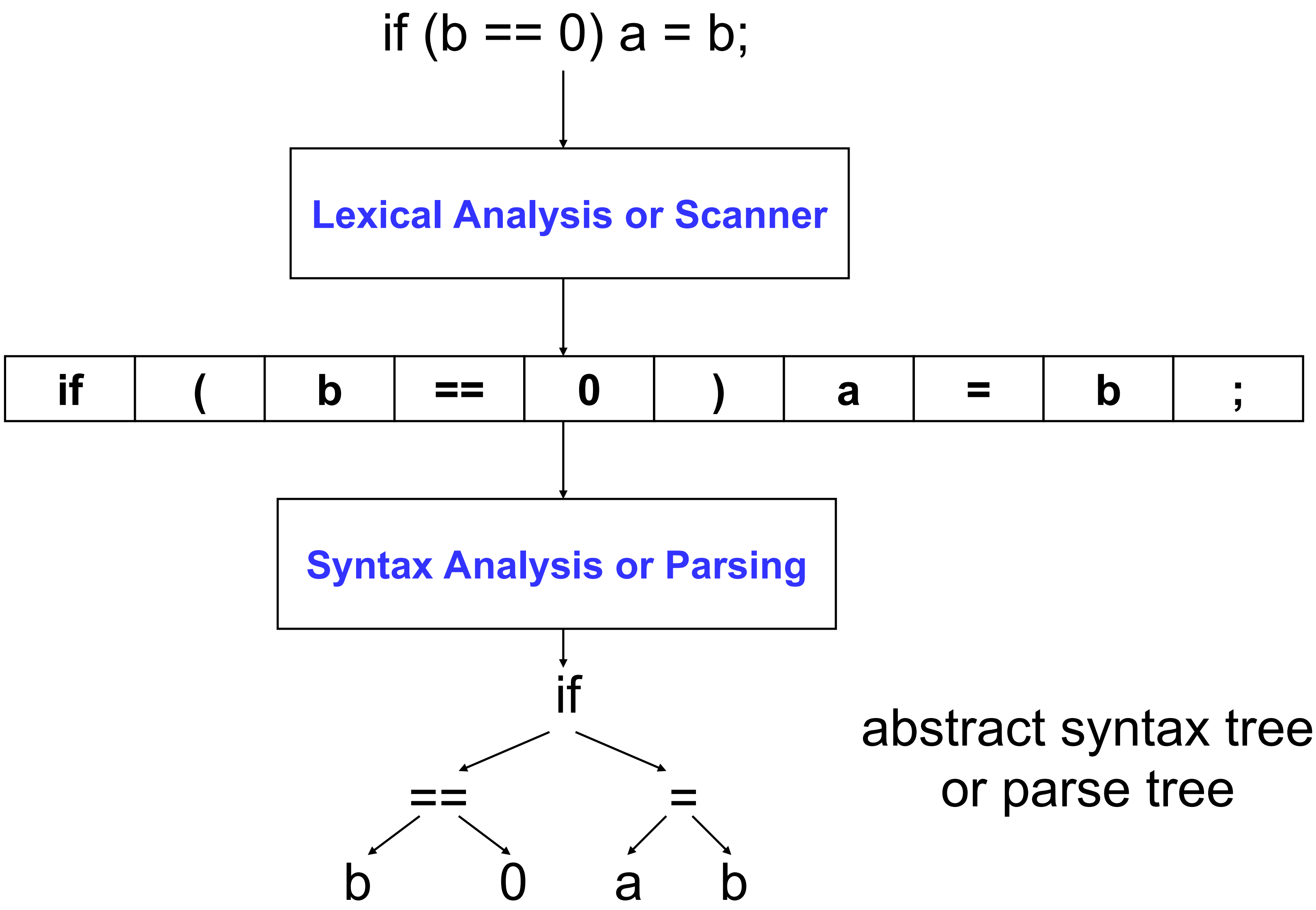
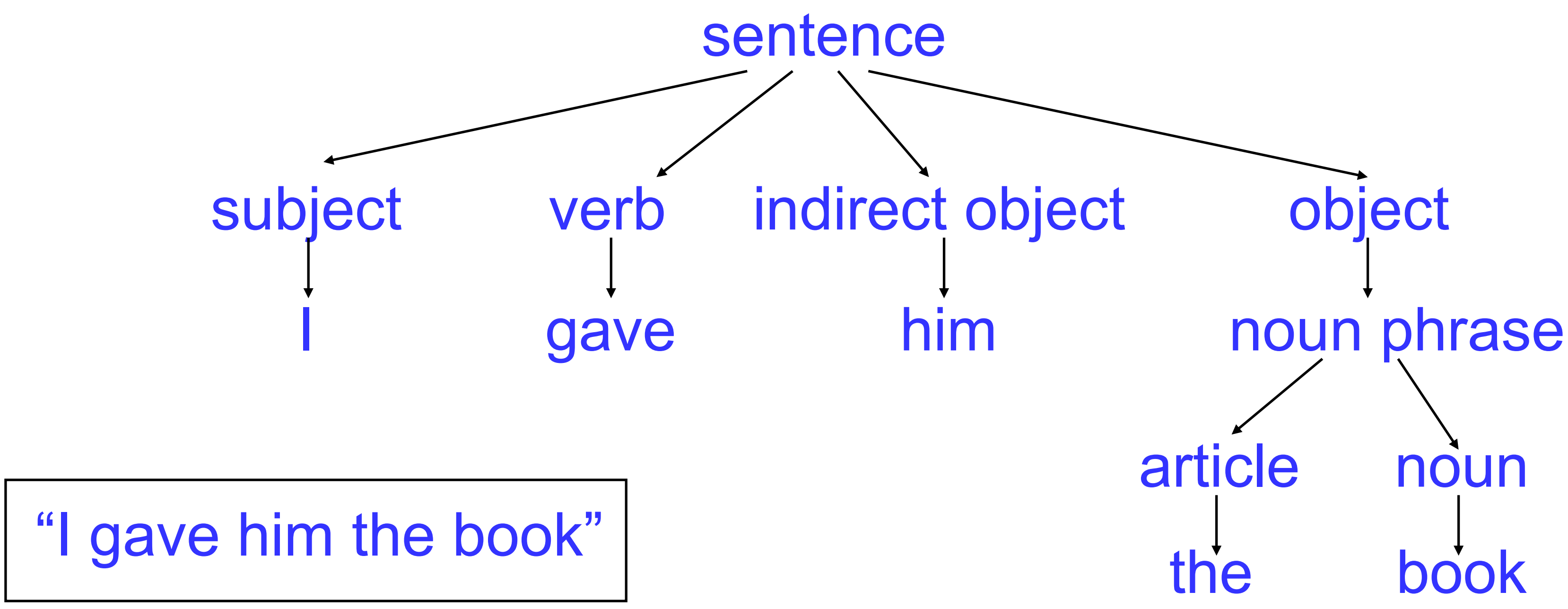
Syntax Analysis

Where is Syntax Analysis Performed?

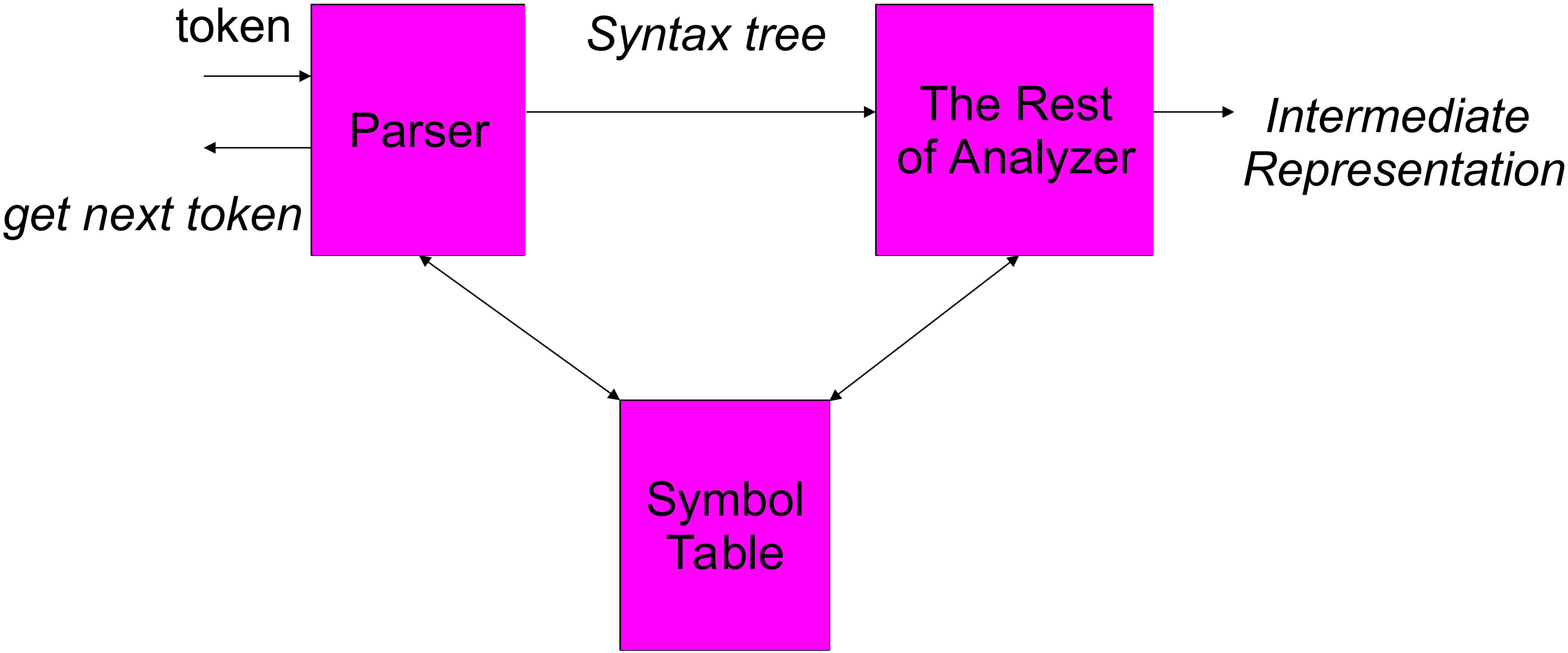


Parsing Analogy

- 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



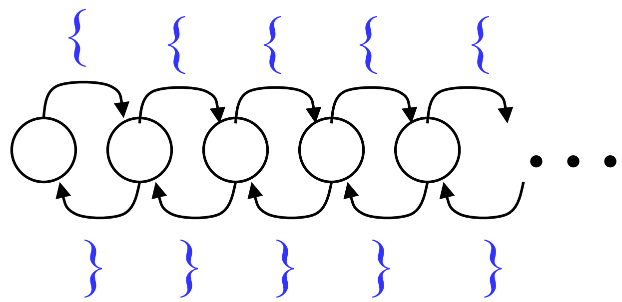
Syntax Analysis Overview

- 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

Just Use Regular Expressions?

