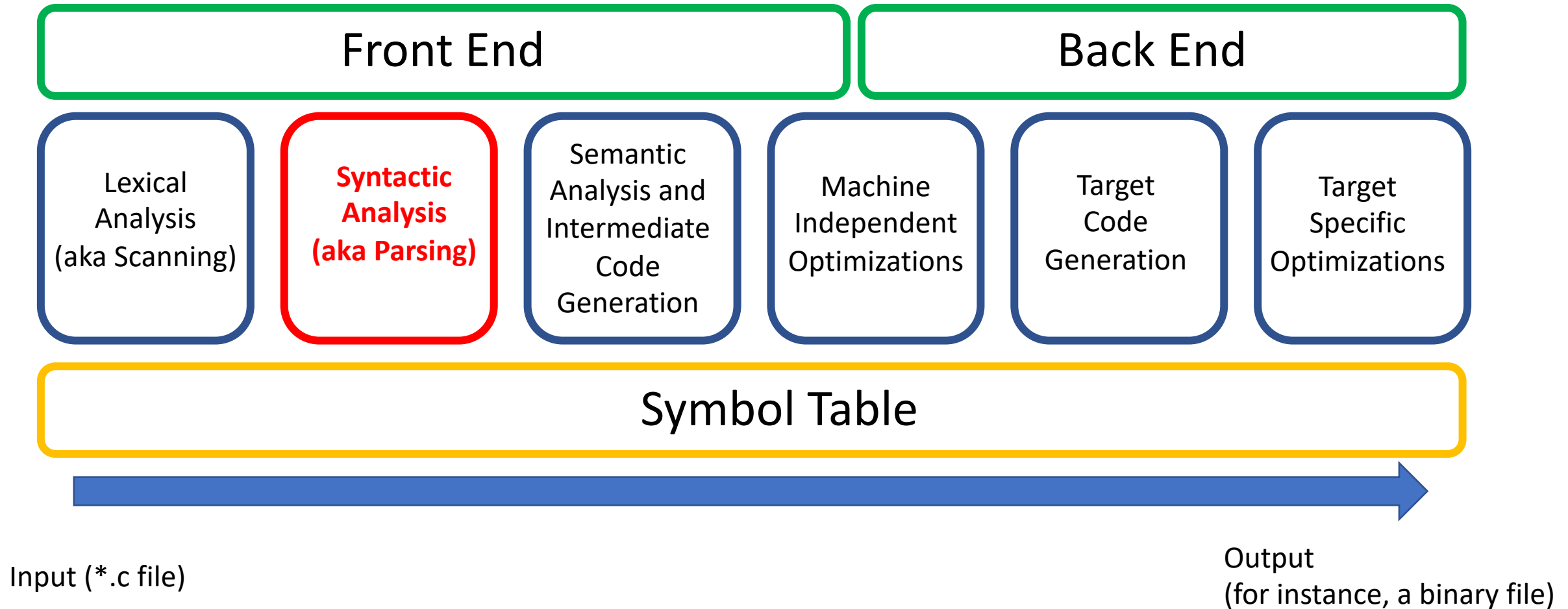


# Syntactic Analysis

# Compilation Overview

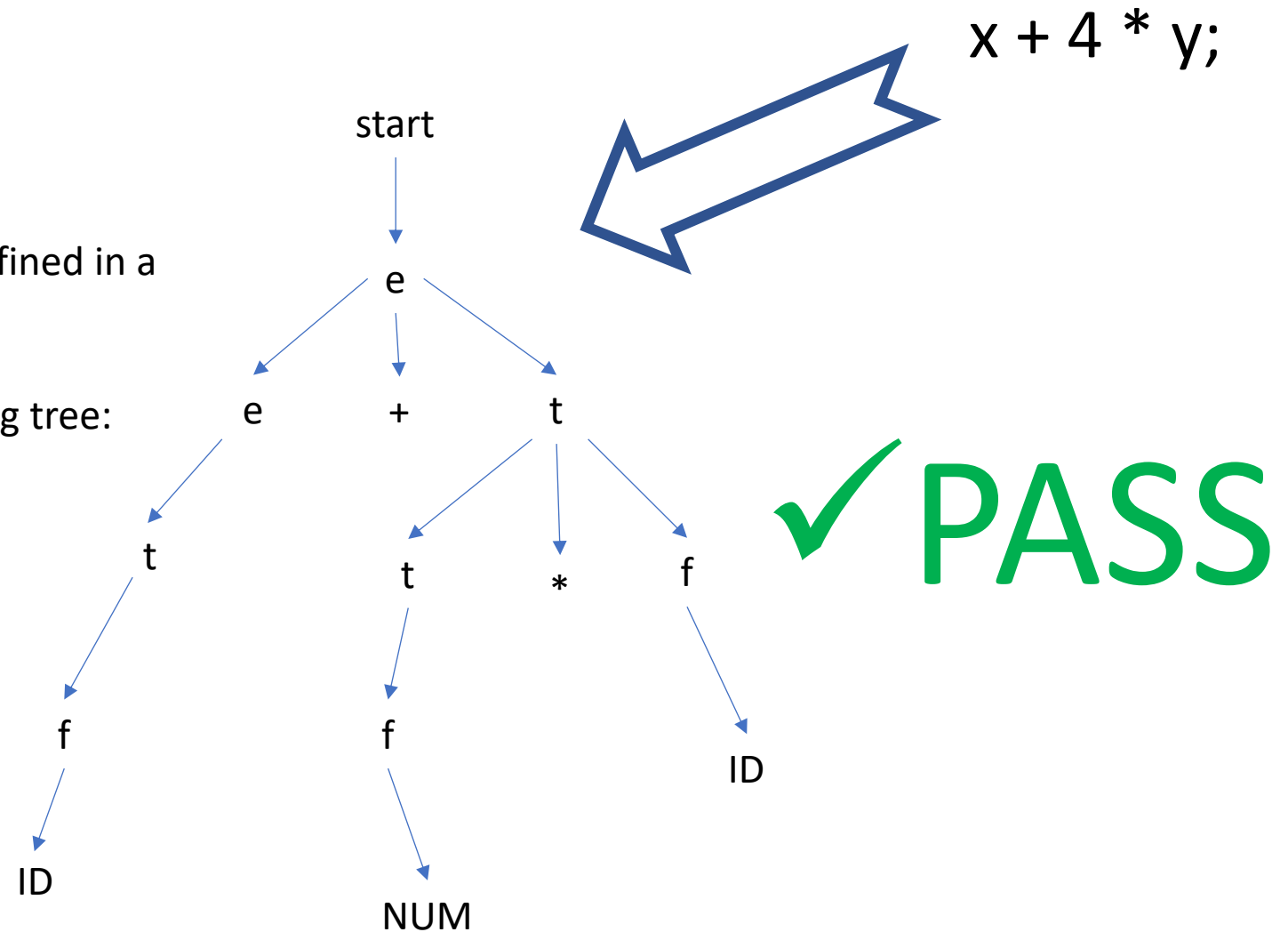


# Syntactic Analysis (Parsing)

- ❑ Syntactic rules of programming language defined in a Context Free Grammar (or just *grammar*)
- ❑ Conceptually, attempts to construct a parsing tree:

Simple arithmetic grammar:

- $\text{start} \rightarrow e$ ;
- $e \rightarrow e + t$
- $e \rightarrow t$
- $t \rightarrow t * f$
- $t \rightarrow f$
- $f \rightarrow \text{ID}$
- $f \rightarrow \text{NUM}$



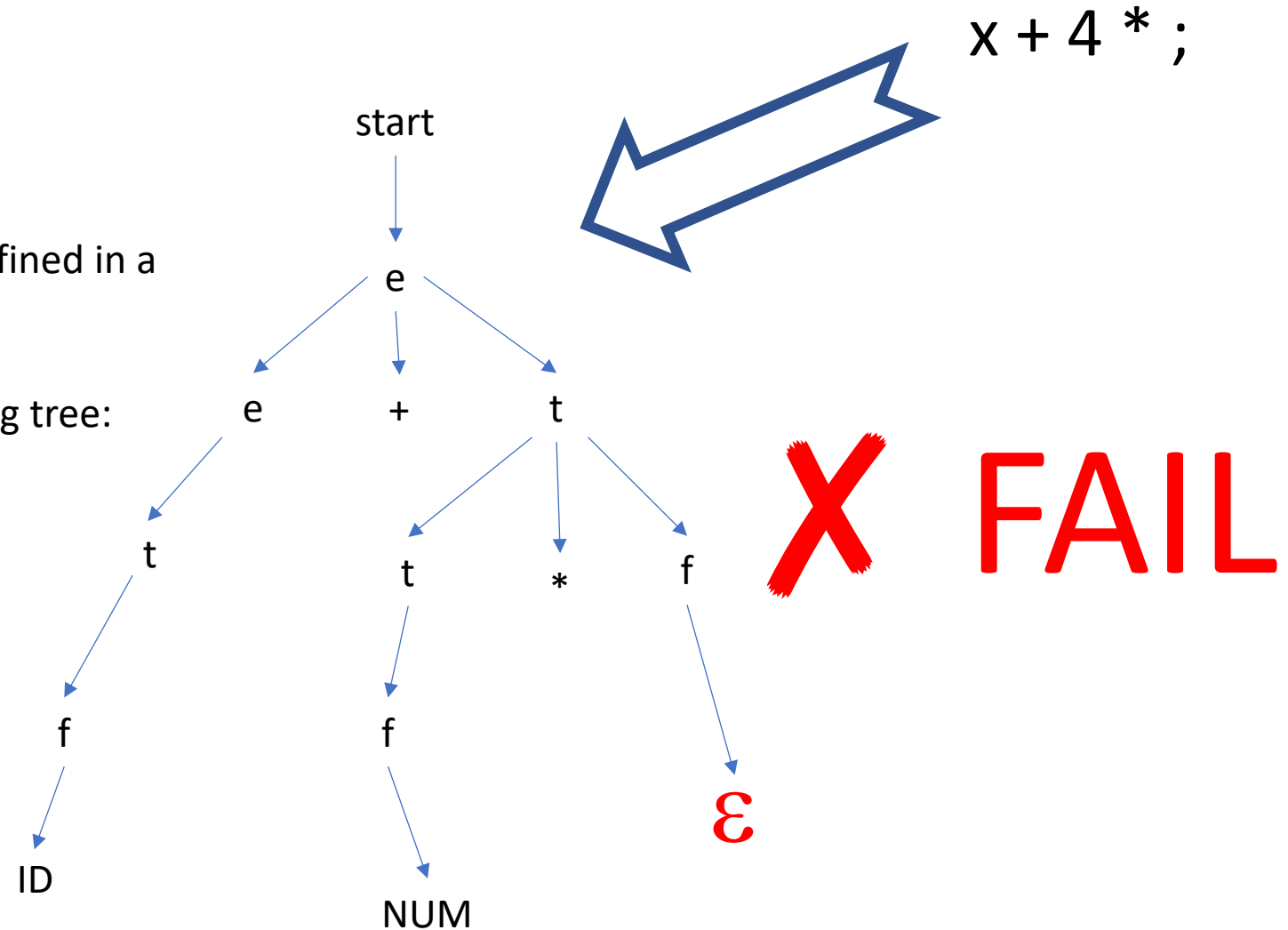
# Syntactic Analysis (Parsing)

