Compiler Design

Lecture 6: Syntax Analysis (Parsing) II

Sahar Selim

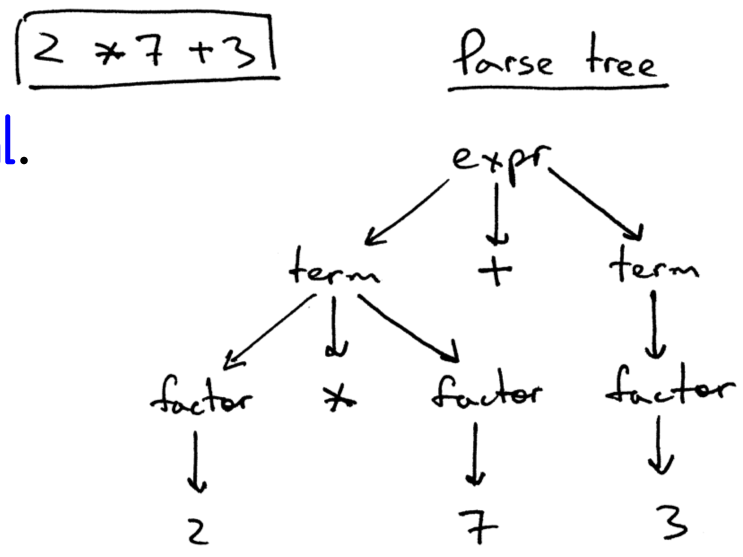
Agenda

1. Parse Tree
2. Abstract Syntax Tree
3. Ambiguous grammars
4. Dangling Else

1 Derivations & Parse Trees

Parsing Tree

- A **parse tree** is a tree encoding the steps in a derivation.
  - Each internal node is labeled with a nonterminal.
  - Each leaf node is labeled with a terminal.
  - Reading the leaves from left to right gives the string that was produced.



- The children of each internal node represent the replacement of the associated non-terminal in one step of the derivation.

Parse Tree

- A **leftmost derivation** is a derivation in which each step expands the leftmost nonterminal (corresponds to preorder).

- A **rightmost derivation** is a derivation in which each step expands the rightmost nonterminal (corresponds to postorder).

Parsing Tree

- Left most derivation

(1) *exp* => *exp op exp*
(2) => ***number*** *op exp*
(3) => ***number*** + *exp*
(4) => *number* + *number*

- Right most derivation

(1) *exp* => *exp op exp*
(2) => *exp op number*
(3) => *exp* + *number*
(4) => *number* + *number*

Parsing Tree

- Neither leftmost nor rightmost derivation

  (1) *exp* => *exp op exp*

  (2) => *exp* + *exp*

  (3) => *number* + *exp*

  (4) => *number* + *number*

- Generally, a parse tree corresponds to many derivations
  - represent the same basic structure for the parsed string of terminals.
- It is possible to distinguish particular derivations that are uniquely associated with the parse tree.

Derivations

- Derivations do not uniquely represent the structure of the strings
  - There are many derivations for the same string.

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

  op → + | – | *

- The string of tokens:

  (*number* - *number*) * *number*

- There exists two different derivations for above string

Rightmost Derivations

Derivation steps **use a different arrow (=>)** from the arrow meta-symbol in the grammar rules (→).

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

*(number - number) * number*

(1) *exp* => *exp op exp*	[*exp* → *exp op exp*]
(2)=> *exp op number*	[*exp* → *number*]
(3)=> *exp* * *number*	[*op* → *]
(4) => (*exp*) * *number*	[*exp* → (*exp*)]
(5) =>(*exp op exp*) * *number*	[*exp* → *exp op exp*]
(6)=> (*exp op* number) * *number*	[*exp* → *number*]
(7)=> (*exp* - number) * *number*	[*op* → -]
(8)=> (number - number) * *number*	[*exp* → *number*]

Leftmost Derivations

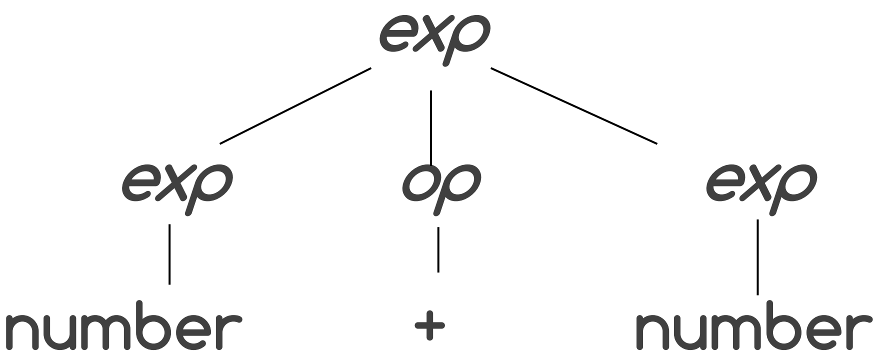
exp → exp op exp | (exp) | number
op → + | – | *

*(number - number) * number*

(1) *exp* => *exp op exp*	[*exp* → *exp op exp*]
(2)=> (*exp*) *op exp*	[*exp* → (*exp*)]
(3)=> (*exp op exp*) *op exp*	[*exp* → *exp op exp*]
(4) => (*number op exp*) *op exp*	[*exp* → *number*]
(5) =>(*number* - *exp*) *op exp*	[*op* → -]
(6)=> (*number* - *number*) *op exp*	[*exp* → *number*]
(7)=> (*number* - *number*) * *exp*	[*op* → *]
(8)=>(*number* - *number*) * *number*	[*exp* → number]

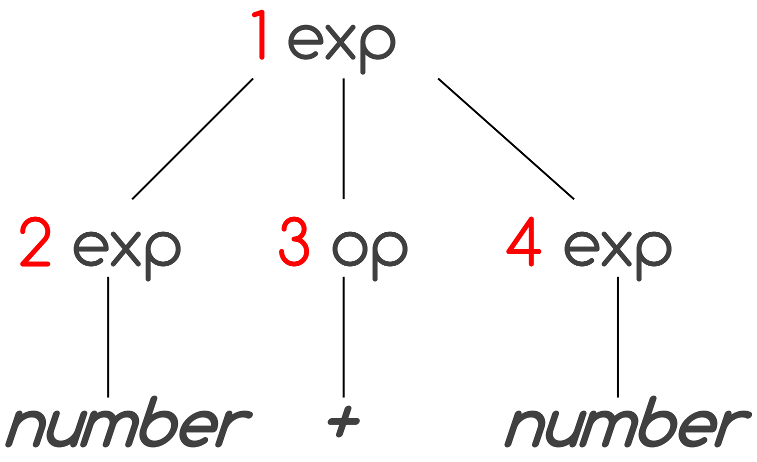
Parsing Tree

- The example:
  - *exp* => *exp op exp* => number *op exp* => number + *exp* => number + number
- Corresponding to the parse tree:



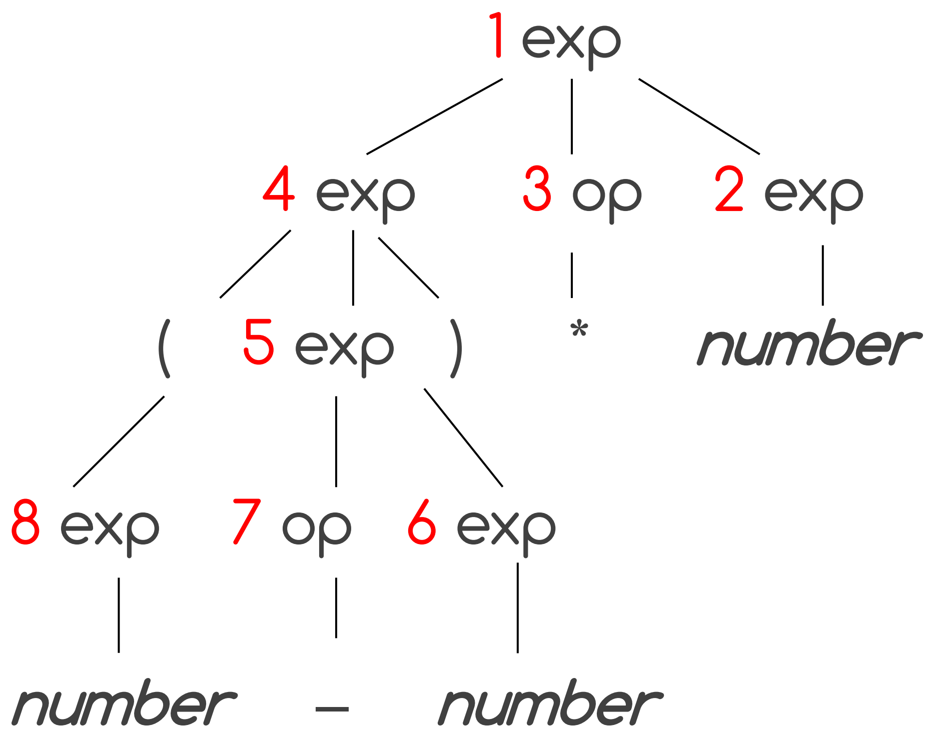
Parsing Tree

- The parse tree corresponds to the first derivation (Left-most).



Example: The expression (34-3)*42

- The parse tree for the above arithmetic expression



Right-most Derivation

Parse Trees:
int * (int + int)

$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

E

Parse Trees:
int * (int + int)

$$E \to E \; Op \; E \mid \texttt{int} \mid (E)$$
$$Op \to + \mid * \mid - \mid /$$

E

E

Parse Trees: int * (int + int)

$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

$\boxed{E}$

E

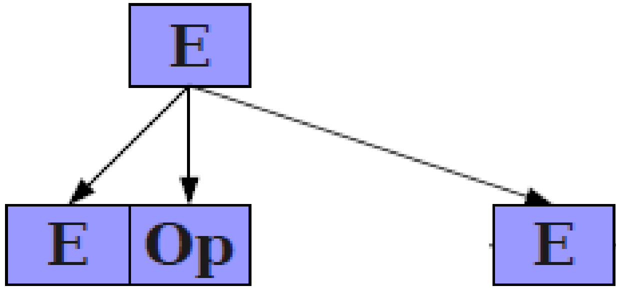
⇒ E Op E

Parse Trees:
int * (int + int)

$$E \to E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \to + \mid * \mid - \mid /$$

E

$\Rightarrow E\ Op\ E$



Parse Trees: int * (int + int)

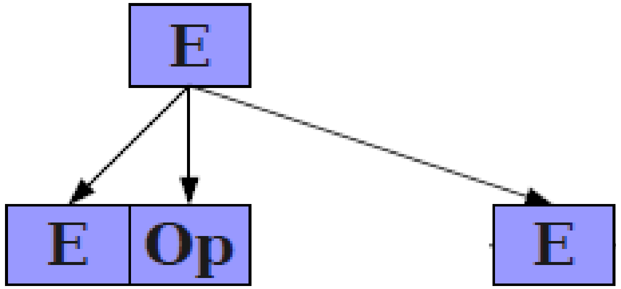
$E \rightarrow E \; Op \; E \mid \mathtt{int} \mid (E)$

$Op \rightarrow + \mid * \mid - \mid /$

E

$\Rightarrow E \; Op \; E$

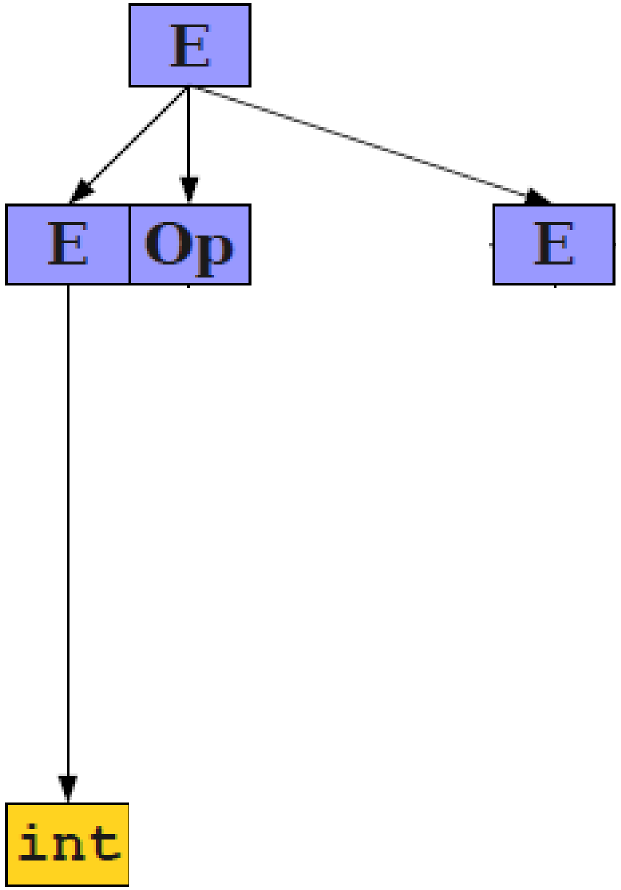
$\Rightarrow \mathtt{int} \; Op \; E$



Parse Trees: int * (int + int)

$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$
$Op \rightarrow + \mid * \mid - \mid /$

