Lecture 04

Syntax Analyzer (Parser)

Part 1: Context Free Grammars

Hyosu Kim

School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

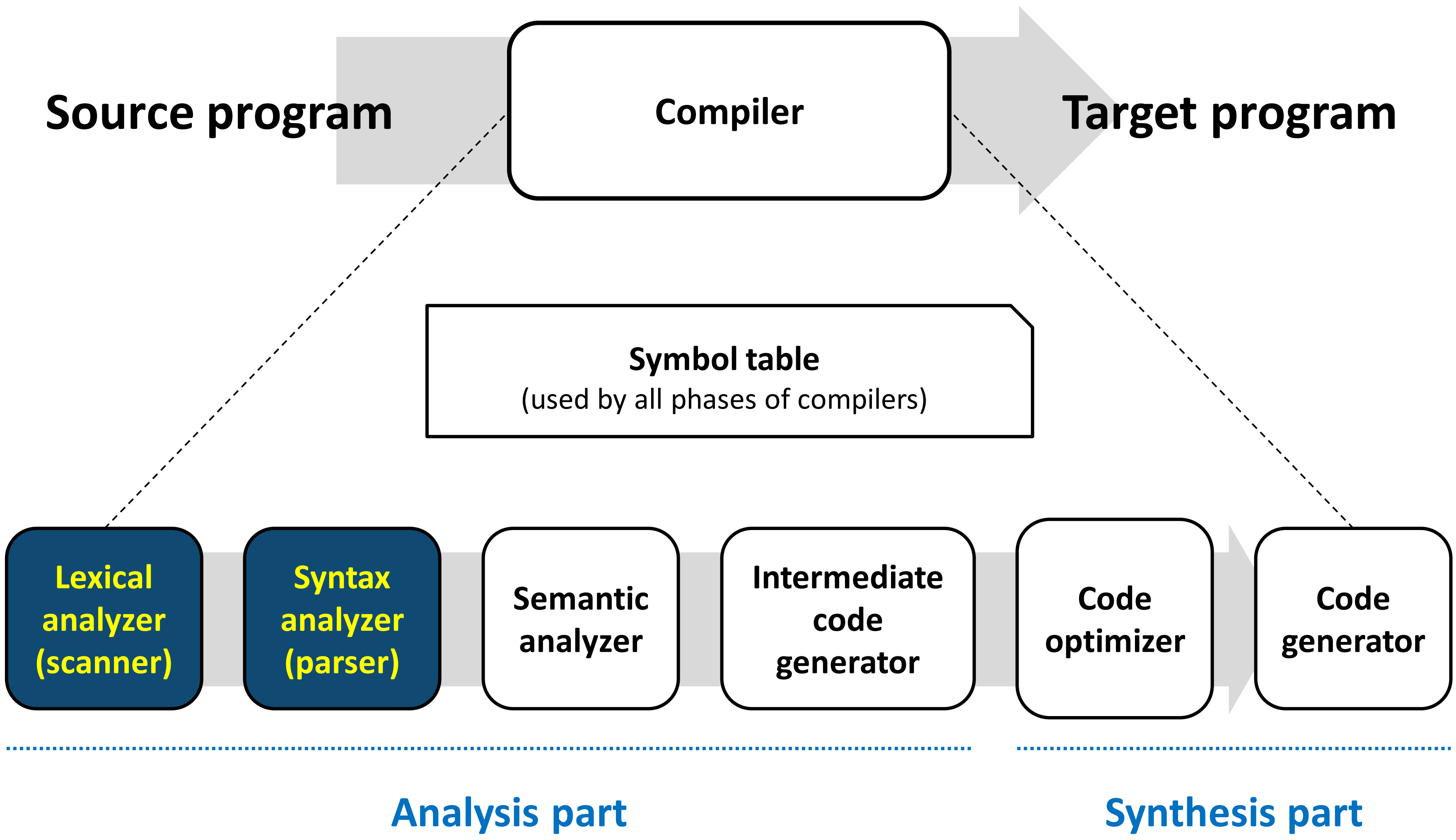
Overview

In this lecture, you will learn

1. **What a syntax analyzer is & how it works**

2. **How to specify the syntax of programming languages**

Overview



Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Overview

What does a syntax analyzer do?

In / this / course / , / you / will / learn /

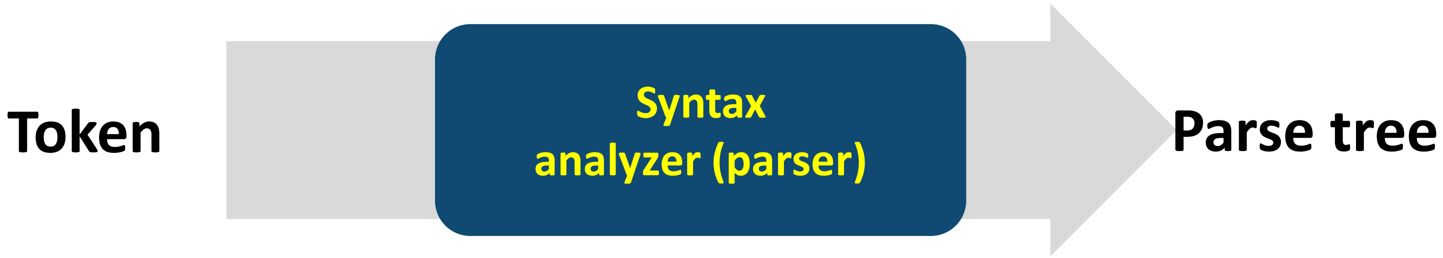
how / to / design / and / implement / compilers

Does this sentence have a valid structure?

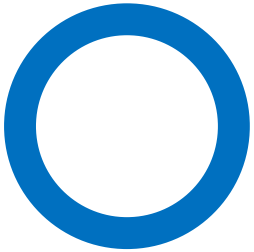
e.g., verb after noun

Syntax analyzer

1. Decides whether a given set of tokens is valid or not



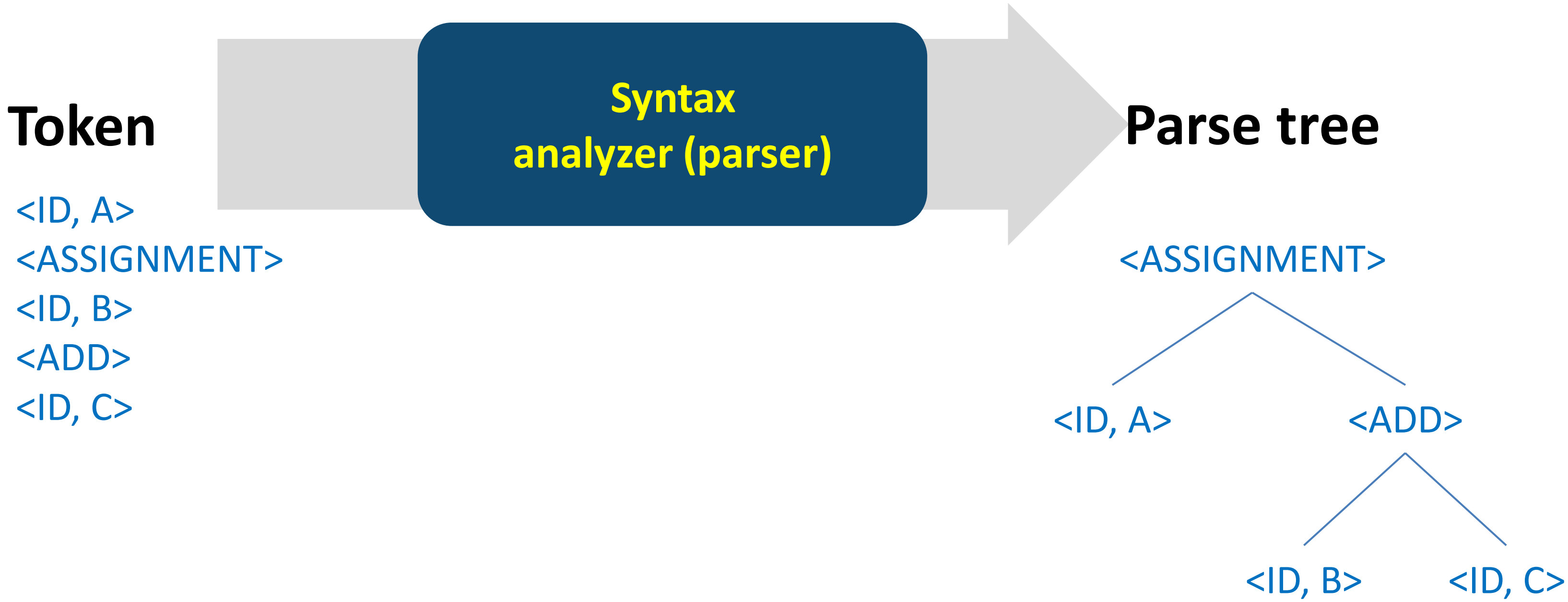
<ID, A>
<ASSIGNMENT>
<ID, B>
<ADD>
<ID, C>