- ❑ Syntactic rules of programming language defined in a Context Free Grammar (or just *grammar*)
- ❑ Conceptually, attempts to construct a parsing tree:

Simple arithmetic grammar:

- $\text{start} \rightarrow e$  ;
- $e \rightarrow e + t$
- $e \rightarrow t$
- $t \rightarrow t * f$
- $t \rightarrow f$
- $f \rightarrow \text{ID}$
- $f \rightarrow \text{NUM}$

// rule f does not produce the empty string



# Overview

- Grammars
- Parsing and derivation trees
- Building LL and LR parsers

# Context Free Grammars

- Regular Expressions are unable to specify nested constructs
- Consider the following:

$expr \rightarrow \mathbf{id} \mid \mathbf{number} \mid - expr \mid ( expr )$   
 $\quad \mid expr \mathit{op} expr$   
 $op \rightarrow + \mid - \mid * \mid /$



- **Tokens in bold**
- *Non-terminals in italic*
- $\mid$  means “or”
- $\rightarrow$  means “can be replaced by”

- Need the ability to represent recursion, something in terms of itself
- For instance, cannot describe matching parenthesis with RE alone

# Context Free Grammars (CFG)

*expr*     $\rightarrow$  **id**  
          | **number**  
          | - *expr*  
          | ( *expr* )  
          | *expr op expr*

*op*         $\rightarrow$  +  
          | -  
          | \*  
          | /

- Notation known as Backus-Naur Form (BNF)
- Owed to John Backus and Peter Naur who designed it for Algol-60

- CFG or grammar for short
- "Context Free" means that rules are applicable independently of the context or surroundings
- Each rule in a CFG is a *production*
- Symbols on the left-hand side are *variables* or *non-terminals*
- A variable can have any number of productions
- *Tokens* (symbols in bold) are also known as *terminals* (because they don't have a right-hand sides or productions)
- Tokens cannot appear on the left-hand side of productions
- Some non-terminal is chosen as the *start symbol*

# Context Free Grammars (CFG)

*expr*     $\rightarrow$  **id**  
           $\rightarrow$  **number**  
*expr*     $\rightarrow$  - *expr*  
          | ( *expr* )  
          | *expr* *op* *expr*  
*op*         $\rightarrow$  +  
          | -  
          | \*  
          | /

- If I forget to mark some symbol in bold or italic, the role of the symbol can be easily determined from its context
- It's totally fine to write productions either with  $\rightarrow$  or with |



# Context-Free Grammars

In summary, a CFG consists of:


- a set of terminals  $T$
- a set of non-terminals  $N$
- a non-terminal  $S$  identified as the start symbol
- a set of productions

# Scanning and Parsing

- Recall that tokens such as IDENTIFIER and NUMBER actually represent sets of strings acceptable by a language
- The parser, however, does not distinguish between “1.5”, “1” or “10000.0000”.
- The actual values (as number, strings or names) are stored in the symbol table along the parsing process

# Derivation and Parse Trees

A CFG allows to generate syntactically valid string of terminals, i.e. a valid program:

- a. Begin with start symbol
  - b. Choose a production with the start symbol on the left-hand side
  - c. Replace start symbol with right-hand side of production
  - d. Choose a non-terminal A in resulting string
  - e. Choose a production P with A on the left-hand side, and replace A with the right-hand side of P
- 

# Context-Free Grammars

Consider the previous grammar for arithmetic expressions:

$$\begin{aligned} \text{expr} &\longrightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid ( \text{expr} ) \\ &\quad \mid \text{expr op expr} \\ \text{op} &\longrightarrow + \mid - \mid * \mid / \end{aligned}$$

How could we produce the string “slope \* x + intercept” ?

# Context-Free Grammars

$$\begin{aligned} \text{expr} &\longrightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid ( \text{expr} ) \\ &\quad \mid \text{expr op expr} \\ \text{op} &\longrightarrow + \mid - \mid * \mid / \end{aligned}$$

slope \* x + intercept

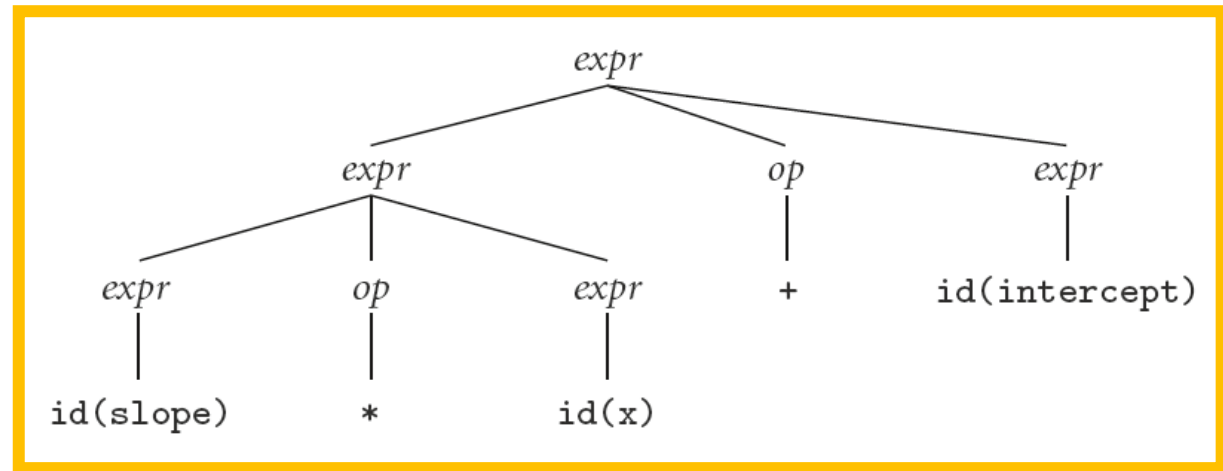
$$\begin{aligned} \text{expr} &\Rightarrow \text{expr op } \underline{\text{expr}} \\ &\Rightarrow \text{expr } \underline{\text{op}} \text{ id} \\ &\Rightarrow \underline{\text{expr}} + \text{id} \\ &\Rightarrow \text{expr op } \underline{\text{expr}} + \text{id} \\ &\Rightarrow \text{expr } \underline{\text{op}} \text{ id} + \text{id} \\ &\Rightarrow \underline{\text{expr}} * \text{id} + \text{id} \\ &\Rightarrow \quad \text{id} \quad * \text{id} + \quad \text{id} \\ &\quad (\text{slope}) \quad (\text{x}) \quad (\text{intercept}) \end{aligned}$$

# Context-Free Grammars

$expr \rightarrow id \mid number \mid -\ expr \mid ( \ expr \ )$   
 $\mid \ expr \ op \ expr$   
 $op \rightarrow + \mid - \mid * \mid /$

$expr \Rightarrow expr \ op \ \underline{expr}$   
 $\Rightarrow expr \ \underline{op} \ id$   
 $\Rightarrow \underline{expr} \ + \ id$   
 $\Rightarrow expr \ op \ \underline{expr} \ + \ id$   
 $\Rightarrow expr \ \underline{op} \ id \ + \ id$   
 $\Rightarrow \underline{expr} \ * \ id \ + \ id$   
 $\Rightarrow \quad id \quad * \ id \ + \quad id$   
 $\quad (slope) \quad (x) \quad (intercept)$

slope \* x + intercept



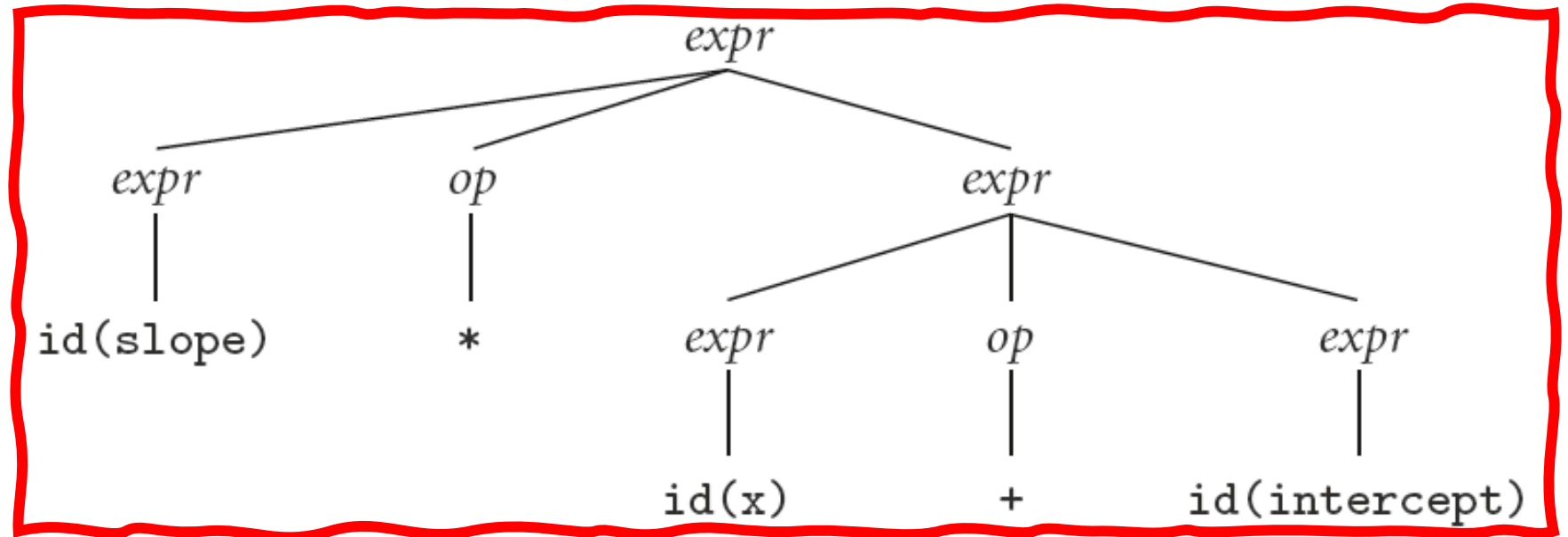
Sentential forms

A parse tree

# Context-Free Grammars

$expr \longrightarrow id \mid number \mid - expr \mid ( expr )$   
 $\quad \quad \quad \mid expr \ op \ expr$   
 $op \longrightarrow + \mid - \mid * \mid /$