- REs can expressively describe tokens
  - Easy to implement via DFAs
- So just use them to describe the syntax of a programming language??
  - NO! – They don't have enough power to express any non-trivial syntax
  - Example – Nested constructs (blocks, expressions, statements) – Detect balanced braces:

{{} {} {{} { }}}



- We need unbounded counting!
- FSAs cannot count except in a strictly modulo fashion

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→RHS
    - LHS = single non-terminal
    - RHS = string of terminals and non-terminals
    - Specify how non-terminals may be expanded

$$
\begin{aligned}
S &\rightarrow a\ S\ a \\
S &\rightarrow T \\
T &\rightarrow b\ T\ b \\
T &\rightarrow \varepsilon
\end{aligned}
$$

- 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

CFG - Example

- Grammar for balanced-parentheses language
  - $S \rightarrow (S) S$
  - $S \rightarrow \varepsilon$
    - 1 non-terminal: S
    - 2 terminals: "(", ")"
    - Start symbol: S
    - 2 productions

Why is the final S required?

- If grammar accepts a string, there is a derivation of that string using the productions
  - "(())"
  - $S \Rightarrow (S)S \Rightarrow (S)\,\varepsilon \Rightarrow ((S)\,S)\,\varepsilon \Rightarrow ((S)\,\varepsilon)\,\varepsilon \Rightarrow ((\varepsilon)\,\varepsilon\,)\,\varepsilon \Rightarrow (())$

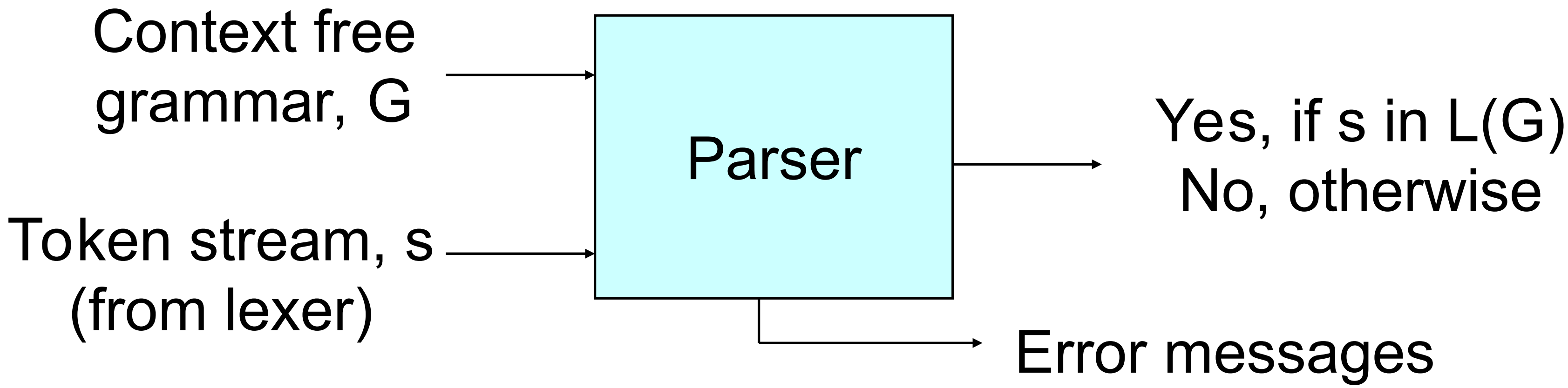
More on CFGs

- Shorthand notation – vertical bar for multiple productions
  - $S \rightarrow a\ S\ a \mid T$
  - $T \rightarrow b\ T\ b \mid \varepsilon$
- CFGs powerful enough to expression the syntax in most programming languages
- Derivation = successive application of productions starting from S
- Acceptance? = Determine if there is <u>a derivation</u> 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 → ε
a	S → a
R1 R2	S → S1 S2
R1 | R2	S → S1 | S2
R1*	S → S1 S | ε

Where

G1 = grammar for R1, with start symbol S1

G2 = grammar for R2, with start symbol S2

Grammar for Sum Expression

- Grammar
  - S → E + S | E
  - E → number | (S)
- Expanded
  - S → E + S
  - S → E
  - E → number
  - E → (S)

4 productions
2 non-terminals (S,E)
4 terminals: “(“, “)”, “+”, number
start symbol: S

Constructing a Derivation

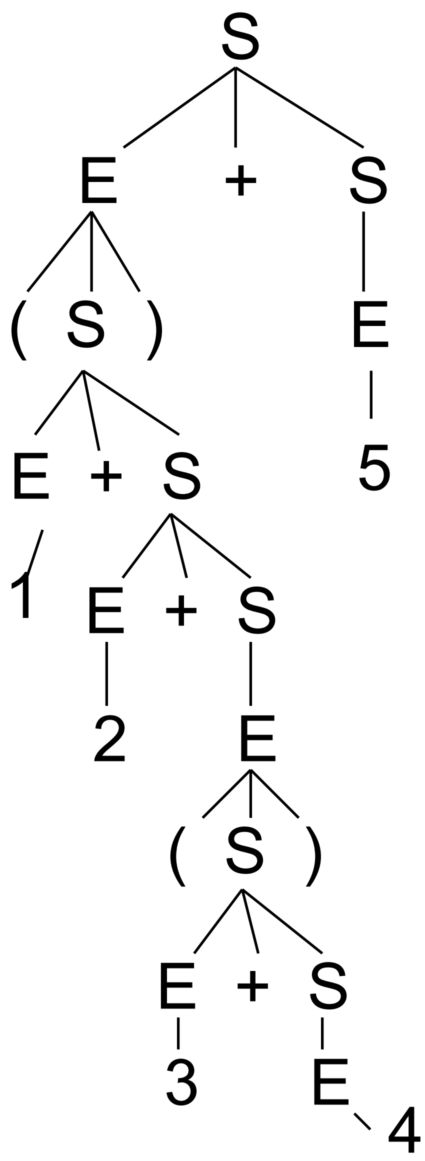
- Start from S (the start symbol)
- Use productions to derive a sequence of tokens
- For arbitrary strings α, β, γ and for a production: A → β
  - A single step of the derivation is
  - α A γ => α β γ   (substitute β for A)
- Example
  - S → E + S
  - $(\underline{S} + E) + E \Rightarrow (\underline{E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

Class Problem

- S → E + S | E
- E → number | (S)

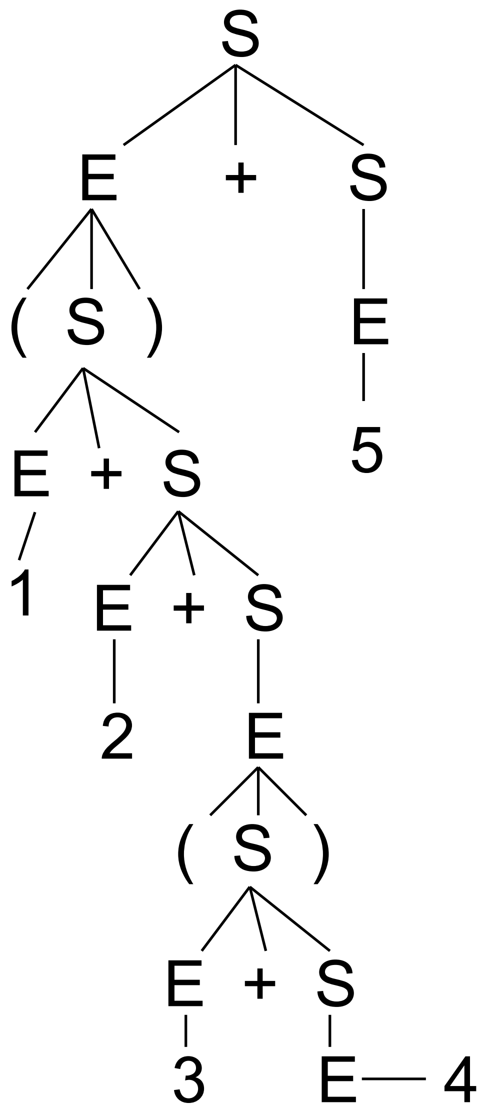
- Derive: (1 + 2 + (3 + 4)) + 5

Parse Tree

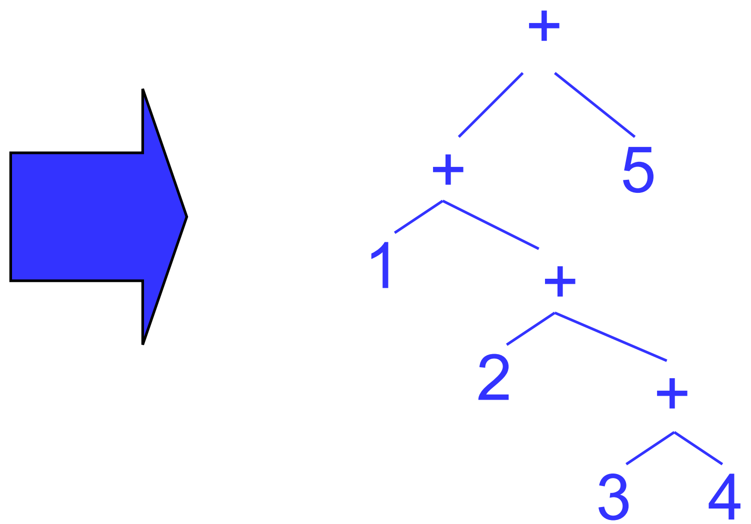


- Parse tree = tree representation of the derivation
- Leaves of the tree are terminals
- Internal nodes are non-terminals
- No information about the order of the derivation steps

Parse Tree vs Abstract Syntax Tree



Parse tree also called “concrete syntax”



