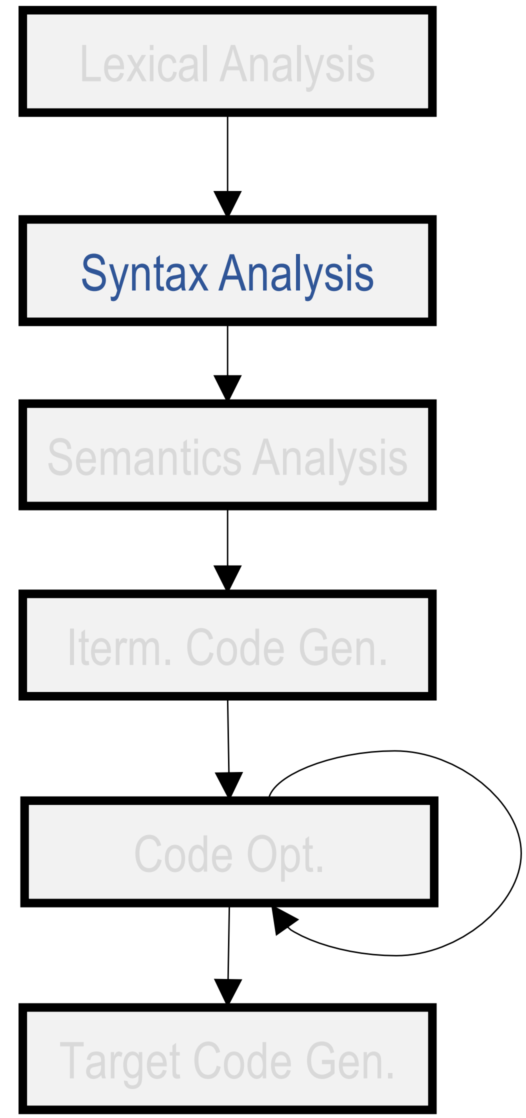
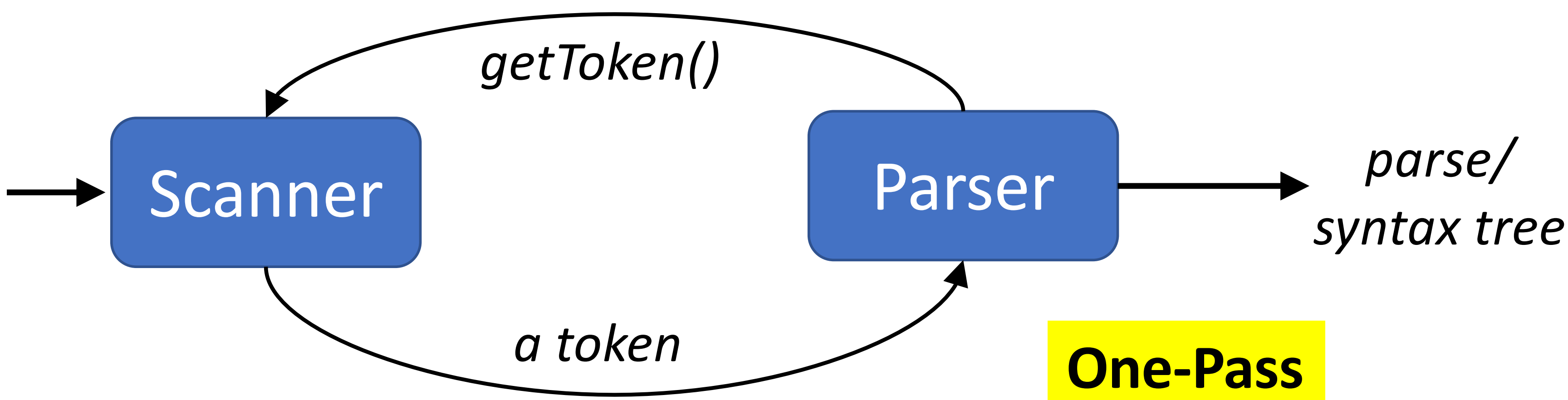
Syntax Analysis

(Chapters 3-5)



Parsing

- Determine the syntax (structure) of a program based on the token sequence; Typically, parser drives scanner

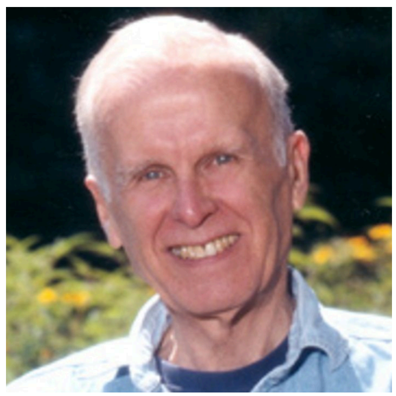


One-Pass

Context Free Grammars

Context-Free Grammars

- Backus-Naur Form (BNF)
  - first used for ALGOL 60
  - developed by **John Backus** and **Peter Naur**



```
<exp> ::= <exp> <op> <exp> | ( <exp> ) | number
<op>  ::= + | - | *
```