slope \* x + intercept

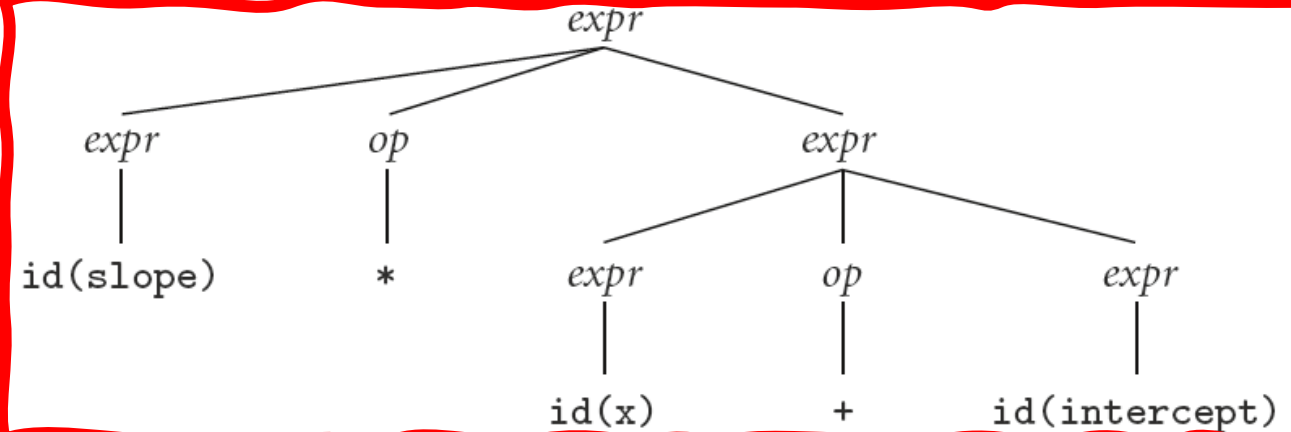
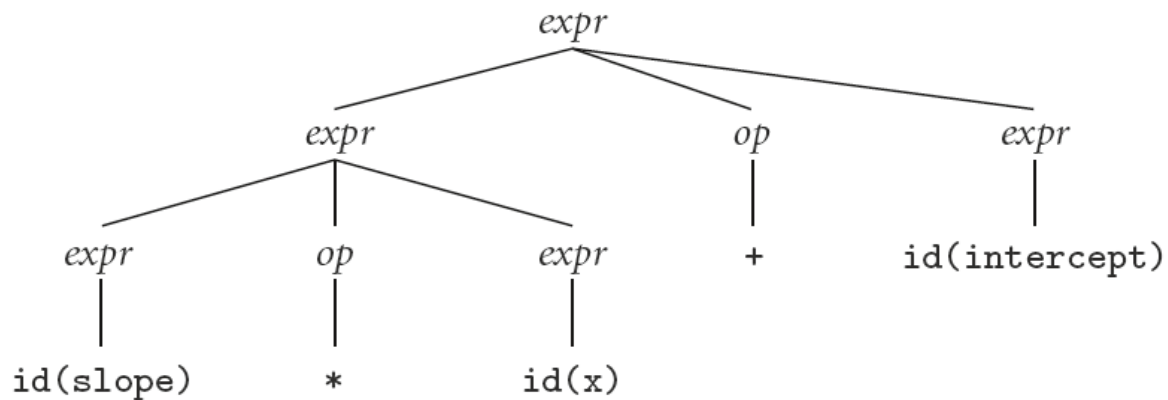


Another parse tree

# Context-Free Grammars

$expr \rightarrow id \mid number \mid -\ expr \mid (\ expr )$   
 $\quad \quad \quad \mid \ expr\ op\ expr$   
 $op \rightarrow + \mid - \mid * \mid /$

slope \* x + intercept



Is there any difference?  
Thoughts?



# Context-Free Grammars

How about we re-write the grammar as:

1.  $expr \longrightarrow term \mid expr \text{ add\_op } term$
2.  $term \longrightarrow factor \mid term \text{ mult\_op } factor$
3.  $factor \longrightarrow id \mid number \mid - factor \mid ( expr )$
4.  $add\_op \longrightarrow + \mid -$
5.  $mult\_op \longrightarrow * \mid /$

# Context-Free Grammars

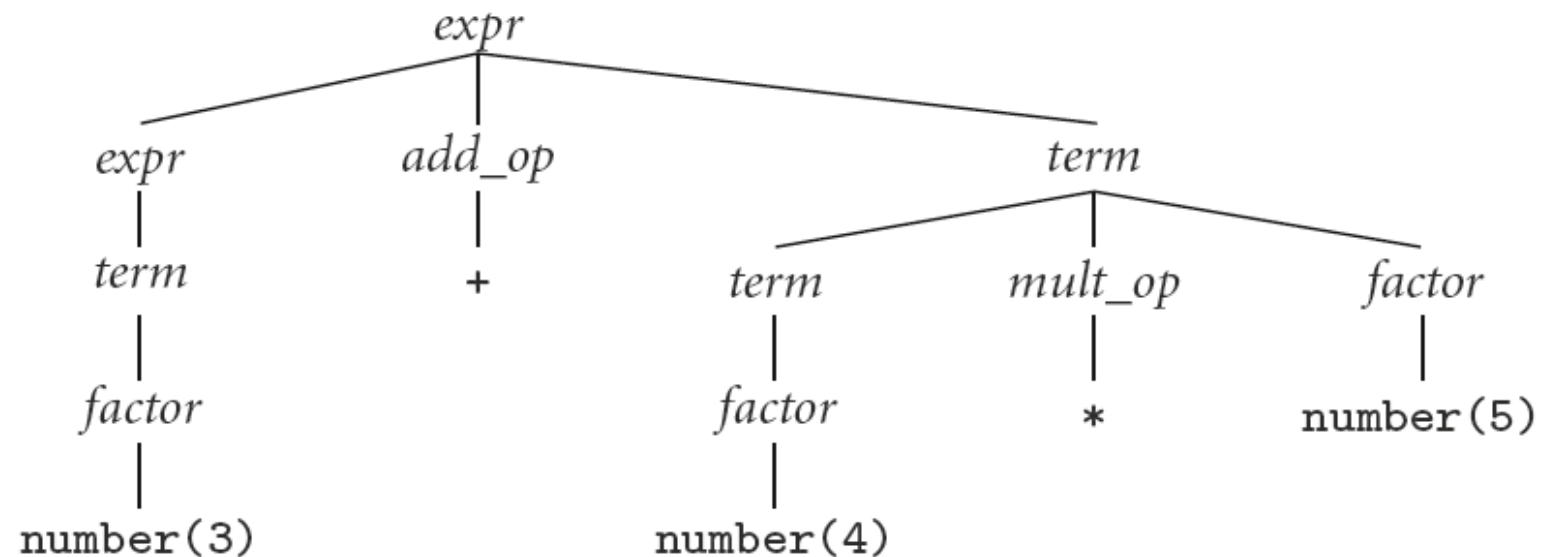
$$\begin{aligned} \text{expr} &\longrightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid ( \text{expr} ) \\ &\quad \mid \text{expr op expr} \\ \text{op} &\longrightarrow + \mid - \mid * \mid / \end{aligned}$$

1.  $\text{expr} \longrightarrow \text{term} \mid \text{expr add\_op term}$
2.  $\text{term} \longrightarrow \text{factor} \mid \text{term mult\_op factor}$
3.  $\text{factor} \longrightarrow \text{id} \mid \text{number} \mid - \text{factor} \mid ( \text{expr} )$
4.  $\text{add\_op} \longrightarrow + \mid -$
5.  $\text{mult\_op} \longrightarrow * \mid /$

# Context-Free Grammars

1.  $expr \rightarrow term \mid expr \text{ add\_op } term$
2.  $term \rightarrow factor \mid term \text{ mult\_op } factor$
3.  $factor \rightarrow id \mid number \mid - factor \mid ( expr )$
4.  $add\_op \rightarrow + \mid -$
5.  $mult\_op \rightarrow * \mid /$

Left-recursion



# Derivations

- Left-most vs right-most derivation: which non-terminal is chosen for the next derivation step
- Left-most derivation: always derive the first non-terminal found in the sentential form, moving from left-to-right
- Right-most derivation: always derive the first non-terminal found in the sentential form, moving from right-to-left
- Do not confuse *left-recursion* or *right-recursion* with *left-most derivation* and *right-most derivation*

# Capturing Structure

- Languages can be designed to capture several aspects, for example associativity and precedence
- Associativity: the order in which a sequence of instances of the same operator is computed

- Example :

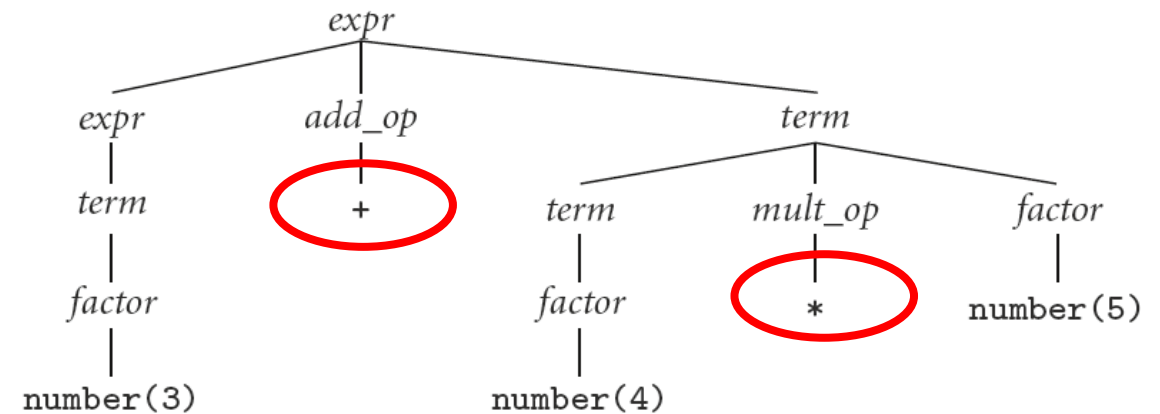
$10 - 4 - 3$  is computed as  $(10 - 4) - 3 \rightarrow$  (left associative operator)  
rather than  $10 - (4 - 3) \rightarrow$  (right associative operator)

- Precedence: some operators are applied before others
- Example:

$3 + 4 * 5 = 3 + (4 * 5)$  instead of  $(3 + 4) * 5$

# Capturing Structure

- In a parse tree, operators with higher precedence appear closer to the leaves of the tree
- Equivalently, operators with lower precedence appear closer to the root of the tree
- In bison/yacc, associativity is managed with a directive:



```
%left '<'
%left '-'
%left '*'
```

More info in: [http://web.mit.edu/gnu/doc/html/bison\\_8.html](http://web.mit.edu/gnu/doc/html/bison_8.html)

# Exercises

1. Design a grammar for defining lists of equalities, inequalities and logical expressions.
2. Create a grammar for drawing points, lines, circles, rectangles. Language should allow to place figures in a plane.
3. Define a grammar for set operations, with variables and assignments.
4. Extend the arithmetic grammar to handle rational numbers.