<ID, A>
<ADD>
<ID, B>
<ASIGNMENT>
<ID, C>

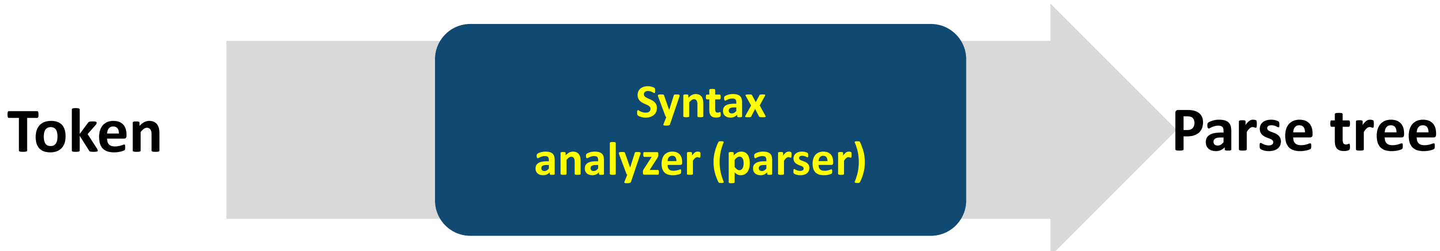
Syntax analyzer

2. Creates a tree-like intermediate representation (e.g., parse tree) that depicts the grammatical structure of the token stream



Syntax analyzer

1. Decides whether a given set of tokens is valid or not



Q. How to specify the rule for deciding valid token set?

Q. How to distinguish between valid and invalid token sets?

Context free grammar!!

Why don't we use regular expressions?

It is not sufficient to depict the syntax of programming languages

An expression is a regular expression

If and only if it can be described by using the basic regular expressions only

Regular expression	Expressed regular language
$\boldsymbol{\epsilon}$	$L(\epsilon) = \{\epsilon\}$
$\boldsymbol{a}$	$L(a) = \{a\}$, *where* a *is a symbol in alphabet* Σ
$\boldsymbol{r_1 \| r_2}$	$L(r_1) \cup L(r_2)$, *where* r_1 *and* r_2 *are regular expressions*
$\boldsymbol{r_1 r_2}$	$L(r_1 r_2) = \{ab \| a \in L(r_1) \text{ and } b \in L(r_2)\}$
$\boldsymbol{r^*}$	$L(r^*) = \bigcup_{i \geq 0} L(r^i)$

Why don’t we use regular expressions?

It is not sufficient to depict the syntax of programming languages

An expression is a regular expression

If and only if it can be described by using the basic regular expressions only

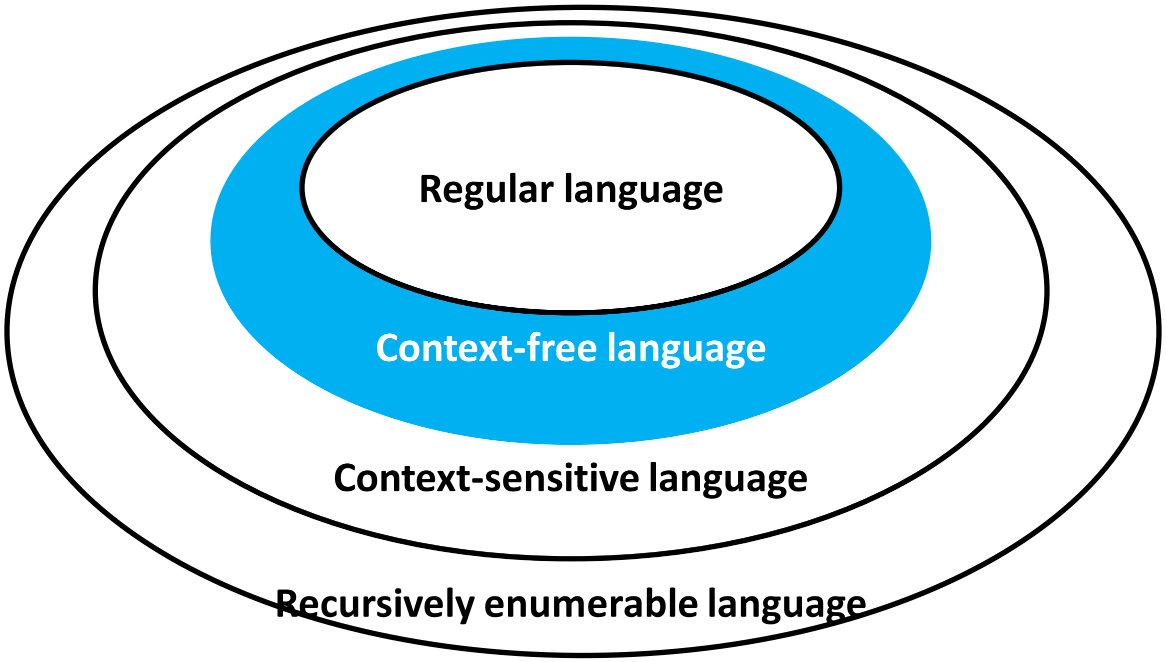
$(^n)^n$ **$(0 \leq n \leq \infty)$ is not a regular expression over alphabet $\Sigma = \{(,)\}$**

Why don't we use regular expressions?

It is not sufficient to depict the syntax of programming languages

Thus, instead of using regular expressions, we use context free grammars!!

- **The coverage of formal languages**



Context free grammars (CFG)

A notation for describing context free languages

A CFG consists of

- **Terminals**: the basic symbols (usually, token name = terminal)

  Terminals can not be replaced

- **Non-terminals**: syntactic variables

  Non-terminals can be replaced by other non-terminals or terminals

- **A start symbol**: one non-terminal (usually, the non-terminal of the first rule)

- **Productions (→)**: a rule for replacement

Context free grammars (CFG)

A notation for describing context free languages

A CFG consists of terminals, non-terminals, a start symbol, and productions

A CFG follows the rule

- $\boldsymbol{A \rightarrow \alpha, \quad B \rightarrow \beta}$

  A and B are non-terminals and A is a start symbol (the non-terminal of the first production)

  α and β are any sequence of non-terminals, terminals, and ϵ

- Example

  Terminal = {+, id}, non-terminals = {S, E}, a start symbol = S, two productions

$$\boldsymbol{S \rightarrow E + E, \qquad E \rightarrow id}$$

NOTE: In this class,

We will use the following notational conventions

- Non-terminals are written in upper-case
- Terminals are written in lower-case
- A sequence of non-terminals, terminals, and ϵ is written in $\alpha,\ \beta,\ \omega$
  - e.g., $\alpha = aABBBcddef$

Based on the CFG rules and notational conventions,

an expression $(^{\boldsymbol{n}})^{\boldsymbol{n}}$ $(\mathbf{0} \leq \boldsymbol{n} \leq \infty)$ can be described as follows:

$$BALANCED \rightarrow (BALANCED)|\epsilon$$

Context free grammars (CFG)

It is good at expressing the recursive structure of a program

In our programming languages,

recursive structures are very frequently observed.

$$STMT \rightarrow if\ (EXPR)\ STMT\ else\ STMT$$
$$|\ if\ (EXPR)\ STMT$$

```
if (xxx) {
  if (yyy) {
    if (…) { … }
  }
} else { … }
```

