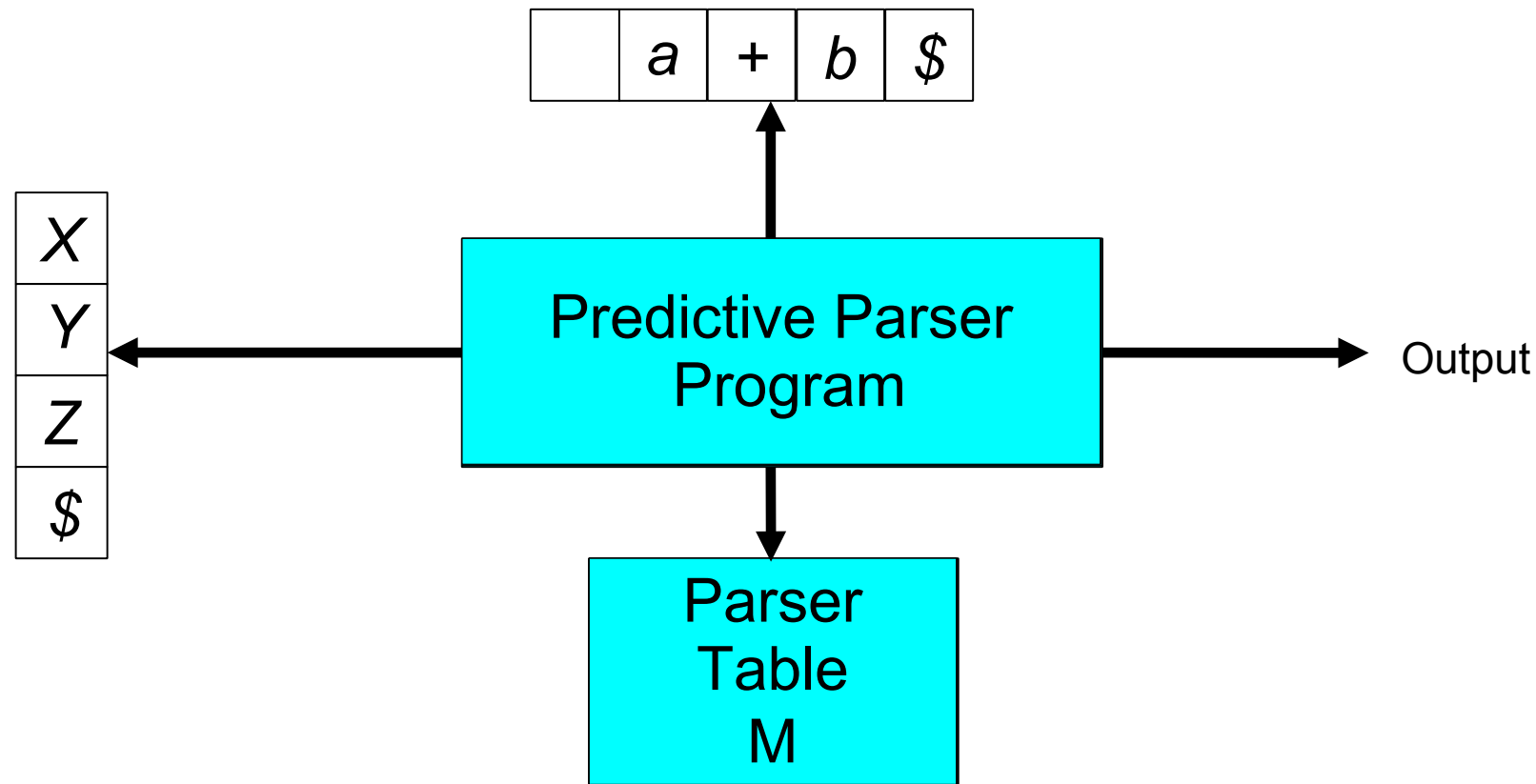
$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid \mathbf{id}$$

Predictive Parser without Recursion



1. If $X=a=\$$ stop and announce success
2. If $X=a \neq \$$ pop X off the stack and advance the input pointer
3. If X is a nonterminal, use production from $M[X,a]$

The M Table for Arithmetic Expressions

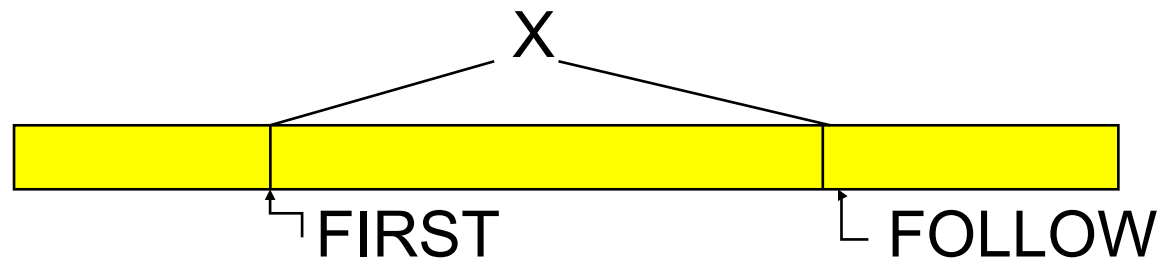
Nonterminal	Input Symbol					
	Id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Class Problem

- Parse the string
 - $\text{id} + \text{id} * \text{id}$

Constructing Parse Tables

- Can construct predictive parser if:
 - For every non-terminal, every lookahead symbol can be handled by at most 1 production
- $FIRST(\beta)$ for an arbitrary string of terminals and non-terminals β is:
 - Set of symbols that might begin the fully expanded version of β
- $FOLLOW(X)$ for a non-terminal X is:
 - Set of symbols that might follow the derivation of X in the input stream



Computation of FIRST(X)

1. If X is a terminal, $\text{FIRST}(X) = \{X\}$
2. If $X \rightarrow \epsilon$ is a production, add ϵ to $\text{FIRST}(X)$
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, place a in $\text{FIRST}(X)$ if for some i , a is in $\text{FIRST}(Y_i)$ and ϵ is in $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$.
If ϵ is in $\text{FIRST}(Y_j)$ for every j , add ϵ to $\text{FIRST}(X)$.

Computation of FOLLOW(X)

1. Place $\$$ in FOLLOW(S), where S is the start symbol
2. If there is a production $A \rightarrow \alpha B \beta$, everything in FIRST(β) except for ϵ is placed in FOLLOW(B)
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ , place all elements from FOLLOW(A) in FOLLOW(B)

Construction of Parsing Table M

1. For every production $A \rightarrow \alpha$ do steps 2 and 3
2. For each terminal a in $\text{FIRST}(\alpha)$ add $A \rightarrow \alpha$ to $M[A, a]$
3. If $\text{FIRST}(\alpha)$ contains ϵ , place $A \rightarrow \alpha$ in $M[A, b]$ for each b in $\text{FOLLOW}(A)$

Grammar is LL(1), if no conflicting entries

Error Handling

Types of errors

- Lexical
- Syntactic
- Semantic
- Logical

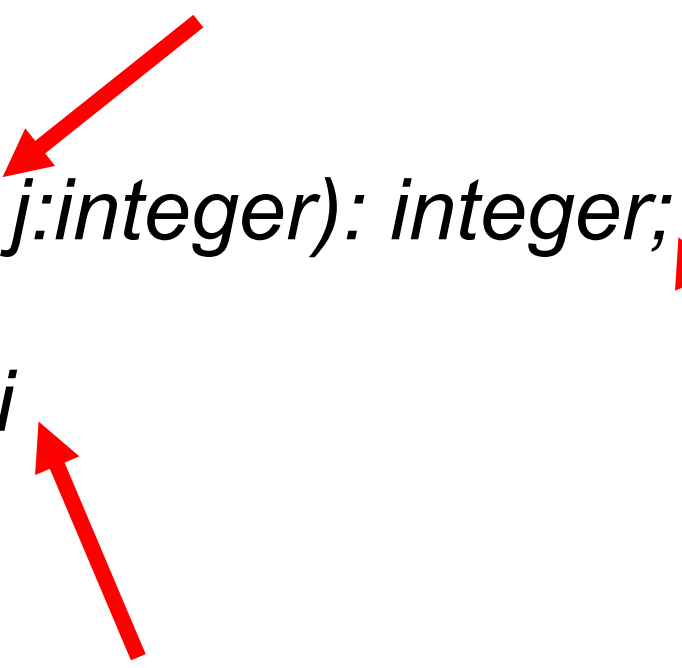
Error handler in a parser

- Should report the presence of errors clearly and accurately
- Should recover from each error quickly enough to be able to detect subsequent errors
- Should not significantly slow down the processing of correct programs