# Recapping Terminology

- Context-Free Grammar (CFG)
- Symbols: terminals (tokens) and non-terminals
- Productions
- Derivations (left-most and right-most)
- Parse trees
- Sentential form



# Context-Free Grammars

- By analogy to RE and DFAs, a context-free grammar (CFG) is a *generator* for a context-free language (CFL)
  - a parser is a language *recognizer*
- There is an infinite number of grammars for every context-free language
  - not all grammars are created equal, however

# Parsing

- We can create parsers that run in  $O(n^3)$  time for any CFG
- Two well-known parsing algorithms that permit this
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- $O(n^3)$  time is clearly unacceptable for a parser in a compiler - too slow

# Parsing

- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called **LL** and **LR**
- LL stands for  
'Left-to-right, Leftmost derivation'.
- LR stands for  
'Left-to-right, Rightmost derivation'

# Parsing

- LL parsers are also called 'top-down', or 'predictive' parsers
- LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them

# Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

# Parsing

- Will commonly see LL(K) or LR (K)
  - K indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use one token of look-ahead
- The expression grammar (with precedence and associativity) previously seen is LR(1), but not LL(1)

# Parsing

Class	Direction of Scanning	Derivation Discovered	Parse Tree Construction	Algorithms Used
LL	Left-to-right	Left-most	Top-down	Predictive
LR	Left-to-right	Right-most	Bottom-up	Shift-reduce

Main classes of  $O(n)$  parsing algorithms

# Top-down and Bottom-up Parsing

Go over the string:

“A, B, C;”

With the grammar:

- a.  $\text{id\_list} \rightarrow \text{id id\_list\_tail}$
- b.  $\text{id\_list\_tail} \rightarrow , \text{id id\_list\_tail}$
- c.  $\text{id\_list\_tail} \rightarrow ;$



# LL Parsing

An example of an LL(1) grammar:

1. program  $\rightarrow$  stmt\_list \$\$
2. stmt\_list  $\rightarrow$  stmt stmt\_list
3.  $\mid \epsilon$
4. stmt  $\rightarrow$  id := expr
5.  $\mid$  read id
6.  $\mid$  write expr
7. expr  $\rightarrow$  term term\_tail
8. term\_tail  $\rightarrow$  add op term term\_tail
9.  $\mid \epsilon$

10. term  $\rightarrow$  factor fact\_tail
11. fact\_tail  $\rightarrow$  mult\_op fact fact\_tail
12.  $\mid \epsilon$
13. factor  $\rightarrow$  ( expr )
14.  $\mid$  id
15.  $\mid$  number
16. add\_op  $\rightarrow$  +
17.  $\mid$  -
18. mult\_op  $\rightarrow$  \*
19.  $\mid$  /

# LL Parsing

Consider the program:

```
read A
```

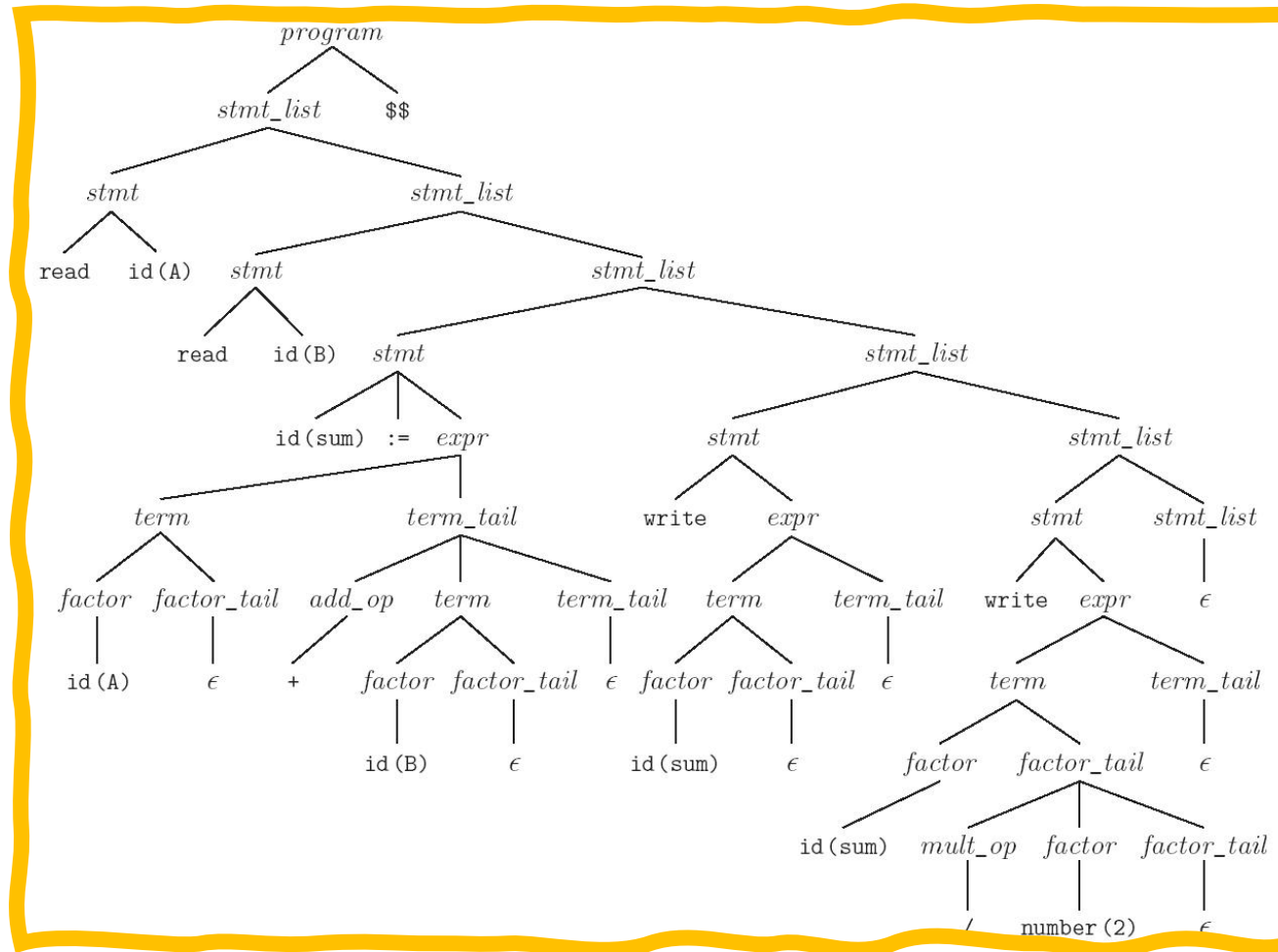
```
read B
```

```
sum := A + B
```

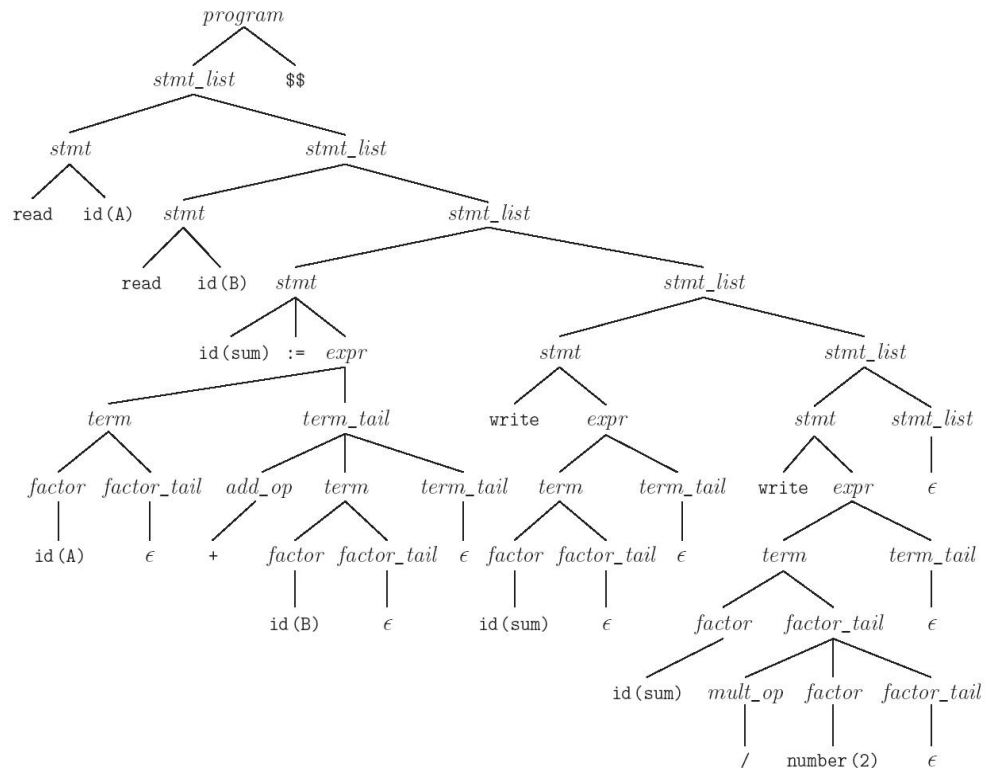
```
write sum
```

```
write sum / 2
```