Examples

Let's express simple arithmetic operations

- Terminals: $id, +, *, (,)$

  e.g., $id + id,\ id * id, (id), id + id * id,\ (id + id) * id$

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

Examples

Let's express simple function calls

- Terminals: $id, (,)$

  e.g., $func(arg1, arg2)$, $print(arg)$, $func2()$ …

$$\boldsymbol{S \rightarrow id(E) \mid id()}$$

$$\boldsymbol{E \rightarrow id, E \mid id}$$

Examples

Let's express a simple while statement

- Terminals: $while, id, comparison, number, (,)$

  e.g., $while(A\ >\ B),\ \ while(A),\ \ while(0),\ \ while(3\ >\ 2),\ while(A\ <=1),\ while()$

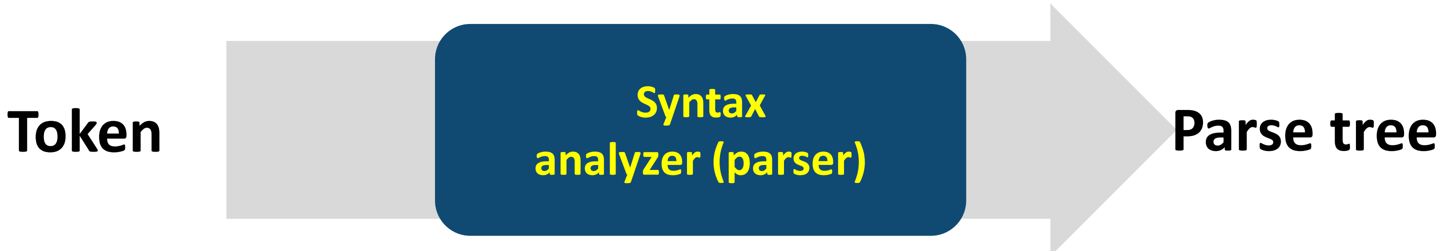
$$\boldsymbol{S \rightarrow while(E)}$$

$$\boldsymbol{E \rightarrow TcomparisonT|T|\epsilon}$$

$$\boldsymbol{T \rightarrow id\ |\ number}$$

Syntax analyzer

1. Decides whether a given set of tokens is valid or not



Q. How to specify the rule for deciding valid token set?

Make **a context free grammar G** based on the rule of a programming language

Q. How to distinguish between valid and invalid token sets?

Derivations

A derivation (⇒) is a sequence of replacement

- $\Rightarrow^*$: Do derivations zero or more times

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

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

$$\boldsymbol{E} \Rightarrow^* \boldsymbol{E} * \boldsymbol{E} + \boldsymbol{E}$$

$$\boldsymbol{E} \Rightarrow (\boldsymbol{E}) \Rightarrow (\boldsymbol{E} + E) \Rightarrow (id + \boldsymbol{E}) \Rightarrow (id + id)$$

$$\boldsymbol{E} \Rightarrow^* (\boldsymbol{id} + \boldsymbol{id})$$

Derivations

A rule for derivations

- Leftmost ($\Rightarrow_{lm}$): replace the lest-most non-terminal first
- Rightmost ($\Rightarrow_{rm}$): replace the right-most non-terminal first

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

$$\boldsymbol{E} \Rightarrow_{lm} \boldsymbol{E} + E \Rightarrow_{lm} \boldsymbol{E} * E + E \Rightarrow_{lm} id * \boldsymbol{E} + E \Rightarrow_{lm} id * id + \boldsymbol{E} \Rightarrow_{lm} id * id + id$$

$$\boldsymbol{E \Rightarrow_{lm}^{*} id * id + id}$$

$$\boldsymbol{E} \Rightarrow_{rm} E + \boldsymbol{E} \Rightarrow_{rm} \boldsymbol{E} + id \Rightarrow_{rm} E * \boldsymbol{E} + id \Rightarrow_{rm} \boldsymbol{E} * id + id \Rightarrow_{rm} id * id + id$$

$$\boldsymbol{E \Rightarrow_{rm}^{*} id * id + id}$$

Token validation test

Definition: **A sentinel form of a CFG G**

- α is a sentinel form of G, if $A \Rightarrow^* \alpha$, where A is the start symbol of G
  - If $A \Rightarrow^*_{lm} \alpha$ or $A \Rightarrow^*_{rm} \alpha$, α is a (left or right) sentinel form of G

Definition: **A sentence of a CFG G**

- α is a sentence form of G,

  if α is a sentinel form of a CFG G which consists of terminals only

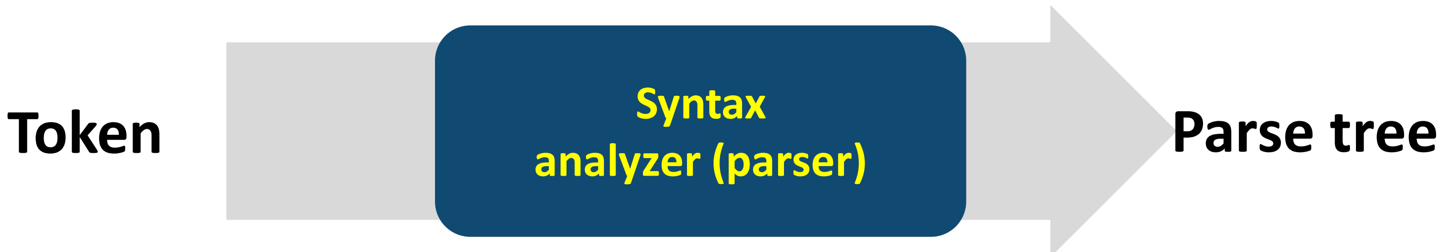
Definition: **A language of a CFG G**

- $L(G)$ is a language of a CFG G (context-free language)
- $L(G) = \{\alpha | \alpha \ is\ a\ sentence\ of\ G\}$

If an input string (e.g., a token set) is in $L(G)$, we can say that it is valid in G

Syntax analyzer

1. Decides whether a given set of tokens is valid or not



Q. How to specify the rule for deciding valid token set?

Make **a context free grammar *G*** based on the rule of a programming language

Q. How to distinguish between valid and invalid token sets?

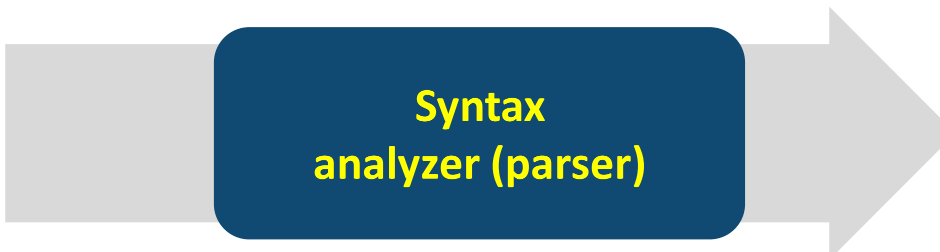
Check **whether the given token set can be derived from the context free grammar *G***

Syntax analyzer

2. Creates a tree-like intermediate representation (e.g., syntax tree) that depicts the grammatical structure of the token stream

Token

<ID, A>
<ASSIGNMENT>
<ID, B>
<ADD>
<ID, C>



Parse tree

How???