Typical Errors in A Pascal Program

```
program prmax(input,output);
var
    x,y: integer;
function max(i:integer; j:integer): integer;
begin
    if I > j then max:=i
    else max :=j
end;

begin
    readln (x,y);
    writeln(max(x,y))
end.
```



Error Handling Strategies

- Panic mode – skip tokens until a synchronizing token is found
- Phrase level – local error correction
- Error productions
- Global correction

Predictive Parser – Error Recovery

- Synchronizing tokens
 - FOLLOW(A)
 - Keywords
 - FIRST(A)
 - Empty production (if exists) as default in case of error
 - Insertion of token from the top of the stack
- Local error correction

Table M with Synchronizing Tokens

Nonterminal	Input symbol					
	Id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

- If $M[A,a]$ blank - skip input symbol a
- If $M[A,a]$ contains synch - pop nonterminal from the stack
- If the token at the top of stack does not match the input - pop terminal from the stack

Class Problem

- Parse the string
 - id^*id

Bottom-Up Parsing

- A more power parsing technology
- LR grammars – more expressive than LL
 - Construct right-most derivation of program
 - Left-recursive grammars, virtually all programming languages are left-recursive
 - Easier to express syntax
- Shift-reduce parsers
 - Parsers for LR grammars
 - Automatic parser generators (yacc, bison)

Bottom-Up Parsing

- Right-most derivation – Backward

$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{num} \mid (S)$$

- Start with the tokens
- End with the start symbol
- Match substring on RHS of production, replace by LHS

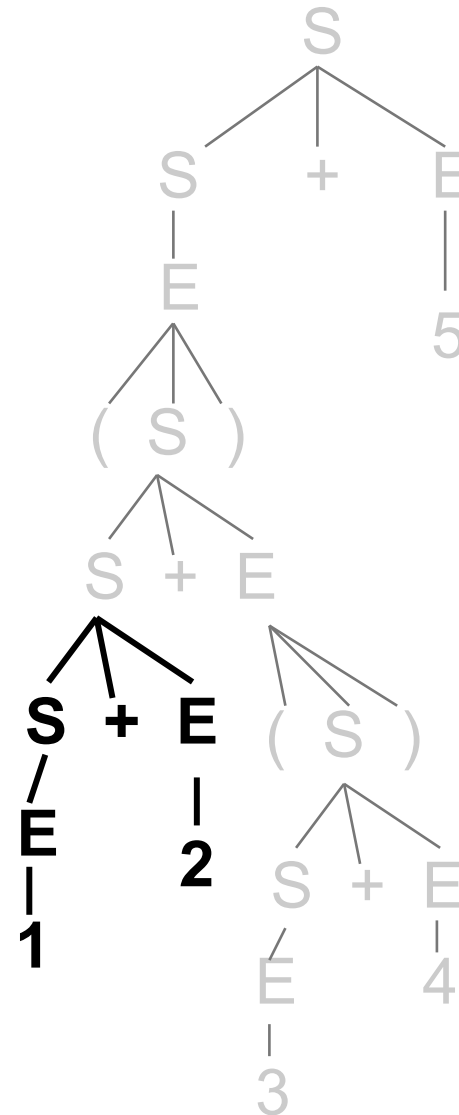
$$(1+2+(3+4))+5 \leq (E+2+(3+4))+5 \leq (S+2+(3+4))+5 \leq (S+E+(3+4))+5$$
$$\leq (S+(3+4))+5 \leq (S+(E+4))+5 \leq (S+(S+4))+5 \leq (S+(S+E))+5 \leq$$
$$(S+(S))+5 \leq (S+E)+5 \leq (S)+5 \leq E+5 \leq S+5 \leq S+E \leq S$$

Bottom-Up Parsing

$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{num} \mid (S)$$

$$(1+2+(3+4))+5$$
$$\leq (E+2+(3+4))+5$$
$$\leq (S+2+(3+4))+5$$
$$\leq (S+E+(3+4))+5$$

Advantage of bottom-up parsing:
can postpone the selection of
productions until more of the
input is scanned

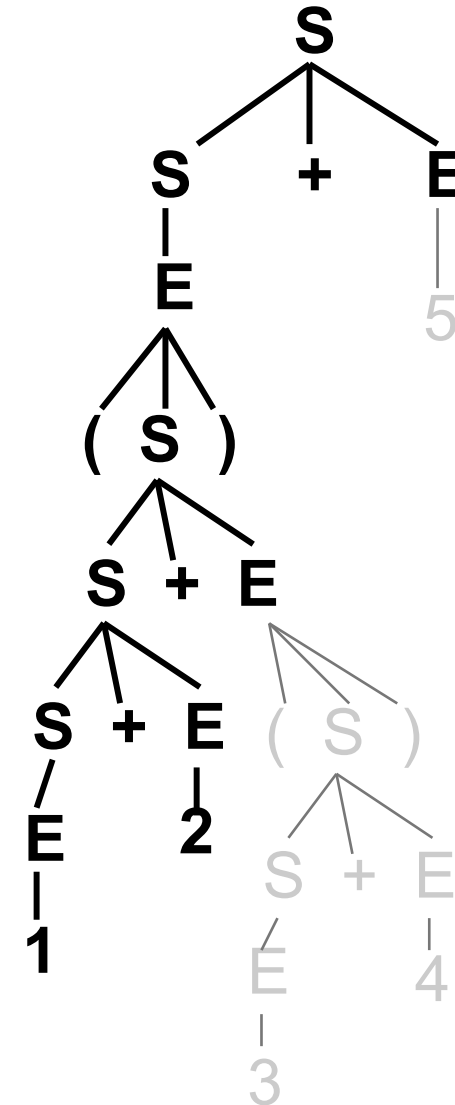


Top-Down Parsing

$S \rightarrow S + E \mid E$
 $E \rightarrow \text{num} \mid (S)$

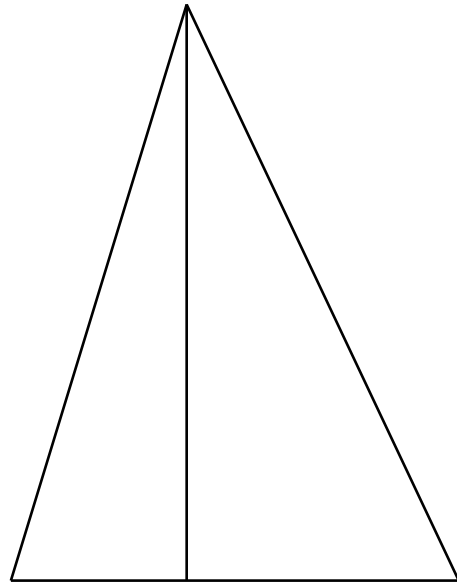
$S \Rightarrow S + E \Rightarrow E + E \Rightarrow$
 $(S) + E \Rightarrow (S + E) + E \Rightarrow$
 $(S + E + E) + E \Rightarrow$
 $(E + E + E) + E \Rightarrow (1 + E + E)$
 $+ E \Rightarrow (1 + 2 + E) + E \dots$

In left-most derivation, entire tree above token (2) has been expanded when encountered