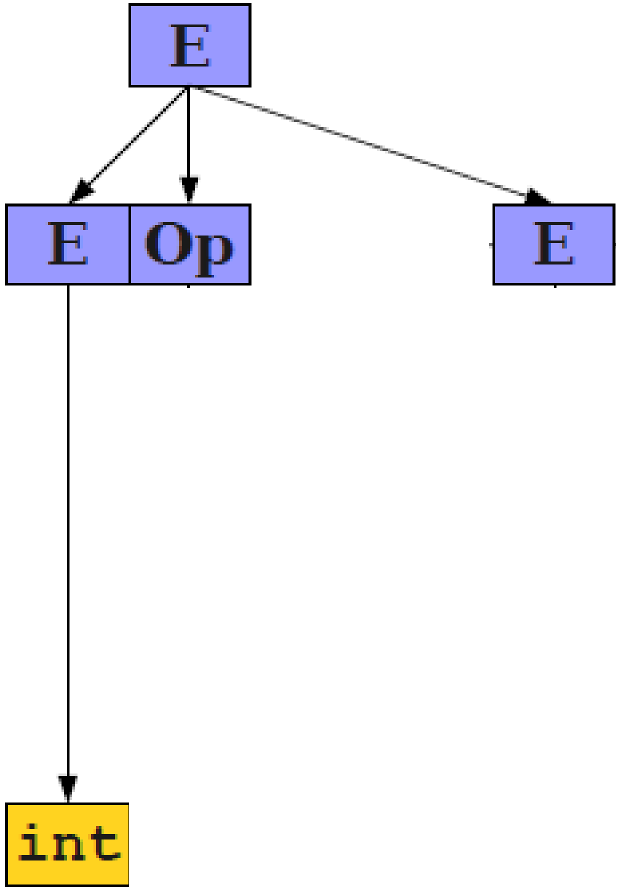
E
⇒ E Op E
⇒ int Op E



Parse Trees: int * (int + int)

$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

E
⇒ E Op E
⇒ int Op E
⇒ int * E



Parse Trees:
int * (int + int)

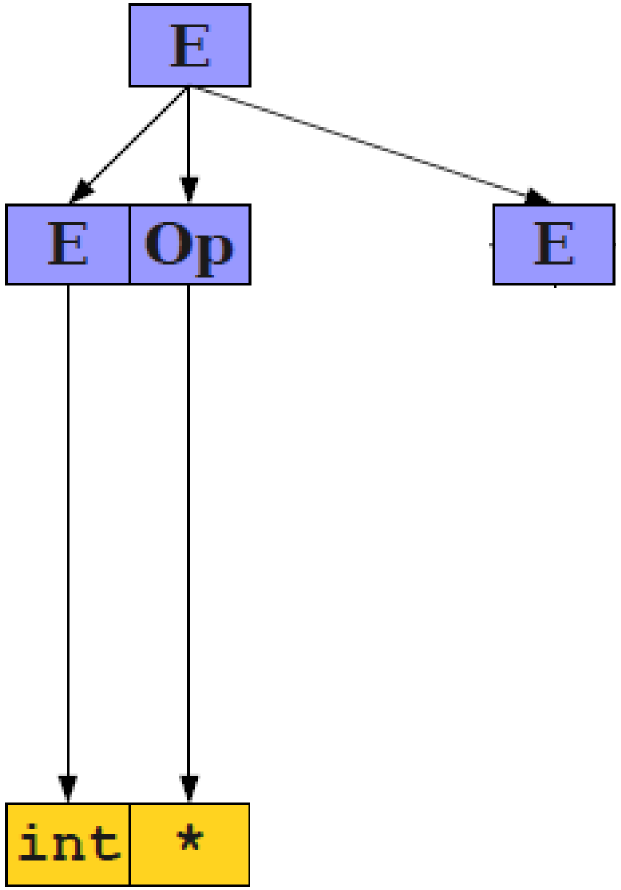
$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

E

⇒ E Op E

⇒ int Op E

⇒ int * E



Parse Trees:
int * (int + int)

$$E \rightarrow E\ Op\ E \mid \mathtt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

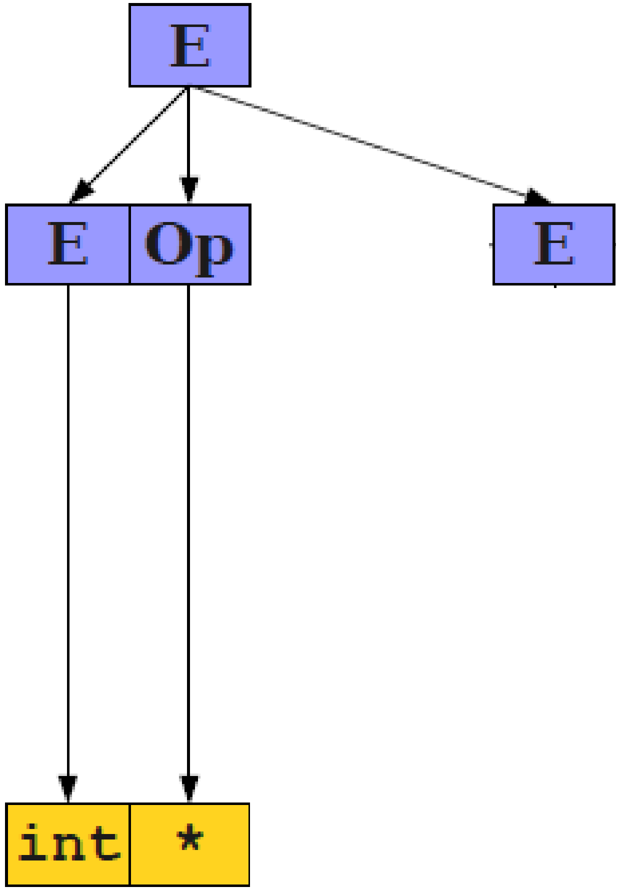
E

⇒ E Op E

⇒ int Op E

⇒ int * E

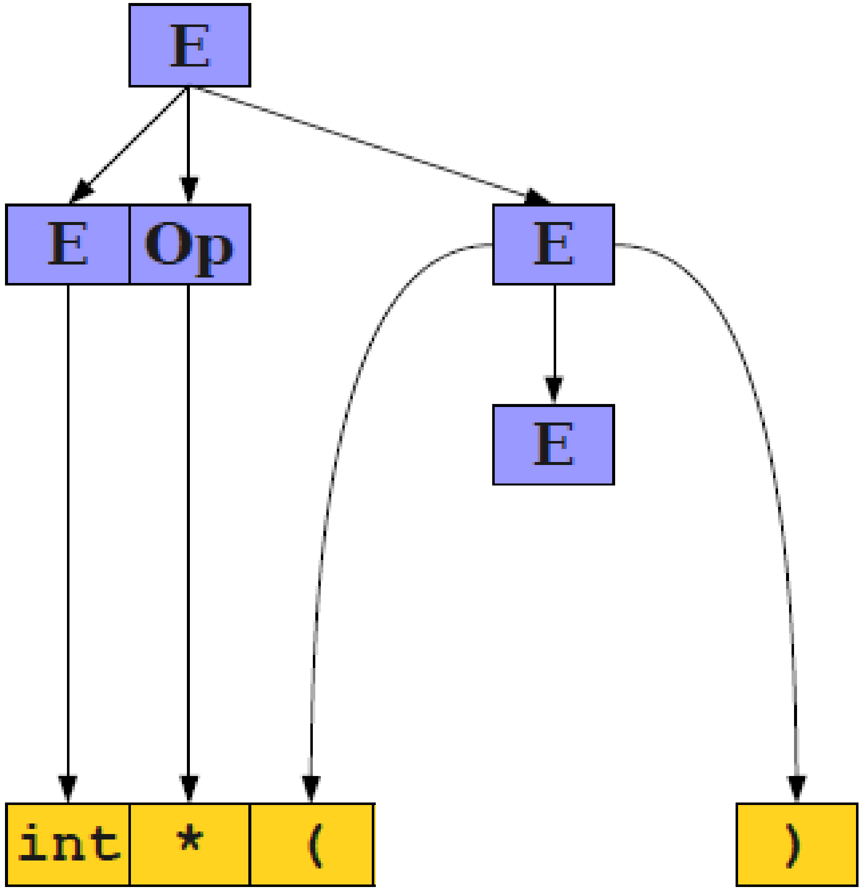
⇒ int * (E)



Parse Trees:
int * (int + int)

$E \rightarrow E \; Op \; E \mid$ `int` $\mid (E)$
$Op \rightarrow + \mid * \mid - \mid /$

E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)



Parse Trees: int * (int + int)

$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

E

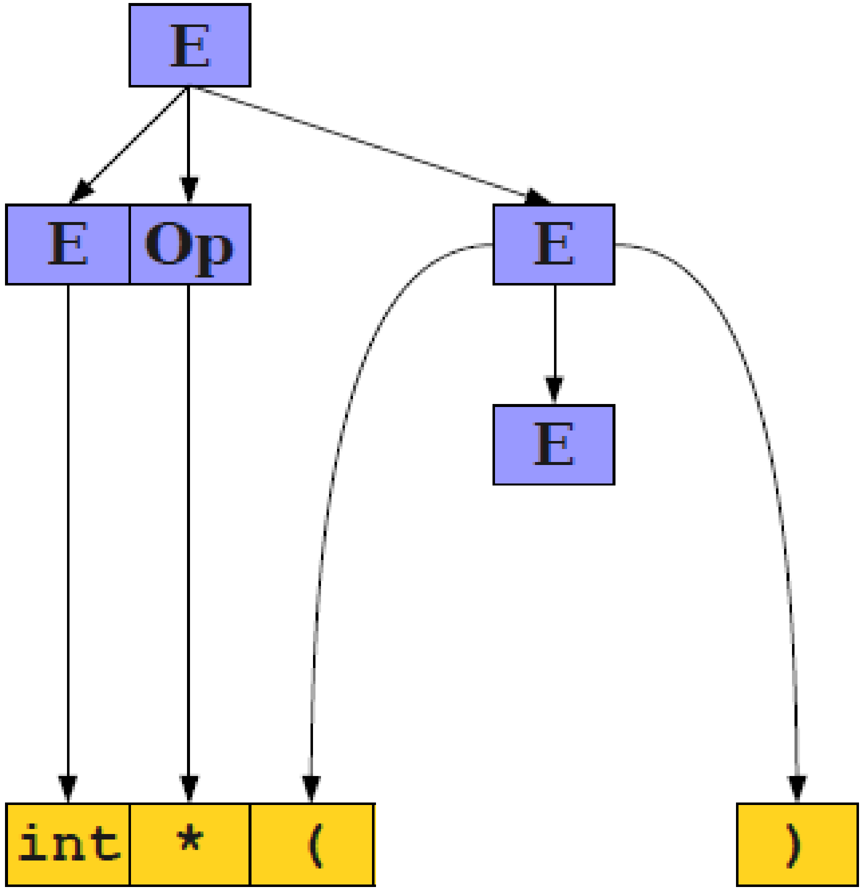
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



Parse Trees: int * (int + int)

E

⇒ E Op E

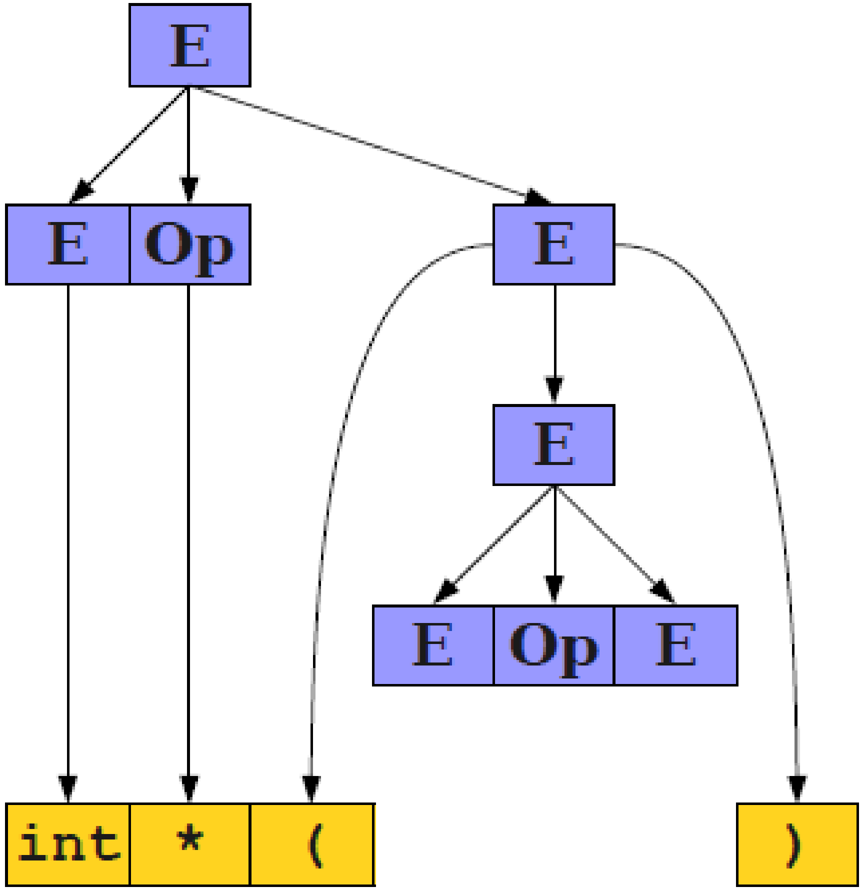
⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