Context-Free Grammars

- A Rule in BNF
  - left-hand side (**LHS**) defines the name of a structure (**nonterminal**)
  - right-hand side (**RHS**) gives its possible layouts (sequence of **terminals** & **nonterminals**)

exp → *exp op exp* | **(** *exp* **)** | **number**

structure name

possible layouts

Context-Free Grammars

- Derivation
  - a sequence of replacements of structure names by layouts on RHS, starting from the ***start nonterminal***

Start Nonterminal

exp ⇒ *exp op exp*
⇒ *exp op number*
⇒ *number op number*
⇒ *number * number*

a legal token string with terminals only

(1) *exp* → *exp op exp*
(2) *exp* → **(** *exp* **)**
(3) *exp* → **number**
(4) *op* → **+**
(5) *op* → **-**
(6) *op* → *****

CFG

Context-Free Grammars

- Derivation
  - **Leftmost derivation**: the leftmost nonterminal is replaced in each step

$$
\begin{aligned}
\mathit{exp} &\Rightarrow \mathit{exp}\ \mathit{op}\ \mathit{exp} \\
&\Rightarrow (\ \mathit{exp}\ )\ \mathit{op}\ \mathit{exp} \\
&\Rightarrow (\ \mathit{exp}\ \mathit{op}\ \mathit{exp}\ )\ \mathit{op}\ \mathit{exp} \\
&\Rightarrow (\ \mathbf{number}\ \mathit{op}\ \mathit{exp}\ )\ \mathit{op}\ \mathit{exp} \\
&\Rightarrow (\ \mathbf{number}\ -\ \mathit{exp}\ )\ \mathit{op}\ \mathit{exp} \\
&\Rightarrow (\ \mathbf{number}\ -\ \mathbf{number}\ )\ \mathit{op}\ \mathit{exp} \\
&\Rightarrow (\ \mathbf{number}\ -\ \mathbf{number}\ )\ *\ \mathit{exp} \\
&\Rightarrow (\ \mathbf{number}\ -\ \mathbf{number}\ )\ *\ \mathbf{number}
\end{aligned}
$$

(1) *exp* → *exp op exp*
(2) *exp* → (*exp*)
(3) *exp* → **number**
(4) *op* → +
(5) *op* → -
(6) *op* → *

CFG

Context-Free Grammars

- Derivation
  - **Rightmost derivation**: the rightmost nonterminal is replaced in each step

```
exp ⇒ exp op exp
    ⇒ exp op number
    ⇒ exp * number
    ⇒ ( exp ) * number
    ⇒ ( exp op exp ) * number
    ⇒ ( exp op number ) * number
    ⇒ ( exp – number ) * number
    ⇒ ( number – number ) * number
```

```
(1) exp → exp op exp
(2) exp → ( exp )
(3) exp → number
(4) op → +
(5) op → -
(6) op → *
```

CFG

Context-Free Grammars

- Exercise

  Use leftmost/rightmost derivation to construct the token string

exp ⇒

(1) *exp* → *exp* *op* *exp*
(2) *exp* → **(** *exp* **)**
(3) *exp* → **number**
(4) *op* → **+**
(5) *op* → **-**
(6) *op* → *****

⇒ **(number * (number – number))**

Context-Free Grammars

- Language
  - the set of *token strings* obtained from ***all possible derivations*** is the ***language*** defined by the CFG

$$\mathbf{L(G) = \{ s \mid exp \Rightarrow^* } s\}$$

```
exp → exp op exp | ( exp ) | number
op → + | - | *
```

derive

`( number - number ) * number`

a legal token string

Context-Free Grammars

- Comparison to Regex
  - both use alternation, concatenation, and name
  - no repetition “*”, but recursion (“→” instead of “=”)
  - more powerful (balanced parentheses S → (S) S | ε)
  - terminals are tokens

recursion

CFG

```
exp → exp op exp | ( exp ) | number
op → + | - | *
```

tokens

regex

```
number = digit digit*
digit = 0|1|2|3|4|5|6|7|8|9
```

Context-Free Grammars

- Exercise
  - What is the language of the following grammar?

$$E \rightarrow (\ E\ )$$
$$E \rightarrow E + \mathbf{a}$$
$$E \rightarrow \mathbf{a}$$

{a, a+a, (a), (a)+a, (a+a), ...}

Context-Free Grammars

- Example
  - Nested if-statments in a C-like form

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
          | if ( exp ) statement else statement
exp → 0 | 1
```

```
other
if (0) other
if (1) other
if (0) other else other
if (0) if (0) other
...
```

Context-Free Grammars

- Example
  - A sequence of statements