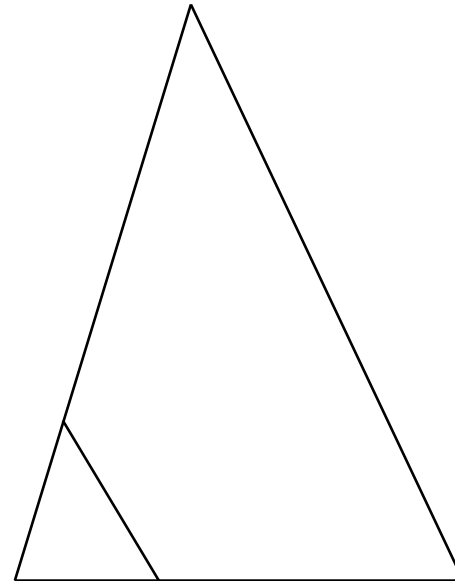
Top-Down vs Bottom-Up

- Bottom-up: Don't need to figure out as much of the parse tree for a given amount of input → More time to decide what rules to apply



scanned unscanned

Top-down



scanned unscanned

Bottom-up

Terminology: LL vs LR

- **LL(k)**
 - Left-to-right scan of input
 - **Left-most derivation**
 - k symbol lookahead
 - [Top-down or predictive] parsing or LL parser
 - Performs pre-order traversal of parse tree
- **LR(k)**
 - Left-to-right scan of input
 - **Right-most derivation**
 - k symbol lookahead
 - [Bottom-up or shift-reduce] parsing or LR parser
 - Performs post-order traversal of parse tree

Handles

- Handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation

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

$$E \Rightarrow \underline{E + E}$$

$$\Rightarrow E + \underline{E * E}$$

$$\Rightarrow E + E * \underline{\mathbf{id}_3}$$

$$\Rightarrow E + \underline{\mathbf{id}_2} * \mathbf{id}_3$$

$$\Rightarrow \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3$$

$$E \Rightarrow \underline{E * E}$$

$$\Rightarrow E * \underline{\mathbf{id}_3}$$

$$\Rightarrow \underline{E + E} * \mathbf{id}_3$$

$$\Rightarrow E + \underline{\mathbf{id}_2} * \mathbf{id}_3$$

$$\Rightarrow \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3$$

Handles

Right-Sentential Form	Handle	Reducing Production
$\mathbf{id_1 + id_2 * id_3}$	$\mathbf{id_1}$	$\mathbf{E \rightarrow id}$
$\mathbf{E + id_2 * id_3}$	$\mathbf{id_2}$	$\mathbf{E \rightarrow id}$
$\mathbf{E + E * id_3}$	$\mathbf{id_3}$	$\mathbf{E \rightarrow id}$
$\mathbf{E + E * E}$	$\mathbf{E * E}$	$\mathbf{E \rightarrow E * E}$
$\mathbf{E + E}$	$\mathbf{E + E}$	$\mathbf{E \rightarrow E + E}$
\mathbf{E}		

Shift-Reduce Parsing

- Parsing is a sequence of shifts and reduces
- Shift: move look-ahead token to stack
- Reduce: Replace symbols β from top of stack with non-terminal symbol X corresponding to the production: $X \rightarrow \beta$ (e.g., pop β , push X)

Stack	Input	Operation
\$	id₁ + id₂ * id₃ \$	Shift
\$id₁	+ id₂ * id₃ \$	Reduce by $E \rightarrow \mathbf{id}$
\$E	+ id₂ * id₃ \$	Shift
\$E +	id₂ * id₃ \$	Shift
\$E + id₂	* id₃ \$	Reduce by $E \rightarrow \mathbf{id}$
\$E + E	* id₃ \$	Shift
\$E + E *	id₃ \$	Shift
\$E + E * id₃	\$	Reduce by $E \rightarrow \mathbf{id}$
\$E + E * E	\$	Reduce by $E \rightarrow E * E$
\$E + E	\$	Reduce by $E \rightarrow E + E$
\$E	\$	Accept

Potential Problems

- How do we know which action to take: whether to shift or reduce, and which production to apply
- Issues
 - Sometimes can reduce but should not
 - Sometimes can reduce in different ways

Action Selection Problem

- Given stack β and look-ahead symbol b , should parser:
 - Shift b onto the stack making it βb ?
 - Reduce $X \rightarrow \gamma$ assuming that the stack has the form $\beta = \alpha\gamma$ making it αX ?
- If stack has the form $\alpha\gamma$, should apply reduction $X \rightarrow \gamma$ (or shift) depending on stack prefix α
 - α is different for different possible reductions since γ 's have different lengths

Shift/Reduce and Reduce/Reduce Conflicts

stmt → **if *expr* then *stmt***
| **if *expr* then *stmt* else *stmt***
| **other**

... if *expr* then *stmt* else ... \$

...
stmt → **id (*parameter_list*)**

...
expr → **id (*expr_list*)**

...
... id (id , id) ... \$

yacc / bison – Parser Generators

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line:    expr '\n'    { printf("%d\n", $1); }
        ;
expr:    expr '+' term    { $$ = $1 + $3; }
        | term
        ;
term:    term '*' factor  { $$ = $1 * $3; }
        | factor
        ;
factor  :    '(' expr ')' { $$ = $2; }
        | DIGIT
        ;
%%

int yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

Operator Precedence in bison

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
}%

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines:    lines expr '\n'    { printf("%g\n", $2); }
        |    lines '\n'
        |    /* empty */
        ;

expr :    expr '+' expr { $$ = $1 + $3; }
        |    expr '-' expr { $$ = $1 - $3; }
        |    expr '*' expr { $$ = $1 * $3; }
        |    expr '/' expr { $$ = $1 / $3; }
        |    '(' expr ')' { $$ = $2; }
        |    '-' expr %prec UMINUS { $$ = -$2; }
        |    NUMBER
        ;

%%

yylex() {
    int c;
    while ( ( c = getchar() ) == ' ');
    if ( c == '.' || isdigit(c) ) {
        ungetc(c, stdin);
        scanf("%lf",&yylval);
        return NUMBER;
    }
    return c;
}
```

yacc / bison – Conflict Resolution

1. Reduce/reduce – first production listed in the input file selected
2. Shift/reduce – shift performed

Terminals can be assigned with precedence and associativity in declarative part of the input file.

Precedence of a production is usually the precedence of rightmost terminal. Can be overridden.

For the conflict: reduce $A \rightarrow \alpha$ and shift a

reduce – if precedence of production greater than precedence of a or they are equal and associativity of the production is left

Error Handling

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
}%

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines:  lines expr '\n'  { printf("%g\n", $2); }
      |  lines '\n'
      |  /* empty */
      |  error '\n' { yyerror("reenter last line:"); yyerrok; }
      ;

expr:   expr '+' expr    { $$ = $1 + $3; }
      |  expr '-' expr    { $$ = $1 - $3; }
      |  expr '*' expr    { $$ = $1 * $3; }
      |  expr '/' expr    { $$ = $1 / $3; }
      |  '(' expr ')' { $$ = $2; }
      |  '-' expr %prec UMINUS { $$ = -$2; }
      |  NUMBER
      ;

%%
```

LR Parsing Engine

- Basic mechanism
 - Use a set of parser states
 - Use stack with alternating symbols and states
 - E.g., 1 (6 S 10 + 5 (blue = state numbers)
 - Use parsing table to:
 - Determine what action to apply (shift/reduce)
 - Determine next state
- The parser actions can be precisely determined from the table

LR Parsing Table

	Terminals	Non-terminals
State	Next action and next state	Next state
	Action table	Goto table

- Algorithm: look at entry for current state S and input terminal C
 - If $\text{Action}[S,C] = s(S')$ then shift:
 - $\text{push}(C), \text{push}(S')$
 - If $\text{Action}[S,C] = X \rightarrow \alpha$ then reduce:
 - $\text{pop}(2*|\alpha|), S' = \text{top}(), \text{push}(X), \text{push}(\text{Goto}[S',X])$

LR Parsing Table Example

We want to derive this in an algorithmic fashion

State	Action					Goto	
	()	id	,	\$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