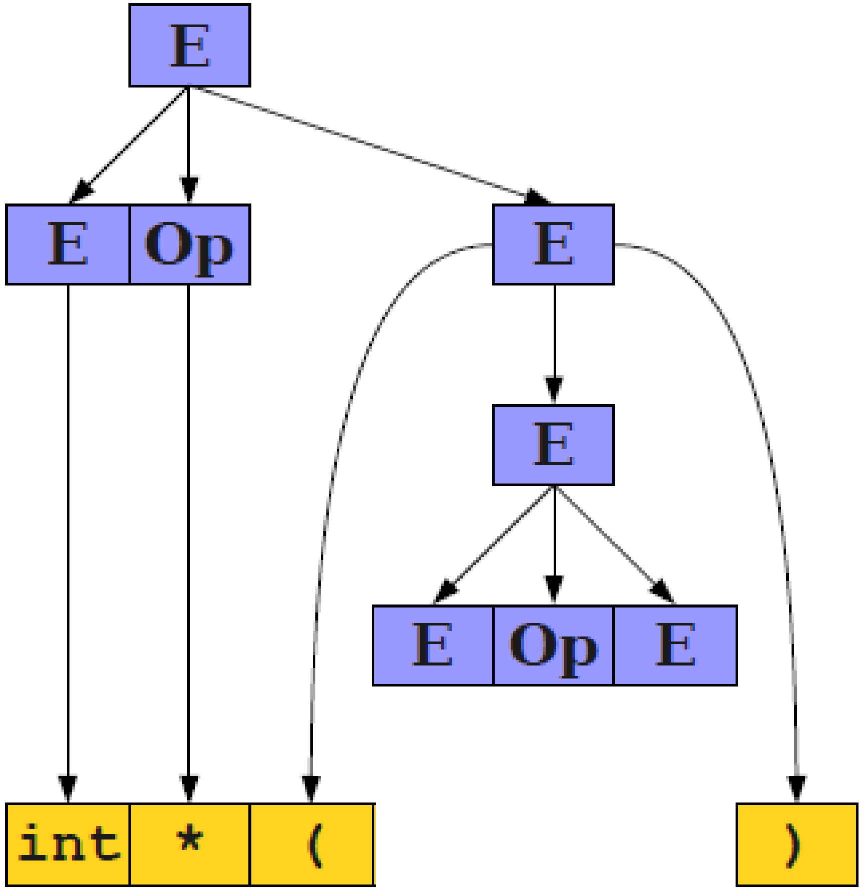
E → E Op E | int | (E)
Op → + | * | - | /



Parse Trees: int * (int + int)

E → E Op E | int | (E)
Op → + | * | - | /

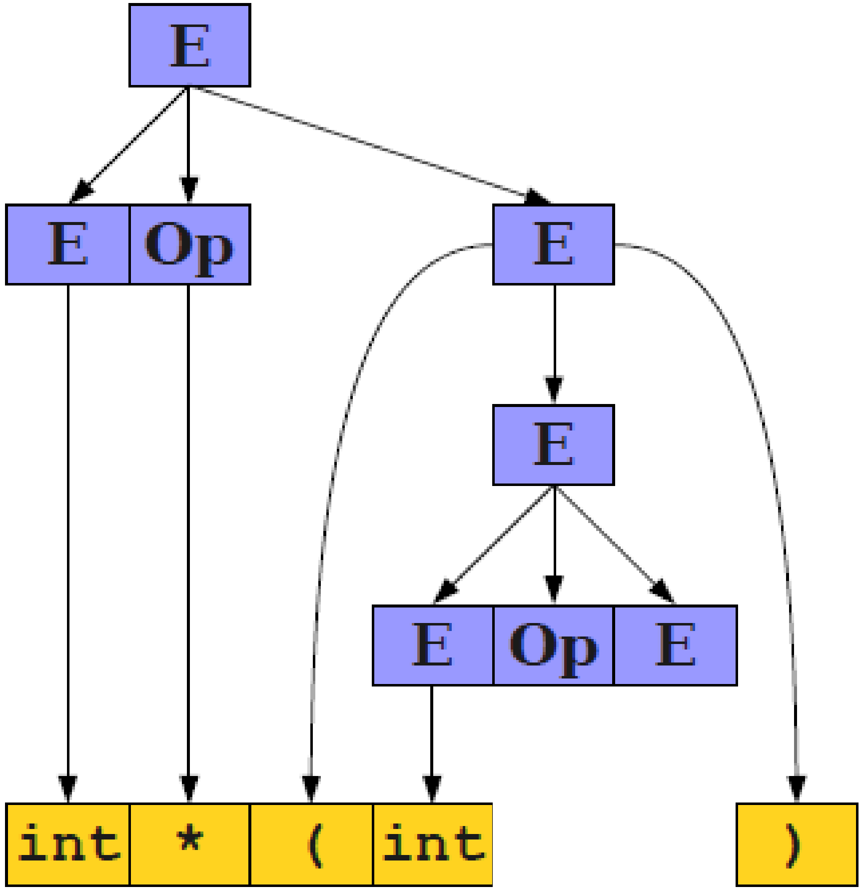
E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)



Parse Trees: int * (int + int)

$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

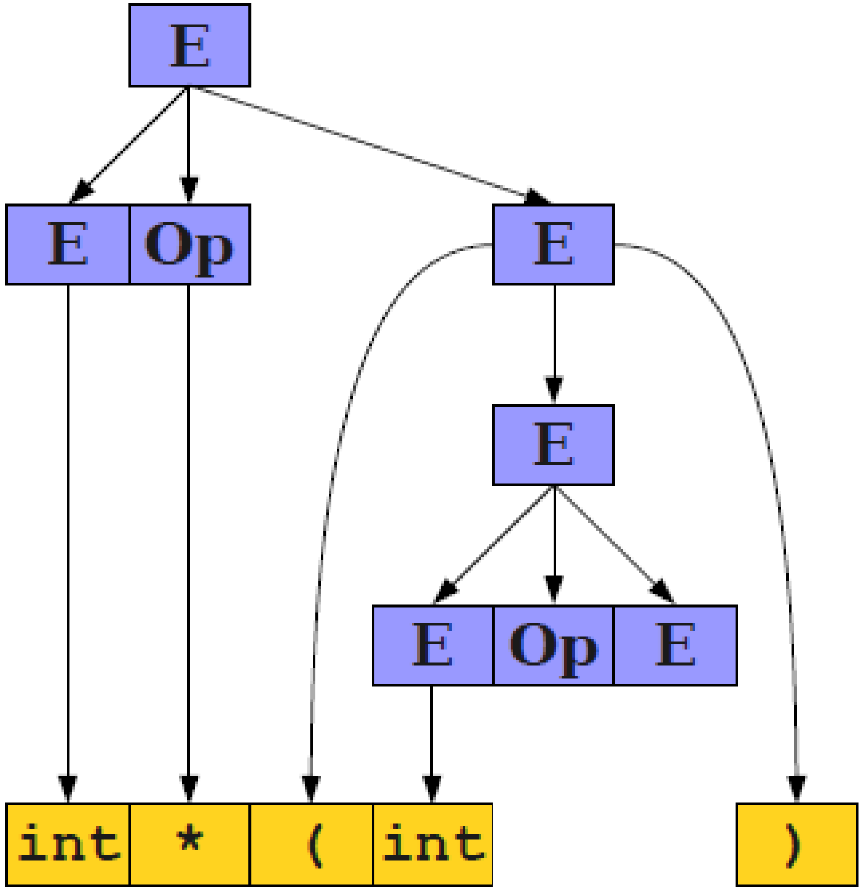
E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)



Parse Trees: int * (int + int)

E → E Op E | int | (E)
Op → + | * | - | /

E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)



Parse Trees: int * (int + int)

E

⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

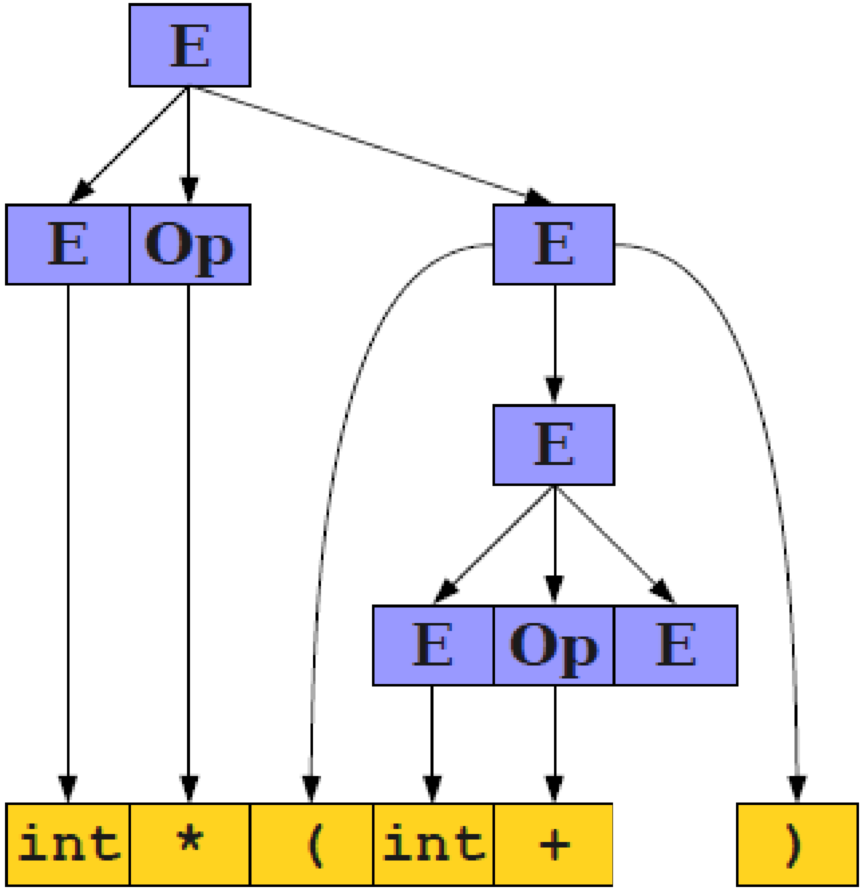
⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

E → E Op E | int | (E)

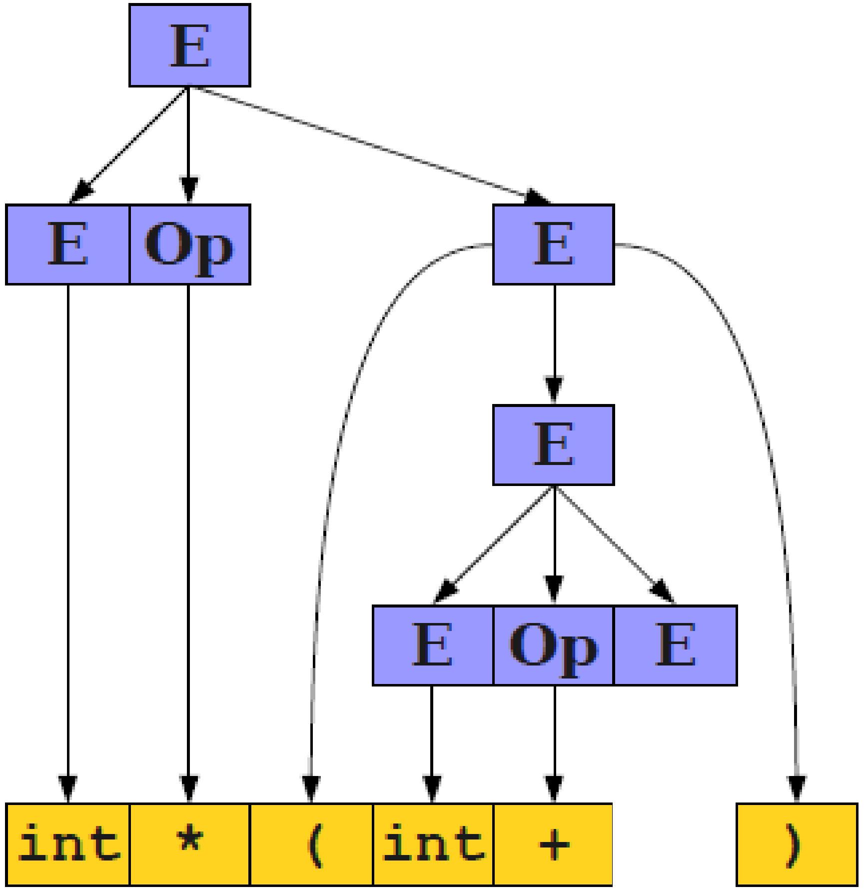
Op → + | * | - | /



Parse Trees: int * (int + int)

$$E \rightarrow E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \rightarrow + \mid * \mid - \mid /$$

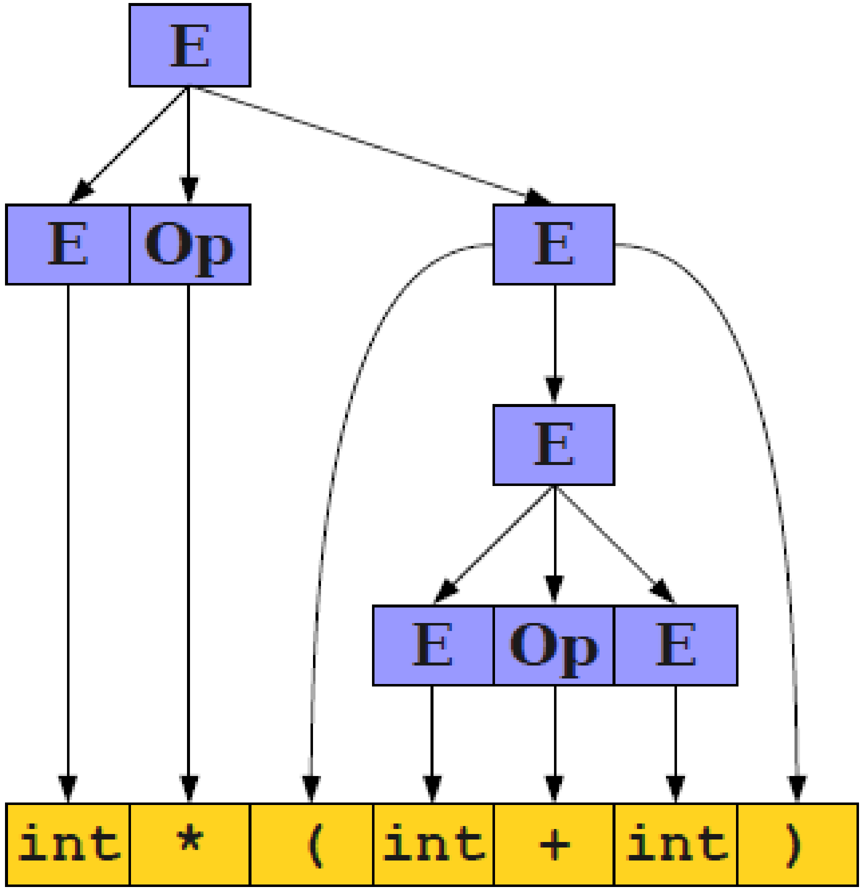
E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)
⇒ int * (int + int)