```
;
s;
s;s;
s;s;s;
...
```

stmt_sequence → *stmt* **;** *stmt-sequence* | *stmt* **;**
stmt → **s** | ε

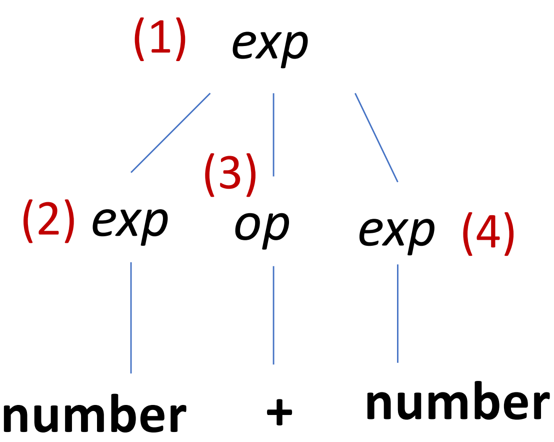
Parse/Syntax Tree

Parse Tree

- A labeled tree corresponding to a derivation
  - **Internal nodes**: nonterminals
  - **Leaf nodes:** terminals
  - **Children – parent relation**: a derivation step

(1) *exp* ⇒ *exp op exp*
(2) ⇒ **number** *op exp*
(3) ⇒ **number** **+** *exp*
(4) ⇒ **number** **+** **number**

Leftmost Derivation

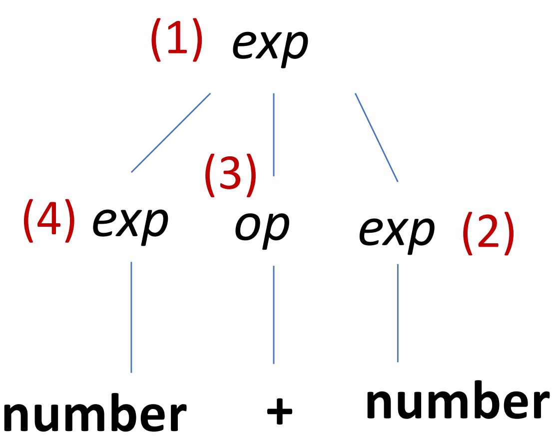


Parse Tree

- A labeled tree corresponding to a derivation
  - **Internal nodes**: nonterminals
  - **Leaf nodes:** terminals
  - **Children – parent relation**: a derivation step

(1) *exp* ⇒ *exp op exp*
(2) ⇒ *exp op* **number**
(3) ⇒ *exp* **+ number**
(4) ⇒ **number + number**

Rightmost Derivation



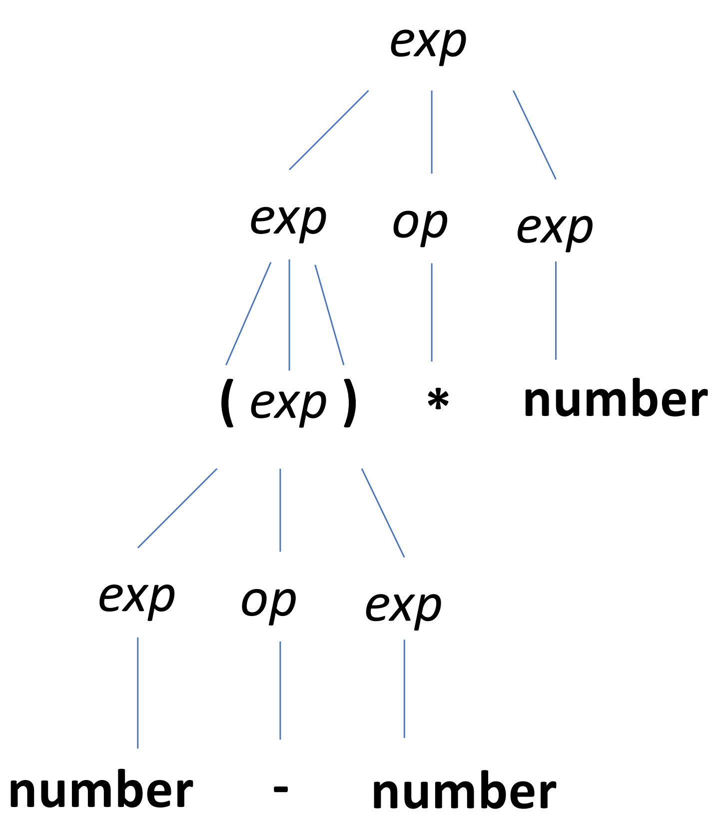
Parse Tree

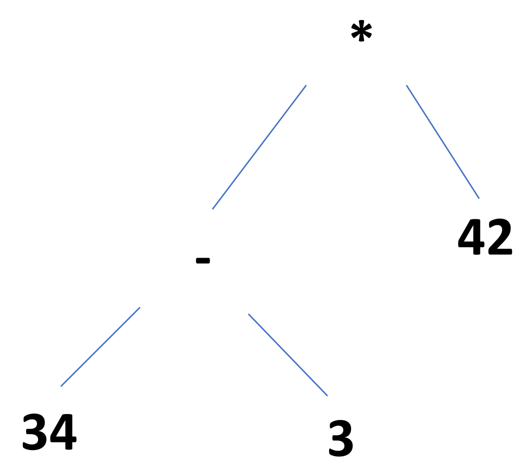
- Exercise: what is the parse tree of this derivation?

(1) *exp* ⇒ *exp* *op* *exp*
(2) ⇒ *exp* *op* **number**
(3) ⇒ *exp* * **number**
(4) ⇒ (*exp*) * **number**
(5) ⇒ (*exp* *op* *exp*) * **number**
(6) ⇒ (*exp* *op* **number**) * **number**
(7) ⇒ (*exp* - **number**) * **number**
(8) ⇒ (**number** - **number**) * **number**

Parse Tree

- **Issue:** match derivation well, but not in a concise form





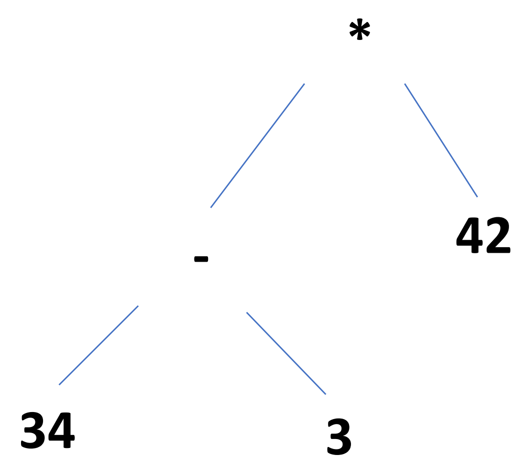
Abstract Syntax Tree (AST)

- a.k.a. Syntax Tree
  - abstract syntactic structure (may miss some syntax details)
  - token sequence may NOT be recoverable from the AST
  - still contains all necessary info for translation