Parsing Example ((a),b)

derivation	stack	input	action	$S \rightarrow (L) \mid id$ $L \rightarrow S \mid L,S$
((a),b)<=	1	((a),b)	shift, goto 3	
((a),b)<=	1(3	(a),b)	shift, goto 3	
((a),b)<=	1(3(3	a),b)	shift, goto 2	
((a),b)<=	1(3(3a2),b)	reduce $S \rightarrow id$	
((S),b)<=	1(3(3(S7),b)	reduce $L \rightarrow S$	
((L),b)<=	1(3(3(L5),b)	shift, goto 6	
((L),b)<=	1(3(3L5)6	,b)	reduce $S \rightarrow (L)$	
(S,b)<=	1(3S7	,b)	reduce $L \rightarrow S$	
(L,b)<=	1(3L5	,b)	shift, goto 8	
(L,b)<=	1(3L5,8	b)	shift, goto 2	
(L,b)<=	1(3L5,8b2)	reduce $S \rightarrow id$	
(L,S)<=	1(3L8,S9)	reduce $L \rightarrow L,S$	
(L)<=	1(3L5)	shift, goto 6	
(L)<=	1(3L5)6	\$	reduce $S \rightarrow (L)$	
S	1S4	\$	done	

LR(k) Grammars

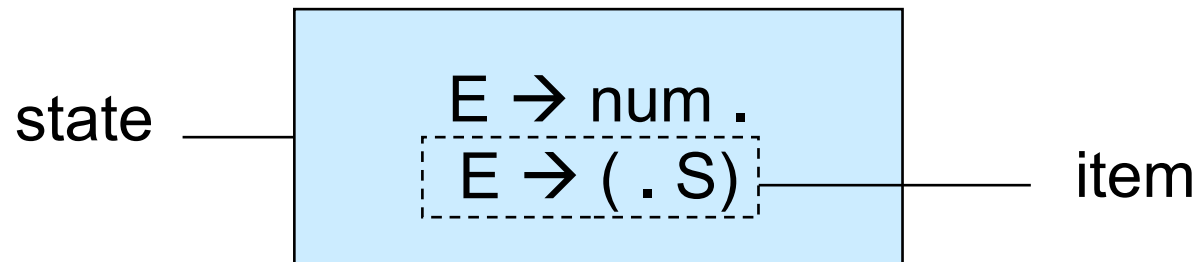
- LR(k) = Left-to-right scanning, right-most derivation, k lookahead chars
- Main cases
 - LR(0), LR(1)
 - Some variations SLR and LALR(1)
- Parsers for LR(0) Grammars:
 - Determine the actions without any lookahead
 - Will help us understand shift-reduce parsing

Building LR(0) Parsing Tables

- To build the parsing table:
 - Define states of the parser
 - Build a DFA to describe transitions between states
 - Use the DFA to build the parsing table
- Each LR(0) state is a set of LR(0) items
 - An LR(0) item: $X \rightarrow \alpha . \beta$ where $X \rightarrow \alpha\beta$ is a production in the grammar
 - The LR(0) items keep track of the progress on all of the possible upcoming productions
 - The item $X \rightarrow \alpha . \beta$ abstracts the fact that the parser already matched the string α at the top of the stack

Example LR(0) State

- An LR(0) item is a production from the language with a separator “.” somewhere in the RHS of the production



- Sub-string before “.” is already on the stack (beginnings of possible γ 's to be reduced)
- Sub-string after “.”: what we might see next

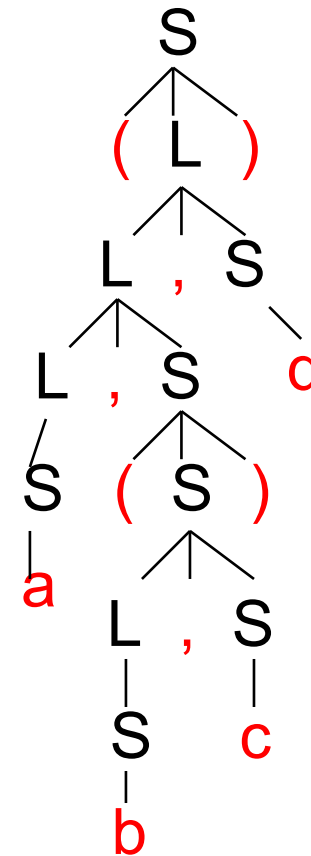
Class Problem

- For the production,
 - $E \rightarrow \text{num} \mid (S)$
- Two items are:
 - $E \rightarrow \text{num} \cdot$
 - $E \rightarrow (\cdot S)$
- Are there any others?
 - If so, what are they?
 - If not, why?

LR(0) Grammar

- Nested lists
 - $S \rightarrow (L) \mid id$
 - $L \rightarrow S \mid L,S$
- Examples
 - (a,b,c)
 - $((a,b), (c,d), (e,f))$
 - $(a, (b,c,d), ((f,g)))$

Parse tree for
 $(a, (b,c), d)$



Start State and Closure

- Start state
 - Augment grammar with production: $S' \rightarrow S \$$
 - Start state of DFA has empty stack: $S' \rightarrow . S \$$
- Closure of a parser state:
 - Start with $\text{Closure}(S) = S$
 - Then for each item in S :
 - $X \rightarrow \alpha . Y \beta$
 - Add items for all the productions $Y \rightarrow \gamma$ to the closure of S : $Y \rightarrow . \gamma$

Closure Example

$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L,S$

DFA start state

$S' \rightarrow \cdot S \$$

closure

$S' \rightarrow \cdot S \$$

$S \rightarrow \cdot (L)$

$S \rightarrow \cdot id$

- Set of possible productions to be reduced next
- Added items have the “.” located at the beginning:
no symbols for these items on the stack yet

The Goto Operation

- Goto operation = describes transitions between parser states, which are sets of items
- Algorithm: for state S and a symbol Y
 - If the item $[X \rightarrow \alpha . Y \beta]$ is in S , then
 - $\text{Goto}(S, Y) = \text{Closure}([X \rightarrow \alpha Y . \beta])$

$S' \rightarrow . S \$$
$S \rightarrow . (L)$
$S \rightarrow . id$

Goto($S, '('$)

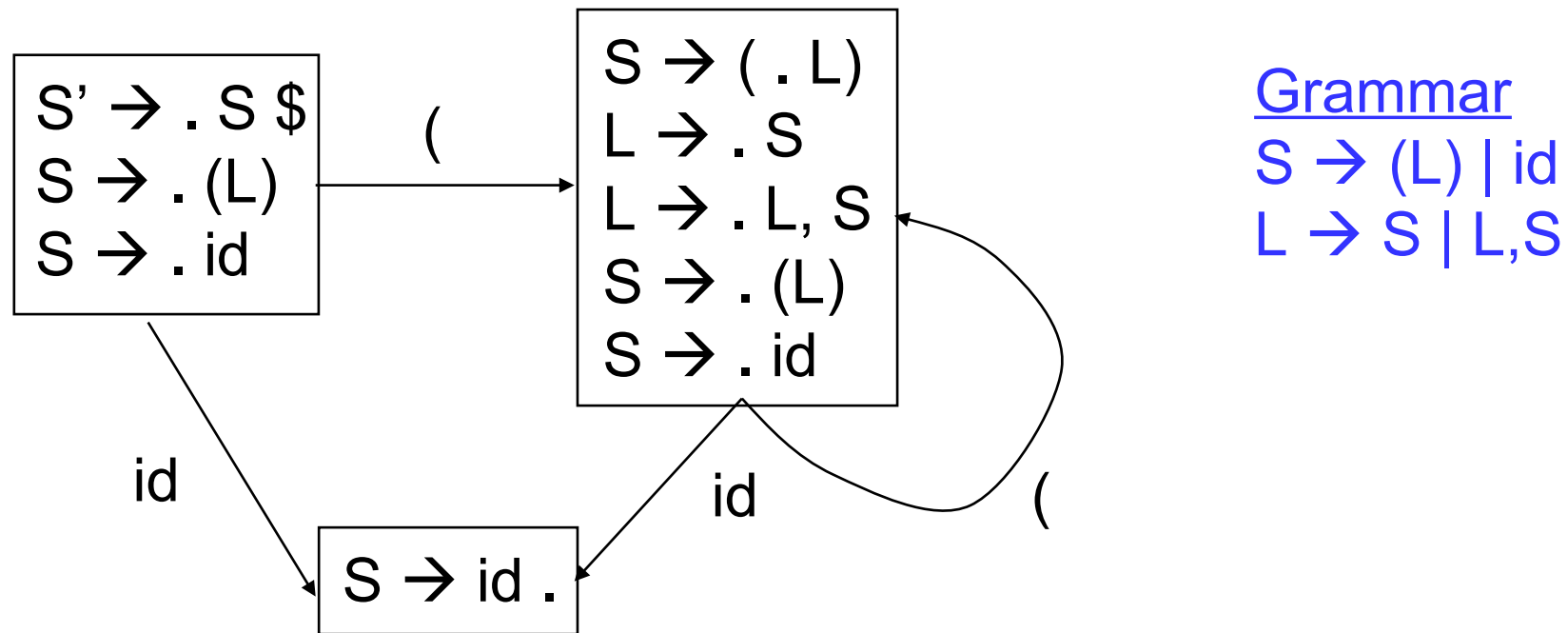
$\text{Closure}([S \rightarrow (.L)])$
--