# LL Parsing



# LL Parsing



- Process tokens in a loop
- Repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token.
- The actions are
  - (1) match a terminal
  - (2) predict a production
  - (3) announce a syntax error

# LL Parsing

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	\$\$
<i>program</i>	1	—	1	1	—	—	—	—	—	—	—	1
<i>stmt_list</i>	2	—	2	2	—	—	—	—	—	—	—	3
<i>stmt</i>	4	—	5	6	—	—	—	—	—	—	—	—
<i>expr</i>	7	7	—	—	—	7	—	—	—	—	—	—
<i>term_tail</i>	9	—	9	9	—	—	9	8	8	—	—	9
<i>term</i>	10	10	—	—	—	10	—	—	—	—	—	—
<i>factor_tail</i>	12	—	12	12	—	—	12	12	12	11	11	12
<i>factor</i>	14	15	—	—	—	13	—	—	—	—	—	—
<i>add_op</i>	—	—	—	—	—	—	—	16	17	—	—	—
<i>mult_op</i>	—	—	—	—	—	—	—	—	—	18	19	—

LL(1) parse table for calculator grammar

Taken from book: Programming Language Pragmatics

# LL Parsing

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	\$\$
<i>program</i>	1	—	1	1	—	—	—	—	—	—	—	1
<i>stmt_list</i>	2	—	2	2	—	—	—	—	—	—	—	3
<i>stmt</i>	4	—	5	6	—	—	—	—	—	—	—	—
<i>expr</i>	7	7	—	—	—	7	—	—	—	—	—	—
<i>term_tail</i>	9	—	9	9	—	—	9	8	8	—	—	9
<i>term</i>	10	10	—	—	—	10	—	—	—	—	—	—
<i>factor_tail</i>	12	—	12	12	—	—	12	12	12	11	11	12
<i>factor</i>	14	15	—	—	—	13	—	—	—	—	—	—
<i>add_op</i>	—	—	—	—	—	—	—	16	17	—	—	—
<i>mult_op</i>	—	—	—	—	—	—	—	—	—	18	19	—

Example string:  
read a;  
write a + b \* 2;

## Grammar

```

program → stmt_list $$
stmt_list → stmt stmt_list
stmt_list → ε
stmt → id := expr
stmt → read id
stmt → write expr
expr → term term_tail
term_tail → add_op term term_tail
term_tail → ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail
factor_tail → ε
factor → ( expr )
factor → id
factor → number
add_op → +
add_op → -
mult_op → *
mult_op → /

```

## Predict Sets

### PREDICT

```

1  program → stmt_list $$ {id, read, write, $$}
2  stmt_list → stmt stmt_list {id, read, write}
3  stmt_list → ε {$$}
4  stmt → id := expr {id}
5  stmt → read id {read}
6  stmt → write expr {write}
7  expr → term term_tail {(, id, number}
8  term_tail → add_op term term_tail {+, -}
9  term_tail → ε {), id, read, write, $$}
10 term → factor factor_tail {(, id, number}
11 factor_tail → mult_op factor factor_tail {*, /}
12 factor_tail → ε {+, -, ), id, read, write, $$}
13 factor → ( expr ) {(}
14 factor → id {id}
15 factor → number {number}
16 add_op → + {+}
17 add_op → - {-}
18 mult_op → * {*}
19 mult_op → / {/}

```

LL(1) parse table for calculator grammar  
Taken from book: Programming Language Pragmatics

# LL Parsing

To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack

- for details see Figure 2.21

The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program

- what you *predict* you will see

# LL Parsing

Some issues with LL parsing:

- Left-recursion
- Common prefixes
- Dangling else


NOTE: eliminating left recursion and common prefixes does NOT make a grammar LL:

- there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
- the few that arise in practice, however, can generally be handled with kludges



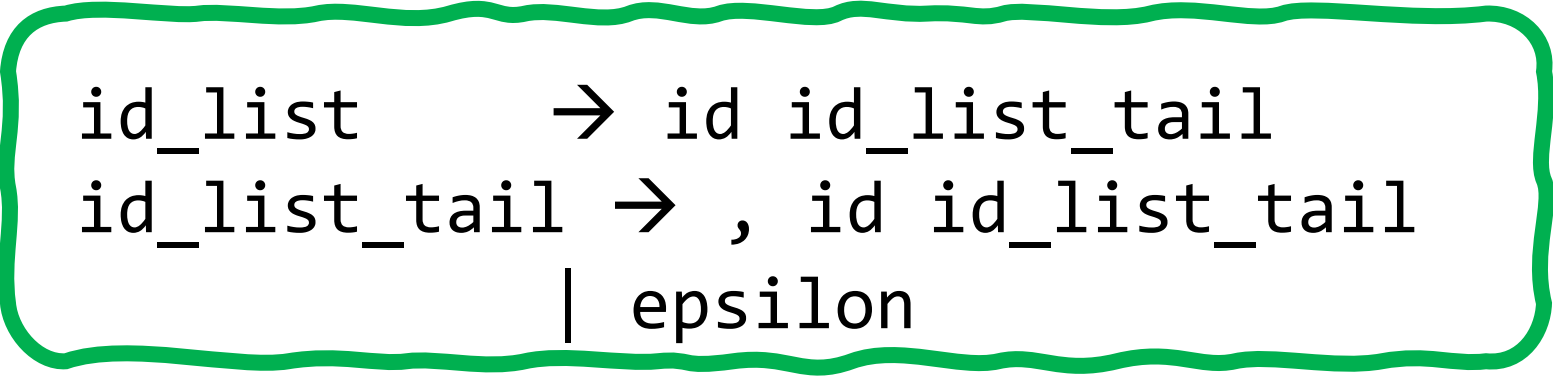
# LL Parsing

Fixing left recursion:



```
id_list → id  
        | id_list , id
```

Becomes



```
id_list      → id id_list_tail  
id_list_tail → , id id_list_tail  
              | epsilon
```

# LL Parsing

- Common prefixes: solved by "left-factoring"
- Example:

`stmt → id := expr | id ( arg_list )`

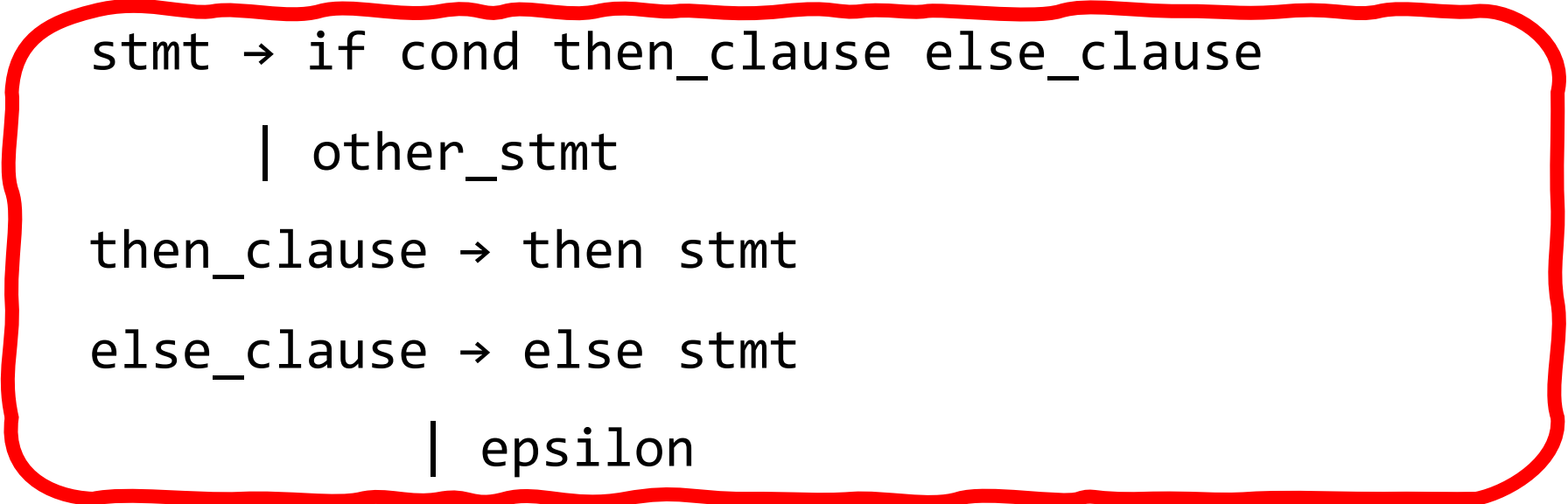
equivalently

`stmt → id id_stmt_tail`  
`id_stmt_tail → := expr`  
`| ( arg_list )`

- we can eliminate left-factor mechanically

# LL Parsing

- The "dangling else" problem prevents grammars from being LL(1) (or in fact LL(k) for any k)
- The following natural grammar fragment is ambiguous (Pascal)



```
stmt → if cond then_clause else_clause  
      | other_stmt  
then_clause → then stmt  
else_clause → else stmt  
             | epsilon
```

# LL Parsing

The less natural grammar fragment can be parsed bottom-up but not top-down

```
stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if cond then balanced_stmt
                else balanced_stmt
                | other_stuff
unbalanced_stmt → if cond then stmt
                | if cond then balanced_stmt
                  else unbalanced_stmt
```

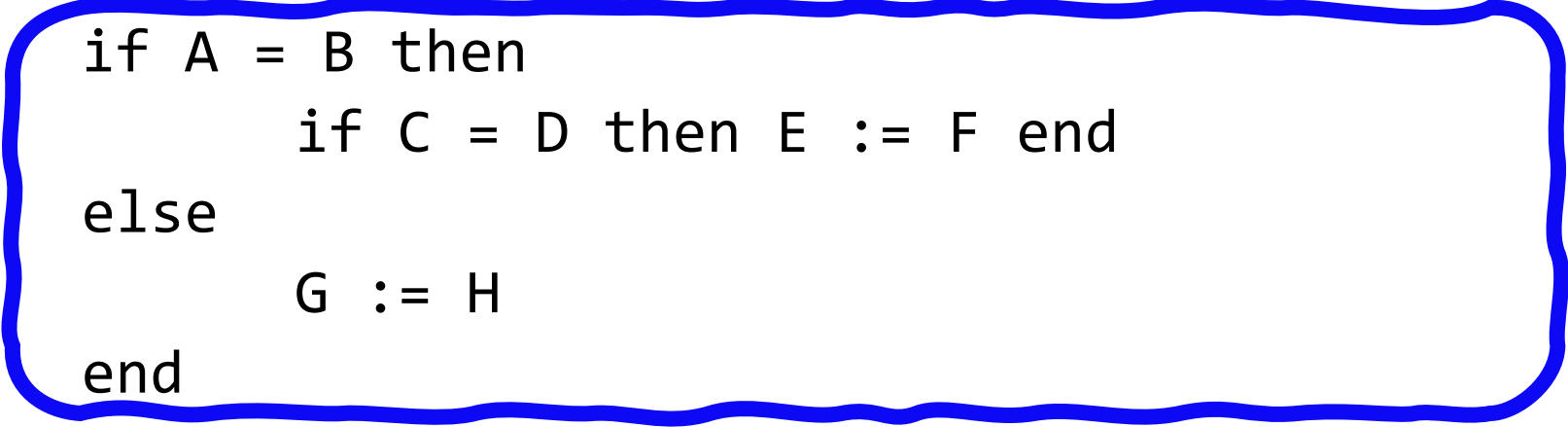
# LL Parsing

Usual fix (top-down OR bottom-up): use the ambiguous grammar together with a *disambiguating rule* that says