```
(34 – 3) * 42
```

```
((34 – 3)) * 42
```

```
((34 – 3) * 42)
```



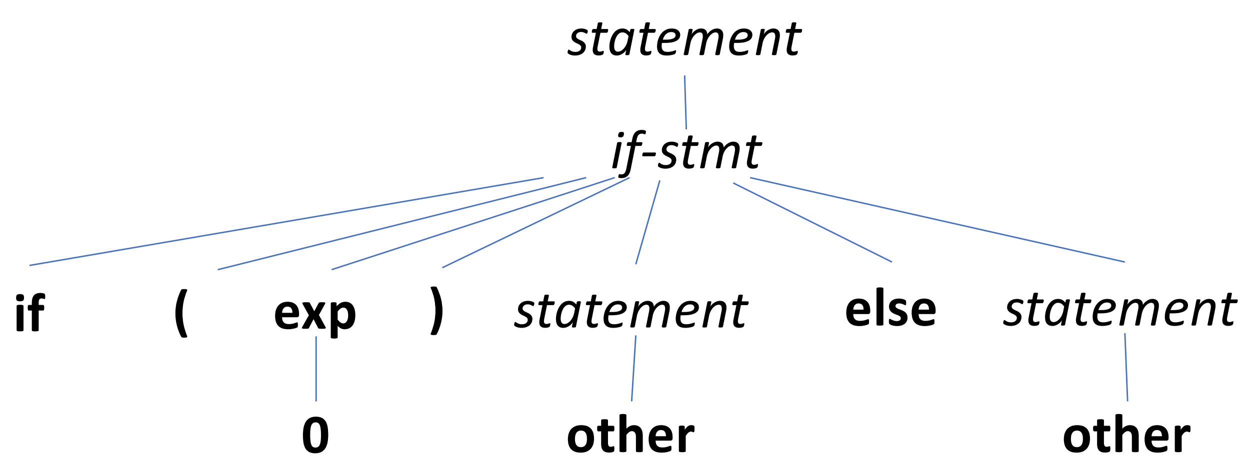
Abstract Syntax Tree (AST)

- 2nd Example

```
if (0) other else other
```

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
        | if ( exp ) statement else statement
exp → 0 | 1
```

Parse Tree



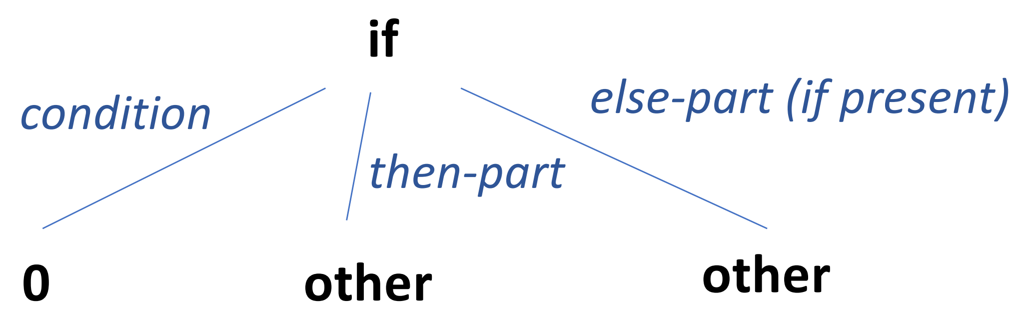
Abstract Syntax Tree (AST)

- 2nd Example

```
if (0) other else other
```

statement → *if-stmt* | **other**
if-stmt → **if** **(** *exp* **)** *statement*
      | **if** **(** *exp* **)** *statement* **else** *statement*
exp → **0** | **1**