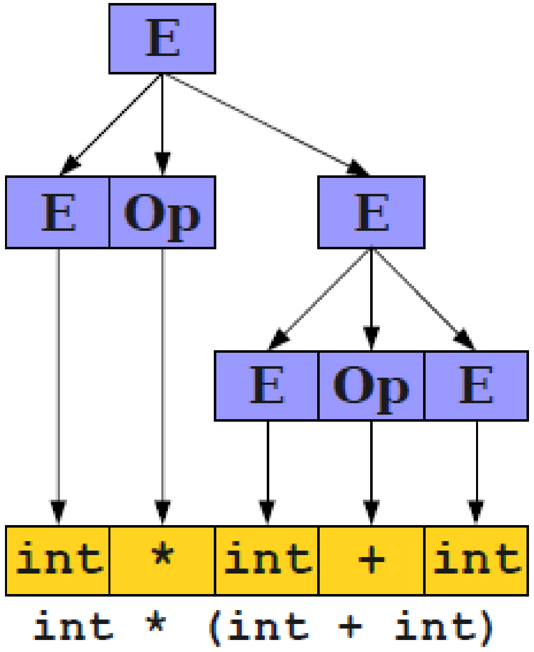
Parse Trees: int * (int + int)

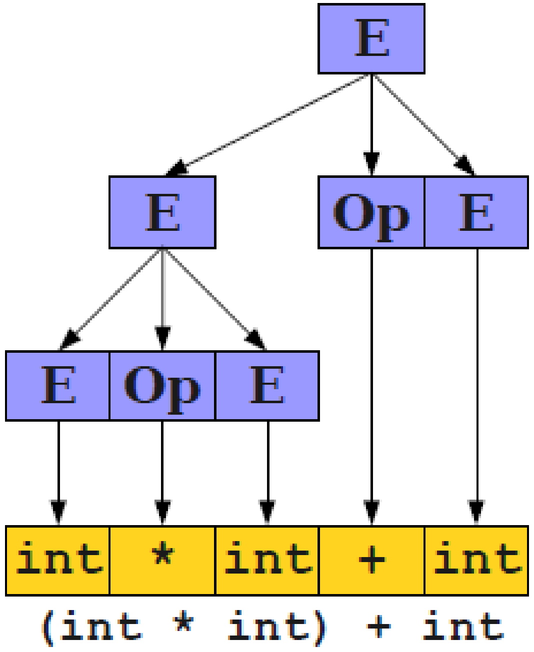
E → E Op E | int | (E)
Op → + | * | - | /

E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)
⇒ int * (int + int)



A Serious Problem





Ambiguous Grammar

$E \rightarrow E\ Op\ E \mid \texttt{int}$
$Op \rightarrow + \mid * \mid - \mid /$

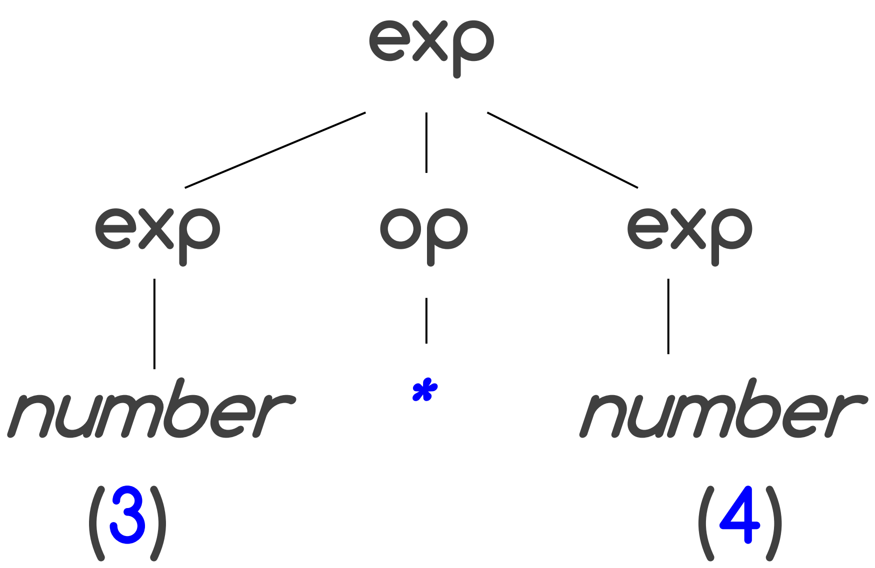
Parse Trees (Recap)

- Goal of syntax analysis:
  - Recover the structure described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the input string.
- A **parse tree** is a tree, encoding the steps in a **derivation**.
- **Internal nodes** represent **nonterminal** symbols used in the production.
- In order walk of the leaves contains the generated string.
- Encodes **what productions** are used, **NOT the order** in which those productions are applied.

2 Abstract Syntax Trees

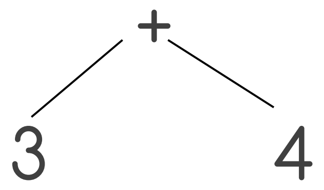
Why Abstract Syntax-Tree ?

- The parse tree contains more information than is absolutely necessary for a compiler
- For the example: 3*4



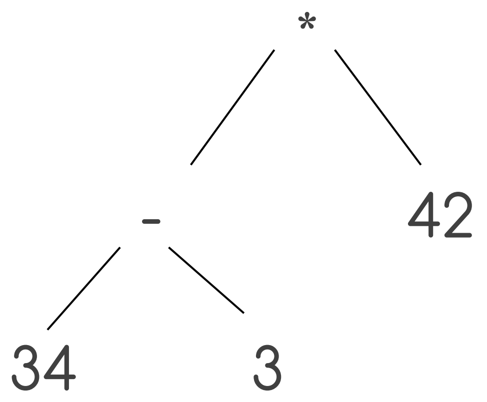
Why Abstract Syntax-Tree ?

- The principle of syntax-directed translation
  - The meaning, or semantics, of the string 3+4 should be directly related to its syntactic structure as represented by the parse tree.
- In this case, the parse tree should imply that the value 3 and the value 4 are to be added.
- A much simpler way to represent this same information, namely, as the tree



Tree for expression (34-3)*42

- The expression (34-3)*42 whose parse tree can be represented more simply by the tree:



- The parentheses tokens have actually disappeared
  - still represents precisely the semantic content of subtracting 3 from 34, and then multiplying by 42

Example 1

- The grammar for simplified if-statements

statement → *if-stmt* | **other**

if-stmt → **if (** *exp* **)** *statement*

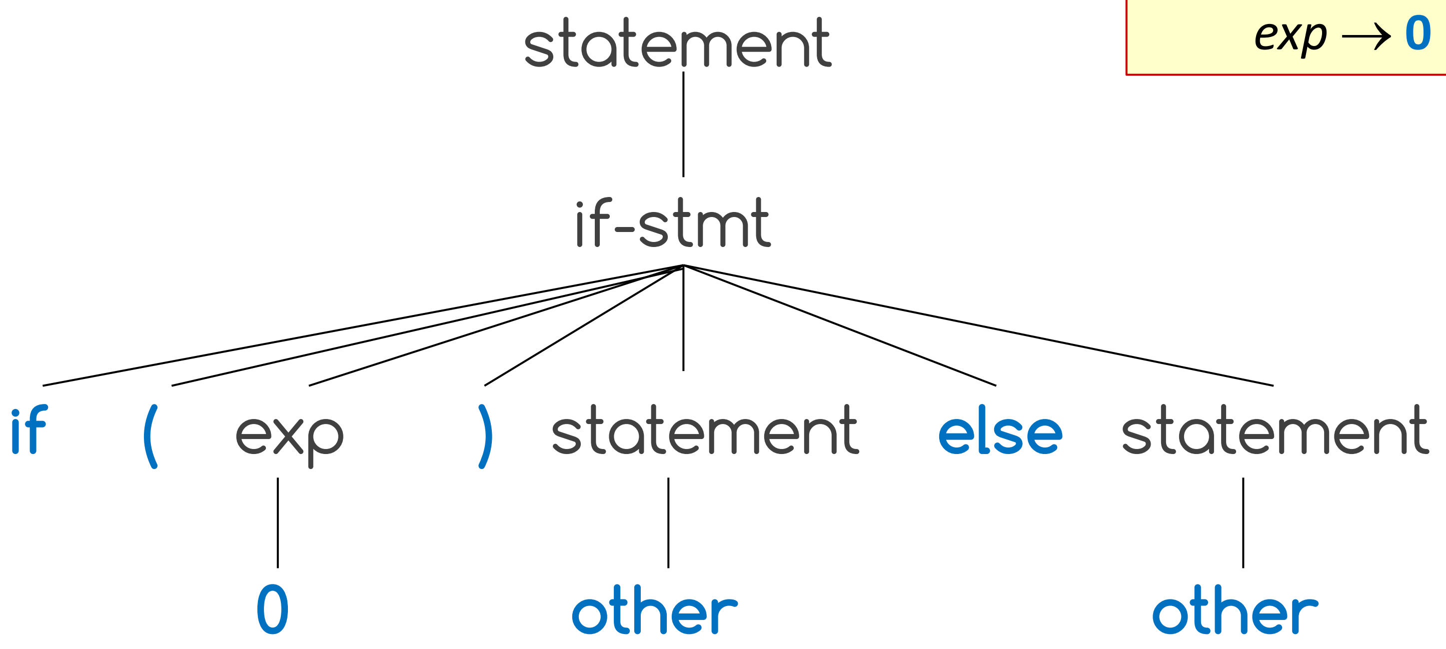
| **if (** *exp* **)** *statement* **else** *statement*

exp → **0** | **1**

Example 1

- The parse tree for the string:
  - if (0) other else other

$$
\begin{aligned}
&\textit{statement} \rightarrow \textit{if-stmt} \mid \textbf{other} \\
&\textit{if-stmt} \rightarrow \textbf{if} \textbf{ (} \textit{exp} \textbf{) } \textit{statement} \\
&\qquad\qquad \mid \textbf{if} \textbf{ (} \textit{exp} \textbf{) } \textit{statement} \textbf{ else } \textit{statement} \\
&\textit{exp} \rightarrow \textbf{0} \mid \textbf{1}
\end{aligned}
$$



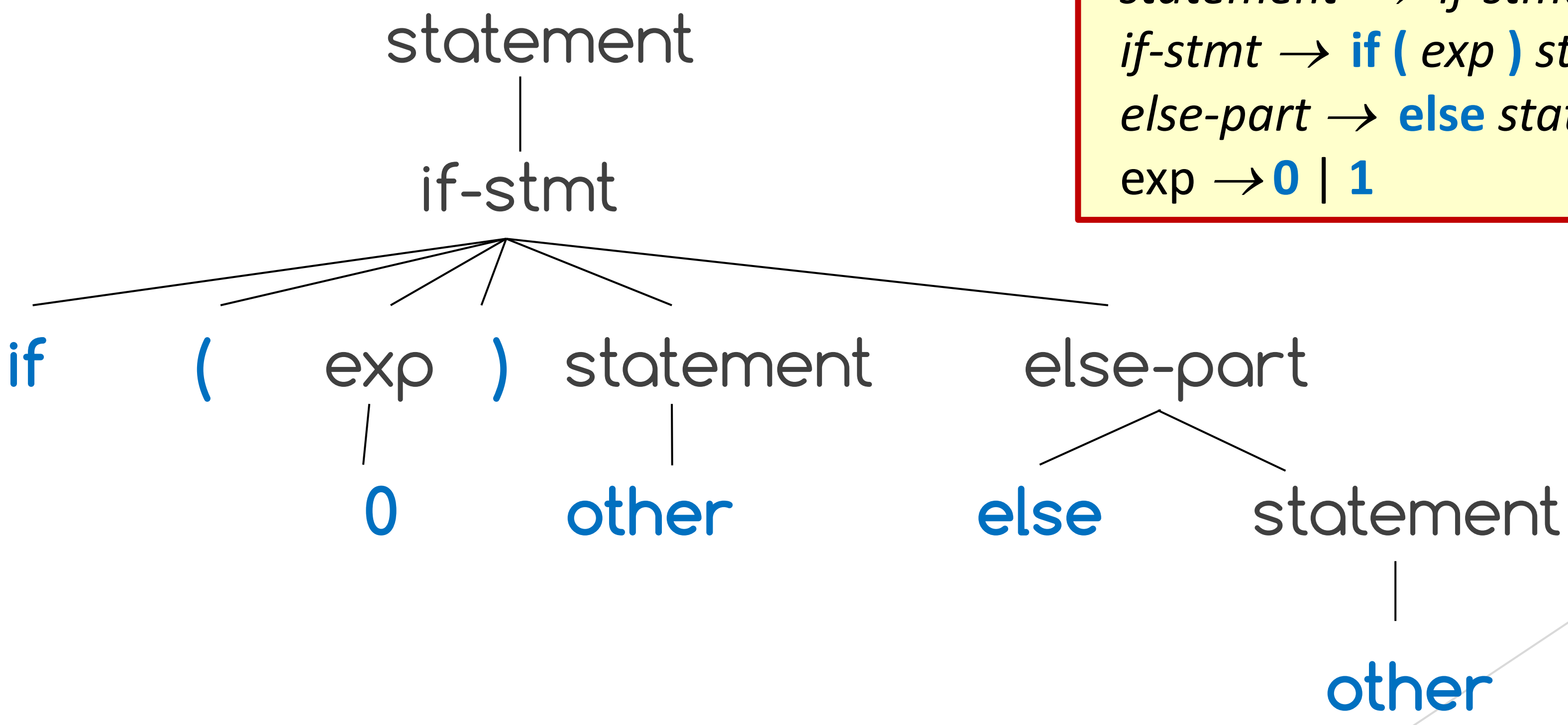
Example 2

- Using another grammar of ***if-statement***

statement $\rightarrow$ *if-stmt* | ***other***
if-stmt $\rightarrow$ **if (** *exp* **)** *statement else-part*
else-part $\rightarrow$ **else** *statement* | ε
exp $\rightarrow$ **0 | 1**