AST discards (abstracts) unneeded information – more compact format

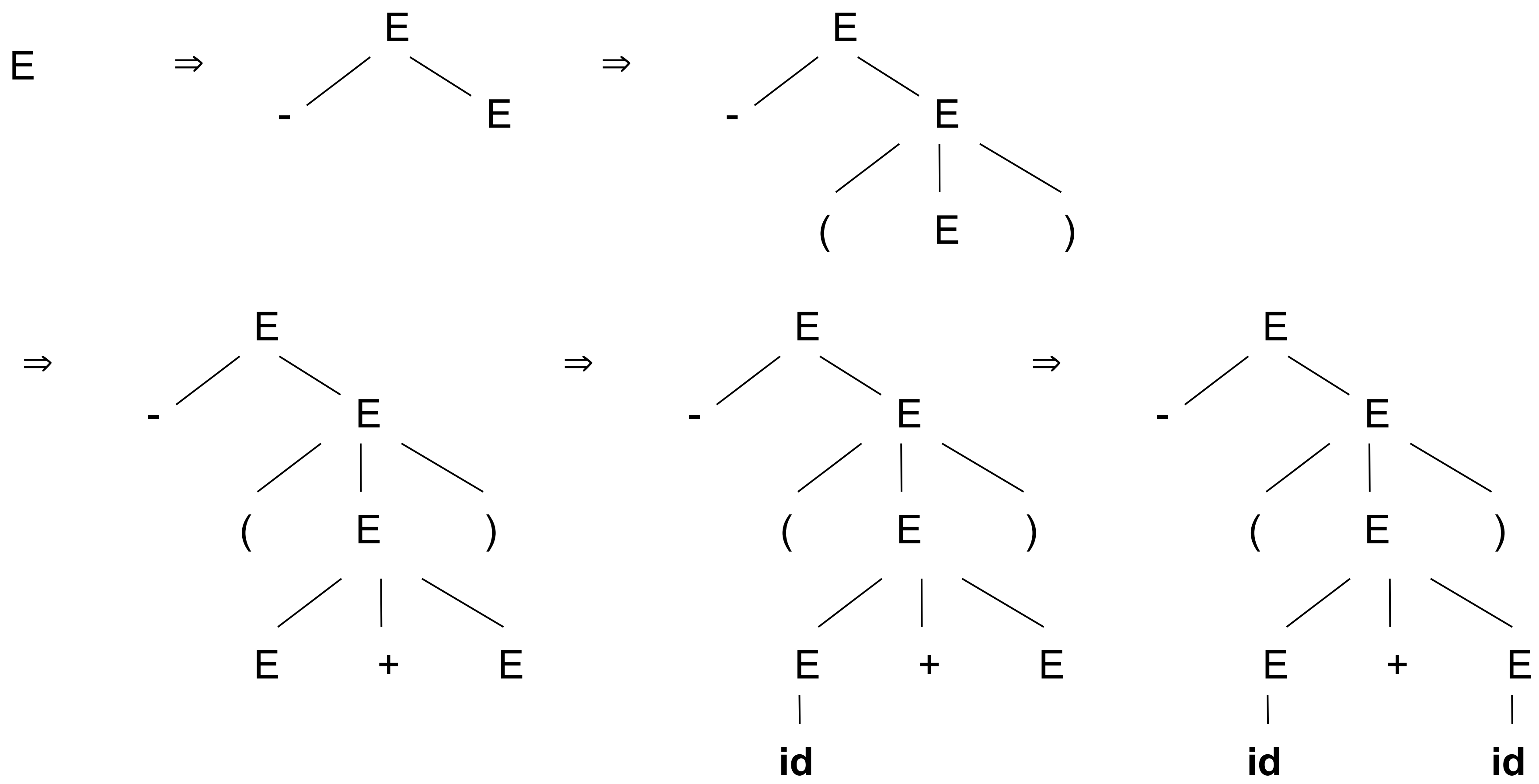
Derivation Order

- 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
  - $E + S \underset{lm}{\Rightarrow} 1 + S$
- Rightmost derivation
  - Same, but find rightmost non-terminal
  - $E + S \underset{rm}{\Rightarrow} E + E + S$

Leftmost Derivation Example

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

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$



Leftmost/Rightmost Derivation Examples

- S → E + S | E
- E → number | (S)
- Leftmost derive: (1 + 2 + (3 + 4)) + 5

S => E + S => (S)+S => (E+S) + S => (1+S)+S => (1+E+S)+S =>
(1+2+S)+S => (1+2+E)+S => (1+2+(S))+S => (1+2+(E+S))+S =>
(1+2+(3+S))+S => (1+2+(3+E))+S => (1+2+(3+4))+S =>
(1+2+(3+4))+E => (1+2+(3+4))+5

- Now, rightmost derive the same input string

S => E+S => E+E => E+5 => (S)+5 => (E+S)+5 => (E+E+S)+5 =>
(E+E+E)+5 => (E+E+(S))+5 => (E+E+(E+S))+5 =>
(E+E+(E+E))+5 => (E+E+(E+4))+5 => (E+E+(3+4))+5 =>
(E+2+(3+4))+5 => (1+2+(3+4))+5

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

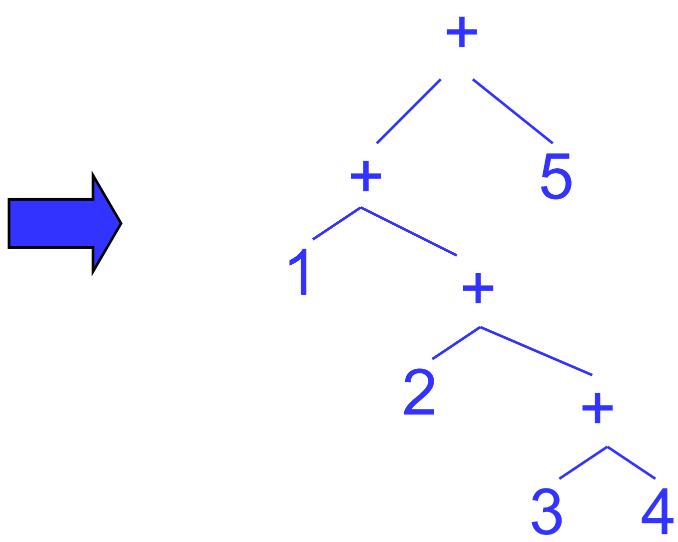
Class Problem

  - $S \rightarrow E + S \mid E$
  - $E \rightarrow \text{number} \mid (S) \mid -S$
- 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 → E + S
- Consider another grammar
  - S → S + S | S * S | 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

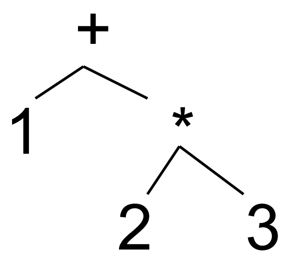
S → S + S | S * S | number

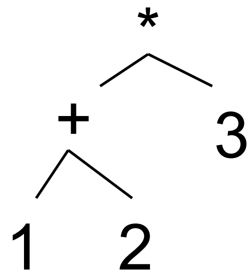
Consider the expression: 1 + 2 * 3

Derivation 1: S => S+S =>
1+S => 1+S*S => 1+2*S => 1+2*3

Derivation 2: S => S*S =>
S+S*S => 1+S*S => 1+2*S => 1+2*3

2 leftmost derivations

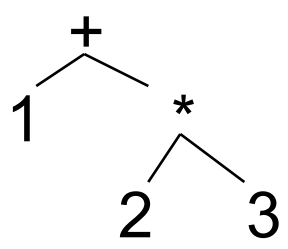




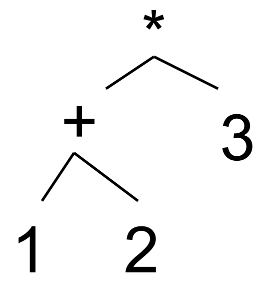
But, obviously not equal!

Impact of Ambiguity

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



= 7



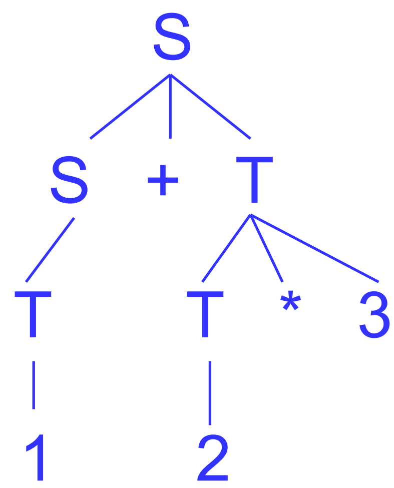
= 9

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
  - $S \rightarrow S + T \mid T$
  - $T \rightarrow T * num \mid num$



  - 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 → left (right) associativity
  - Non: Don’t be recursive, simply reference next higher precedence non-terminal on both sides of operator