AST



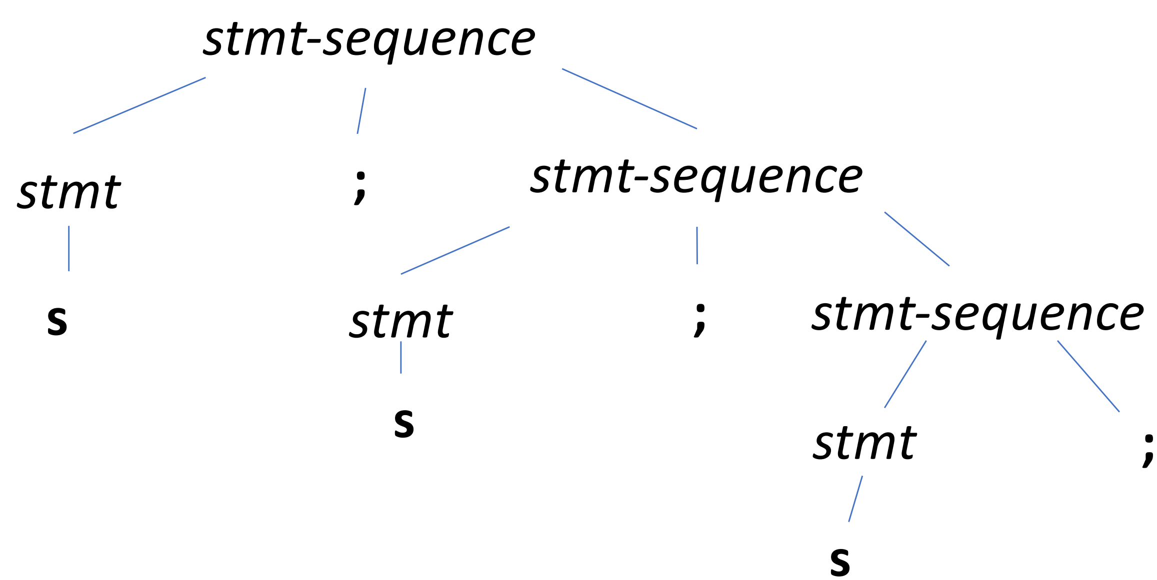
Abstract Syntax Tree (AST)

- 3rd Example

```
s;s;s;
```

```
stmt_sequence → stmt; stmt-sequence | stmt;
stmt → s | ε
```

Parse Tree



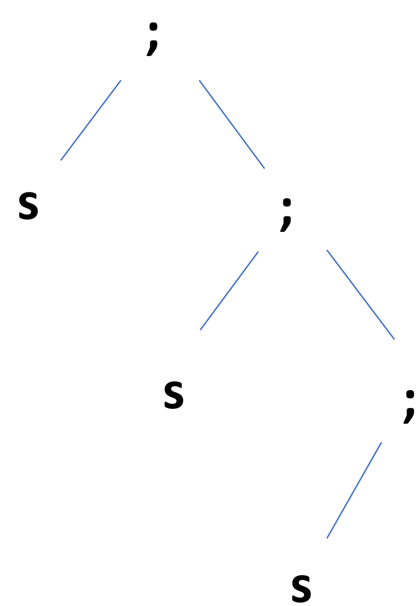
Abstract Syntax Tree (AST)

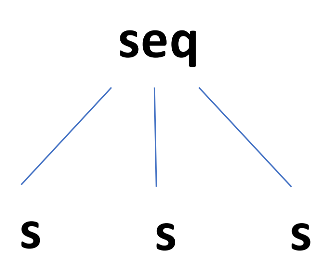
- 3rd Example

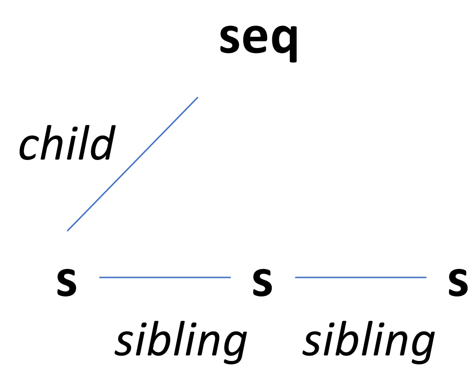
```
s;s;s;
```

stmt_sequence → *stmt; stmt-sequence* | *stmt;*
stmt → **s** | ε

AST







(leftmost-child right-sibling)