Example 2

- This same string has the following parse tree:
  - if (0) other else other



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

Example 3

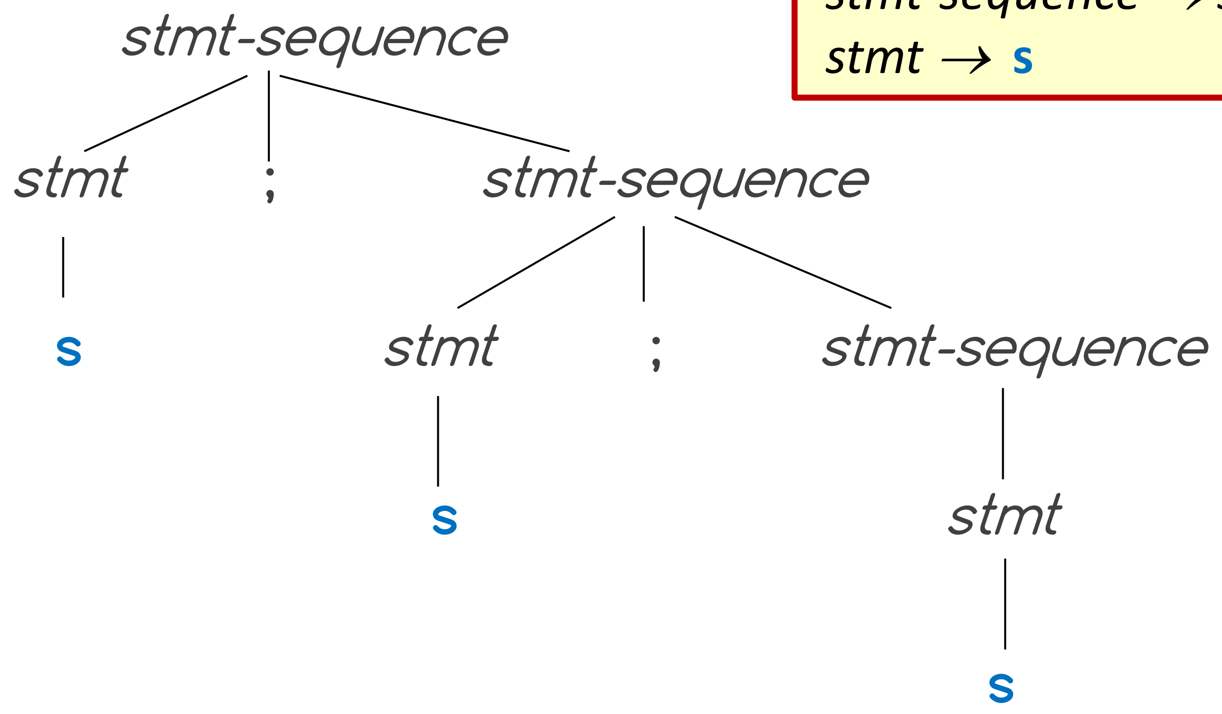
- The grammar of a sequence of statements separated by semicolons:

$$\textit{stmt-sequence} \rightarrow \textit{stmt} \;;\; \textit{stmt-sequence} \mid \textit{stmt}$$
$$\textit{stmt} \rightarrow \mathbf{s}$$

Example 3

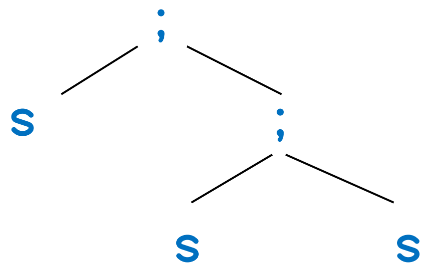
- The string *s; s; s* has the following ***parse tree*** with respect to this grammar:

$$stmt\text{-}sequence \rightarrow stmt\ ;\ stmt\text{-}sequence \mid stmt$$
$$stmt \rightarrow \mathbf{s}$$

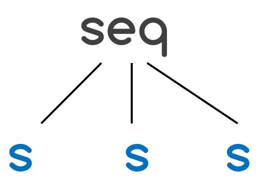


Example 3

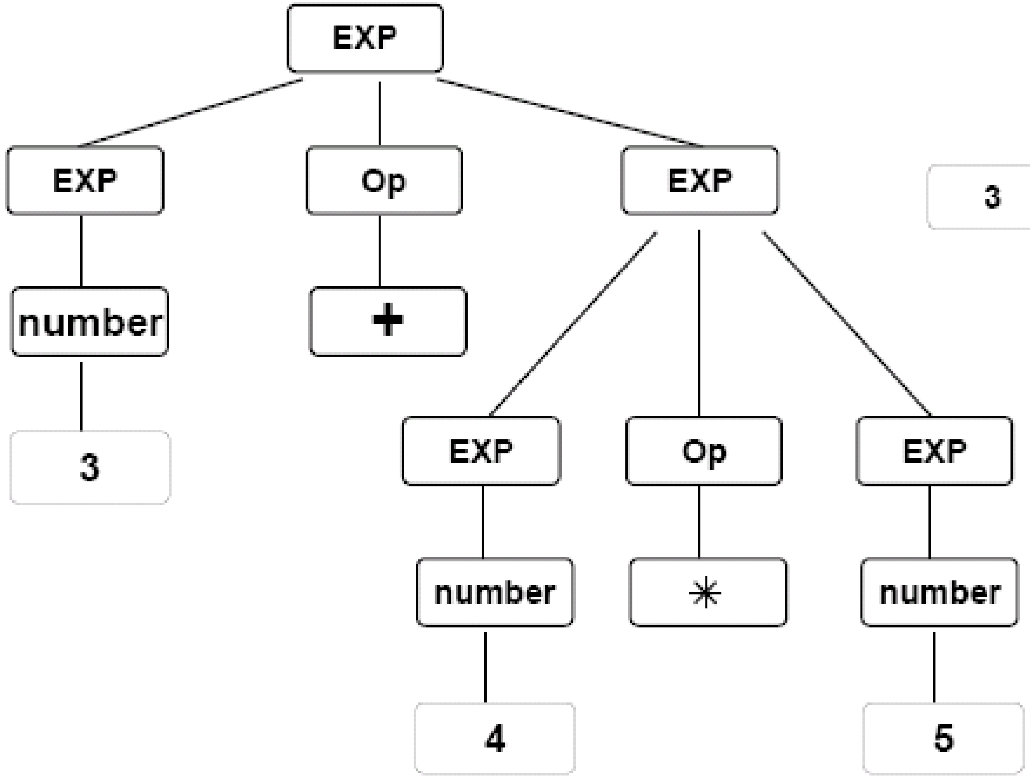
- A possible syntax tree for this same string is:

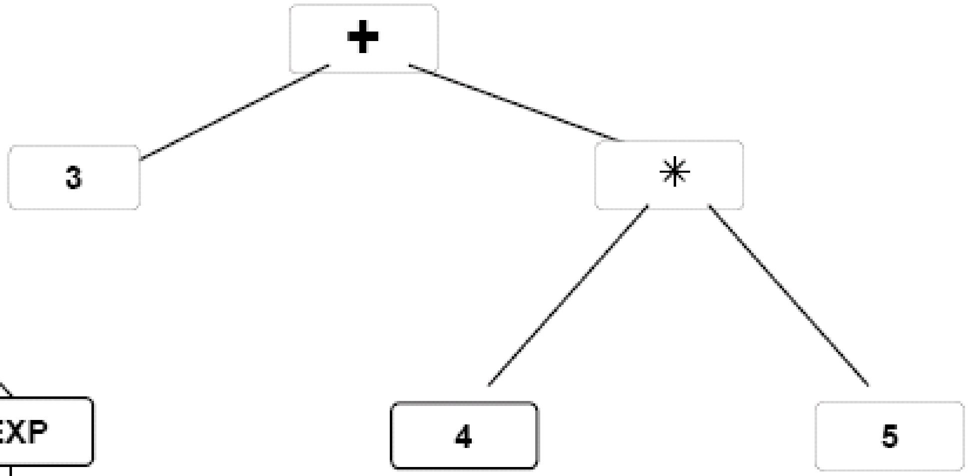


- Bind all the statement nodes in a sequence together with just one node, so that the previous syntax tree would become



Parse tree vs Syntax tree

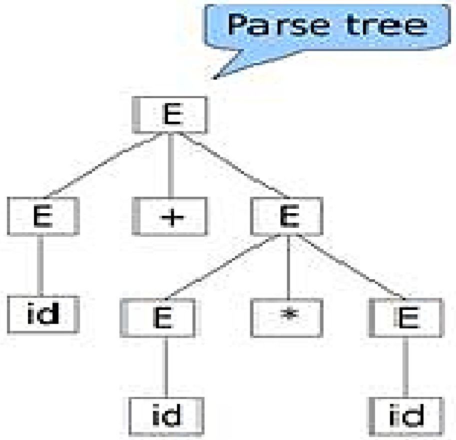


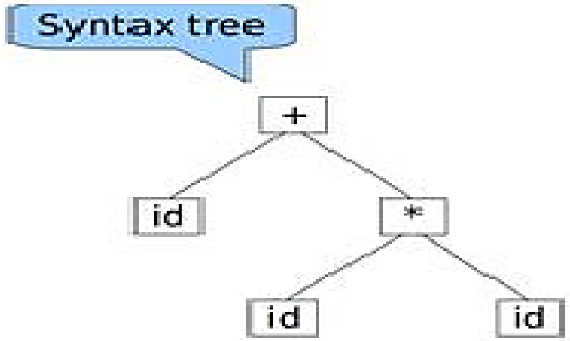


Parse Tree	Syntax Tree
Interior nodes are non-terminals, leaves are terminals	Interior nodes are “operators”, leaves are operands
Rarely constructed as a data structure	When representing a program in a tree structure usually use a syntax tree
Represents the concrete syntax of a program	Represents the abstract syntax of a program (the semantics)
Contains unusable information also	Contains only meaningful information

Grammar: $E \rightarrow E * E \mid E + E \mid id$

Program: $a + b * c$

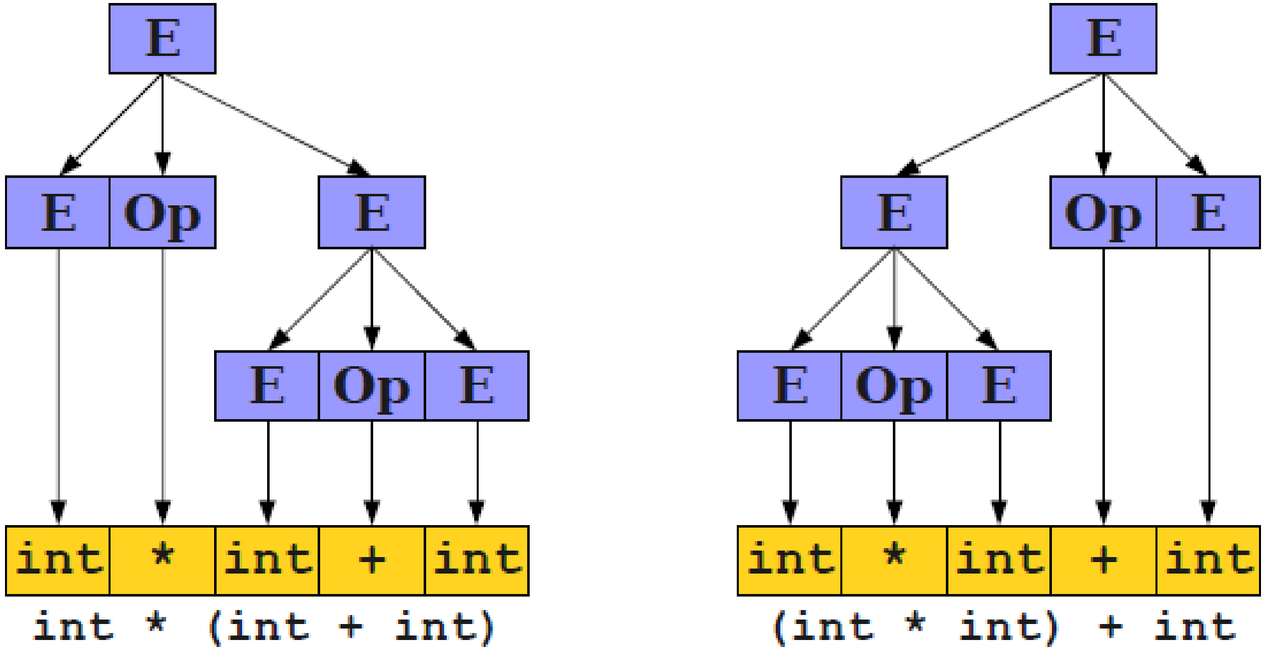





Q
A

3 Ambiguity

A Serious Problem (Recap)