- ❑ else goes with the closest then or
- ❑ more generally, the first of two possible productions is the one to predict (or reduce)

# LL Parsing

- Better yet, languages (since Pascal) generally employ explicit end-markers, which eliminate this problem
- In Modula-2, for example, one says:



```
if A = B then  
    if C = D then E := F end  
else  
    G := H  
end
```

- Ada says 'end if'; other languages say 'fi'

# LL Parsing

- Overall parsing algorithm
- Build **First** and **Follow** sets
  - **First** set: terminals that can start some symbol (including terminals)
  - **Follow** set: terminals that can appear after replacing some non-terminal; focuses on the right-hand side of a production.
- Example:  $A \rightarrow B C D$ 
  - What terminals will **Follow** C? Intuitively, those that D can start with
  - What terminals can A start with? By inspection, at least the same as B
- Build parsing table and driver with sets (NOTE: We will not cover this in class, but please try to read about this somewhere)

# LL Parsing

**repeat**

```
next_sym = stack.pop ()
```

```
if (next_sym in Terminals)
```

```
    match (next_sym)    // removes symbol from input and compares against expected token
```

```
    if (next_sym == $) return ACCEPT
```

```
else if (table[next_sym,next_token].action == ERROR)
```

```
    parse_error
```

```
else
```

```
    prod = table[next_sym,next_token].prod
```

```
    foreach sym in reverse(rhs(prod))
```

```
        tack.push (sym)
```

```
until stack.top () == $ // $ is the End-of-File; typical convention in compilers
```



# Example of LL parsing Table

Stack	id	Num	Read	Write	:=	(	)	+	-	*	/	\$
prog	1		1	1								1
s_list	2		2	2								3
stmt	4		5	6								-
expr	7	7				7						-
t_tail	9		9	9			9	8	8			9
term	10	10				10						-
f_tail	12		12	12			12	12	12	11	11	12
factor	14	15				13						-
add_op	--							16	17			-
mul_op	--									18	19	-

# LL Parsing

Consider the following simpler grammar:

$$E \rightarrow T E'$$

$$E' \rightarrow + E$$

$$E' \rightarrow \varepsilon$$

$$T \rightarrow \text{int } T'$$

$$T \rightarrow ( E )$$

$$T' \rightarrow * T$$

$$T' \rightarrow \varepsilon$$

# LL Parsing: Building the First Sets

Building the FIRST set

- $\text{FIRST}(a)$ : set of terminals that can start a string of terminals
- $T$ : Set of terminal symbols
- $N$ : Set of non-terminal symbols

Rules:

1. If  $t$  in  $T$ :  $\text{First}(t) = \{t\}$
2. If  $X$  in  $N$  and  $X \rightarrow \varepsilon$  exists: add  $\varepsilon$  to  $\text{First}(X)$
3. If  $X$  in  $N$  and  $X \rightarrow Y_1 Y_2 \dots Y_m$ ,  $Y_i$  in  $N$ , then:  
    for  $i$  in  $1..m$ :  
        if ( $i = 1$  or  $Y_1 \dots Y_{i-1}$  is nullable)  
             $\text{First}(X) = \text{First}(X) \cup \text{First}(Y_i)$

# LL Parsing: Building the First Sets

```
foreach t in T
  Eps(t) = false
  First(t) = { t }
foreach X in N
  Eps(X) = if  $X \rightarrow \epsilon$  then true else false
  First(X) = {}
repeat
  foreach production  $X \rightarrow Y_1 Y_2 \dots Y_k$ ,
    for i in 1..k
      add First( $Y_i$ ) to First(X)
      if (not Eps( $Y_i$ )) continue with foreach loop
    Eps(X) = true
until no further progress
```

Algorithm to build First sets

# Building the First Sets: Example

- Apply rule (1):


Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	
E'	
T	
E	

# Building the First Sets: Example

- Apply rule (2), we have  $E' \rightarrow \varepsilon$  and  $T' \rightarrow \varepsilon$

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	$\varepsilon$
E'	$\varepsilon$
T	
E	

# Building the First Sets: Example

- Rule (3):  Non-terminal set
3. If  $X$  in  $N$  and  $X \rightarrow Y_1 Y_2 \dots Y_m$ ,  $Y_i$  in  $N$  then:  
for  $i$  in  $1..m$ :  
if ( $i = 1$  or  $Y_1 \dots Y_{i-1}$  is nullable)  
 $\text{First}(x) = \text{First}(x) \cup \text{First}(Y_i)$

$E \rightarrow T E'$   
 $E' \rightarrow + E$   
 $E' \rightarrow \varepsilon$   
 $T \rightarrow \text{int } T'$   
 $T \rightarrow ( E )$   
 $T' \rightarrow * T$   
 $T' \rightarrow \varepsilon$

$E \rightarrow T E$   
 $\text{First}(E) = \text{First}(E) \cup \text{First}(T)$

Computing  $\text{First}(T) = \{\text{int}, ( \}$  (By rule a)

So  $\text{First}(E) = \text{First}(T)$

We don't include  $\text{First}(E')$  because  $T$  is not nullable

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	$\varepsilon$
E'	$\varepsilon$
T	{int, ( }
E	{int, ( }

# Building the First Sets: Example

Rule (3):

3. If  $X$  in  $N$  and  $X \rightarrow Y_1 Y_2 \dots Y_m$ ,  $Y_i$  in  $N$ , then:  
for  $i$  in  $1..m$ :  
if ( $i = 1$  or  $Y_1 \dots Y_{i-1}$  is nullable)  
 $\text{First}(x) = \text{First}(x) \cup \text{First}(Y_i)$

$E \rightarrow T E'$   
 $E' \rightarrow + E$   
 $E' \rightarrow \varepsilon$   
 $T \rightarrow \text{int } T'$   
 $T \rightarrow ( E )$   
 $T' \rightarrow * T$   
 $T' \rightarrow \varepsilon$

$\text{First}(E') = \{ \varepsilon, + \}$

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	$\varepsilon$
E'	$\varepsilon, +$
T	int, (
E	int, (



# Building the First Sets: Example

Rule (3):

3. If  $X$  in and  $X \rightarrow Y_1 Y_2 \dots Y_m$ ,  $Y_i$  in  $N$ , then:  
for  $i$  in  $1..m$ :  
if ( $i = 1$  or  $Y_1 \dots Y_{i-1}$  is nullable)  
 $\text{First}(x) = \text{First}(x) \cup \text{First}(Y_i)$

$E \rightarrow T E'$   
 $E' \rightarrow + E$   
 $E' \rightarrow \varepsilon$   
 $T \rightarrow \text{int } T'$   
 $T \rightarrow ( E )$   
 $T' \rightarrow * T$   
 $T' \rightarrow e$

$\text{First}(T') = \{ \varepsilon, * \}$

Symbol	First
(	(
)	)
+	+
*	*
int	int
T'	$\varepsilon, *$
E'	$\varepsilon, +$
T	int, (
E	int, (

# LL Parsing: Building the **Follow** Sets

## Computing the **Follow** sets of non-terminals

1. If  $\$$  is the input end-marker, and  $S$  is the start symbol,  $\$ \in \text{FOLLOW}(S)$ .

**Intuition:** The only symbol that can follow a complete program, is the end of file.

2. If there is a production,  $A \rightarrow \alpha \underline{B} \beta$ , then  $(\text{FIRST}(\beta) - \epsilon) \subseteq \text{FOLLOW}(B)$ .

**Intuition:** The symbols that follow  $B$  are those that start  $\beta$  (skip  $\epsilon$ )

3. If there is a production,  $A \rightarrow \alpha \underline{B}$ , or a production  $A \rightarrow \alpha \underline{B} \beta$ , where  $\epsilon \in \text{FIRST}(\beta)$ , then  $\text{Follow}(A) \subseteq \text{Follow}(B)$ .

**Intuition:**  $B$  is (effectively) the last symbol on the right-hand side of the production, so its Follow set should include the Follow of  $A$

# LL Parsing: Building the Follow Sets

```
foreach symbol X
```

```
    Follow(X) = { } // Initialize to empty set
```

```
repeat
```

```
    foreach production A → a B b
```

```
        Follow(B) = Follow(B) U First(b)
```

```
    foreach production A → a B or A → a B b with Eps(b)=True
```

```
        Follow(B) = Follow(B) U Follow(A)
```

```
until no further progress
```

Predicate that evaluates  
if a symbol can derive in the  
empty string  $\epsilon$



This is a fixed point  
algorithm: continue to (re-)compute  
until nothing changes

# LL Parsing: Building the Follow Sets

Computing the Follow set of non-terminals

1. If \$ is the input end-marker, and S is the start symbol,  $\$ \in \text{FOLLOW}(S)$ .

→ Include \$ in Follow(S)

$E \rightarrow T E'$

$E' \rightarrow + E$

$E' \rightarrow \varepsilon$

$T \rightarrow \text{int } T'$

$T \rightarrow ( E )$

$T' \rightarrow * T$

$T' \rightarrow \varepsilon$



**Reminder:** *int* here is a token representing numbers

# LL Parsing: Building the Follow Sets

2. If there is a production,  $A \rightarrow \alpha B \beta$ , then  $(\text{FIRST}(\beta) - \epsilon) \subseteq \text{FOLLOW}(B)$ .