Ambiguous Grammar

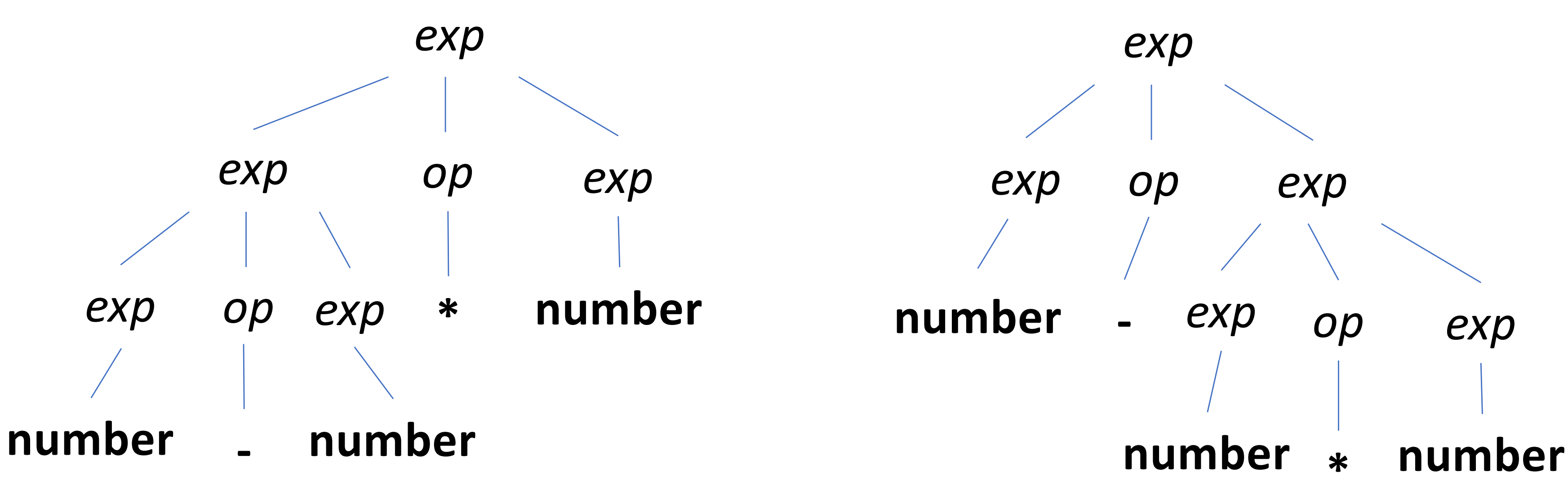
Ambiguous Grammar

- A grammar allowing a string to have

  more than one parse tree

```
exp → exp op exp | ( exp ) | number
op → + | - | * | /
```

```
34 – 3 * 42
```



Ambiguous Grammar

- A grammar allowing a string to have

  more than one parse tree

```
exp → exp op exp | ( exp ) | number
op → + | - | * | /
```

`34 – 3 * 42`

Ambiguous Grammar

- Problem – multiple leftmost / multiple rightmost derivations

```
exp → exp op exp | ( exp ) | number
op → + | - | * | /
```

```
34 – 3 * 42
```

$exp \Rightarrow exp\ op\ exp$
$\Rightarrow exp\ op\ exp\ op\ exp$
$\Rightarrow$ **number** $op\ exp\ op\ exp$
$\Rightarrow$ **number** – $exp\ op\ exp$
$\Rightarrow$ **number** – **number** $op\ exp$
$\Rightarrow$ **number** – **number** * exp
$\Rightarrow$ **number** – **number** * **number**

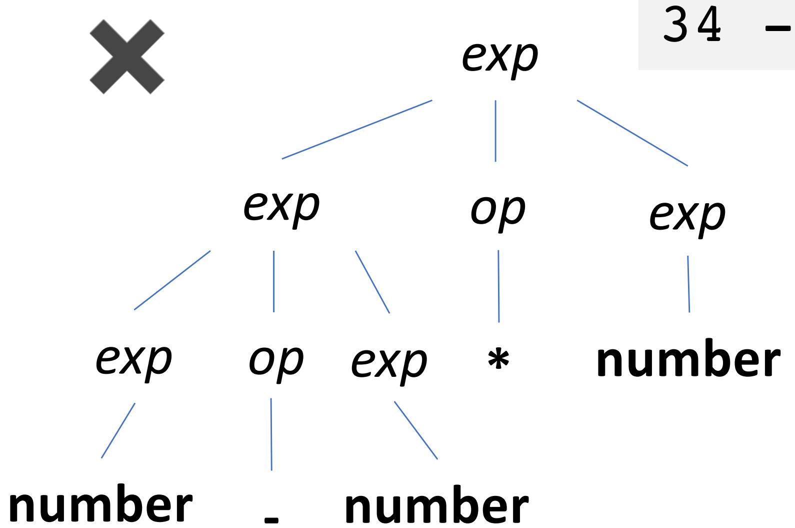
$exp \Rightarrow exp\ op\ exp$
$\Rightarrow$ **number** $op\ exp$
$\Rightarrow$ **number** – exp
$\Rightarrow$ **number** – $exp\ op\ exp$
$\Rightarrow$ **number** – **number** $op\ exp$
$\Rightarrow$ **number** – **number** * exp
$\Rightarrow$ **number** – **number** * **number**