Ambiguous Grammar

$E \rightarrow E\ Op\ E \mid \texttt{int}$

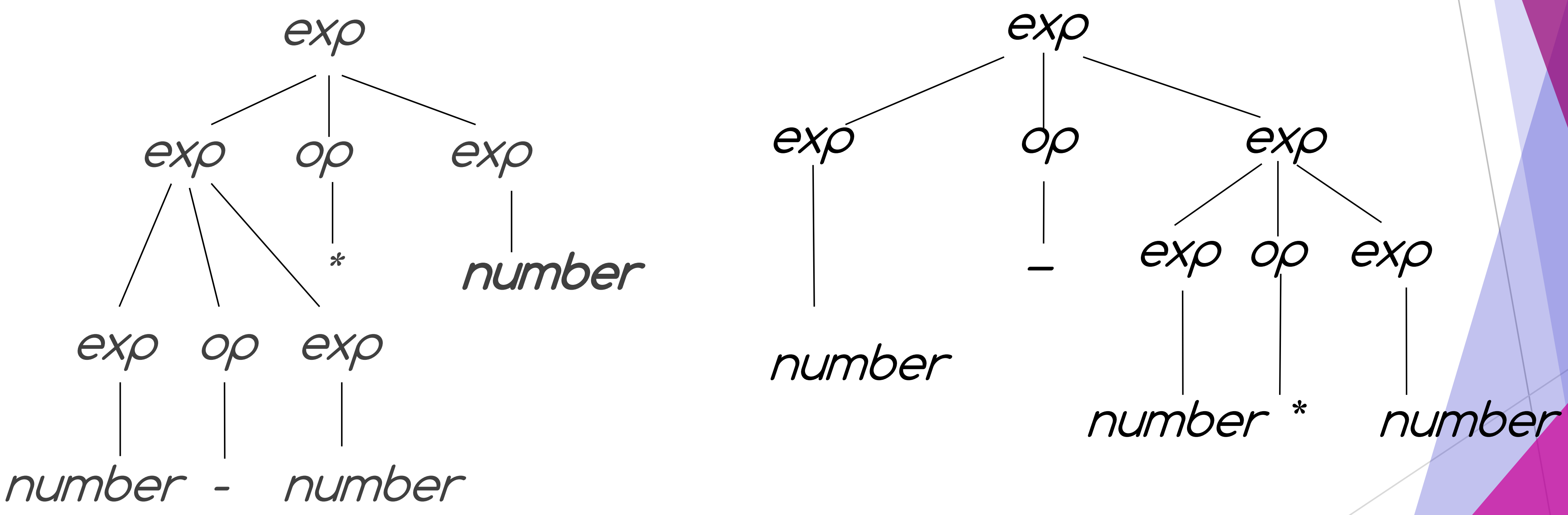
$Op \rightarrow + \mid * \mid - \mid /$

Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more parse trees.
- Note that *ambiguity is a property of grammars, not languages*:
  - there can be multiple grammars for the same language, where some are ambiguous and some aren't.
- Some languages are inherently ambiguous:
  - there are no unambiguous grammars for those languages.

What is Ambiguity?

This string has two different parse trees.



What is Ambiguity?

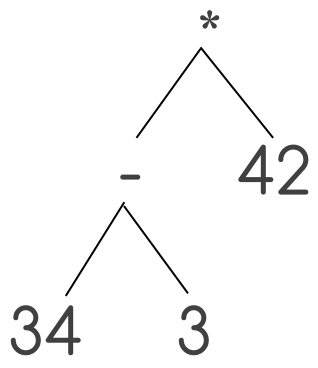
Corresponding to the two leftmost derivations

exp => *exp op exp*
=> *exp op exp op exp* ,
=> ***number*** *op exp op exp*
=> ***number*** - *exp op exp*
=> ***number*** - ***number*** op exp
=> ***number*** - ***number*** * *exp*
=> ***number*** - ***number*** * ***number***

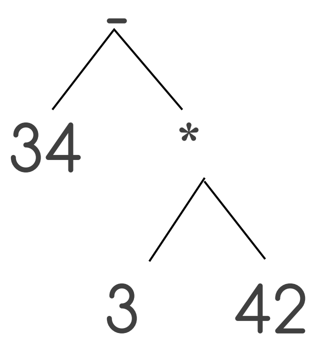
exp => exp *op exp*
=> *number op exp*
=> *number* - *exp*
=> *number* - *exp* op *exp*
=> *number* - *number* op *exp*
=> *number* - *number* * *exp*
=> *number* - *number* * *number*

What is Ambiguity?

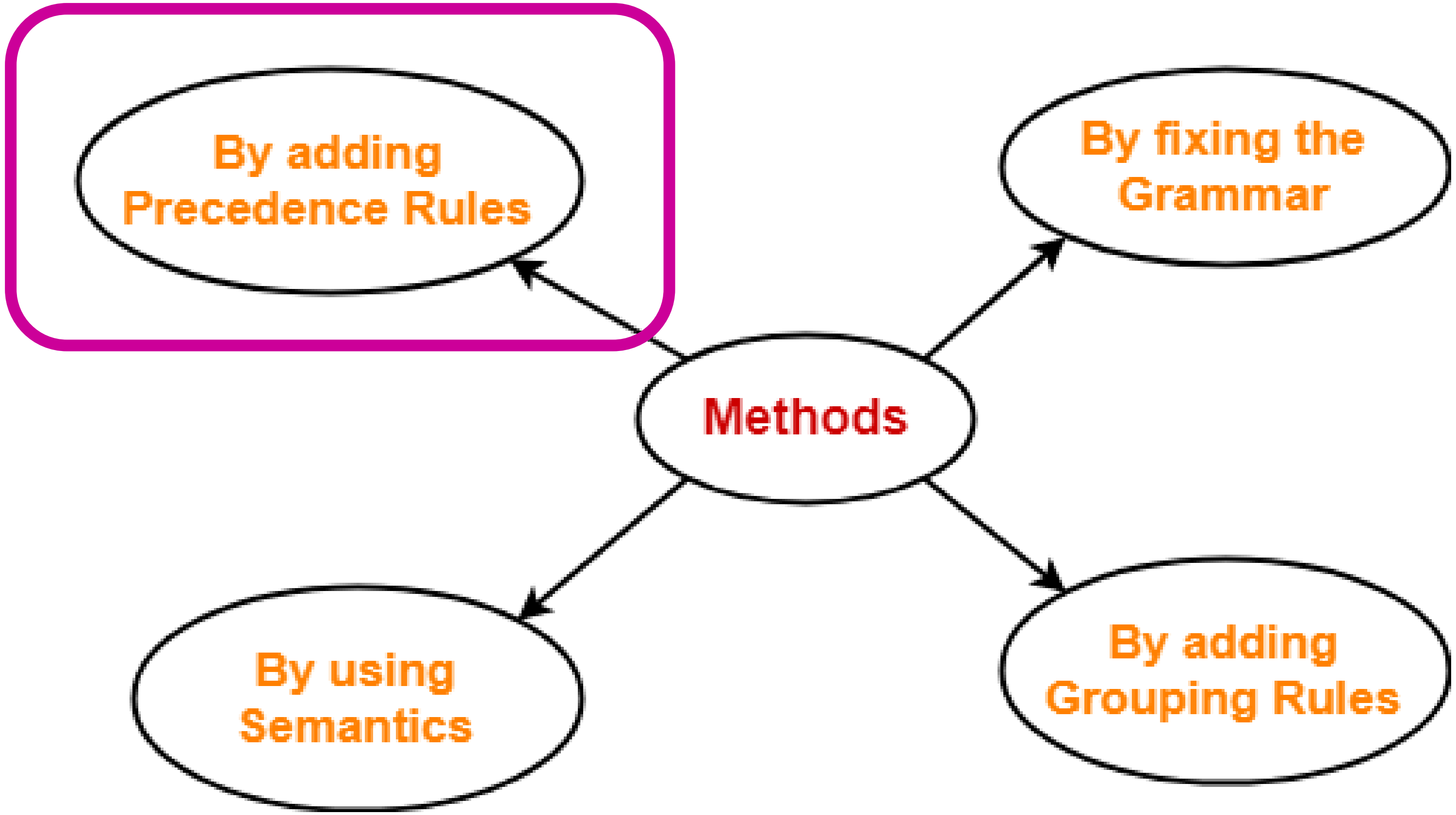
The associated syntax trees are



AND



Resolving


By adding
Precedence Rules
By fixing the
Grammar
Methods
By using
Semantics
By adding
Grouping Rules

Resolving Ambiguity

- Designing unambiguous grammars is tricky and requires planning from the start.
- It's hard to start with an ambiguous grammar and to manually massage it into an unambiguous one.
- Often, have to throw the whole thing out and start over.

Resolving Ambiguity

- We have just seen that this grammar is ambiguous:

$$E \rightarrow E \; Op \; E \mid \text{int}$$

$$Op \rightarrow + \mid - \mid * \mid /$$

Goals

- Eliminate the ambiguity from the grammar
- Make the only parse trees for the grammar the ones corresponding to operator precedence

Operator Precedence

- Can often eliminate ambiguity from grammars with operator precedence issues by *building precedencies into the grammar*.
- Since * and / bind more tightly than + and -, think of an expression as a series of “blocks” of terms multiplied and divided together joined by +s and -s.

int	*	int	*	int	+	int	*	int	-	int

Operator Precedence

- Can often eliminate ambiguity from grammars with operator precedence issues by *building precedencies into the grammar*.
- Since * and / bind more tightly than + and -, think of an expression as a series of “blocks” of terms multiplied and divided together joined by +s and -s.

int	*	int	*	int	+	int	*	int	-	int

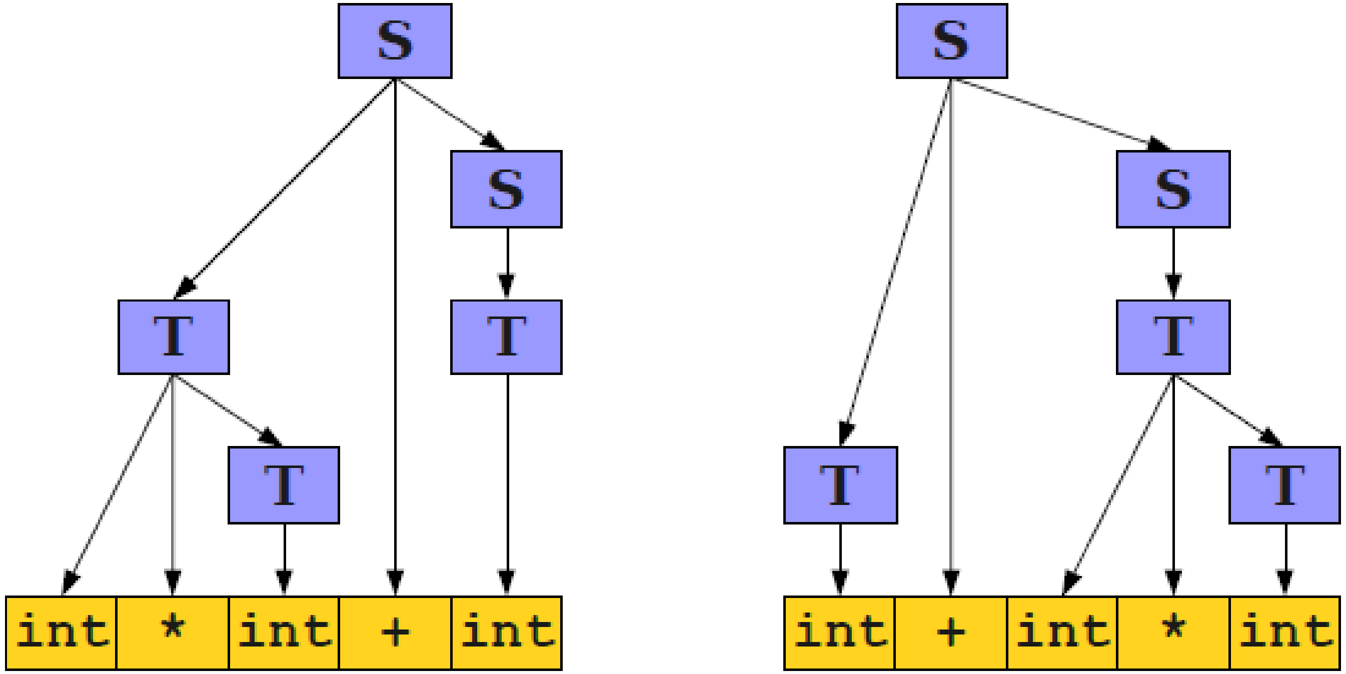
Rebuilding the Grammar

- Idea: Force a construction order where
  - First decide how many “blocks” there will be of terms joined by + and – (lower precedence).
  - Then, expand those blocks by filling in the integers multiplied and divided together (higher precedence).
- One possible grammar:

S → T | T + S | T - S

T → int | int * T | int / T

An Unambiguous Grammar



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

$T \rightarrow \texttt{int} \mid \texttt{int} * T \mid \texttt{int} / T$

Precedence and Associativity

Group of Equal Precedence

- The precedence can be added to our simple expression grammar as follows:

  exp → *exp addop exp* | *term*

  addop → + | -

  term → *term mulop term* | *factor*

  mulop → *

  factor → (*exp*) | number

- Addition and subtraction will appear "higher" (that is, closer to the root) in the parse and syntax trees
  - Receive lower precedence

Precedence Cascade

- Grouping operators into different precedence levels
  - Cascade is a standard method in syntactic specification using BNF
- Replacing the rule

$$exp \rightarrow exp\ addop\ exp\ |\ term$$

by $exp \rightarrow exp\ addop\ term\ |term$ (*Left* Recursive/Associative)

or $exp \rightarrow term\ addop\ exp\ |term$ (*Right* Recursive/Associative)

  - A left recursive rule makes operators associate on the left
  - A right recursive rule makes them associate on the right