Class Problem

S → S + S | S – S | S * S | S / S | (S) | -S | S ^ S | num

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

<u>Precedence (high to low)</u>

(), unary –

^

*, /

+, -

<u>Associativity</u>

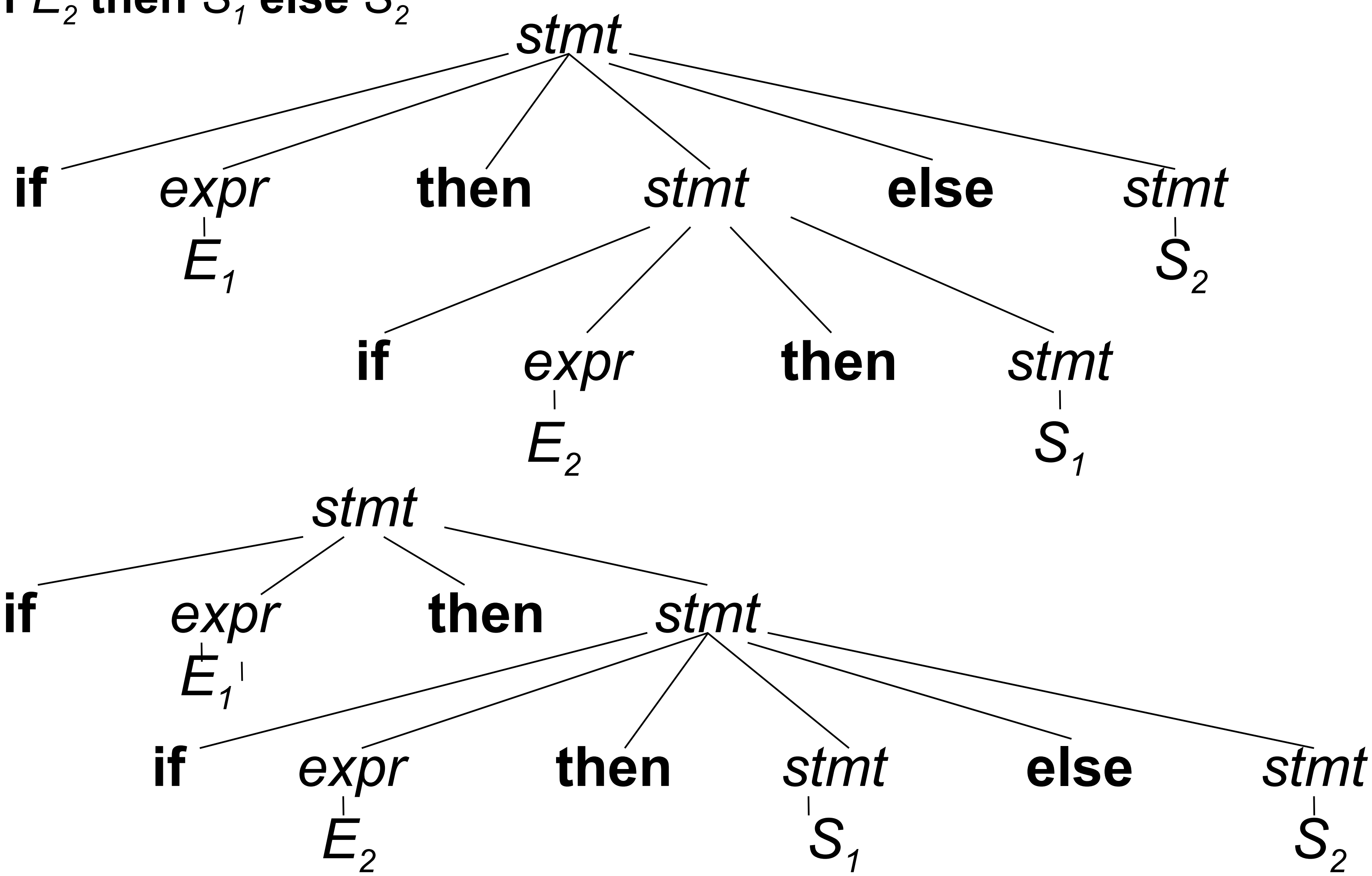
^ = right

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) <u>if (E) S</u> 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 → E + S | E
E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ E + S	(	(1+2+(3+4))+5
→ (S) + S	1	(1+2+(3+4))+5
→ (E+S)+S	1	(1+2+(3+4))+5
→ (1+S)+S	2	(1+2+(3+4))+5
→ (1+E+S)+S	2	(1+2+(3+4))+5
→ (1+2+S)+S	2	(1+2+(3+4))+5
→ (1+2+E)+S	(	(1+2+(3+4))+5
→ (1+2+(S))+S	3	(1+2+(3+4))+5
→ (1+2+(E+S))+S	3	(1+2+(3+4))+5
→ …		

Problem with Top-Down Parsing

Want to decide which production to apply based on next symbol

S → E + S | E
E → num | (S)

Ex1: "(1)" S => E => (S) => (E) => (1)
Ex2: "(1)+2" S => <u>E+S</u> => (S)+S => (E)+S => (1)+E => (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

$S \rightarrow E + S \mid E$
$E \rightarrow num \mid (S)$

- 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)

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

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

- 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 → ES'   S' → ε | +S   E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ES'	(	(1+2+(3+4))+5
→(S)S'	1	(1+2+(3+4))+5
→(ES')S'	1	(1+2+(3+4))+5
→(1S')S'	+	(1+2+(3+4))+5
→(1+ES')S'	2	(1+2+(3+4))+5
→(1+2S')S'	+	(1+2+(3+4))+5
→(1+2+S)S'	(	(1+2+(3+4))+5
→(1+2+ES')S'	(	(1+2+(3+4))+5
→(1+2+(S)S')S'	3	(1+2+(3+4))+5
→(1+2+(ES')S')S'	3	(1+2+(3+4))+5
→(1+2+(3S')S')S'	+	(1+2+(3+4))+5
→(1+2+(3+E)S')S'	4	(1+2+(3+4))+5
→ ...		

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 → productions

Adaptation for Predictive Parsing

- Elimination of left recursion
  $expr \rightarrow expr + term \mid term$

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

  $A \rightarrow \beta R$
  $R \rightarrow \alpha R \mid \in$

- Left factoring
  $stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt$
  $\quad\quad \mid \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$

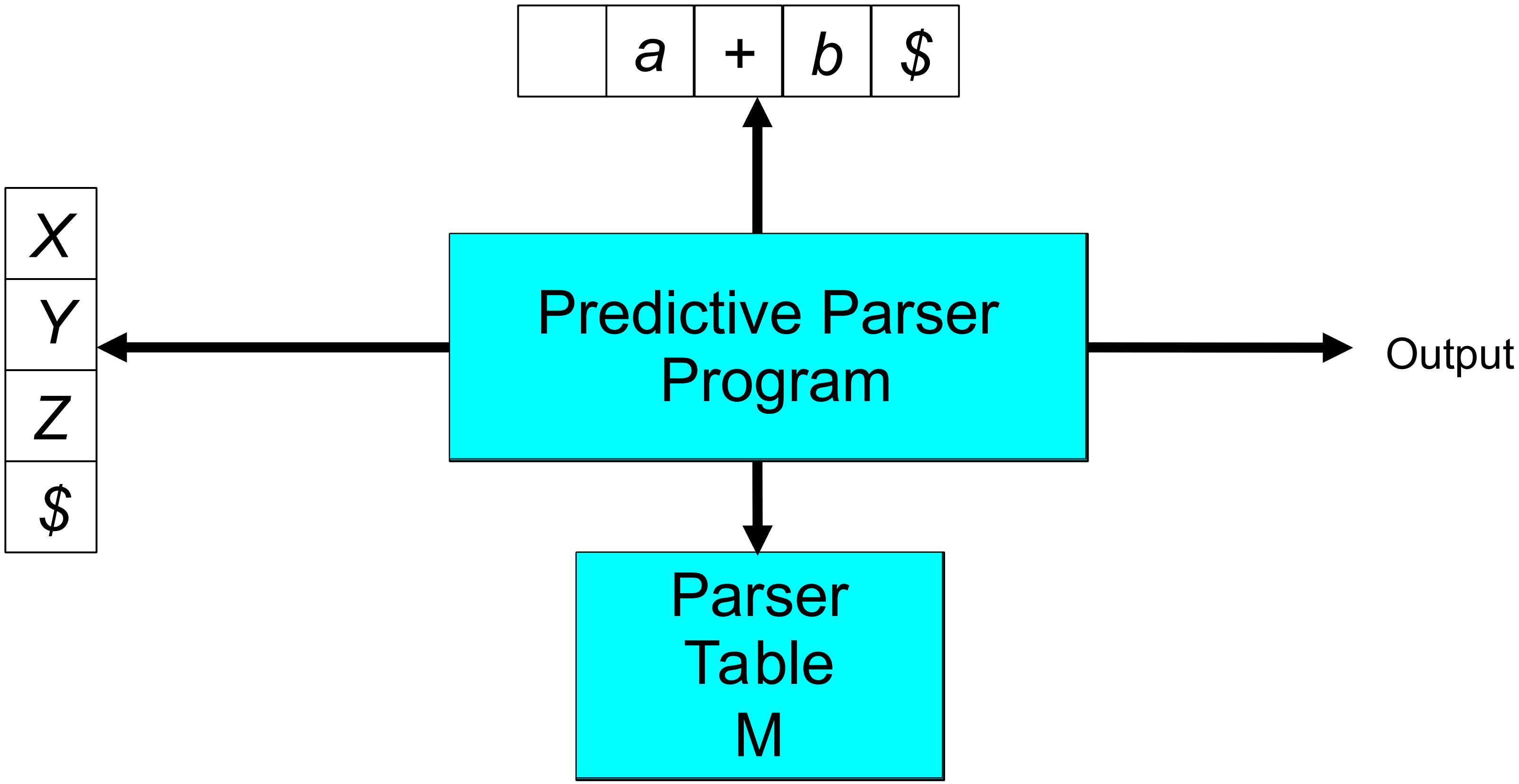
  $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$
  $A \rightarrow \alpha\ A'$
  $A' \rightarrow \beta_1 \mid \beta_2$

Transformation for Arithmetic Expression Grammar

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathbf{id}$$

$$E \to TE'$$
$$E' \to +TE' \mid \in$$
$$T \to FT'$$
$$T' \to *FT' \mid \in$$
$$F \to (E) \mid \mathbf{id}$$

Predictive Parser without Recursion



1. If X=a=$ stop and announce success
2. If X=a<>$ 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

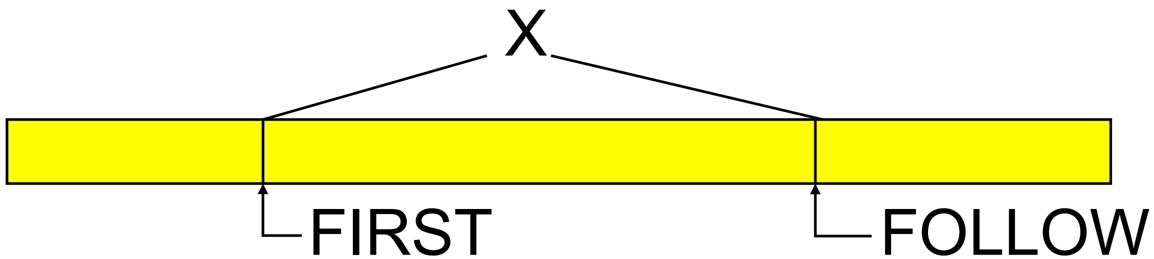
Nonterminal	Input Symbol					
	Id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \in$	$T' \rightarrow *FT'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F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

Class Problem

- Parse the string
  - id + id * 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(β) 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, FIRST(X) = {X}

2. If $X \rightarrow \in$ is a production,
   add $\in$ to FIRST(X)

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

Computation of FOLLOW(X)

1. Place \$ in FOLLOW(S), where S is the start symbol

2. If there is a production
   $A \to \alpha B \beta$,
   everything in FIRST(β) except for $\in$ is placed in FOLLOW(B)

3. If there is a production $A \to \alpha B$
   or a production $A \to \alpha B \beta$ where FIRST(β) contains $\in$, 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 FIRST(α) add $A \rightarrow \alpha$ to M[A,a]

3. If FIRST(α) contains $\in$, place $A \rightarrow \alpha$ in M[A,b] for each b in 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'		$E' \rightarrow +TE'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \in$	$T' \rightarrow *FT'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow$ **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
  - Start with the tokens
  - End with the start symbol
  - Match substring on RHS of production, replace by LHS