We look at the occurrences of non-terminals on the right-hand side of productions which are followed by something

2a.  $E \rightarrow T . E'$

Follow(T) contains  $\text{First}(E') = \{\epsilon, +\} - \{\epsilon\} = \{+\}$

2b.  $T \rightarrow ( E . )$

Follow(E) contains (at least)  $\text{First}(')') = \{ ) \}$

Follow(E) now becomes  $\{\$, ) \}$



The . (DOT) is not part of the grammar. It marks the end of a non-terminal, i.e. what we are trying to compute

# LL Parsing: Building the Follow Sets

- a.  $E \rightarrow T E'$  .  
Follow(  $E'$  ) contains (at least) Follow(  $E$  ), so Follow( $E'$ ) = { }, \$ } (from step 2b)
- b.  $\epsilon$  in First(  $E'$  ) so:  
Follow(  $T$  ) contains (at least) Follow(  $E$  ), so Follow( $T$ ) = { }, \$, +} (from step 2a and 2b)
- c.  $E' \rightarrow + E$  .  
Follow(  $E$  ) contains (at least) Follow(  $E'$  ), so Follow( $E$ ) = { }, \$ } (from step 3a)
- d.  $T \rightarrow \text{int } T'$  .  
Follow(  $T'$  ) contains (at least) Follow(  $T$  ), so Follow( $T'$ ) = { }, \$, +}. (from step 3b)
- e.  $T' \rightarrow * T$  .  
Follow(  $T$  ) contains (at least) Follow(  $T'$  ), so Follow( $T$ ) = { }, \$ } (from step 3d)

# LL Parsing: Building the Follow Sets

We do this whole process again until no more additions happen:

- f.  $E \rightarrow T E'$  .  
Follow(  $E'$  ) contains (at least) Follow(  $E$  ), Follow( $E'$ ) = { ), \$ } (no change)
- g.  $\epsilon$  in First(  $E'$  ) so:  
Follow(  $T$  ) contains (at least) Follow(  $E$  ), Follow( $T$ ) = { ), \$, + } (no change)
- h.  $E' \rightarrow + E$  .  
Follow(  $E$  ) contains (at least) Follow(  $E'$  ), Follow( $E$ ) = { ), \$ } (no change)
- i.  $T \rightarrow \text{int } T'$  .  
Follow(  $T'$  ) contains (at least) Follow(  $T$  ), Follow( $T'$ ) = { ), \$, + } (no change)
- j.  $T' \rightarrow^* T$  .  
Follow(  $T$  ) contains (at least) Follow(  $T'$  ), Follow( $T$ ) = { ), \$ } (no change)

# LL Parsing: Building the First and Follow Sets

Symbol	First	Follow
(	(	N/A
)	)	
+	+	
*	*	
int	int	
T'	$\epsilon$ , *	), \$, +
E'	$\epsilon$ , +	), \$
T	int, (	), \$, +
E	int, (	), \$

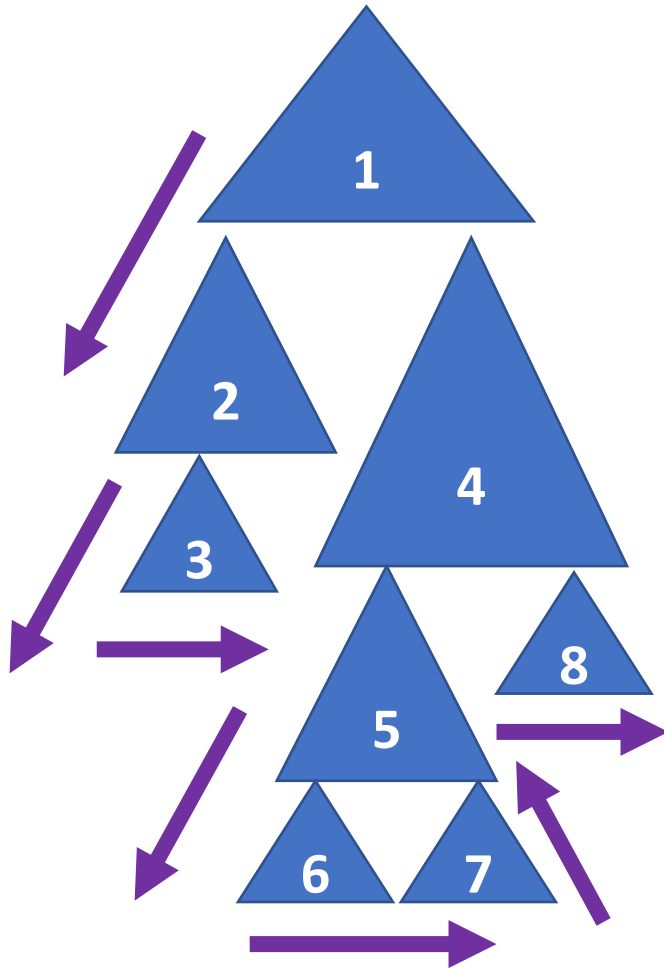
Final First and Follow sets



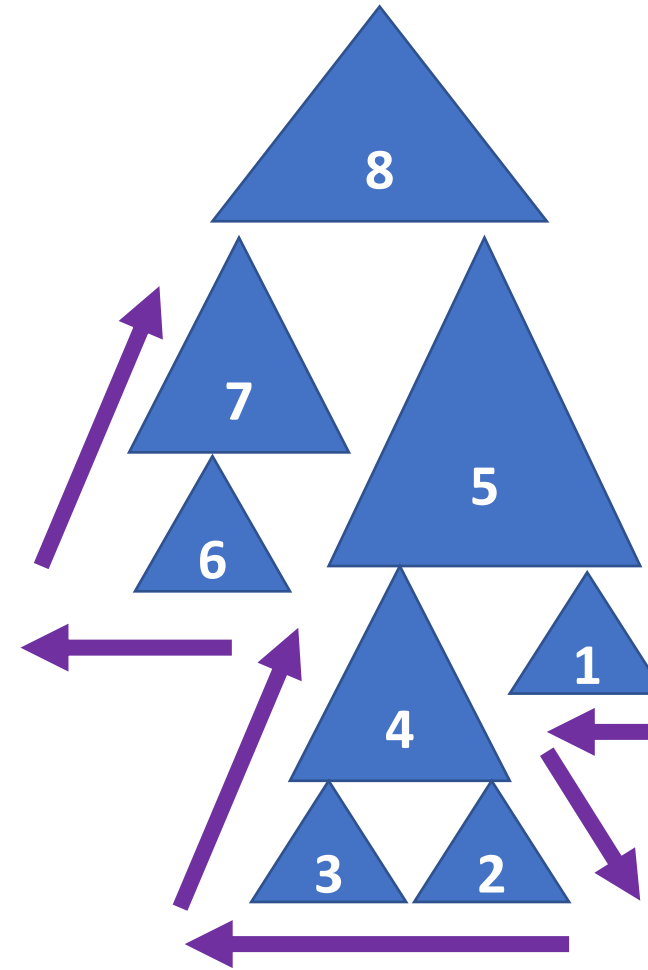
# LR Parsing

- Builds a forest of partial subtrees of the parse tree
- Performs a replacement of the form  $Y_1 Y_2 \dots Y_k \rightarrow A$  when the production  $A \rightarrow Y_1 Y_2 \dots Y_k$  is recognized
- Table driven
- Roots of partially recognized sub-trees stored in a stack
- Produces reversed right-most (canonical) derivations

# LL vs LR Parsing: Visually



- Assumes input string was produced via a **left-most** derivation
- Top-down discovery
- In general: moves down and right



- Assumes input string was produced via a **right-most** derivation
- Bottom-up discovery
- In general: moves up and left

# LR Parsing

- Shift action: performed when a new token is found in the input, it's shifted to the stack
- Reduce action: performed when the right-hand side of a production is recognized, pop symbols from stack and insert non-terminal of the left-hand side of the production
- Role of stack is the main difference between LL and LR parsing:
  - Top-down parsing: stack contains symbols that expects to see in the future
  - Bottom-up parsing: stack contains symbols of what the parsing already has seen

# LR Parsing

A grammar producing list of identifiers:

- $\text{id\_list} \rightarrow \text{ID id\_list\_tail}$
- $\text{id\_list\_tail} \rightarrow , \text{ID id\_list\_tail}$
- $\text{id\_list\_tail} \rightarrow ;$

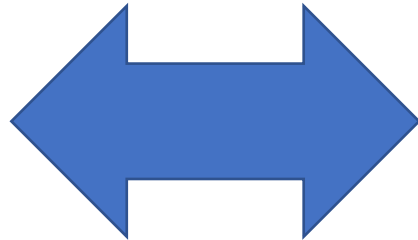
Symbols in right-hand side of production that are being replaced by the left-hand side (non-terminal) constitute the handle of the sentential form

Stack Contents (Sentential Forms)	Remaining Input String	Action
\$	A , B , C ;	SHIFT
\$ ID (A)	, B , C ;	SHIFT
\$ ID (A) ,	B , C ;	SHIFT
\$ ID (A) , ID (B)	, C ;	SHIFT
\$ ID (A) , ID (B),	C ;	SHIFT
\$ ID (A) , ID (B) , ID (C)	;	SHIFT
\$ ID (A), ID (B) , ID (C) ;		REDUCE
\$ ID (A), ID (B) , ID (C) id_list_tail		REDUCE
\$ ID (A) , ID (B) id_list_tail		REDUCE
\$ ID (A) id_list_tail		REDUCE
\$ id_list		REDUCE

# LR Parsing

1.  $e \rightarrow t e'$
2.  $e' \rightarrow + t e'$
3.  $e' \rightarrow \varepsilon$
4.  $t \rightarrow f t'$
5.  $t' \rightarrow * f$
6.  $t' \rightarrow \varepsilon$
7.  $f \rightarrow ( e )$
8.  $f \rightarrow \text{ID}$
9.  $f \rightarrow \text{NUM}$

**LL Grammar**



1.  $e \rightarrow e + t$
2.  $e \rightarrow t$
3.  $t \rightarrow t * f$
4.  $t \rightarrow f$
5.  $f \rightarrow ( e )$
6.  $f \rightarrow \text{ID}$
7.  $f \rightarrow \text{NUM}$

**LR Grammar**

- Left recursion allows the parser to collapse lists as it goes along (contrast with previous example by trying a list from  $e \rightarrow e + t$ )
- Left recursion also captures left associativity while keeping operators and operands together in the same right-hand side
- This wasn't possible in top-down grammars