Class Problem

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

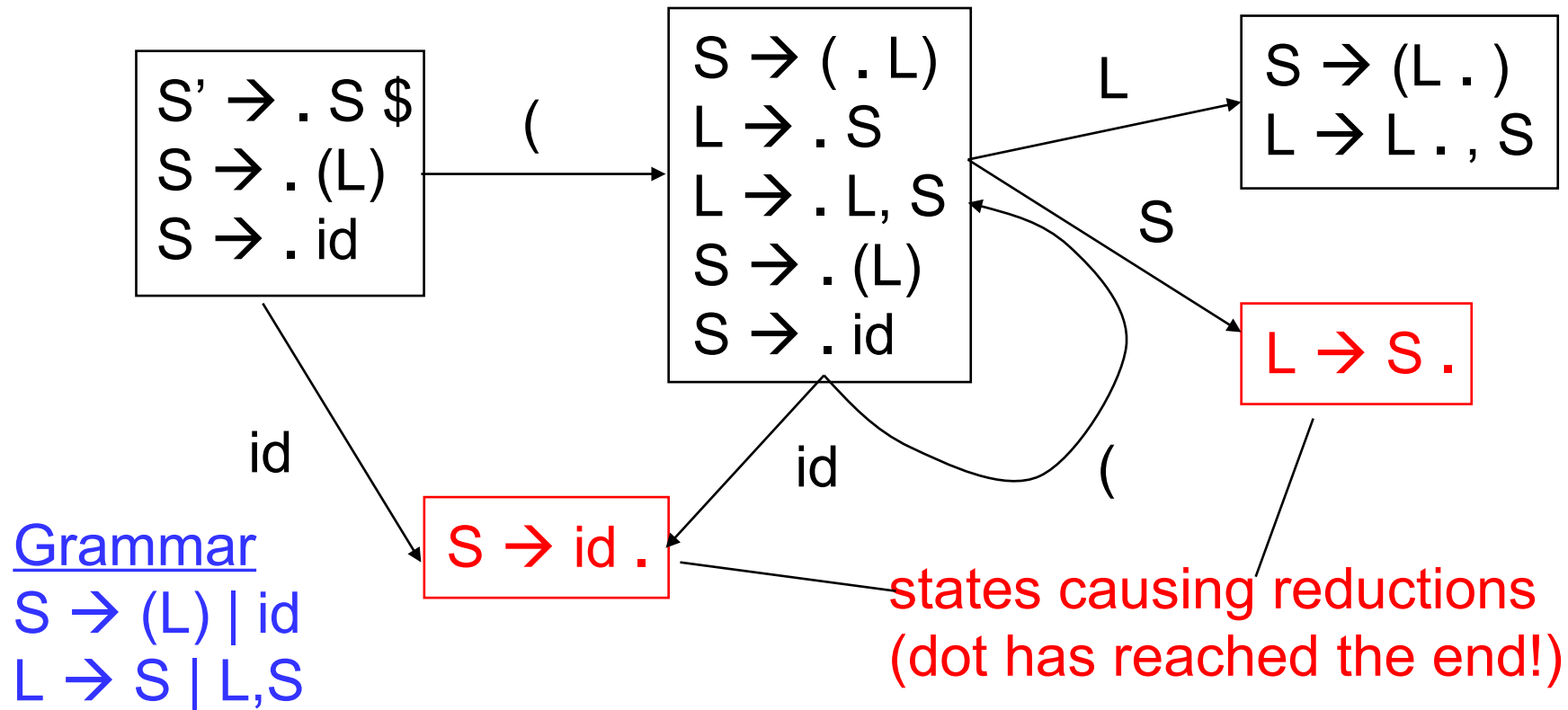
- If $I = \{ [E' \rightarrow \cdot E] \}$, then $\text{Closure}(I) = ??$
- If $I = \{ [E' \rightarrow E \cdot], [E \rightarrow E \cdot + T] \}$, then $\text{Goto}(I, +) = ??$

Goto: Terminal Symbols



In new state, include all items that have appropriate input symbol just after dot, advance dot in those items and take closure

Applying Reduce Actions



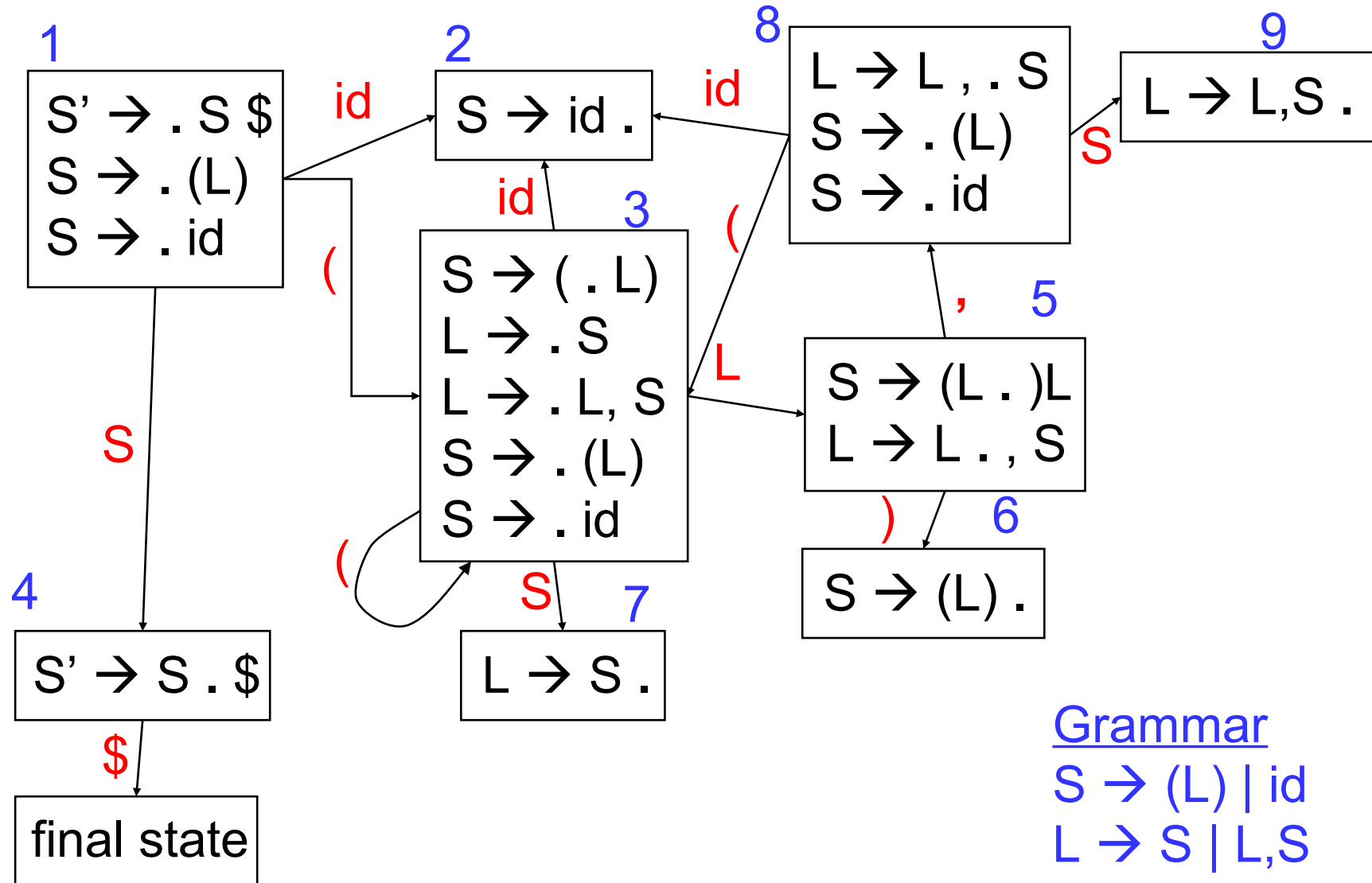
Pop RHS off stack, replace with LHS X ($X \rightarrow \beta$), then rerun DFA