$S \rightarrow S + E \mid E$
$E \rightarrow num \mid (S)$

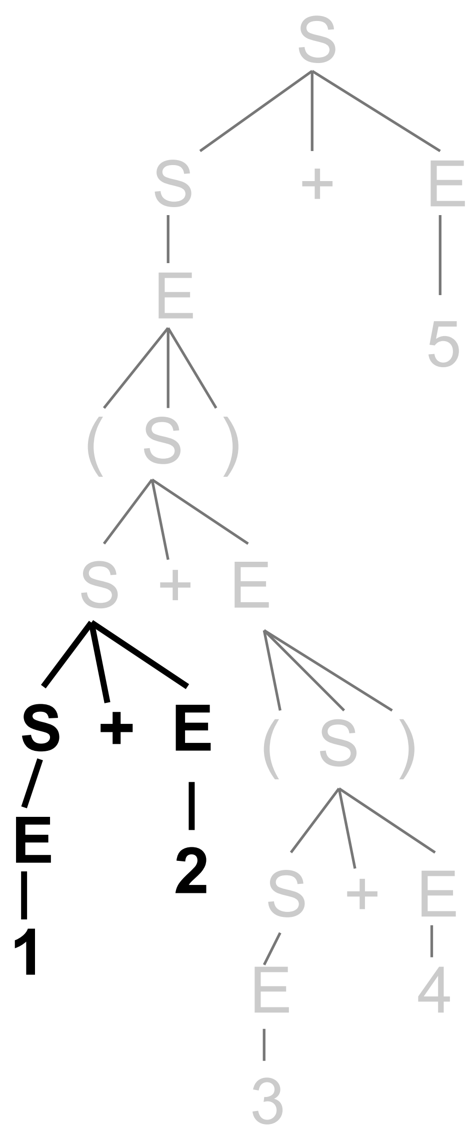
(1+2+(3+4))+5 <= (E+2+(3+4))+5 <= (S+2+(3+4))+5 <= (S+E+(3+4))+5 <= (S+(3+4))+5 <= (S+(E+4))+5 <= (S+(S+4))+5 <= (S+(S+E))+5 <= (S+(S))+5 <= (S+E)+5 <= (S)+5 <= E+5 <= S+5 <= S+E <= S

Bottom-Up Parsing

S → S + E | E
E → num | (S)

(1+2+(3+4))+5
 <= (E+2+(3+4))+5
<= (S+2+(3+4))+5
<= (S+E+(3+4))+5

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

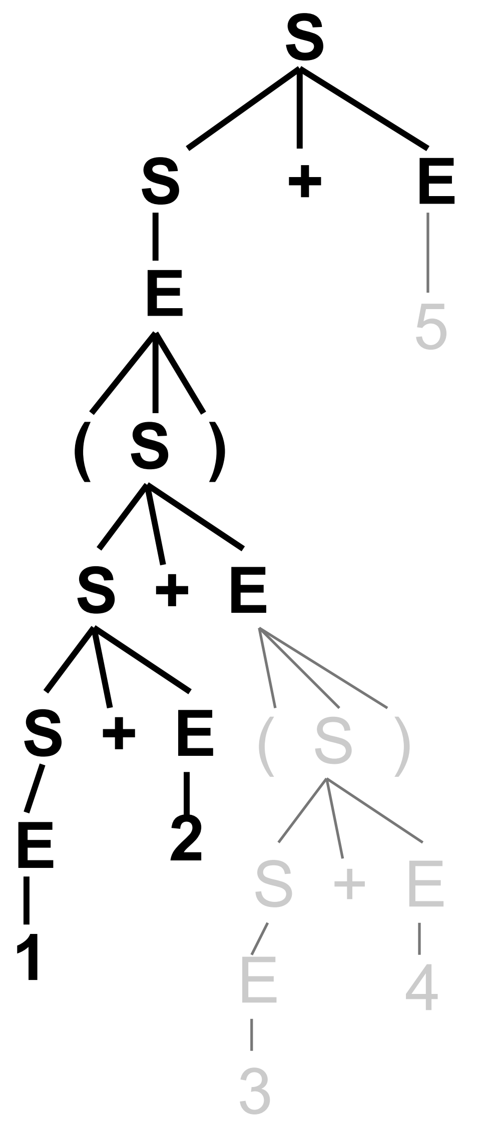


Top-Down Parsing

$S \rightarrow S + E \mid E$
$E \rightarrow num \mid (S)$

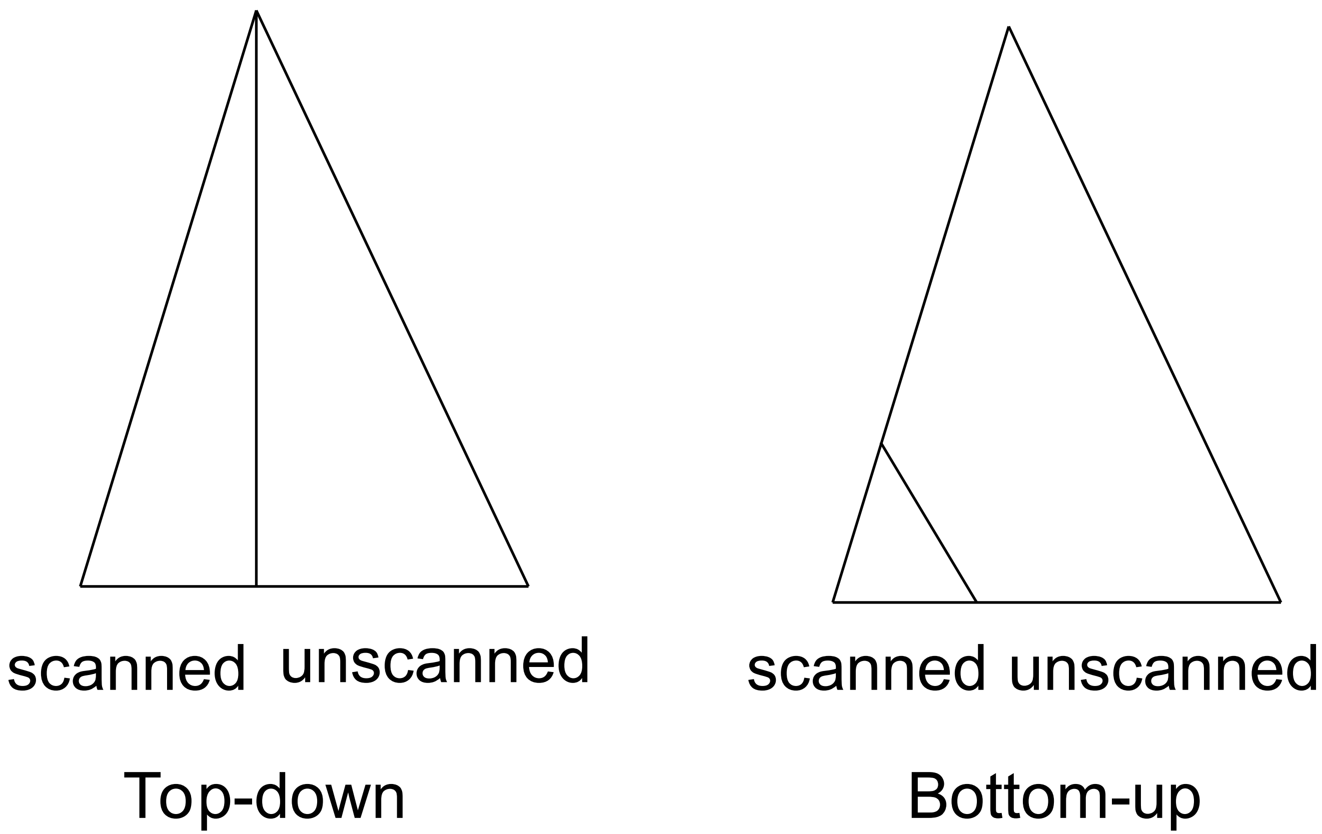
S => S+E => E+E => (S)+E => (S+E)+E=> (S+E+E)+E => (E+E+E)+E=> (1+E+E) +E => (1+2+E)+E ...

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



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 size of the production represents one step along the reverse of a rightmost derivation

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

$$
\begin{aligned}
E &\Rightarrow \underline{E + E} \\
&\Rightarrow E + \underline{E * E} \\
&\Rightarrow E + E * \underline{\mathbf{id}_3} \\
&\Rightarrow E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
&\Rightarrow \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

$$
\begin{aligned}
E &\Rightarrow \underline{E * E} \\
&\Rightarrow E * \underline{\mathbf{id}_3} \\
&\Rightarrow \underline{E + E} * \mathbf{id}_3 \\
&\Rightarrow E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
&\Rightarrow \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{aligned}
$$

Handles

Right-Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \rightarrow \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \rightarrow \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \rightarrow \mathbf{id}$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
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→ β (e.g., pop β, push X)

Stack	Input	Operation
\$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3\$$	Shift
$\$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	Reduce by $E \to \mathbf{id}$
$\$E$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	Shift
$\$E +$	$\mathbf{id}_2 * \mathbf{id}_3\$$	Shift
$\$E + \mathbf{id}_2$	$* \mathbf{id}_3\$$	Reduce by $E \to \mathbf{id}$
$\$E + E$	$* \mathbf{id}_3\$$	Shift
$\$E + E *$	$\mathbf{id}_3\$$	Shift
$\$E + E * \mathbf{id}_3$	\$	Reduce by $E \to \mathbf{id}$
$\$E + E * E$	\$	Reduce by $E \to E * E$
$\$E + E$	\$	Reduce by $E \to E + E$
$\$E$	\$	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 → γ assuming that the stack has the form $\beta = \alpha\gamma$ making it αX ?
- If stack has the form $\alpha\gamma$, should apply reduction X → γ (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 overriden.