# LR Parsing Example

Remaining Input String	Stack Contents (Sentential Forms)	Action
ID + NUM * ID \$	\$ (0)	SHIFT ID, GOTO 1
+ NUM * ID \$	\$ (0) ID (1)	REDUCE by 6 (fact $\rightarrow$ ID)
+ NUM * ID \$	\$ (0) fact	Back to state 0
+ NUM * ID \$	\$ (0) fact	GOTO 7
+ NUM * ID \$	\$ (0) fact (7)	REDUCE by 4 (term $\rightarrow$ fact)
+ NUM * ID \$	\$ (0) term	GOTO 6
+ NUM * ID \$	\$ (0) term (6)	REDUCE by 2 (expr $\rightarrow$ term)
+ NUM * ID \$	\$ (0) expr	GOTO 5
+ NUM * ID \$	\$ (0) expr (5)	SHIFT +, GOTO 11
NUM * ID \$	\$ (0) expr (5) + (11)	SHIFT NUM, GOTO 2
* ID \$	\$ (0) expr (5) + (11) NUM (2)	REDUCE by 5 (fact $\rightarrow$ NUM)
* ID \$	\$ (0) expr (5) + (11) fact	GOTO 7

Use together with grammar in canvas (.output file)

# LR Parsing Example

Remaining Input String	Stack Contents (Sentential Forms)	Action
* ID \$	\$ (0) expr (5) + (11) fact	GOTO 7
* ID \$	\$ (0) expr (5) + (11) fact (7)	REDUCE by 4 (term $\rightarrow$ fact)
* ID \$	\$ (0) expr (5) + (11) term	GOTO 14
* ID \$	\$ (0) expr (5) + (11) term (14)	SHIFT *, GOTO 12
ID \$	\$ (0) expr (5) + (11) term (14) * (12)	SHIFT ID, GOTO 1
\$	\$ (0) expr (5) + (11) term (14) * (12) ID (1)	REDUCE by 6 (fact $\rightarrow$ ID)
\$	\$ (0) expr (5) + (11) term (14) * (12) fact	GOTO 15
\$	\$ (0) expr (5) + (11) term (14) * (12) fact (15)	REDUCE by 3 (term $\rightarrow$ term * fact)
\$	\$ (0) expr (5) + (11) term	GOTO 14
\$	\$ (0) expr (5) + (11) term (14)	REDUCE by 1 (expr $\rightarrow$ expr + term)
\$	\$ (0) expr	GOTO 5
\$	\$ (0) expr (5)	SHIFT \$, GOTO 10
	\$ (0) expr (5) \$ (10)	ACCEPT

# LR Parsing

- Key: keeping track of the set of productions on which the parser might be, and the position within the productions
- At the beginning, the stack's parser is empty; our state is at the beginning of the start non-terminal symbol
- We represent the position in some production (i.e. the location associated to the top of the parser stack) with a dot symbol (.) in some position of the right-hand side of a production:

$$A \rightarrow . B \beta$$

$$A \rightarrow \alpha B . \beta$$



# LR Parsing

- When augmented with a (.) a production becomes an LR item
- The LR item tells us in which production we can be, and it what position:

1.  $e \rightarrow . e + t$   
2.  $e \rightarrow t$   
3.  $t \rightarrow t * f$   
4.  $t \rightarrow f$   
5.  $f \rightarrow ( e )$   
6.  $f \rightarrow ID$   
7.  $f \rightarrow NUM$

1.  $e \rightarrow e + t$   
2.  $e \rightarrow . t$   
3.  $t \rightarrow t * f$   
4.  $t \rightarrow f$   
5.  $f \rightarrow ( e )$   
6.  $f \rightarrow ID$   
7.  $f \rightarrow NUM$

1.  $e \rightarrow e + t$   
2.  $e \rightarrow t$   
3.  $t \rightarrow . t * f$   
4.  $t \rightarrow f$   
5.  $f \rightarrow ( e )$   
6.  $f \rightarrow ID$   
7.  $f \rightarrow NUM$

1.  $e \rightarrow e + t$   
2.  $e \rightarrow t$   
3.  $t \rightarrow t * f$   
4.  $t \rightarrow . f$   
5.  $f \rightarrow ( e )$   
6.  $f \rightarrow ID$   
7.  $f \rightarrow NUM$

1.  $e \rightarrow e + t$   
2.  $e \rightarrow t$   
3.  $t \rightarrow t * f$   
4.  $t \rightarrow f$   
5.  $f \rightarrow . ( e )$   
6.  $f \rightarrow . ID$   
7.  $f \rightarrow . NUM$

# LR Parsing

1.  $e \rightarrow \cdot e + t$



The original item is the basis of the list

2.  $e \rightarrow t$

3.  $t \rightarrow t * f$

4.  $t \rightarrow f$

5.  $f \rightarrow (e)$

6.  $f \rightarrow \text{ID}$

7.  $f \rightarrow \text{NUM}$

Remaining items are the closure:

1.  $e \rightarrow e + t$

2.  $e \rightarrow \cdot t$

3.  $t \rightarrow t * f$

4.  $t \rightarrow f$

5.  $f \rightarrow (e)$

6.  $f \rightarrow \text{ID}$

7.  $f \rightarrow \text{NUM}$

1.  $e \rightarrow e + t$

2.  $e \rightarrow t$

3.  $t \rightarrow \cdot t * f$

4.  $t \rightarrow f$

5.  $f \rightarrow (e)$

6.  $f \rightarrow \text{ID}$

7.  $f \rightarrow \text{NUM}$

1.  $e \rightarrow e + t$

2.  $e \rightarrow t$

3.  $t \rightarrow t * f$

4.  $t \rightarrow \cdot f$

5.  $f \rightarrow (e)$

6.  $f \rightarrow \text{ID}$

7.  $f \rightarrow \text{NUM}$

1.  $e \rightarrow e + t$

2.  $e \rightarrow t$

3.  $t \rightarrow t * f$

4.  $t \rightarrow f$

5.  $f \rightarrow \cdot (e)$

6.  $f \rightarrow \cdot \text{ID}$

7.  $f \rightarrow \cdot \text{NUM}$

# LR Parsing

1.  $e \rightarrow . e + t$
2.  $e \rightarrow . t$
3.  $t \rightarrow . t * f$
4.  $t \rightarrow . f$
5.  $f \rightarrow . ( e )$
6.  $f \rightarrow . ID$
7.  $f \rightarrow . NUM$

- The list of LR items represents the state of the parser
- By shifting and reducing, the set of items will change
- If a state in which some item has the (.) at the end of the right-hand side of a production, we reduce by that production
- If we need to shift, but the next token cannot follow the (.) in any item of the current state, then we detect a syntax error

# LR Parsing

- Suppose the input string is: ID \* NUM + ID
- Next token is ID, which is shifted into the stack
- Our options are now narrowed down to #6 → single basis and empty closure
- Since the (.) is at the end #6, we must reduce

1.  $e \rightarrow .e + t$   
2.  $e \rightarrow .t$   
3.  $t \rightarrow .t * f$   
4.  $t \rightarrow .f$   
5.  $f \rightarrow .(e)$   
6.  $f \rightarrow .ID$   
7.  $f \rightarrow .NUM$



6.  $f \rightarrow ID.$

3.  $t \rightarrow t * f.$   
4.  $t \rightarrow f.$

Stack: \$

Stack: \$ ID

Stack: \$ f

# LR Parsing

- Remaining input string is: \* NUM + ID
- We could be either in #3 or #4
- (.) is at the end of #4, so reduce again
- Next token is \*, so we shift it into the stack

3.  $t \rightarrow t * f$   
4.  $t \rightarrow f.$

Stack: \$ f

3.  $t \rightarrow t . * f$   
4.  $t \rightarrow f$

Stack: \$ t

3.  $t \rightarrow t * . f$

Stack: \$ t \*

# LR Parsing

- Remaining input string is: NUM + ID
- We could be either in #3, #5, #6 or 7
- Next token is NUM, we should then be in #7
- Shift token from input to stack

3.  $t \rightarrow t * . f$   
5.  $f \rightarrow . ( e )$   
6.  $f \rightarrow . ID$   
7.  $f \rightarrow . NUM$

Stack: \$ t \*

3.  $t \rightarrow t * f$   
5.  $f \rightarrow ( e )$   
6.  $f \rightarrow ID$   
7.  $f \rightarrow NUM .$

Stack: \$ t \* ID

3.  $t \rightarrow t * f .$

Stack: \$ t \* f

1.  $e \rightarrow e + t .$   
2.  $e \rightarrow t .$

Stack: \$ t

# LR Parsing

- Remaining input string is: + ID
- We have reached the end of #2: reduce
- Next token is +, shift it to stack
- New item list mark the beginning with t

1.  $e \rightarrow e + t$   
2.  $e \rightarrow t.$

Stack: \$ t

1.  $e \rightarrow e. + t$   
2.  $e \rightarrow t$

Stack: \$ e

1.  $e \rightarrow e + . t$   
2.  $e \rightarrow t$

Stack: \$ e +

First item is the basis;  
remaining items form the  
closure

1.  $e \rightarrow e + . t$   
3.  $t \rightarrow . t * f$   
4.  $t \rightarrow . f$   
5.  $f \rightarrow . ( e )$   
6.  $f \rightarrow . ID$   
7.  $f \rightarrow . NUM$

Stack: \$ e +

# LR Parsing

- Remaining input string is ID, we shift it
- Reduce ID by f
- Reduce f by t
- Reduce e + t by e

6.  $f \rightarrow .ID$

6.  $f \rightarrow ID.$

4.  $t \rightarrow f.$

1.  $e \rightarrow e + t.$

4.  $e \rightarrow t.$

Stack: \$ e +

Stack: \$ e + ID

Stack: \$ e + f

Stack: \$ e + t

Stack: \$ e



# LR Parsing

- Shift rules work as transition functions in the automaton
- Each state of the automaton corresponds to a list of items
- Set of items represent where we can be at some point in the parsing process
- Sets of items constructed during the parser construction, but not needed during actual parsing

# LR Parsing

Family of parsers:

- LR(0): cannot handle shift-reduce conflicts
- SLR(1) or Simple LR: parser peeks at input and use Follow sets to resolve conflicts (reduction  $A \rightarrow \alpha$  is applied if the next token is in  $\text{Follow}(\alpha)$ )

Issues arise when the token is also in the First set of the symbols following a (.)

- LALR(1):
  - Use state-specific look-ahead to disambiguate
  - Most common parsers; resolve more conflicts
- Full LR:
  - Much larger and complex than LALR and
  - Duplicates states