Derivations-to-parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

- E

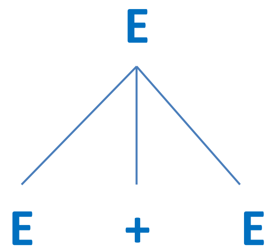
E

Derivations-to-parse trees

$$E \ \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

- E
- $\Rightarrow_{lm} E + E$

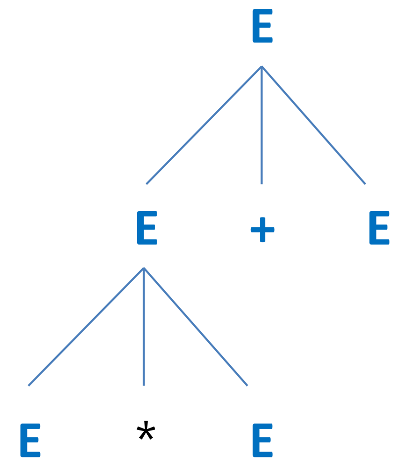


Derivations-to-parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

- E
- $\Rightarrow_{lm} E + E$
- $\Rightarrow_{lm} E * E + E$

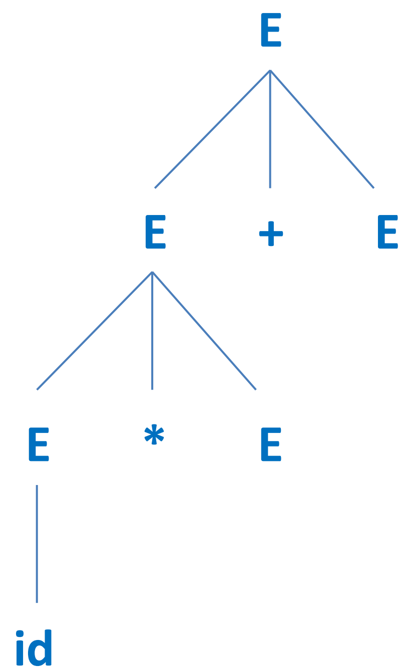


Derivations-to-parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

- E
- $\Rightarrow_{lm} E + E$
- $\Rightarrow_{lm} E * E + E$
- $\Rightarrow_{lm} id * E + E$

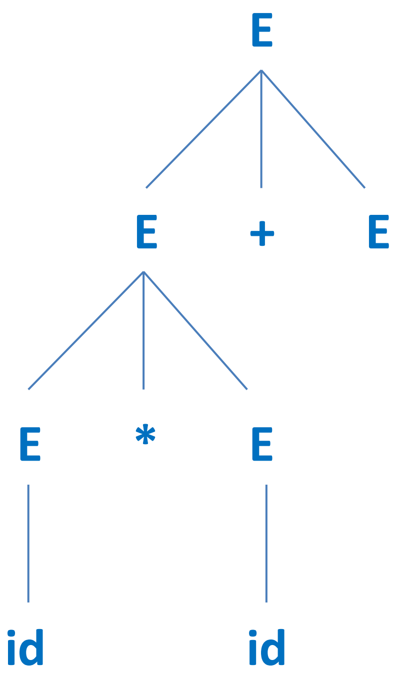


Derivations-to-parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

- E
- $\Rightarrow_{lm} E + E$
- $\Rightarrow_{lm} E * E + E$
- $\Rightarrow_{lm} id * E + E$
- $\Rightarrow_{lm} id * id + E$

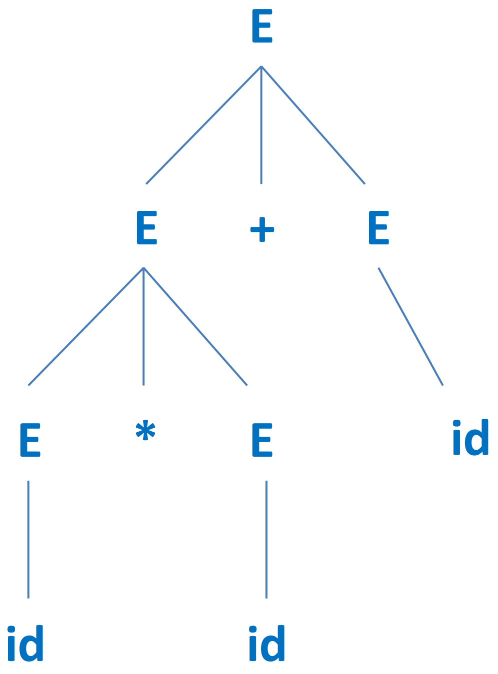


Derivations-to-parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

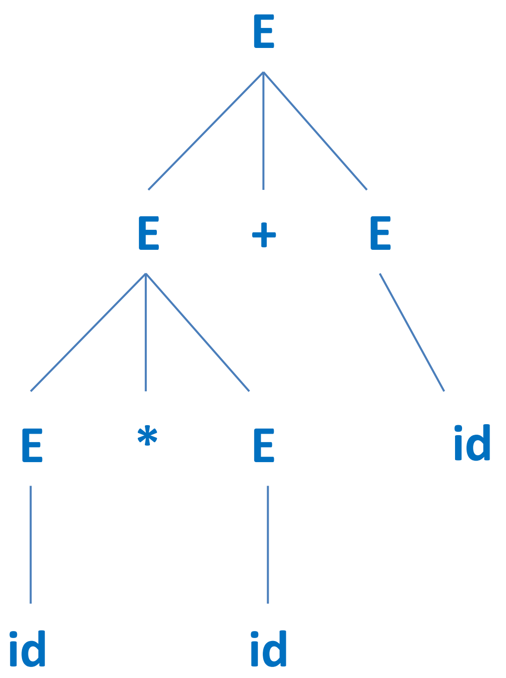
- E
- $\Rightarrow_{lm} E + E$
- $\Rightarrow_{lm} E * E + E$
- $\Rightarrow_{lm} id * E + E$
- $\Rightarrow_{lm} id * id + E$
- $\Rightarrow_{lm} id * id + id$



Derivations-to-parse trees

A parse tree

- Leaf nodes = terminals
- Non-leaf nodes = non-terminals
- An inorder traversal of a parse tree = original strings



※ The same tree is created regardless of the derivation method

Derivations-to-parse trees

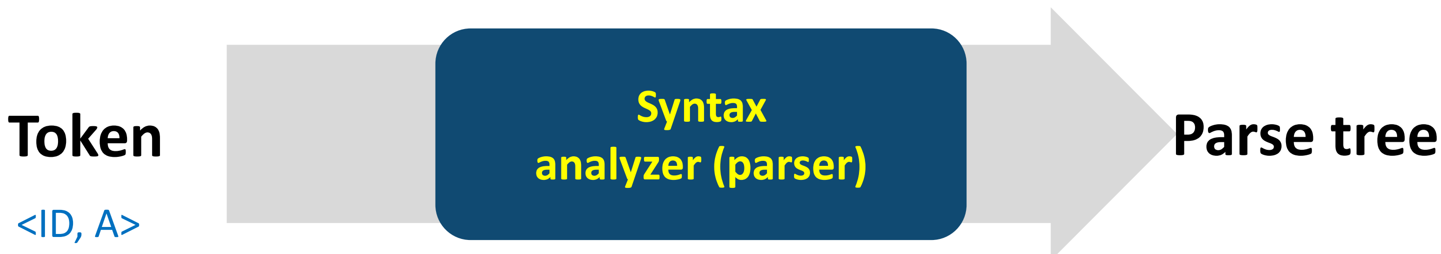
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

- E
- $\Rightarrow_{rm} E + E$
- $\Rightarrow_{rm} E + id$
- $\Rightarrow_{rm} E * E + id$
- $\Rightarrow_{rm} E * id + id$
- $\Rightarrow_{rm} id * id + id$

Syntax analyzer

2. Creates a tree-like intermediate representation (e.g., syntax tree) that depicts the grammatical structure of the token stream



<ID, A>
<ASSIGNMENT>
<ID, B>
<ADD>
<ID, C>

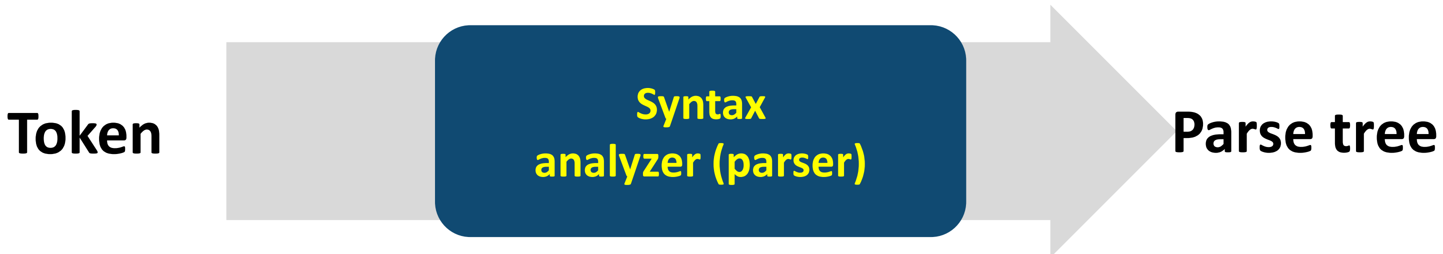
How???