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

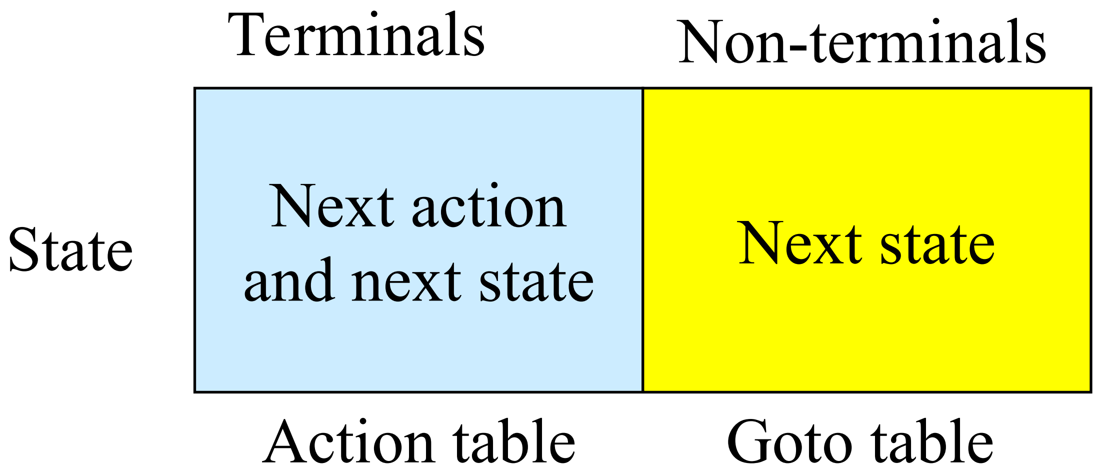
```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}

%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



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

LR Parsing Table Example

We want to derive this in an algorithmic fashion

	Action					Goto	
State	(	)	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)

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

derivation	stack	input	action
((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

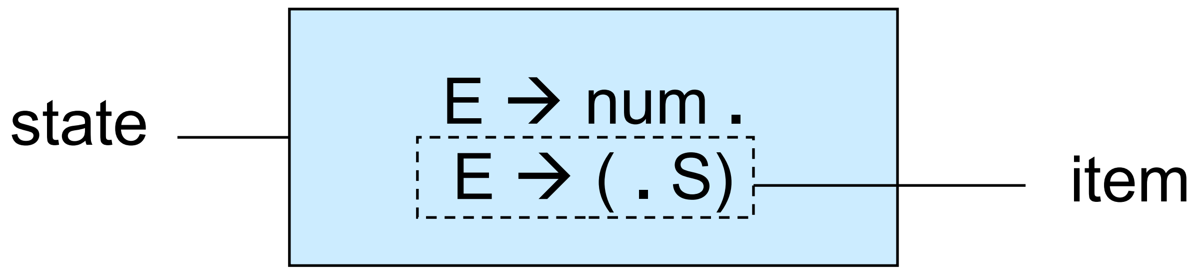
- LR(k) = Left-to-right scanning, right-most derivation, k lookahead chars
- Main cases
  - LR(0), LR(1)
  - Some variations SLR and LALR(1)
- Parsers for LR(0) Grammars:
  - Determine the actions <u>without any lookahead</u>
  - Will help us understand shift-reduce parsing

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

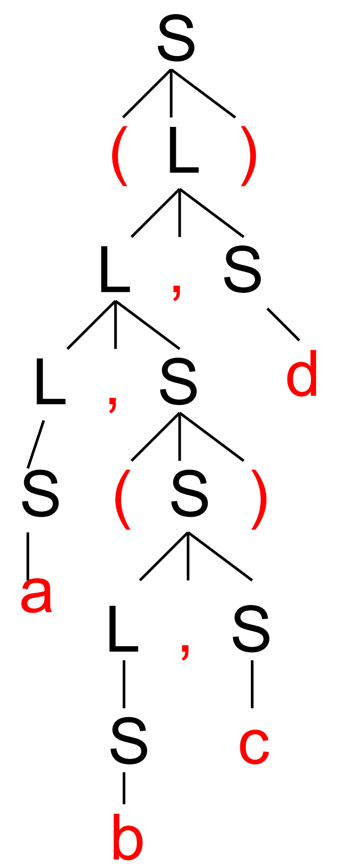
Class Problem

- For the production,
  - E → num | (S)
- Two items are:
  - E → num .
  - E → (. S)
- Are there any others?
  - If so, what are they?
  - If not, why?

LR(0) Grammar

- Nested lists
  - S → (L) | id
  - L → S | 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 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

$S' \rightarrow . S \$$

closure

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

- 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, '(')

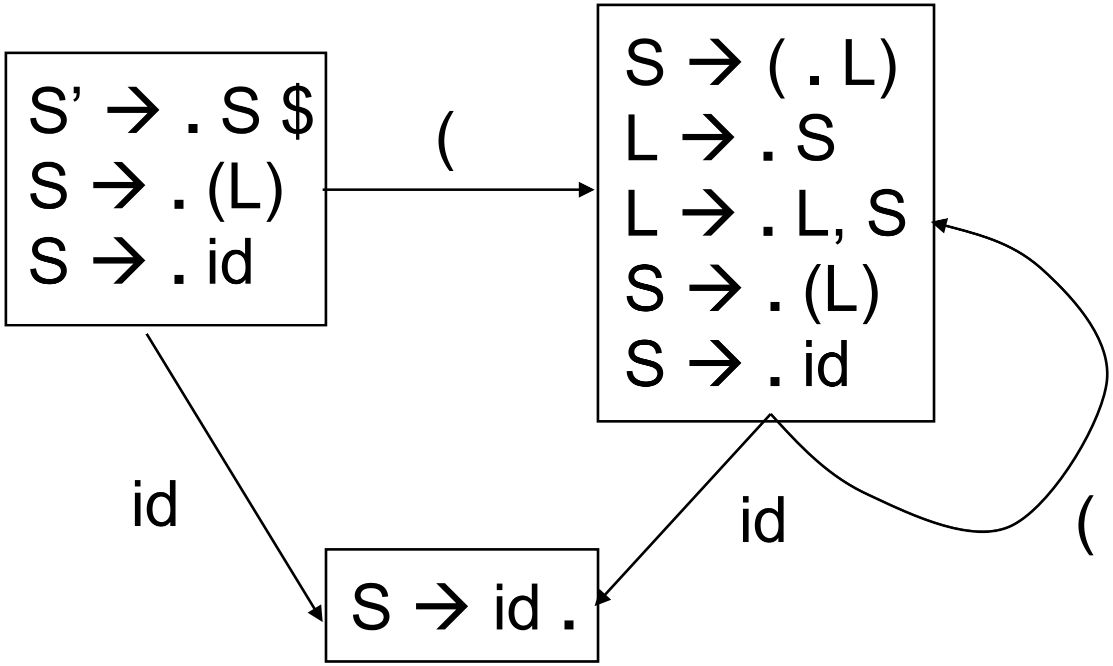
$\text{Closure}(\, [\, S \rightarrow (\, .\, L)\,]\,)$

Class Problem

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

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

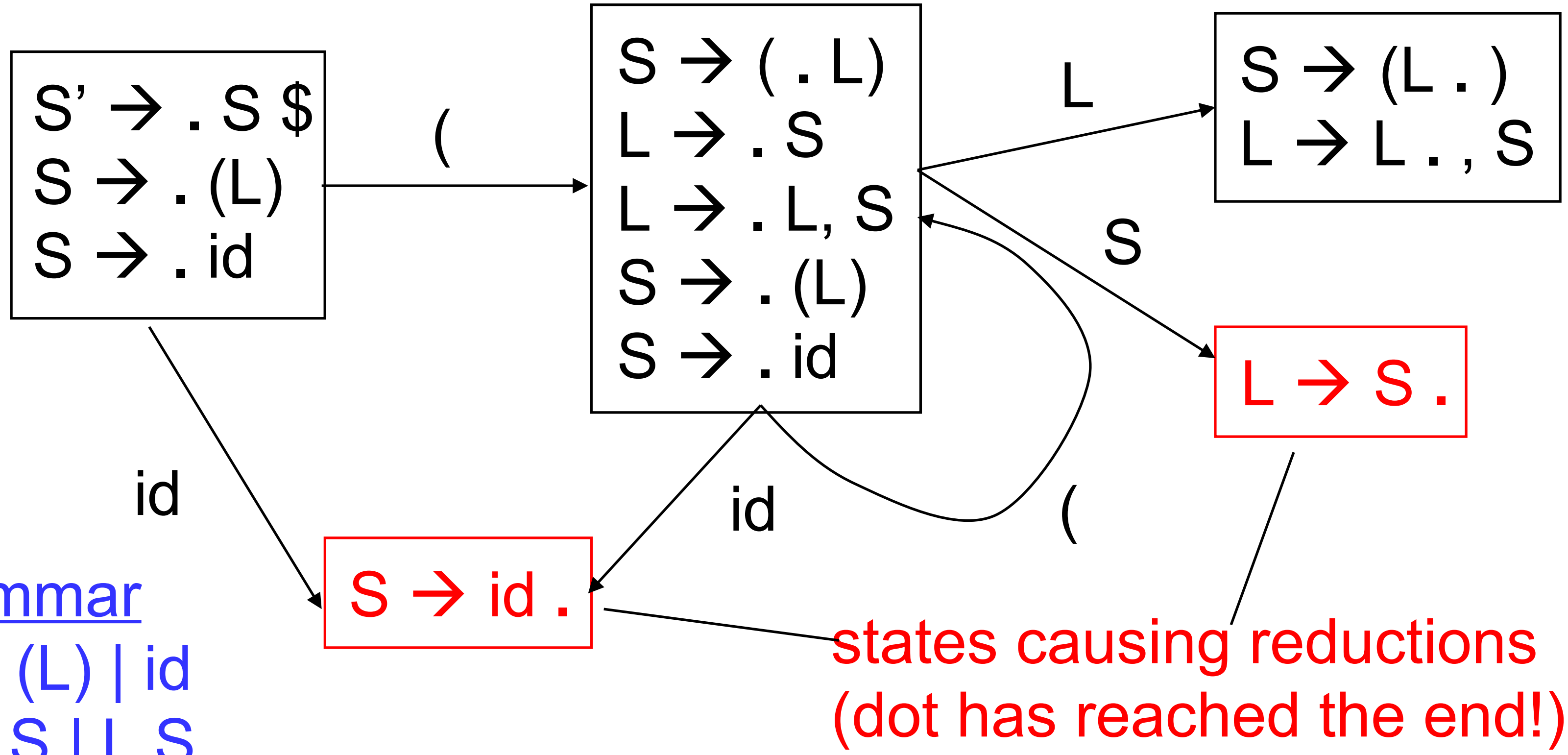
Goto: Terminal Symbols



Grammar
S → (L) | id
L → S | L,S

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



Grammar
S → (L) | id
L → S | L,S

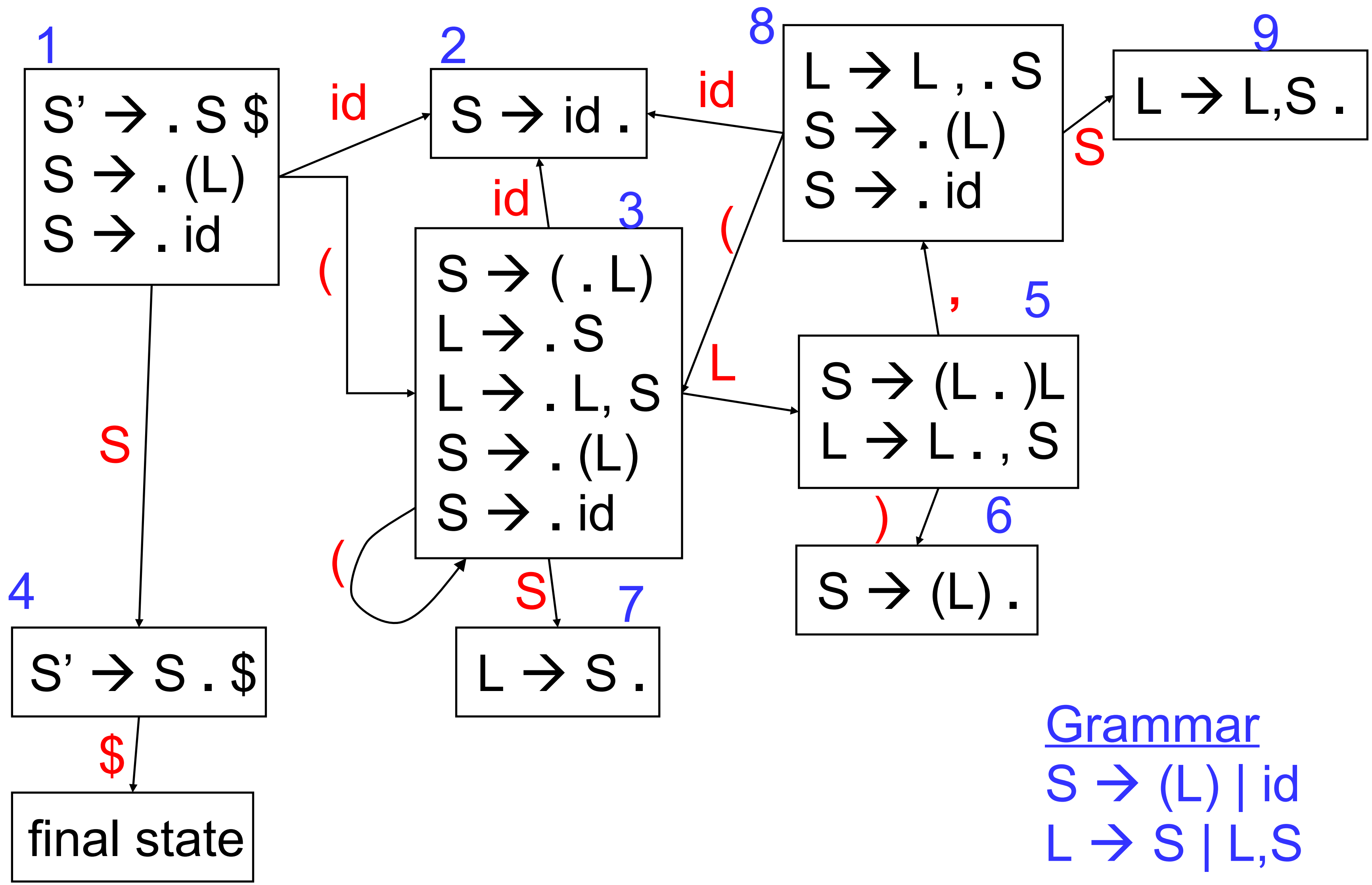
Pop RHS off stack, replace with LHS X (X → β),
then rerun DFA

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) <=	1 (3 (3	a),b)	shift, goto 2
((a),b) <=	1 (3 (3 a 2	),b)	reduce $S \rightarrow id$
((S),b) <=	1 (3 (3 S 7	),b)	reduce $L \rightarrow S$

Full DFA


1