Removal of Ambiguity

- Removal of ambiguity in the BNF rules for simple arithmetic expressions
  - write the rules to make all the operations left associative

exp → *exp addop term* | *term*

addop → **+** | **-**

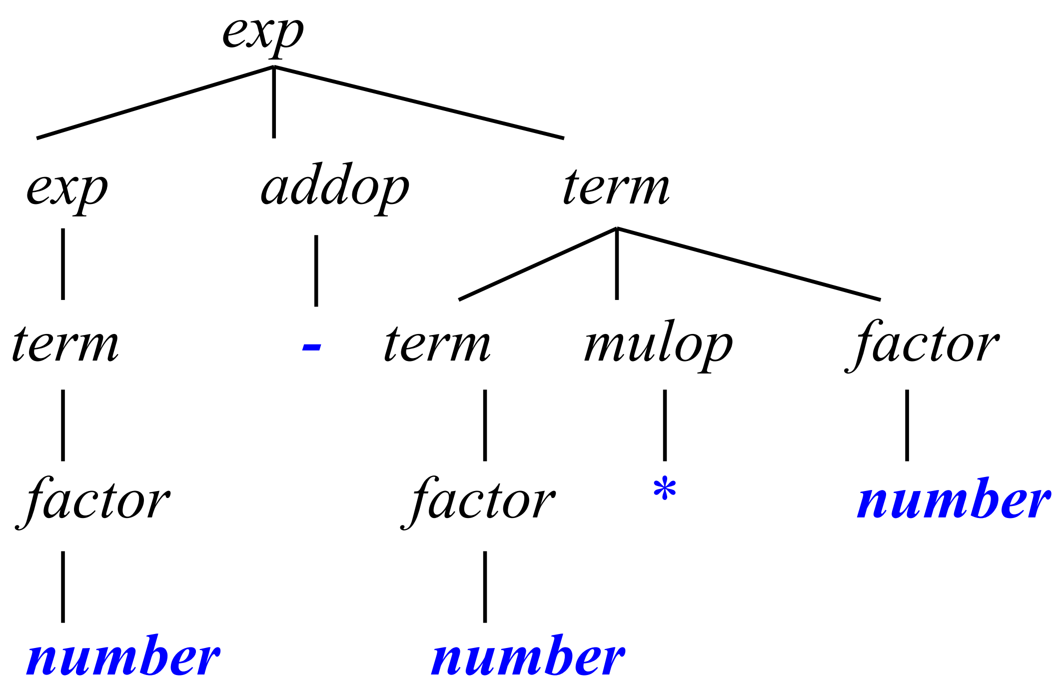
term → *term mulop factor* | *factor*

mulop → *****

factor → (*exp*) | **number**

New Parse Tree

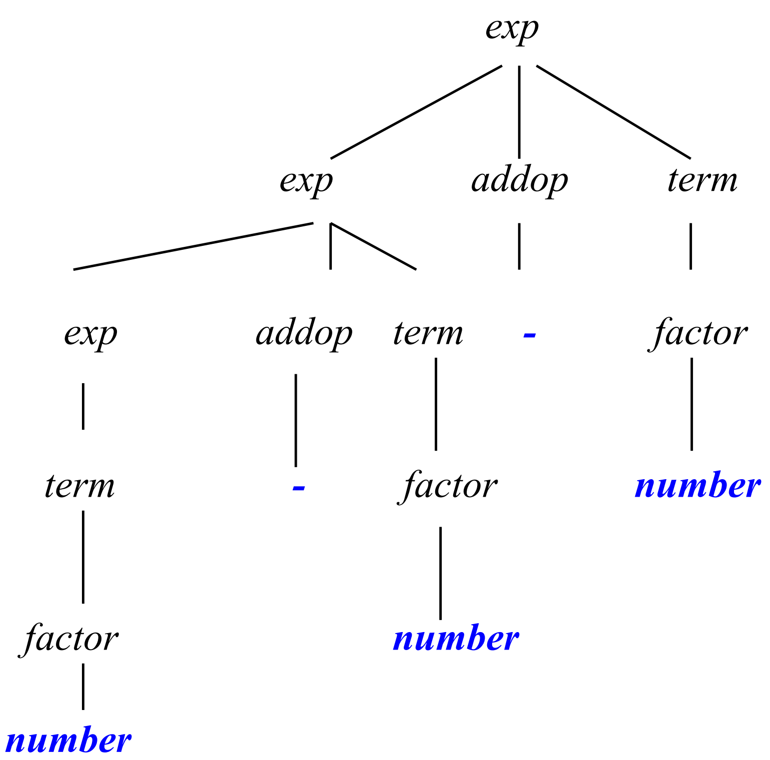
- The parse tree for the expression 34-3*42 is



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

New Parse Tree

- The parse tree for the expression 34-3-42



- The precedence cascades cause the parse trees to become much more complex
- The syntax trees, however, are not affected

4 The dangling else problem

The dangling else problem:
An Ambiguity Grammar

- Consider the grammar from:

  statement → *if-stmt* | ***other***

  if-stmt → **if** (*exp*) *statement* | **if** (*exp*) *statement* **else** *statement*

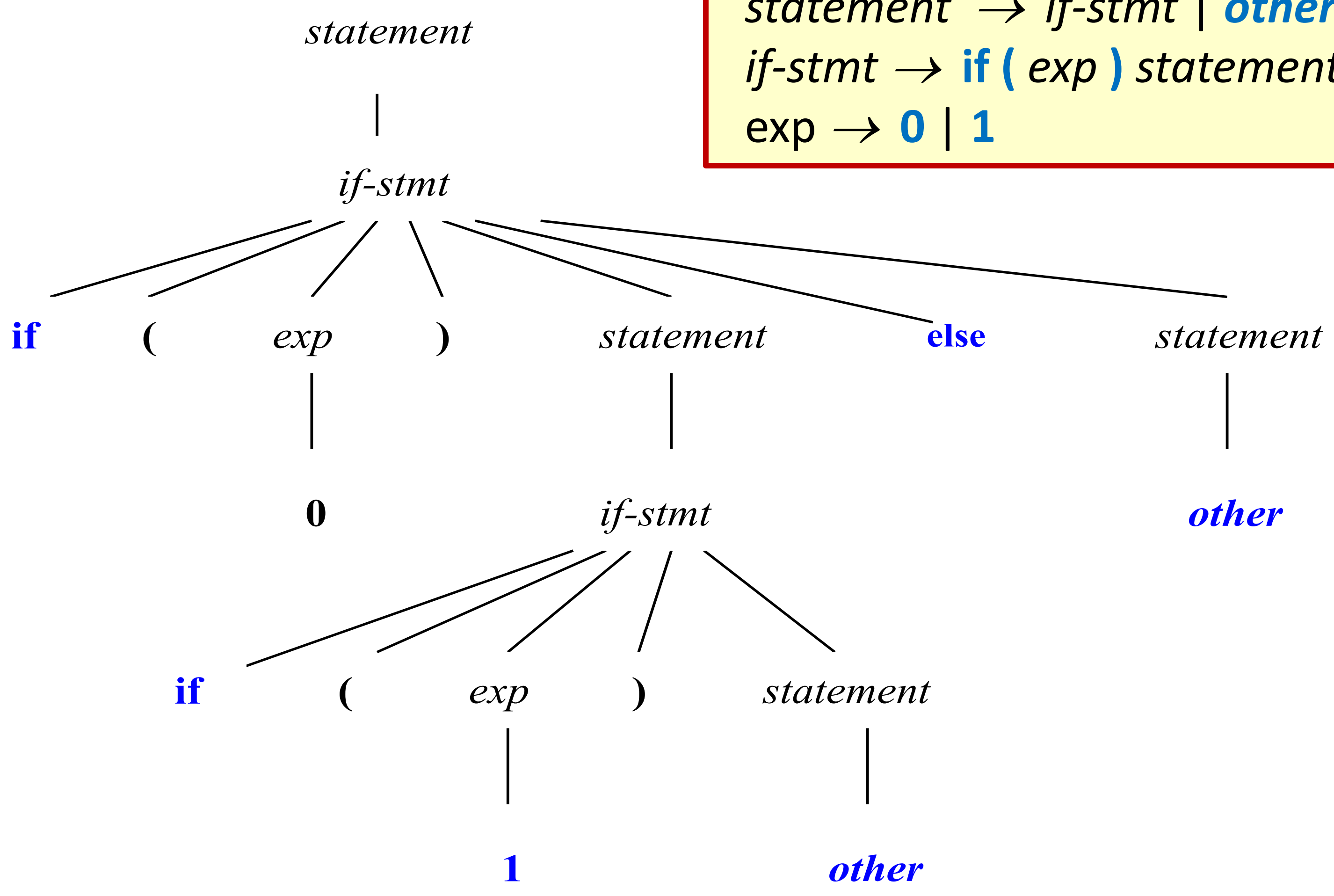
  exp → 0 | 1

- This grammar is ambiguous as a result of the optional else.
- Consider the string

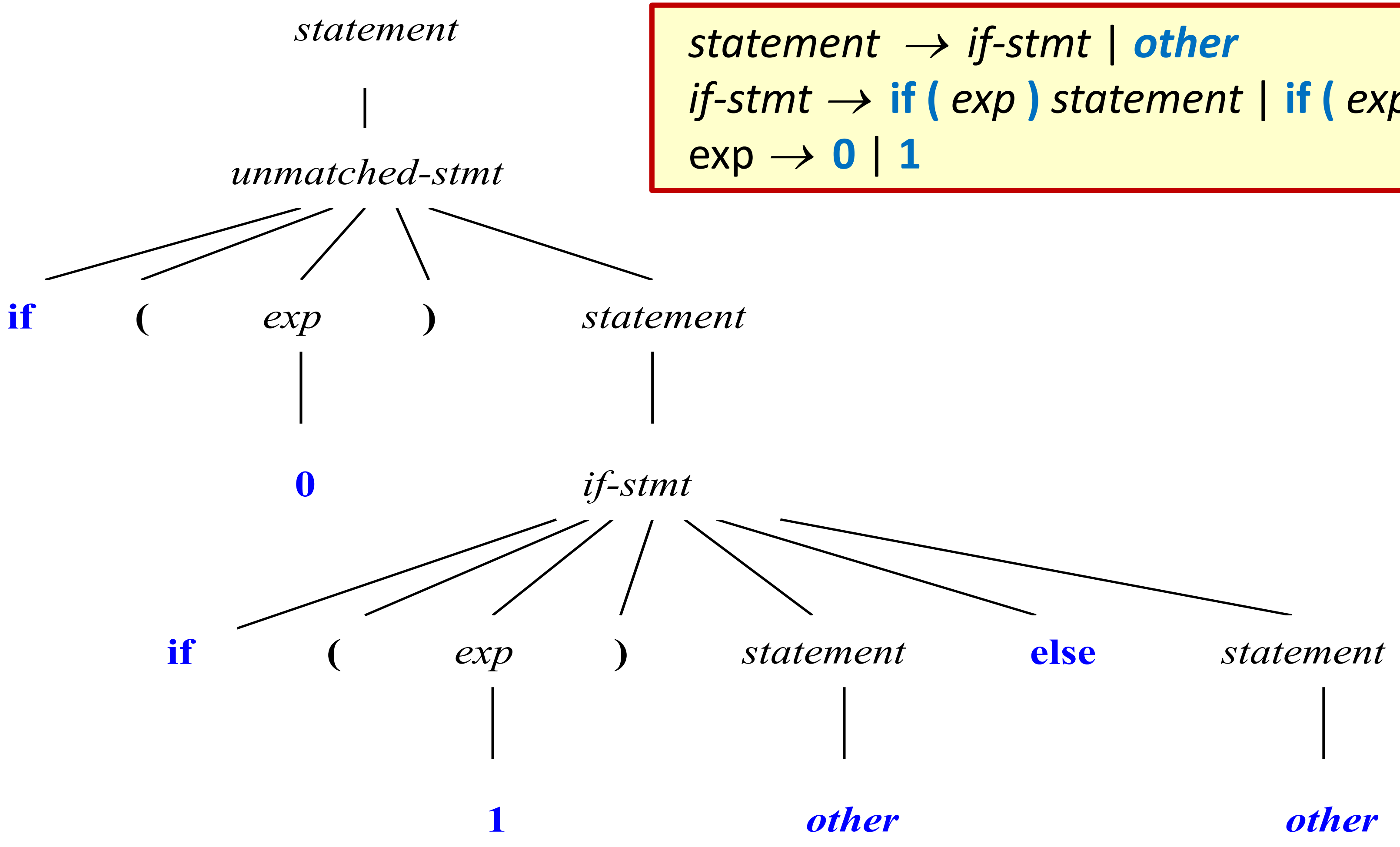
  if (0) if (1) other else other

if (0) if (1) other else other

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



if (0) if (1) other else other



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

Dangling else problem

- Which tree is correct depends on associating the single else-part with the first or the second if-statement.
  - The first associates the else-part with the first if-statement;
  - The second associates it with the second if-statement.
- This ambiguity is called dangling else problem
- **This disambiguating rule is the most closely nested rule**
  - **implies that the second parse tree is the correct one.**

Example

```
if (x != 0)
   if (y = = 1/x)  ok = TRUE;
        else  z = 1/x;
```

- Note that, if we wanted we *could* associate the else-part with the first if-statement by using brackets {...} in C, as in