CFG, derivations, derivations-to-parse trees

Syntax analyzer

1. Decides whether a given set of tokens is valid or not

2. Creates a tree-like intermediate representation (e.g., syntax tree) that depicts the grammatical structure of the token stream



Then, how to do these processes 1) efficiently and 2) automatically?

For efficient parsing: creating a good CFG

1. A good CFG is **non-ambiguous**

2. A good CFG has **no left recursion**

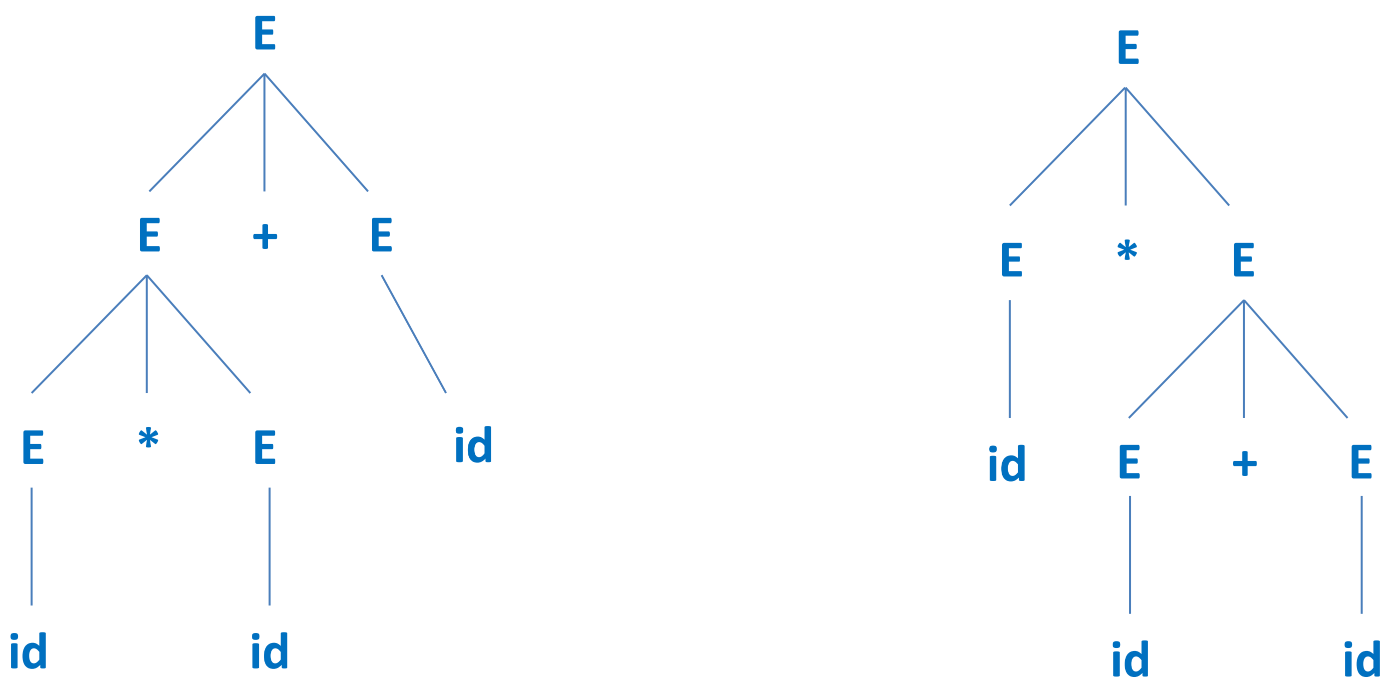
3. For each non-terminal, a good CFG has **only one choice of production** starting from a specific input symbol

Ambiguity

One input string can have multiple different parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $\boldsymbol{id * id + id}$

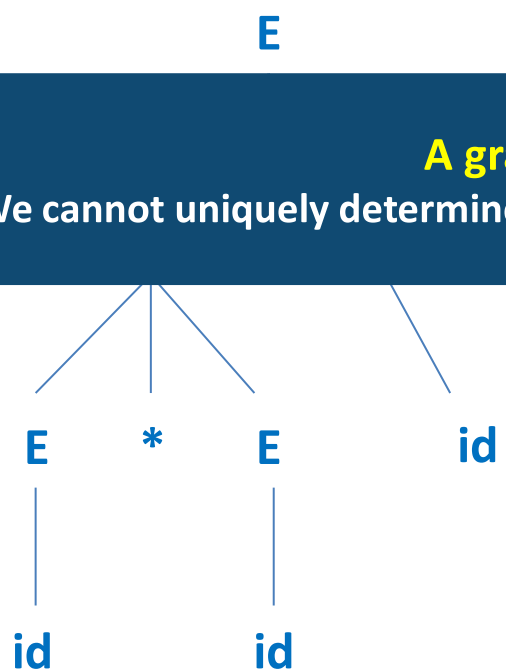


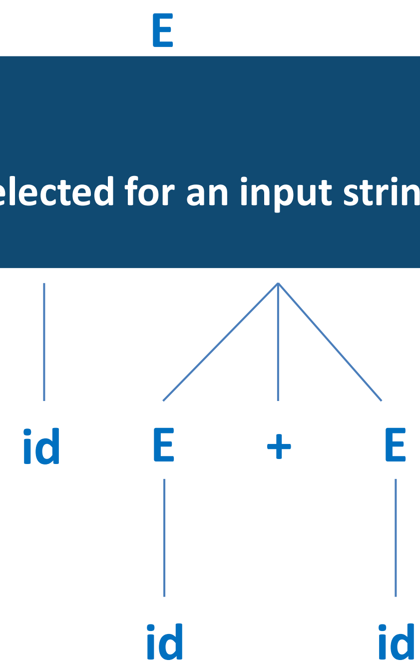
Ambiguity

One input string can have multiple different parse trees

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

For $id * id + id$





A grammar is ambiguous!!!

We cannot uniquely determine which parse tree to be selected for an input string

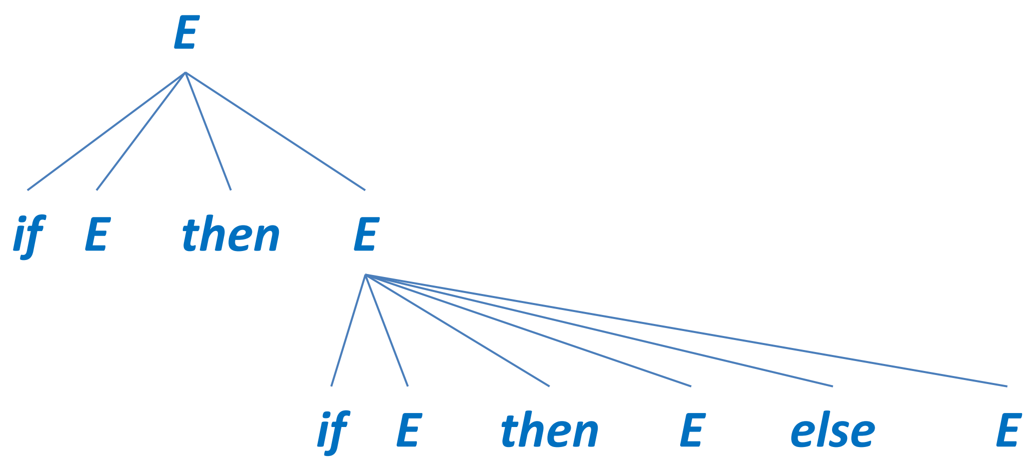
How to eliminate ambiguity?