S' → . S $
S → . (L)
S → . id
2
S → id .
3
S → (. L)
L → . S
L → . L, S
S → . (L)
S → . id
4
S' → S . $
final state
5
S → (L .)L
L → L . , S
6
S → (L) .
7
L → S .
8
L → L , . S
S → . (L)
S → . id
9
L → L,S .
id
id
id
(
(
(
S
S
S
L
,
)
$
Grammar
S → (L) | id
L → S | L,S

Building the Parsing Table

- States in the table = states in the DFA
- For transition S → S’ on terminal C:
  - Action[S,C] += Shift(S’)
- For transition S → S’ on non-terminal N:
  - Goto[S,N] += Goto(S’)
- If S is a reduction state X → β then:
  - Action[S,*] += Reduce(X → β)

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 → E + S | E
  - E → num

LR(0) Limitations

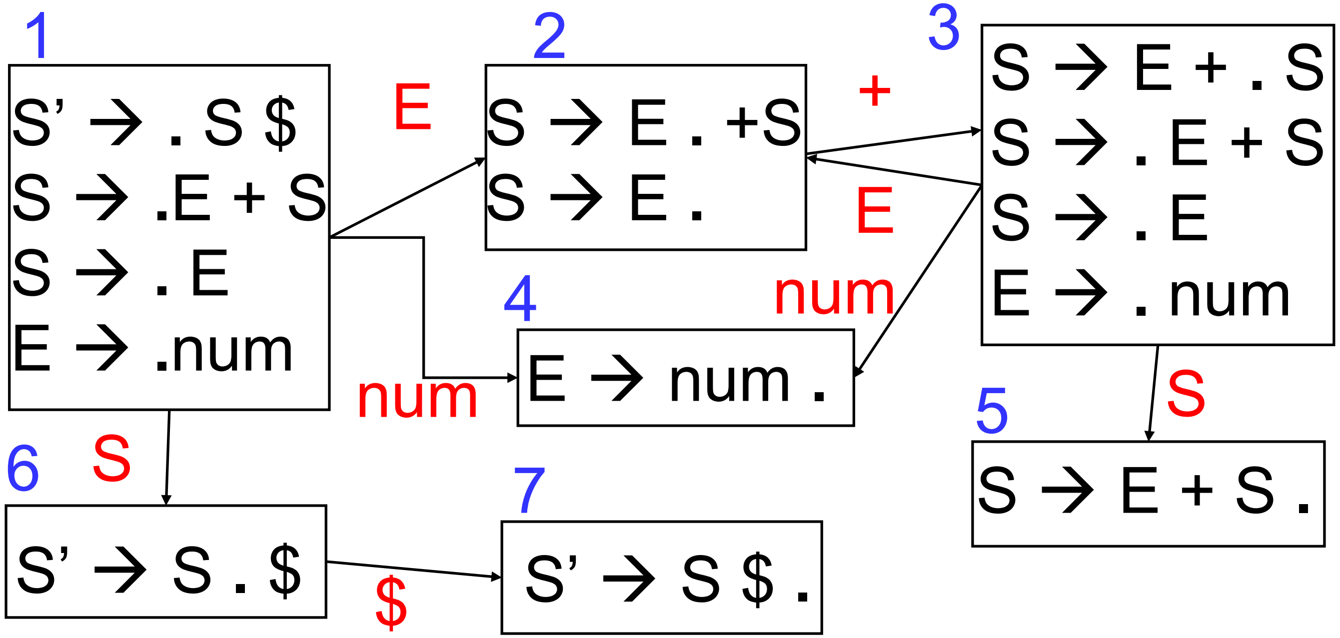
- An LR(0) machine only works if states with reduce actions have a single reduce action
  - Always reduce regardless of lookahead
- With a more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use lookahead to choose

OK	shift/reduce	reduce/reduce
$L \rightarrow L , S .$	$L \rightarrow L , S .$ $S \rightarrow S . , L$	$L \rightarrow S , L .$ $L \rightarrow S .$

A Non-LR(0) Grammar

- Grammar for addition of numbers
  - S → S + E | E
  - E → num
- Left-associative version is LR(0)
- Right-associative is not LR(0) as you saw with the previous class problem
  - S → E + S | E
  - E → num

LR(0) Parsing Table



Grammar
S → E + S | E
E → num

Shift or reduce in state 2?

	num	+	$	E	S
1	s4			g2	g6
2	S→E	s3/S→E	S→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 → E + S | E
E → num

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		

SLR Parsing Table

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

Grammar

S → E + S | E

E → num

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		
3	s4			g2	g5
4		E→num	E→num		
5			S→E+S		
6			s7		
7			accept		

Class Problem

- Consider:
  - S → L = R
  - S → R
  - L → *R
  - L → ident
  - R → 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 → L . = R R → L .

S → 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 = $(X \rightarrow \alpha \,.\, \beta \,,\, y)$
  - Meaning: α already matched at top of the stack, next expect to see β y
- Shorthand notation
  - $(X \rightarrow \alpha \,.\, \beta \,,\, \{x1, ..., xn\})$