Reductions

- On reducing $X \rightarrow \beta$ with stack $\alpha\beta$
 - Pop β off stack, revealing prefix α and state
 - Take single step in DFA from top state
 - Push X onto stack with new DFA state
- Example

derivation	stack	input	action
$((a),b) \leq$	1 (3 (3	a),b)	shift, goto 2
$((a),b) \leq$	1 (3 (3 a 2),b)	reduce $S \rightarrow id$
$((S),b) \leq$	1 (3 (3 S 7),b)	reduce $L \rightarrow S$

Full DFA



Building the Parsing Table

- States in the table = states in the DFA
- For transition $S \rightarrow S'$ on terminal C :
 - $\text{Action}[S,C] += \text{Shift}(S')$
- For transition $S \rightarrow S'$ on non-terminal N :
 - $\text{Goto}[S,N] += \text{Goto}(S')$
- If S is a reduction state $X \rightarrow \beta$ then:
 - $\text{Action}[S,*] += \text{Reduce}(X \rightarrow \beta)$

LR(0) Summary

- LR(0) parsing recipe:
 - Start with LR(0) grammar
 - Compute LR(0) states and build DFA:
 - Use the closure operation to compute states
 - Use the goto operation to compute transitions
 - Build the LR(0) parsing table from the DFA
- This can be done automatically

Class Problem

- Generate the DFA for the following grammar
 - $S \rightarrow E + S \mid E$
 - $E \rightarrow \text{num}$

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
 - Always reduce regardless of lookahead
- With a more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use lookahead to choose

OK

$L \rightarrow L, S.$

shift/reduce

$L \rightarrow L, S.$
 $S \rightarrow S., L$

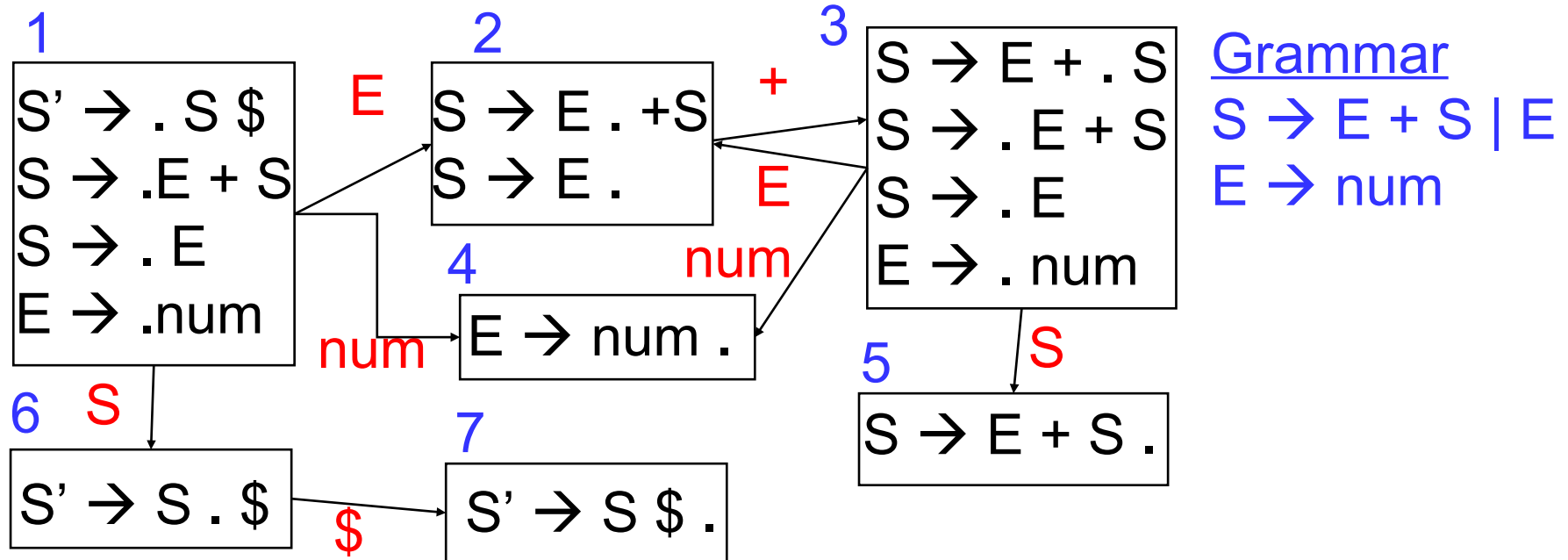
reduce/reduce

$L \rightarrow S, L.$
 $L \rightarrow S.$

A Non-LR(0) Grammar

- Grammar for addition of numbers
 - $S \rightarrow S + E \mid E$
 - $E \rightarrow \text{num}$
- Left-associative version is LR(0)
- Right-associative is **not LR(0)** as you saw with the previous class problem
 - $S \rightarrow E + S \mid E$
 - $E \rightarrow \text{num}$

LR(0) Parsing Table



Shift or
 reduce
 in state 2?

	num	+	\$	E	S
1	s4			g2	g6
2	$S \rightarrow E$	s3/ $S \rightarrow E$	$S \rightarrow E$		

Solve Conflict With Lookahead

- 3 popular techniques for employing lookahead of 1 symbol with bottom-up parsing
 - SLR – Simple LR
 - LALR – LookAhead LR
 - LR(1)
- Each as a different means of utilizing the lookahead
 - Results in different processing capabilities

SLR Parsing

- SLR Parsing = Easy extension of LR(0)
 - For each reduction $X \rightarrow \beta$, look at next symbol C
 - Apply reduction only if C is in FOLLOW(X)
- SLR parsing table eliminates some conflicts
 - Same as LR(0) table except reduction rows
 - Adds reductions $X \rightarrow \beta$ only in the columns of symbols in FOLLOW(X)

Example: FOLLOW(S) = {\$}

Grammar

$S \rightarrow E + S \mid E^1$

$E \rightarrow \text{num}^2$

	num	+	\$	E	S
s4				g2	g6
		s3	S → E		

SLR Parsing Table

- Reductions do not fill entire rows as before
- Otherwise, same as LR(0)

Grammar
 $S \rightarrow E + S \mid E$
 $E \rightarrow \text{num}$

	num	+	\$	E	S
1	s4			g2	g6
2		s3	$S \rightarrow E$		
3	s4			g2	g5
4		$E \rightarrow \text{num}$	$E \rightarrow \text{num}$		
5			$S \rightarrow E + S$		
6			s7		
7			accept		

Class Problem

- Consider:

- $S \rightarrow L = R$
 - $S \rightarrow R$
 - $L \rightarrow *R$
 - $L \rightarrow \text{ident}$
 - $R \rightarrow L$
- Think of L as l-value, R as r-value, and * as a pointer dereference

When you create the states in the SLR(1) DFA, 2 of the states are the following:

$S \rightarrow L . = R$
$R \rightarrow L .$

$S \rightarrow R .$

Do you have any shift/reduce conflicts?