Just rewrite the ambiguous grammars based on disambiguating rules

- The most common ambiguity problem: **dangling-else**

$$E \ \rightarrow if\ E\ then\ E\ |\ if\ E\ then\ E\ else\ E\ |other$$

For $if\ other\ then\ if\ other\ then\ other\ else\ other,$



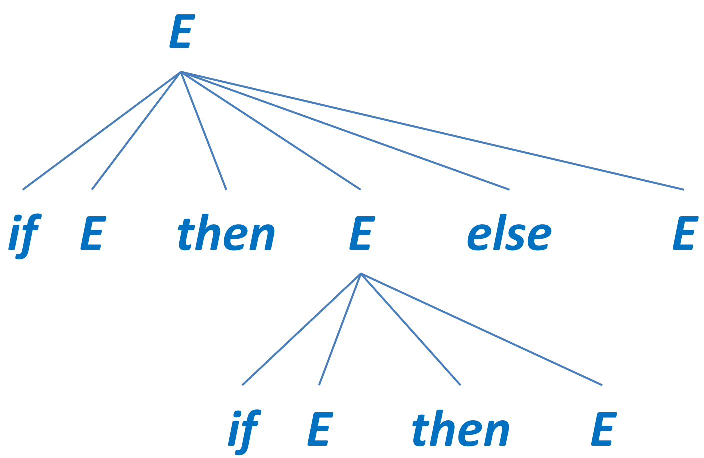
How to eliminate ambiguity?

Just rewrite the ambiguous grammars based on disambiguating rules

- The most common ambiguity problem: **dangling-else**

$$E \rightarrow if\ E\ then\ E \mid if\ E\ then\ E\ else\ E \mid other$$

For $if\ other\ then\ if\ other\ then\ other\ else\ other,$



How to eliminate ambiguity?

Just rewrite the ambiguous grammars based on disambiguating rules

- The most common ambiguity problem: **dangling-else**

$$E \rightarrow if\ E\ then\ E \mid if\ E\ then\ E\ else\ E \mid other$$

For $if\ other\ then\ if\ other\ then\ other\ else\ other,$

Disambiguating rule for the dangling-else problem

"Match each else with the closest unmatched then"

$$E \rightarrow MATCHED \mid UNMATCHED$$

$$MATCHED \rightarrow if\ MATCHED\ then\ MATCHED\ else\ MATCHED \mid other$$

$$UNMATCHED \rightarrow if\ E\ then\ E \mid if\ E\ then\ MATCHED\ else\ UNMATCHED$$

How to eliminate ambiguity?

Just rewrite the ambiguous grammars based on disambiguating rules

- The most common ambiguity problem: **dangling-else**

$$E \rightarrow if\ E\ then\ E \mid if\ E\ then\ E\ else\ E \mid other$$

For $if\ other\ then\ if\ other\ then\ other\ else\ other,$

Disambiguating rule for the dangling-else problem

“Match each else with the closest unmatched then”

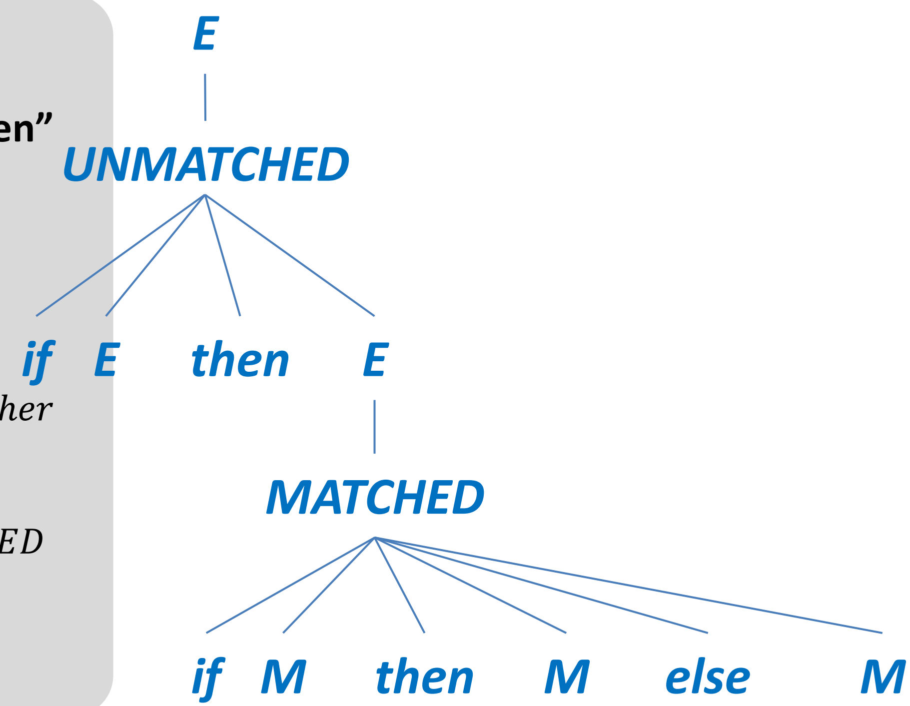
$E \rightarrow MATCHED \mid UNMATCHED$

$MATCHED$

$\rightarrow if\ MATCHED\ then\ MATCHED\ else\ MATCHED \mid other$

$UNMATCHED$

$\rightarrow if\ E\ then\ E \mid if\ E\ then\ MATCHED\ else\ UNMATCHED$



How to eliminate ambiguity?

Just rewrite the ambiguous grammars based on disambiguating rules

- Another example: arithmetic expressions

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

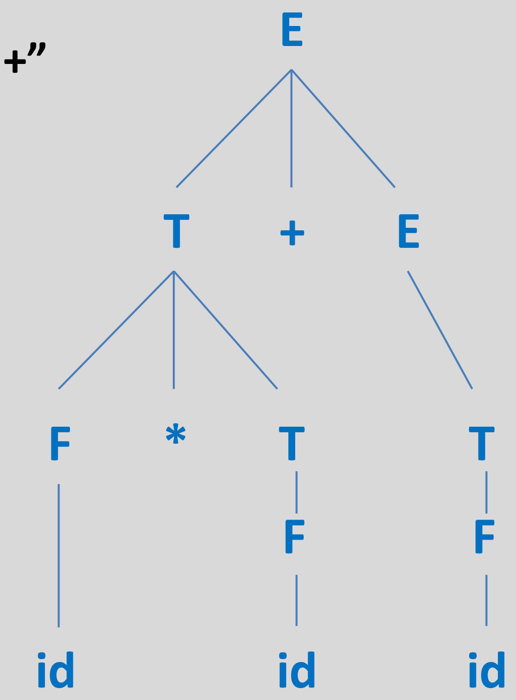
Disambiguating rule for the arithmetic expressions

"* has a higher priority than +"

$$E \rightarrow T + E \mid \mathrm{T}$$

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

$$F \rightarrow (E) \mid id$$



Left recursion

Some parsing techniques (called top-down parsing) cannot handle

left recursive grammars because they use a leftmost derivation policy

- A grammar is **left recursive** if it has **a nonterminal** $\boldsymbol{A}$ with a derivation $\boldsymbol{A} \Rightarrow^{+} \boldsymbol{A\alpha}$, where α is any sequence of non-terminals and terminals

  - $S \rightarrow Sa|b$ is a left recursive grammar
    - e.g., try a leftmost derivation for an input string a,

      $S \Rightarrow_{lm} Sa \Rightarrow_{lm} Saa \Rightarrow_{lm} Saaa \Rightarrow_{lm} \ldots$ **Infinite loop!!**

  - Q. Is $S \rightarrow Aa|b, A \rightarrow Sb$ a left recursive grammar?

How to eliminate left recursion?

Answer: **rewrite using right-recursion!!**

- e.g., $S \rightarrow Sa|b$ can be rewritten as:

  $\boldsymbol{S \rightarrow bA, \quad A \rightarrow aA|\epsilon}$

- e.g., $S \rightarrow S\alpha_1|S\alpha_2| \ldots |S\alpha_m|\beta_1|\beta_2| \ldots |\beta_n$ can be rewritten as:

  (α_i *and* β_i are any sequence of terminals and non-terminals)

  - Step 1: Make a new nonterminal A and add a production rule $\alpha_i A$ for all α_i and ϵ
    - $\boldsymbol{A \rightarrow \alpha_1 A|\alpha_2 A| \ldots |\alpha_m A|\epsilon}$

  - Step 2: For a nonterminal S, add a production rule $\beta_i A$ for all β_i and discard other rules
    - $\boldsymbol{S \rightarrow \beta_1 A|\beta_2 A| \ldots |\beta_n A, \quad A \rightarrow \alpha_1 A|\alpha_2 A| \ldots |\alpha_m A|\epsilon}$

Left factoring

For a non-terminal, if there are two or more productions

which start with the same input symbol….

- e.g., $E \rightarrow T + E|T,\ T \rightarrow F * T|F,\ F \rightarrow\ (E)|\ id$

  Then, which production should be selected? ….