  - means:
    - $(X \rightarrow \alpha \,.\, \beta \,,\, x1)$
    - . . .
    - $(X \rightarrow \alpha \,.\, \beta \,,\, xn)$
- Need to extend closure and goto operations

$S \rightarrow S \,.\, + E \quad +,\$$
$S \rightarrow S + . \, E \quad num$

LR(1) Closure

- LR(1) closure operation:
  - Start with 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: $Y \rightarrow . \, \gamma$, FIRST(βz)
  - 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' → . S , $), then apply closure operation
- Example: sum grammar

S' → S $
S → E + S | E
E → num

S' → . S , $

closure ⇒

S' → . S , $
S → . E + S , $
S → . E , $
E → . 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
  - Goto(S, Y) = Closure($[X \rightarrow \alpha \, Y \, . \, \beta]$)

S1

$S \rightarrow E \, . + S \, , \$$

$S \rightarrow E \, . \, , \$$

Goto(S1, '+')

S2

Closure($\{S \rightarrow E + . \, S \, , \$\}$)

Grammar:

$S' \rightarrow S\$$

$S \rightarrow E + S \mid E$

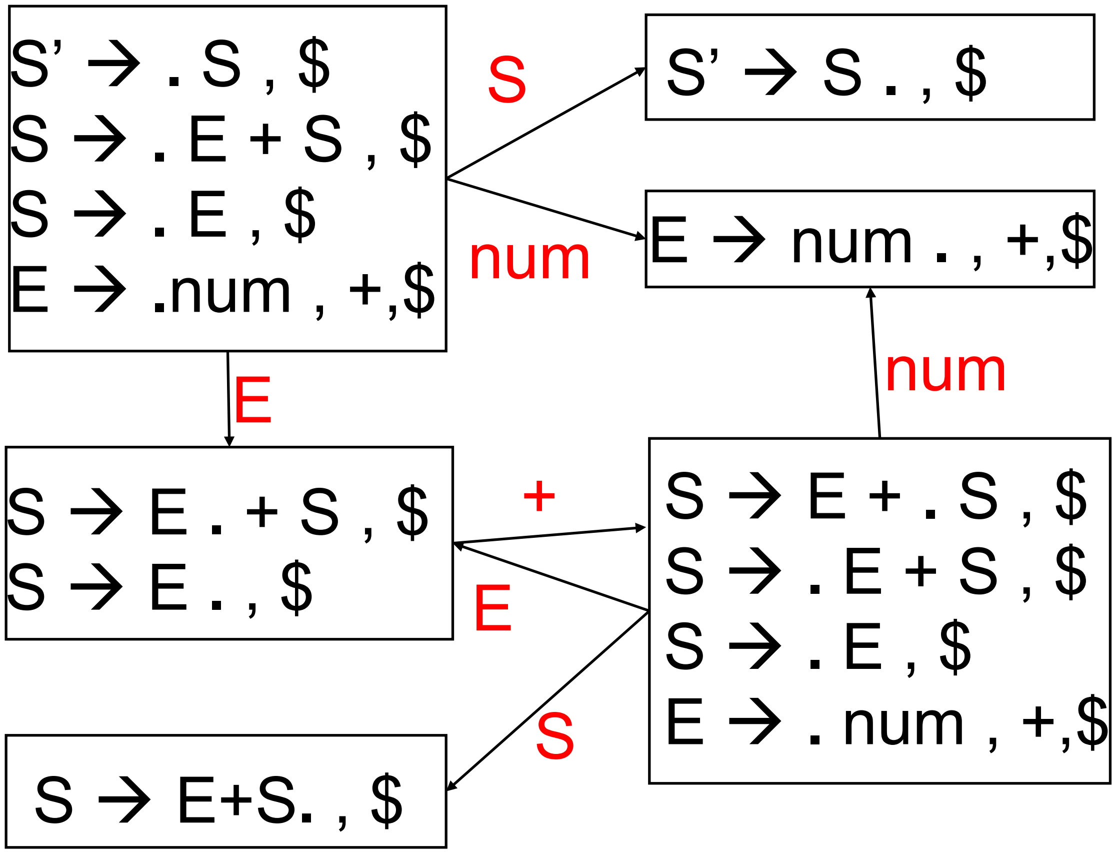
$E \rightarrow num$

Class Problem

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

S' → S $
S → E + S | E
E → num

LR(1) DFA Construction



<u>Grammar</u>

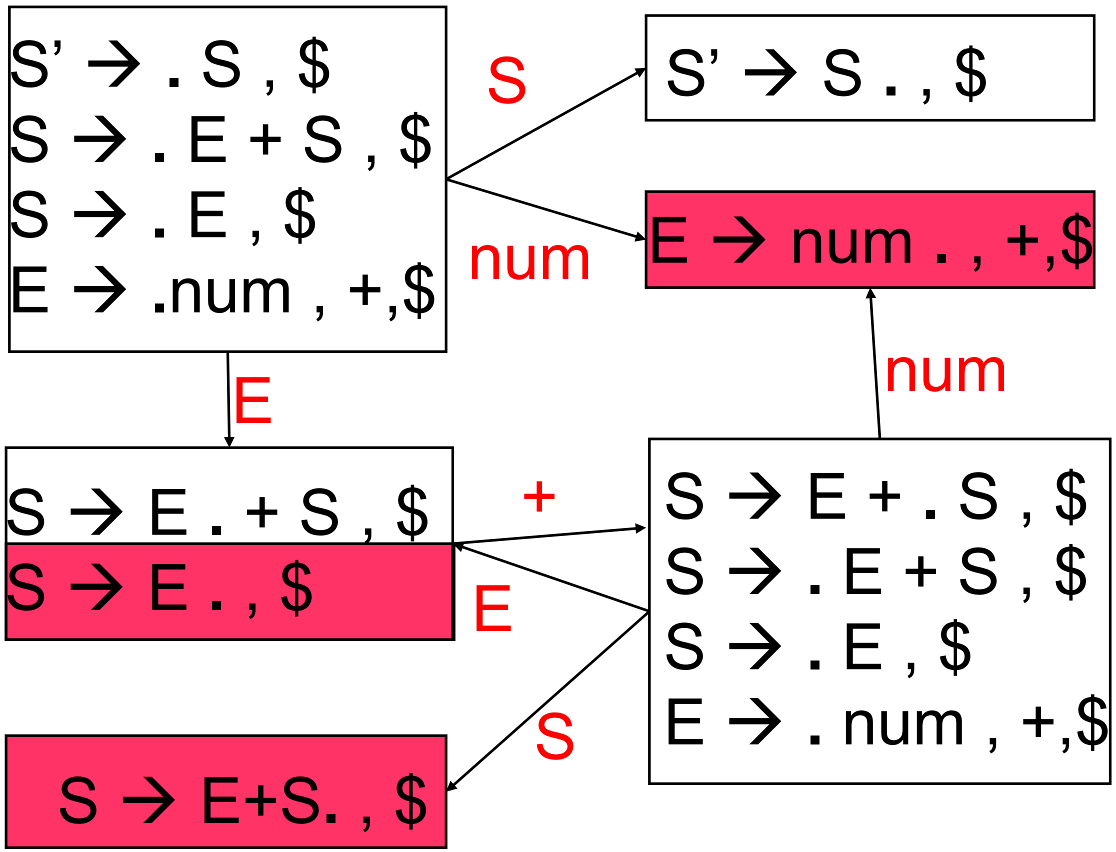
S’ → S$

S → E + S | E

E → num

LR(1) Reductions

- Reductions correspond to LR(1) items of the form (X → γ . , y)

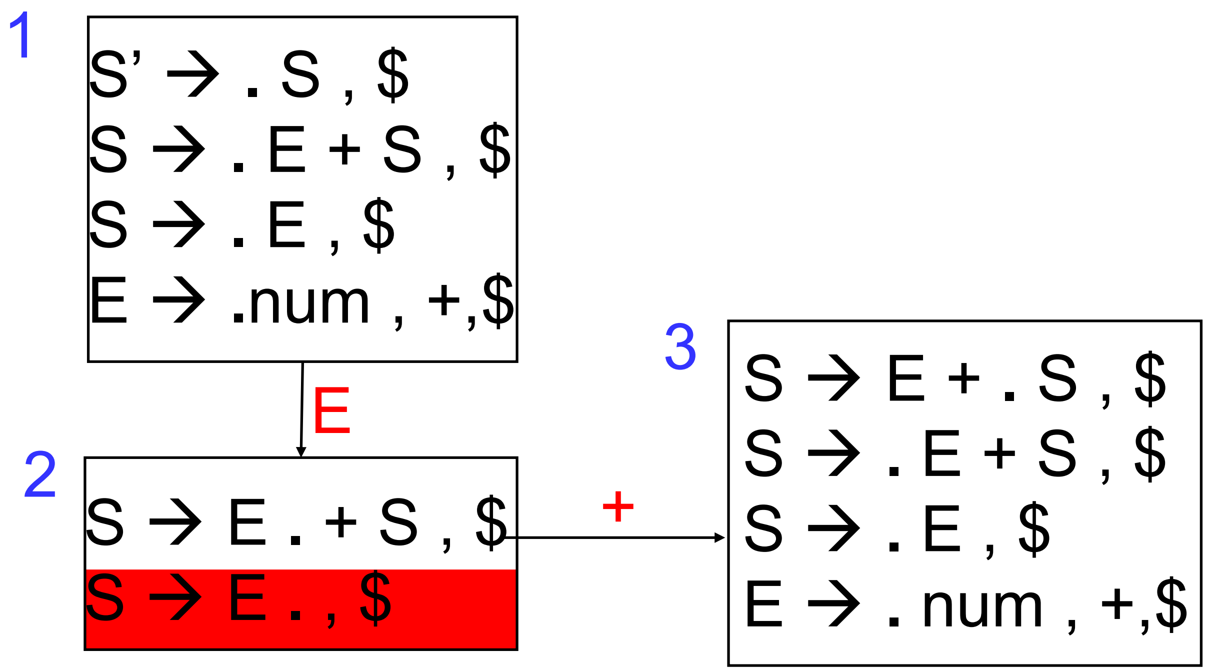


Grammar
S' → S$
S → E + S | E
E → num

LR(1) Parsing Table Construction

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

LR(1) Parsing Table Example



Grammar

S' → S$

S → E + S | E

E → num

Fragment of the parsing table

	+	$	E
1			g2
2	s3	S→E	

Class Problem

- Compute the LR(1) DFA for the following grammar
  - E → E + T | T
  - T → TF | F
  - F → F* | a | 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)

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

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

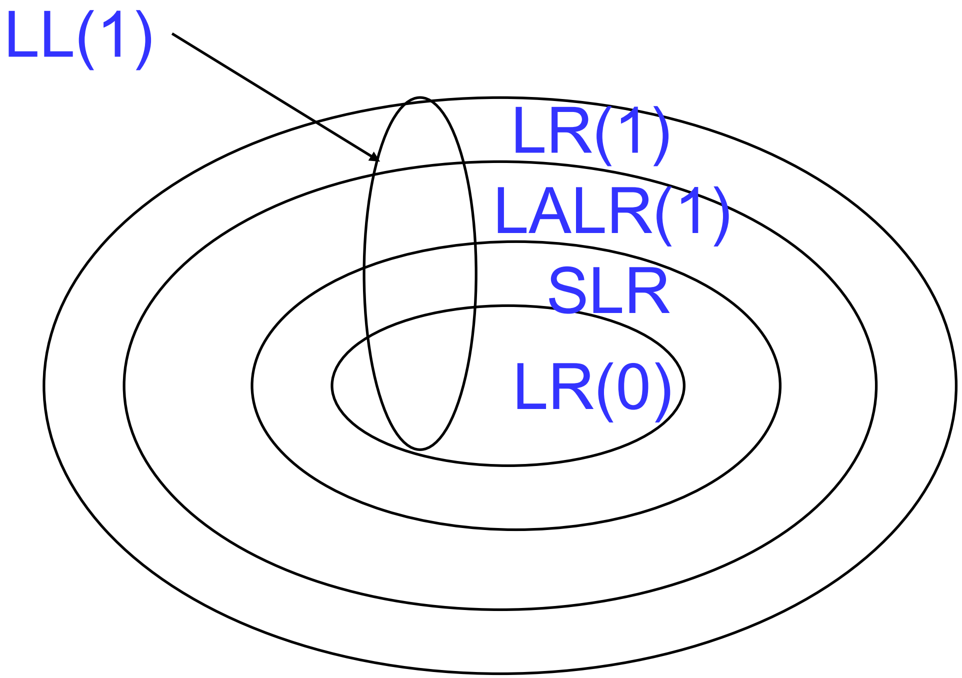
LALR Parsers

- 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 → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables
  - LR states x terminals → {shift/reduce}
  - LR states x non-terminals → 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