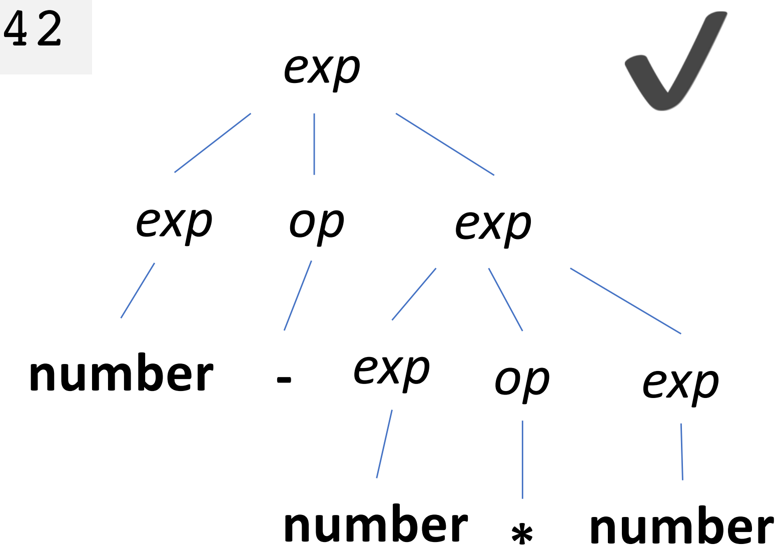
Two leftmost derivations

Ambiguous Grammar

- **Disambiguation rule needed** → * has **precedence** over –

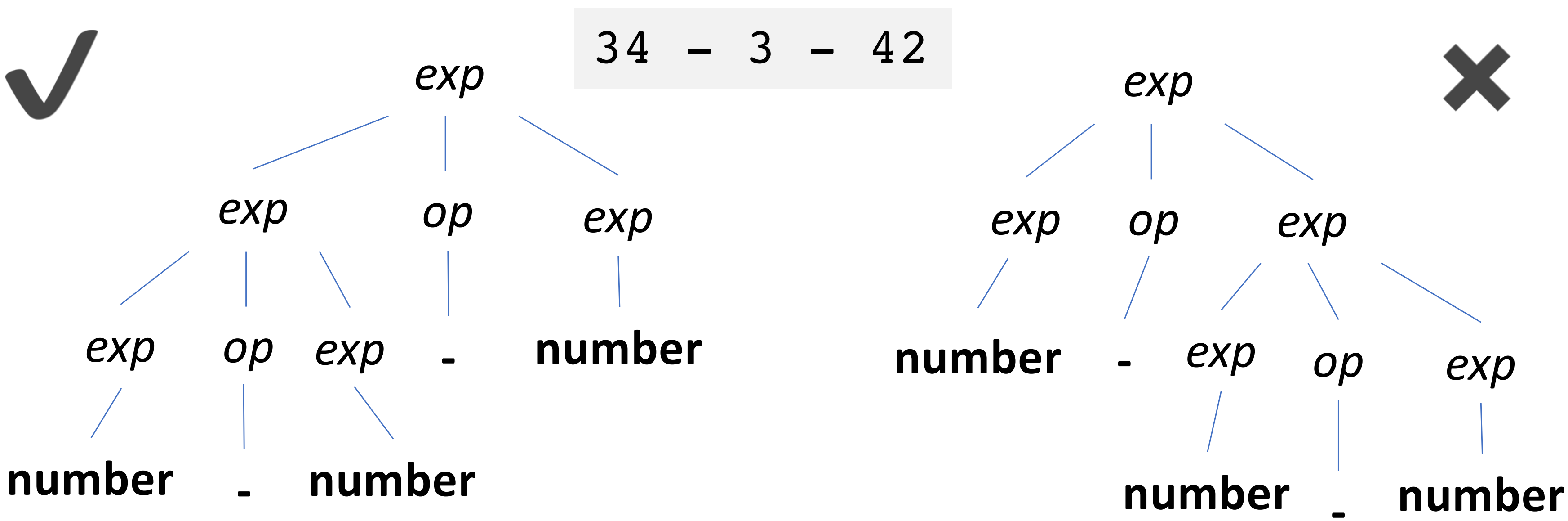
`34 – 3 * 42`





Ambiguous Grammar

- **Disambiguation rule needed →** – is **left associative**: left to right calculation



Ambiguous Grammar

- **Add disambiguation rules to grammars**

`*` **and** `/` have **precedence** over `–` and `+`

`–` and `/` are **left associative**

```
exp → exp op exp | ( exp ) | number
op → + | – | * | /
```

Ambiguous Grammar

Precedence Rules

Rewrite the grammar

group operators of same precedence;

write a different production rule for each precedence group

```
exp → exp op exp | ( exp ) | number
op → + | - | * | /
```

⬇

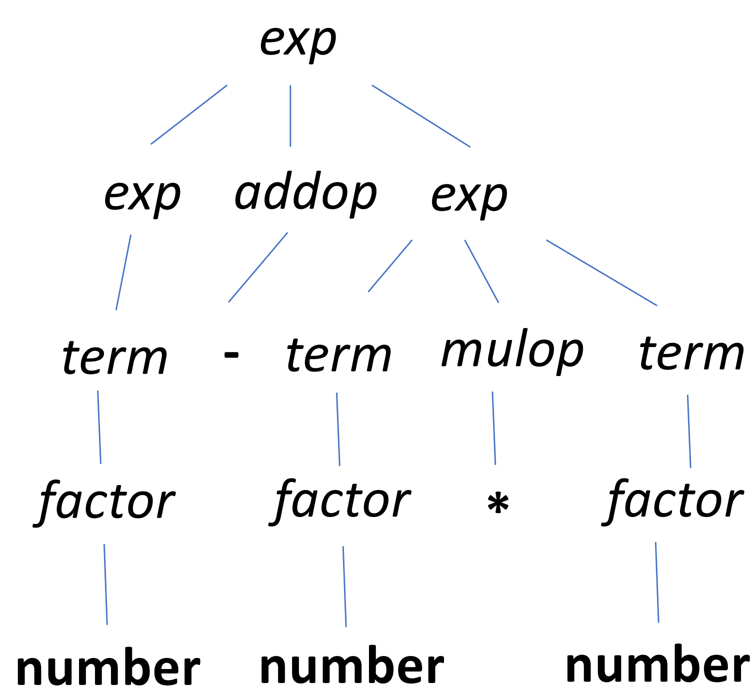
```
exp → exp addop exp | term
addop → + | -
term → term mulop term | factor
mulop → * | /
factor → ( exp ) | number
```