We need left factoring to discard this confusion

The procedure of left factoring

$$E \rightarrow T + E|T, \qquad T \rightarrow F * T|F, \qquad F \rightarrow (E)|\,id$$

- Step 1: For each non-terminal A, find the longest common prefix of productions α
  - e.g., for E, $\alpha = T$
- Step 2: Discard all productions which have the form of $A \rightarrow \alpha\beta$, and add $A \rightarrow \alpha A'$
  - e.g., $E \rightarrow TE'$
- Step 3: For the new non-terminal A', add $A' \rightarrow \beta$ for all discarded productions in step 2
  - e.g., $E' \rightarrow +E|\epsilon$
- Step 4: Repeat step 1 ~ 3 until there is no more common prefix for all non-terminals
  - $E \rightarrow TE', \quad E' \rightarrow +E|\epsilon, \quad T \rightarrow FT', \quad T' \rightarrow * T|\epsilon, \quad F \rightarrow (E)|id$

For efficient parsing: creating a good CFG

1. **A good CFG is non-ambiguous**
   - We can achieve this by defining disambiguating rules
   - **But, it's not easy..**

2. **A good CFG has no left recursion**
   - We can easily achieve this by rewriting with right recursion

3. **For each non-terminal, a good CFG has only one choice of production starting from a specific input symbol**
   - We can easily achieve this through left factoring

For efficient parsing: creating a good CFG

Examples

Let's rewrite a CFG G: $\boldsymbol{DECL \rightarrow DECL\ type\ id; | DECL\ type\ id = id; | \epsilon}$

(G is non-ambiguous)

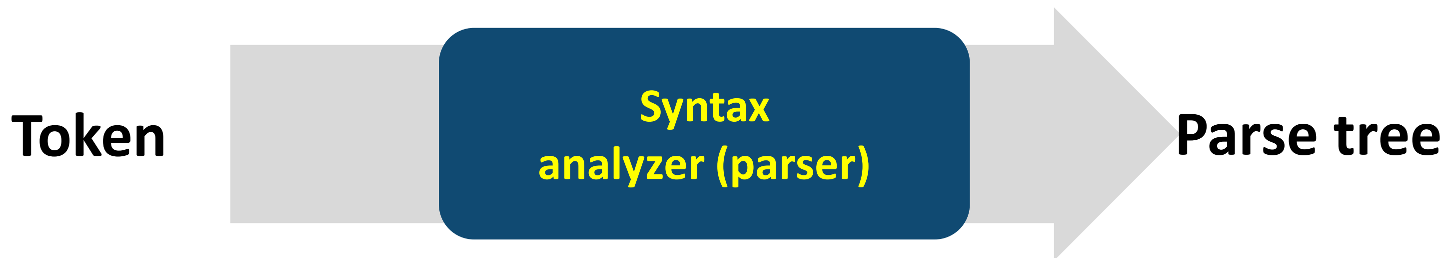
- Step 1: rewrite G by using right recursion

- Step 2: rewrite G by using left factoring

Syntax analyzer

1. Decides whether a given set of tokens is valid or not

2. Creates a tree-like intermediate representation (e.g., syntax tree) that depicts the grammatical structure of the token stream



Then, how to do these processes 1) efficiently and 2) automatically?

A good output: Abstract Syntax Tree (AST)

Abstract syntax trees look like parse trees, but without some parsing details

Example

For an input stream $(A\ + B)\ * C$

$$(\boldsymbol{A}\ + \boldsymbol{B})\ * \boldsymbol{C}$$

$$(\boldsymbol{id}\ +\ \boldsymbol{id})\ *\ \boldsymbol{id}$$

A good output: Abstract Syntax Tree (AST)

Abstract syntax trees look like parse trees, but without some parsing details

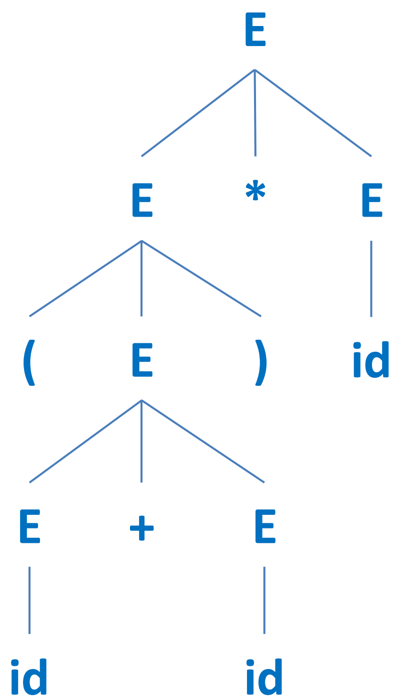
Example

For a token stream $(id\ + id)\ * id$ with **a CFG** $\boldsymbol{G}$: $\boldsymbol{E\ \rightarrow E + E|E * E|(E)|id}$

- An example sequence of derivations

  $E \Rightarrow_{lm} E * E \Rightarrow_{lm} (E) * E$

  $\Rightarrow_{lm} (E + E) * E \Rightarrow_{lm}^{*} (id + id) * id$



- **A parse tree for** $(\boldsymbol{id} + \boldsymbol{id}) * \boldsymbol{id}$ **describes**
  - The sequence of derivations
  - The nesting structure
  - But, too much information…
- **Q) Which nodes can be reduced?**

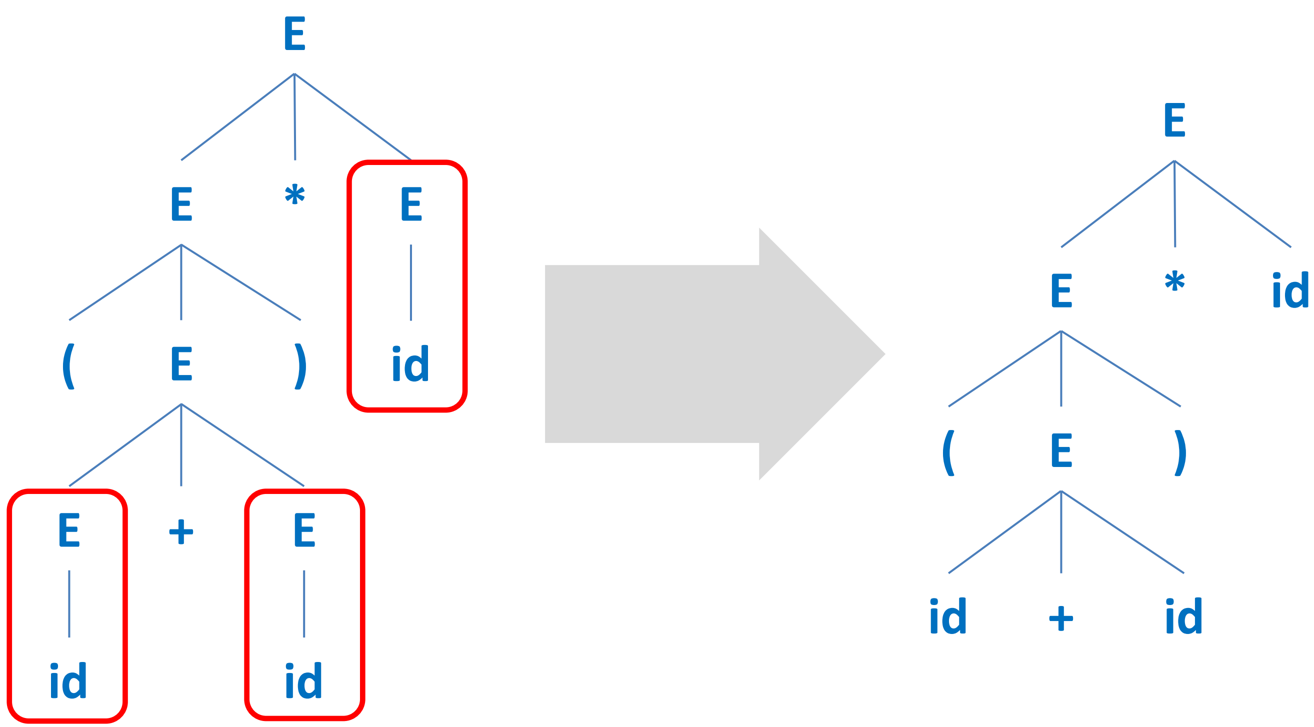
A good output: Abstract Syntax Tree (AST)

Abstract syntax trees look like parse trees, but without some parsing details

Q) Which nodes can be reduced?