LR(1) Parsing

- Get as much as possible out of 1 lookahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 lookahead
- LR(1) parsing uses similar concepts as LR(0)
 - Parser states = set of items
 - LR(1) item = LR(0) item + lookahead symbol possibly following production
 - LR(0) item: $S \rightarrow . S + E$
 - LR(1) item: $S \rightarrow . S + E \underline{, +}$
 - Lookahead only has impact upon REDUCE operations, apply when lookahead = next input

LR(1) States

- **LR(1) state = set of LR(1) items**
- LR(1) item = $(X \rightarrow \alpha \cdot \beta, y)$
 - Meaning: α already matched at top of the stack, next expect to see βy
- Shorthand notation
 - $(X \rightarrow \alpha \cdot \beta, \{x_1, \dots, x_n\})$
 - means:
 - $(X \rightarrow \alpha \cdot \beta, x_1)$
 - \dots
 - $(X \rightarrow \alpha \cdot \beta, x_n)$
- Need to extend closure and goto operations

$S \rightarrow S \cdot + E$	$+, \$$
$S \rightarrow S + \cdot E$	num

LR(1) Closure

- LR(1) closure operation:
 - Start with $\text{Closure}(S) = S$
 - For each item in S :
 - $X \rightarrow \alpha . Y \beta, z$
 - and for each production $Y \rightarrow \gamma$, add the following item to the closure of S : $Y \rightarrow . \gamma, \text{FIRST}(\beta z)$
 - Repeat until nothing changes
- Similar to LR(0) closure, but also keeps track of lookahead symbol

LR(1) Start State

- Initial state: start with $(S' \rightarrow \cdot S, \$)$, then apply closure operation
- Example: sum grammar

$S' \rightarrow S \$$
 $S \rightarrow E + S \mid E$
 $E \rightarrow \text{num}$

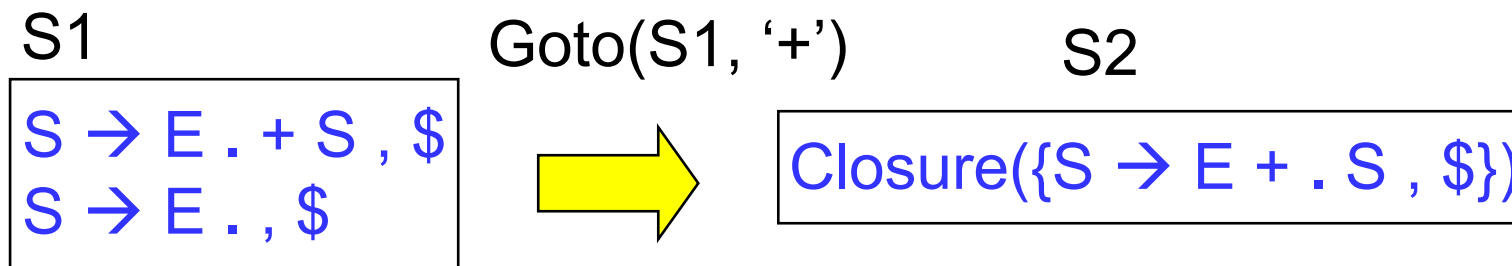
$S' \rightarrow \cdot S, \$$

closure

$S' \rightarrow \cdot S, \$$
 $S \rightarrow \cdot E + S, \$$
 $S \rightarrow \cdot E, \$$
 $E \rightarrow \cdot \text{num}, +, \$$

LR(1) Goto Operation

- LR(1) goto operation = describes transitions between LR(1) states
- Algorithm: for a state S and a symbol Y (as before)
 - If the item $[X \rightarrow \alpha . Y \beta]$ is in S , then
 - $\text{Goto}(S, Y) = \text{Closure}([X \rightarrow \alpha Y . \beta])$



Grammar:

$S' \rightarrow S\$$

$S \rightarrow E + S \mid E$

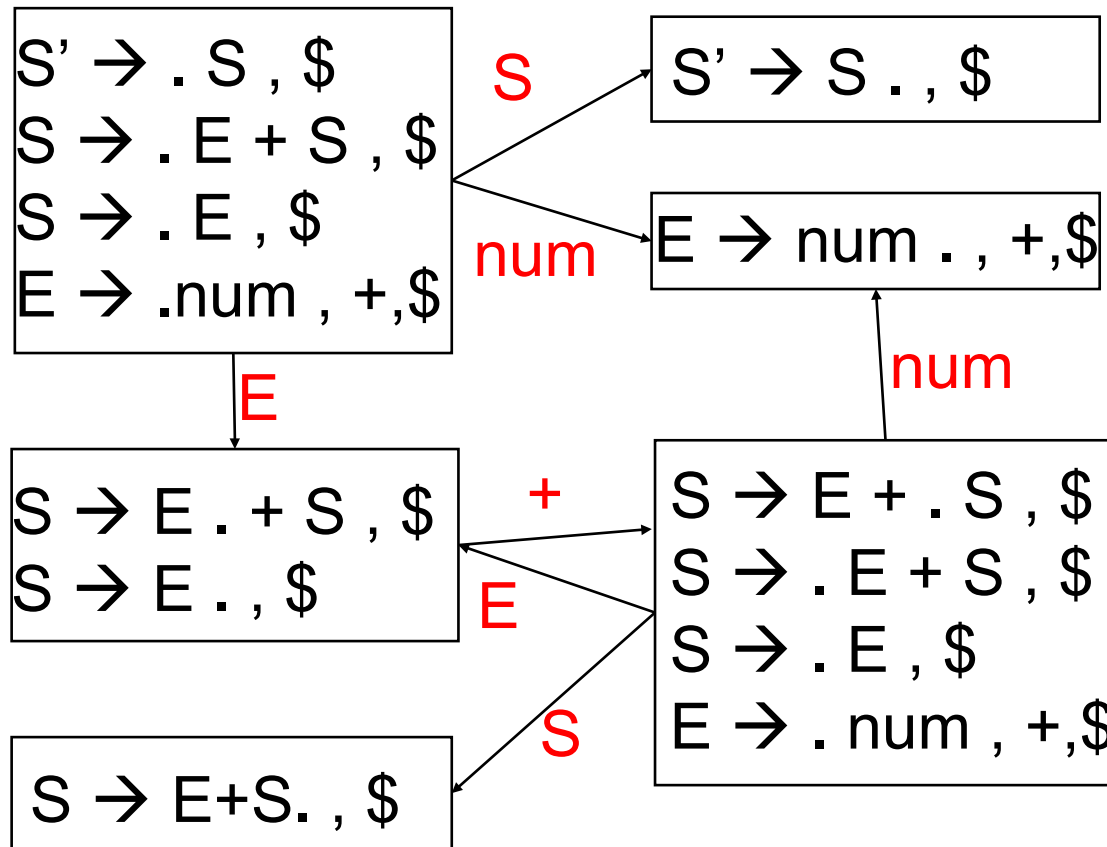
$E \rightarrow \text{num}$

Class Problem

1. Compute: $\text{Closure}(I = \{S \rightarrow E + \cdot S, \$\})$
2. Compute: $\text{Goto}(I, \text{num})$
3. Compute: $\text{Goto}(I, E)$