Ambiguous Grammar

- What is the parse tree with the new grammar?

```
34 – 3 * 42
```

```
exp → exp addop exp | term
addop → + | -
term → term mulop term | factor
mulop → * | /
factor → ( exp ) | number
```



Ambiguous Grammar

- Some ambiguity still exists due to associativity rules

```
34 - 3 - 42
```

```
exp → exp addop exp | term
addop → + | -
term → term mulop term | factor
mulop → * | /
factor → ( exp ) | number
```

recursion occuring on both sides

Ambiguous Grammar

- Associativity Constraints
  - **Rewrite the grammar**

    Replace one of the recurisons with the base case

```
exp → exp op exp | ( exp ) | number
op → + | - | * | /
```

```
exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number
```

Ambiguous Grammar

- Example parse tree for the new grammar

```
34 - 3 - 42
```

```
exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number
```

Ambiguous Grammar

- Revisit the if-statments

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
         | if ( exp ) statement else statement
exp → 0 | 1
```

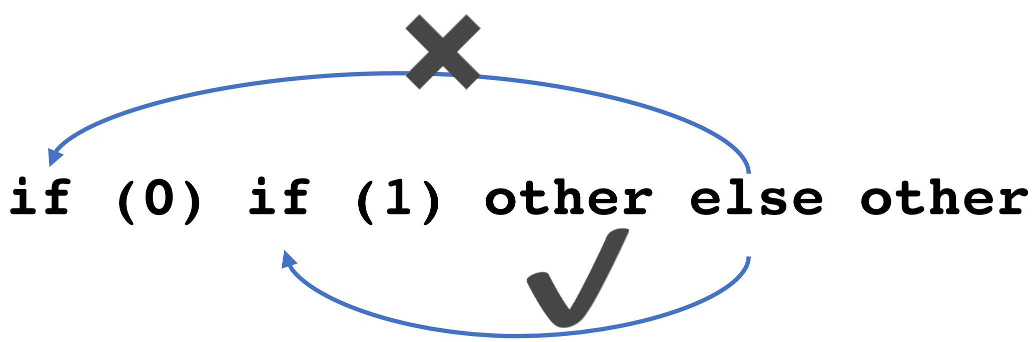
```
if (0) if (1) other else other
```

“Dangling Else” Problem

Ambiguous Grammar

- Revisit the if-statments

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
            | if ( exp ) statement else statement
exp → 0 | 1
```



disambiguation rule: <u>most closely nested rule</u>

Ambiguous Grammar

- Revisit the if-statments

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
        | if ( exp ) statement else statement
exp → 0 | 1
```

```
statement → matched-stmt | unmatched-stmt
matched-stmt → if (exp) matched-stmt else matched-stmt
             | other
unmatched-stmt → if (exp) statement
               | if (exp) matched-stmt else unmatched-stmt
exp → 0 | 1
```

Harder to write (often not used in practice)

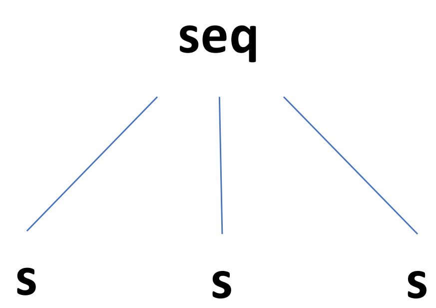
Ambiguous Grammar

- <u>Harmless Ambiguity</u>
  - ambiguous grammar may always produce unique AST

    (but may produce different parse trees)

stmt_sequence → *stmt-sequence* **;** *stmt-sequence* | *stmt* **;**
stmt → **s** | ε

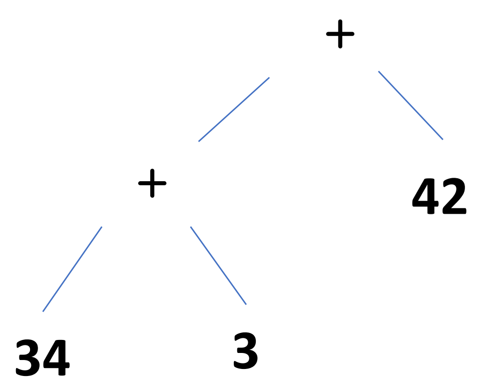
```
s; s; s;
```

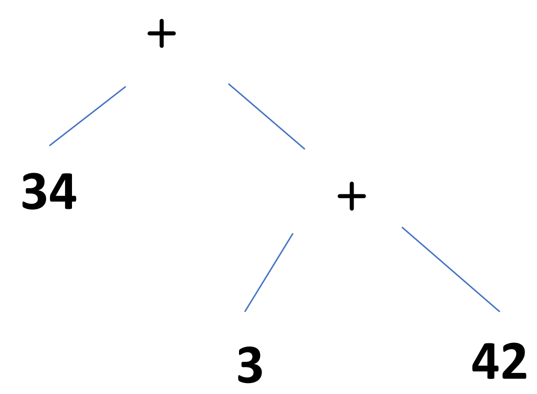


Ambiguous Grammar

- <u>Harmless Ambiguity</u>
  - even for ambiguous grammar producing different ASTs

```
34 + 3 + 42
```





<u>Both carry the same semantics</u>