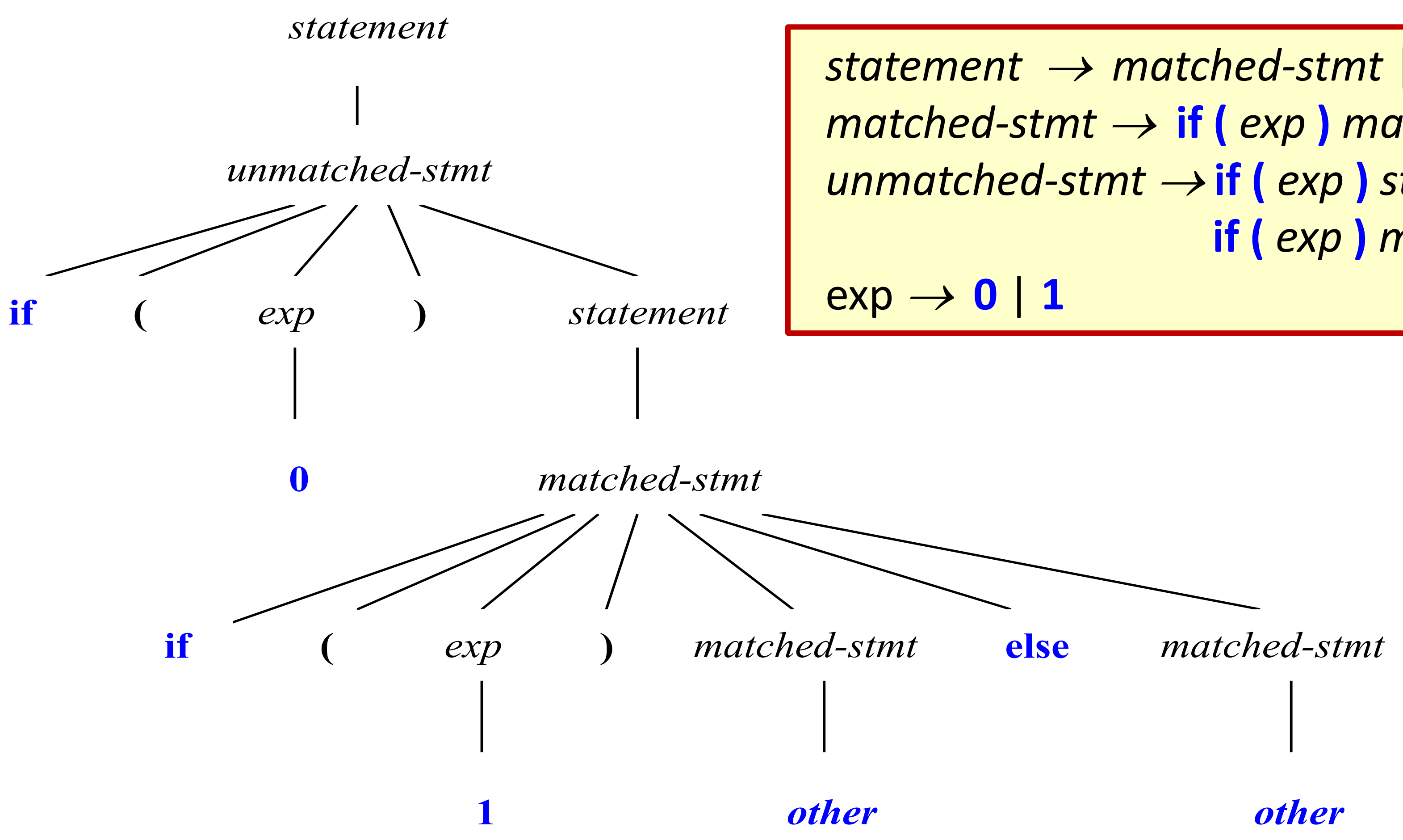
```
if  (x != 0)
{ if (y = = 1/x)  ok = TRUE;   }
else  z = 1/x;
```

A Solution to the dangling else ambiguity in the BNF

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**

- Permitting only a *matched-stmt* to come before an else in an if-statement
  - forcing all else-parts to be matched as soon as possible.

if (0) if (1) other else other



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**


Q
A

Review Questions

Problem 1

- Check whether the following grammar G is ambiguous or not given the production rules:

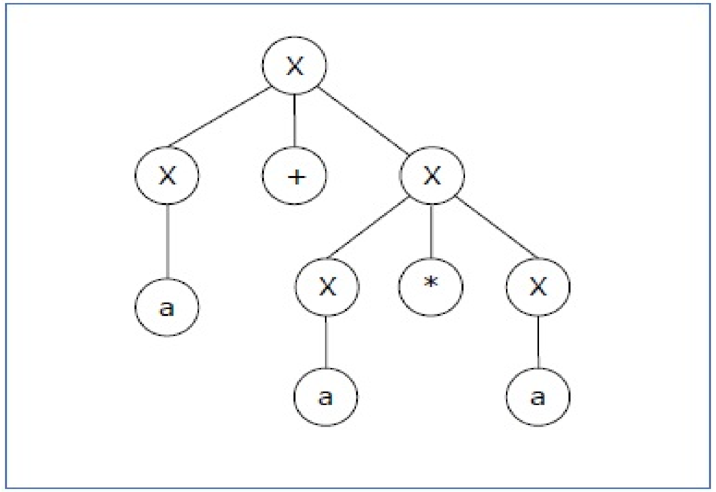
$$X \rightarrow X+X \mid X^*X \mid |X| \mid a$$

Problem 1 Solution

- find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

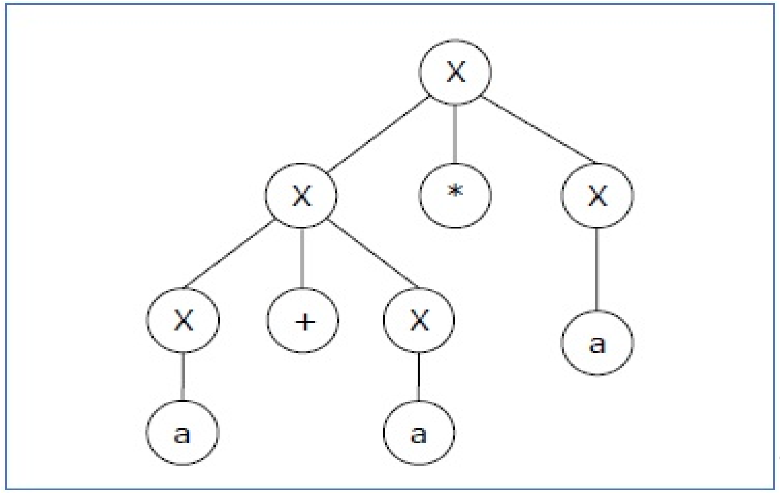
Derivation 1 –

X → X+X → a +X → a+ X*X → a+a*X → a+a*a



Derivation 2 –

X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a



Then this grammar is ambiguous

Problem 2

Show that the grammar G with the following production is ambiguous.

$S \rightarrow a \mid aAb \mid abSb$

$A \rightarrow aAAb \mid bS$

Problem 2 Solution

$$S \Rightarrow abSb \qquad (\because S \rightarrow abSb)$$
$$\Rightarrow abab \qquad (\because S \rightarrow a)$$

- Similarly,

$$S \Rightarrow aAb \qquad (\because S \rightarrow aAb)$$
$$\Rightarrow abSb \qquad (\because A \rightarrow bS)$$
$$\Rightarrow abab \qquad (\because S \rightarrow a)$$

- Since ‘abab’ has two different derivations, the grammar G is ambiguous.

Problem 3

- Convert the following ambiguous grammar into unambiguous grammar.

$$R \rightarrow R + R \mid R \,.\, R \mid R^* \mid a \mid b$$

- where * is kleen closure and . is concatenation.

Problem 3 Solution

- To convert the given grammar into its corresponding unambiguous grammar, we implement the precedence and associativity constraints.
- We have
  - Given grammar consists of the following operators + , . , *
  - Given grammar consists of the following operands a , b
- The priority order is (a , b) > * > . > + where
  - . operator is left associative
  - + operator is left associative

Problem 3 Solution

- Using the precedence and associativity rules, we write the corresponding unambiguous grammar as

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \,.\, F \mid F \\ F &\rightarrow F * F \mid G \\ G &\rightarrow a \mid b \end{aligned}$$

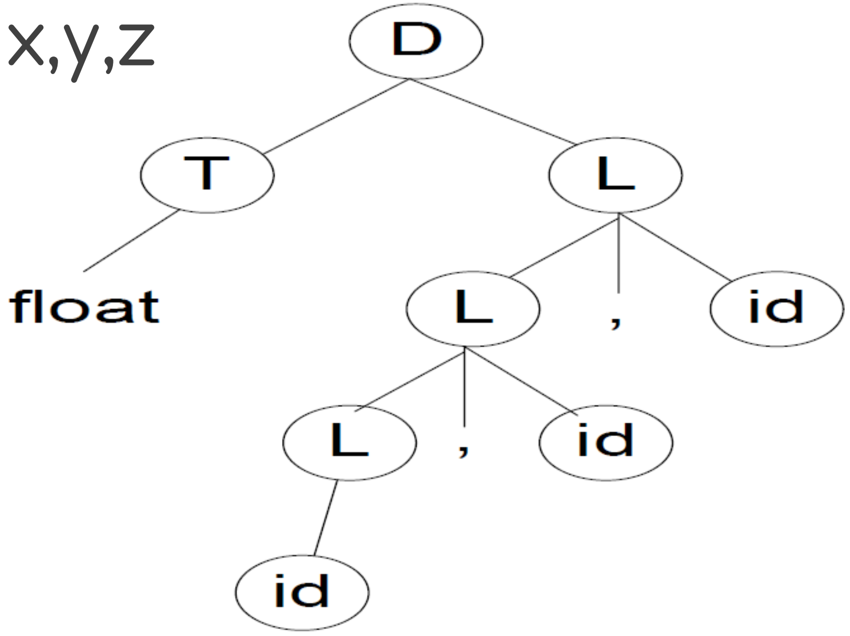
OR

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \,.\, F \mid F \\ F &\rightarrow F * F \mid a \mid b \end{aligned}$$

Problem 4

- Write the CFG and Parse tree that can generates the following expression

float x,y,z



Summary

- Syntax analysis (parsing) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by context-free grammars (CFGs).
- A parse tree shows how a string can be derived from a grammar.
- Abstract syntax trees (ASTs) contain an abstract representation of a program's syntax.

Summary

- A grammar is ambiguous if it can derive the same string multiple ways.
- A CFG is said to be **ambiguous** if there is **at least one string with two or more parse trees.**
- Note that **ambiguity is a property of grammars, not languages.**
- There is no algorithm for eliminating ambiguity; it must be done by hand.
  - *Some Grammars are inherently ambiguous, meaning that no unambiguous grammar exists for them.*
- There is **no algorithm for detecting whether an arbitrary grammar is ambiguous.**

Summary

- If a grammar can be made unambiguous at all, it is usually made unambiguous through **layering**.
- Have exactly one way to build each piece of the string.
- Have exactly one way of combining those pieces back together.

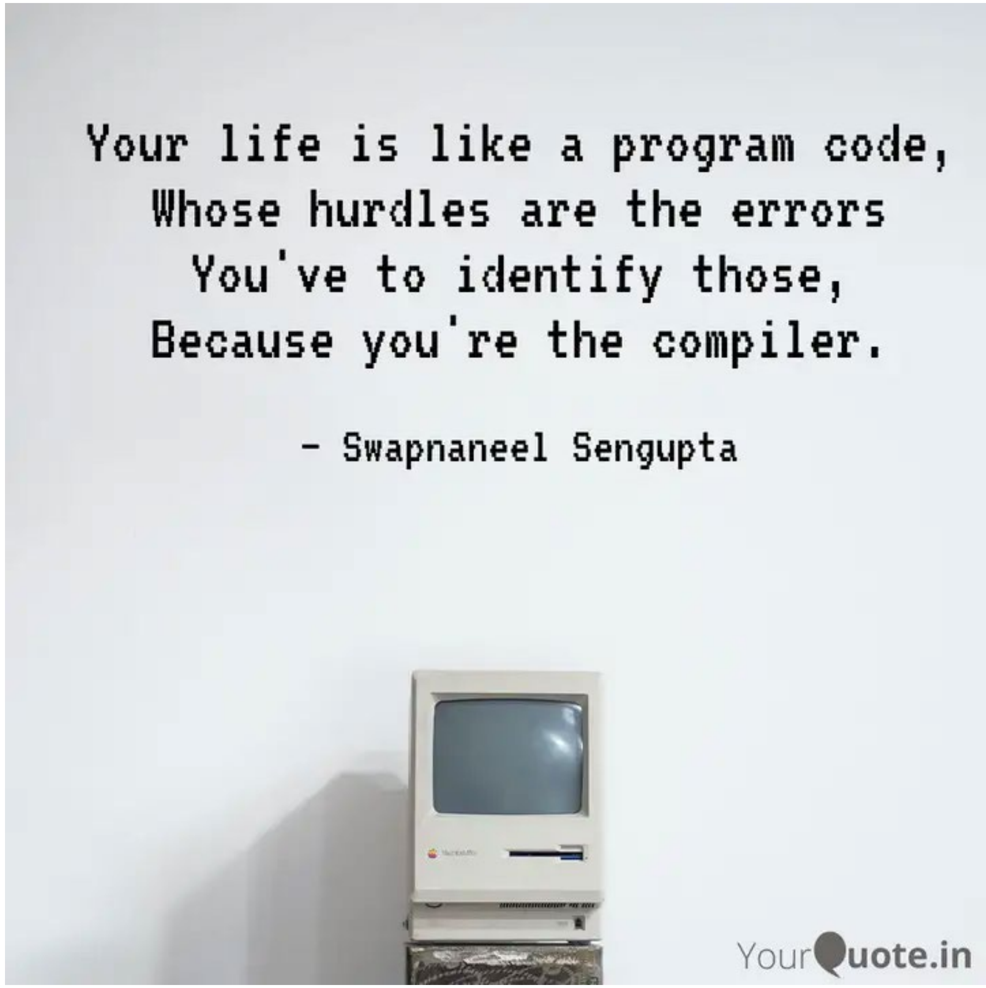
References of this lecture

- Presentation slides of the book: COMPILER CONSTRUCTION, Principles and Practice, by Kenneth C. Louden
- Credits for Dr. Sally Saad, Prof. Mostafa Aref, Dr. Islam Hegazy, and Dr. Abd ElAziz for help in content preparation and aggregation (FCIS-ASU)

Supplementary References

- A good video explaining ambiguity and disambiguating the CFG

https://youtu.be/9vmhcBpZDcE



See you next lecture