$S' \rightarrow S \$$
 $S \rightarrow E + S \mid E$
 $E \rightarrow \text{num}$

LR(1) DFA Construction

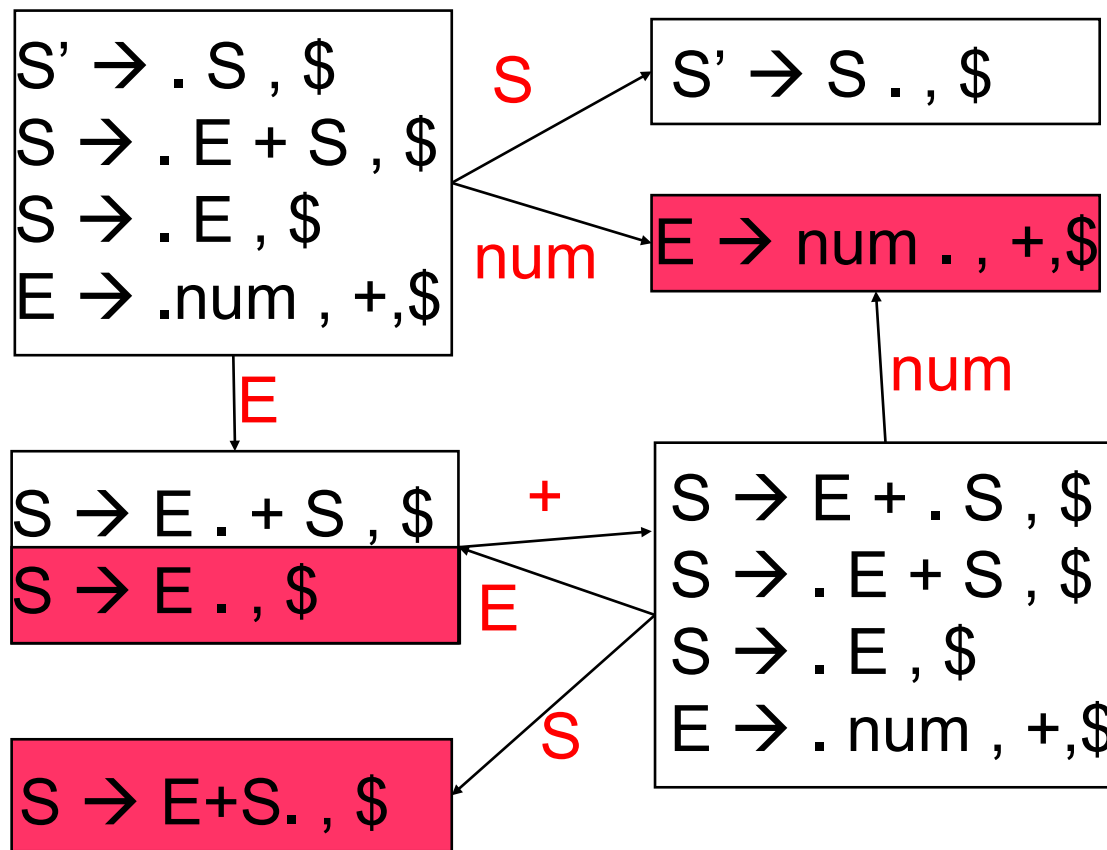


Grammar

$S' \rightarrow S\$$
 $S \rightarrow E + S \mid E$
 $E \rightarrow \text{num}$

LR(1) Reductions

- Reductions correspond to LR(1) items of the form $(X \rightarrow \gamma \cdot, y)$



Grammar

$S' \rightarrow S\$$
 $S \rightarrow E + S \mid E$
 $E \rightarrow \text{num}$

LR(1) Parsing Table Construction

- Same as construction of LR(0), except for reductions
- For a transition $S \rightarrow S'$ on terminal x :
 - $\text{Table}[S,x] += \text{Shift}(S')$
- For a transition $S \rightarrow S'$ on non-terminal N :
 - $\text{Table}[S,N] += \text{Goto}(S')$
- If I contains $\{(X \rightarrow \gamma \cdot, y)\}$ then:
 - $\text{Table}[I,y] += \text{Reduce}(X \rightarrow \gamma)$

LR(1) Parsing Table Example

1

$S' \rightarrow \cdot S, \$$
 $S \rightarrow \cdot E + S, \$$
 $S \rightarrow \cdot E, \$$
 $E \rightarrow \cdot \text{num}, +, \$$

E

2

$S \rightarrow E \cdot + S, \$$
 $S \rightarrow E \cdot, \$$

+

3

$S \rightarrow E + \cdot S, \$$
 $S \rightarrow \cdot E + S, \$$
 $S \rightarrow \cdot E, \$$
 $E \rightarrow \cdot \text{num}, +, \$$

Grammar

$S' \rightarrow S\$$

$S \rightarrow E + S \mid E$

$E \rightarrow \text{num}$

Fragment of the parsing table

	+	\$	E
1			g2
2	s3	$S \rightarrow E$	

Class Problem

- Compute the LR(1) DFA for the following grammar
 - $E \rightarrow E + T \mid T$
 - $T \rightarrow TF \mid F$
 - $F \rightarrow F^* \mid a \mid b$

LALR(1) Grammars

- Problem with LR(1): too many states
- LALR(1) parsing (aka LookAhead LR)
 - Constructs LR(1) DFA and then merge any 2 LR(1) states whose items are identical except lookahead
 - Results in smaller parser tables
 - Theoretically less powerful than LR(1)

$$\begin{array}{|l} S \rightarrow id . , + \\ S \rightarrow E . , \$ \end{array} + \begin{array}{|l} S \rightarrow id . , \$ \\ S \rightarrow E . , + \end{array} = ??$$

- LALR(1) grammar = a grammar whose LALR(1) parsing table has no conflicts

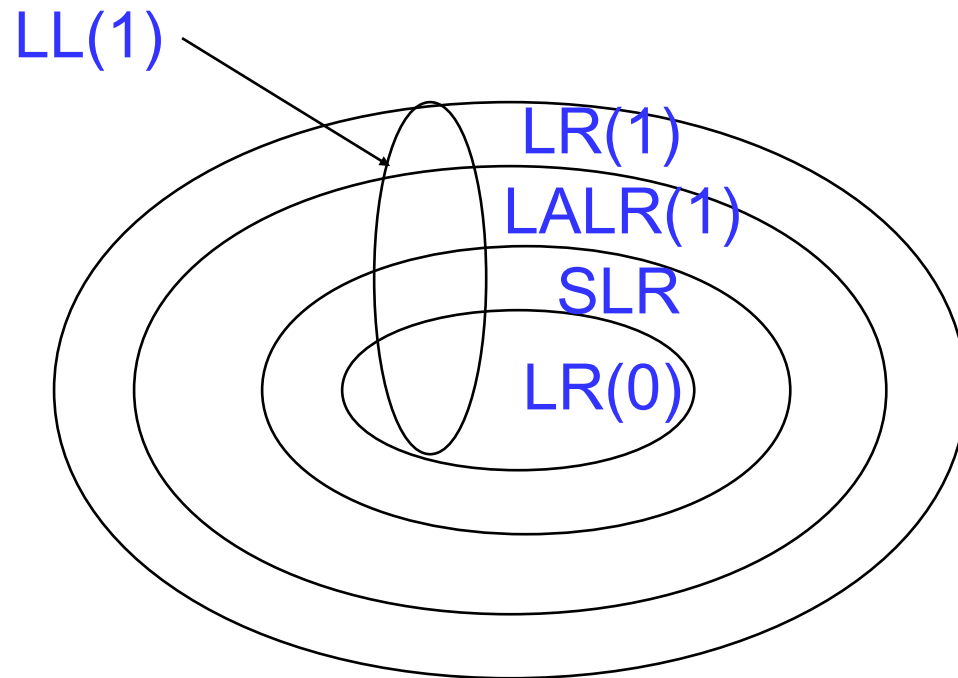
LALR Parsers

- LALR(1)
 - Generally same number of states as SLR (much less than LR(1))
 - But, with same lookahead capability of LR(1) (much better than SLR)
 - Example: Pascal programming language
 - In SLR, several hundred states
 - In LR(1), several thousand states

LL/LR Grammar Summary

- LL parsing tables
 - Non-terminals x terminals \rightarrow productions
 - Computed using FIRST/FOLLOW
- LR parsing tables
 - LR states x terminals \rightarrow {shift/reduce}
 - LR states x non-terminals \rightarrow goto
 - Computed using closure/goto operations on LR states
- A grammar is:
 - LL(1) if its LL(1) parsing table has no conflicts
 - same for LR(0), SLR, LALR(1), LR(1)

Classification of Grammars



not to scale 😊

$$LR(k) \subseteq LR(k+1)$$

$$LL(k) \subseteq LL(k+1)$$

$$LL(k) \subseteq LR(k)$$

$$LR(0) \subseteq SLR$$

$$LALR(1) \subseteq LR(1)$$

Automate the Parsing Process

- Can automate:
 - The construction of LR parsing tables
 - The construction of shift-reduce parsers based on these parsing tables
- LALR(1) parser generators
 - yacc, bison
 - Not much difference compared to LR(1) in practice
 - Smaller parsing tables than LR(1)
 - Augment LALR(1) grammar specification with declarations of **precedence, associativity**
 - Output: LALR(1) parser program