1. **Single-successor nodes** which have exactly one child node

   Our main focus is their single child, not the parent nodes.



A good output: Abstract Syntax Tree (AST)

Abstract syntax trees look like parse trees, but without some parsing details

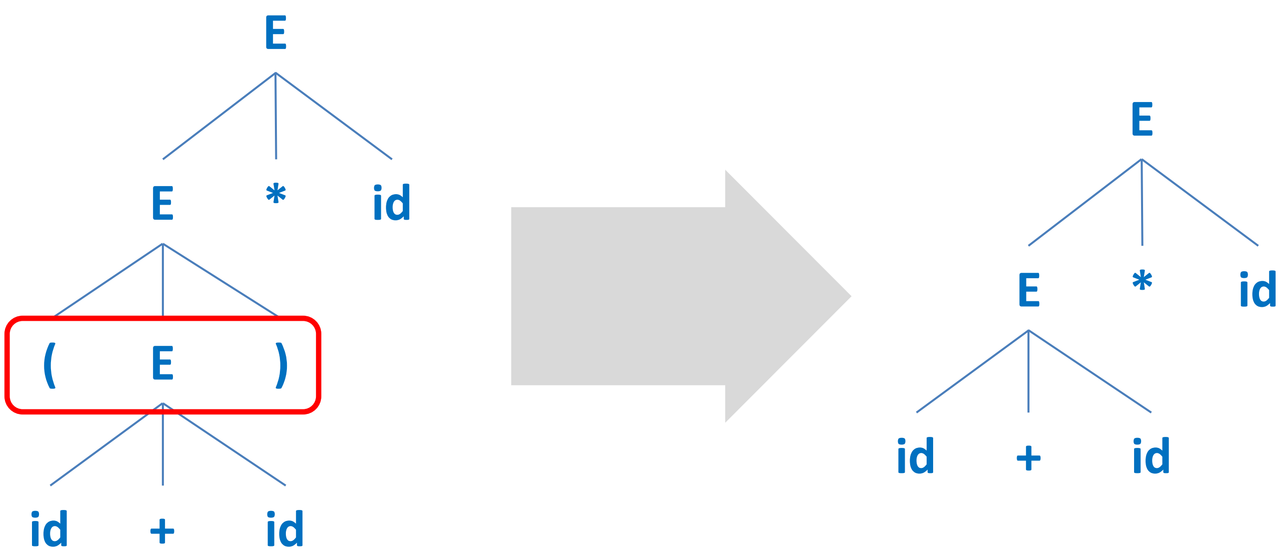
Q) Which nodes can be reduced?

1. **Single-successor nodes** which have exactly one child node

   Our main focus is their single child, not the parent nodes.

2. **Symbols for describing syntactic details** (e.g., parenthesis, comma)

   A parse tree already describes such syntactic information



A good output: Abstract Syntax Tree (AST)

Abstract syntax trees look like parse trees, but without some parsing details

Q) Which nodes can be reduced?

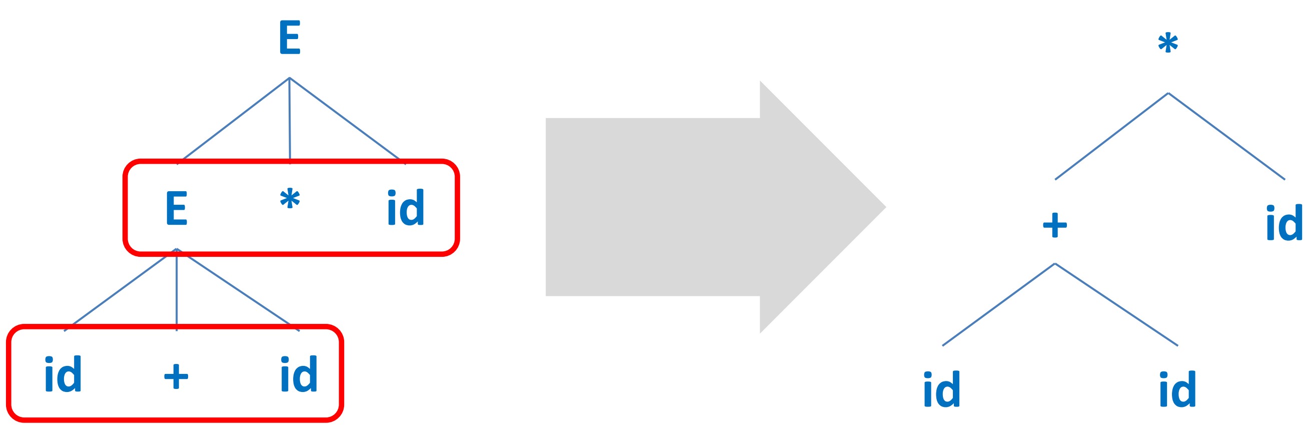
1. **Single-successor nodes** which have exactly one child node

   Our main focus is their single child, not the parent nodes.

2. **Symbols for describing syntactic details** (e.g., parenthesis, comma)

   A parse tree already describes such syntactic information

3. **Non-terminals with an operator and arguments as their child nodes**



A good output: Abstract Syntax Tree (AST)

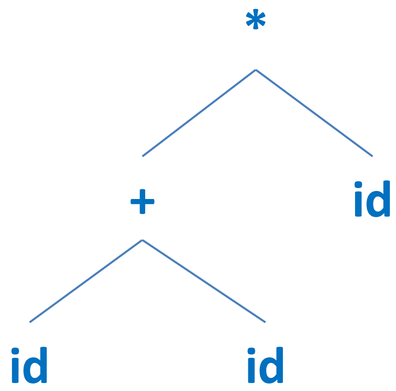
Abstract syntax trees look like parse trees, but without some parsing details

AST for $(id\ *\ id) +\ id$ describes

- The nesting structure (core syntactic information)

Compared to a parse tree

- **More compact**
- **Easier to use and understand**



AST construction

Make semantic actions for each production of a CFG G

Semantic action? An action related with grammar productions

It is also used for type checking, code generation, …

- **Example**

  For a CFG G: $E \rightarrow E + E|E * E|(E)|id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node('+', E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node('*', E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$

AST construction

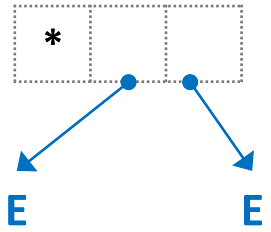
- **Example**

For a CFG G: $E \rightarrow E+E|E*E|(E)|id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node('+', E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node('*', E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$

An example sequence of derivations for $(\boldsymbol{id} + \boldsymbol{id}) * \boldsymbol{id}$

$E \Rightarrow_{lm} E * E$



AST construction

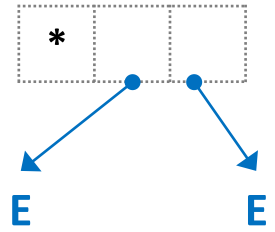
- **Example**

For a CFG G: $E \rightarrow E + E | E * E | (E) | id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node('+', E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node('*', E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$

An example sequence of derivations for $(\boldsymbol{id} + \boldsymbol{id}) * \boldsymbol{id}$

$E \Rightarrow_{lm} E * E \Rightarrow_{lm} (E) * E$



AST construction

- **Example**

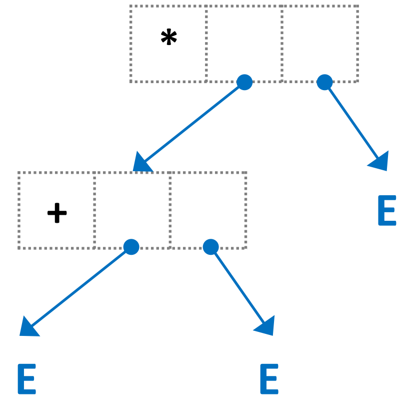
  For a CFG G: $E \rightarrow E + E | E * E | (E) | id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node('+', E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node('*', E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$

An example sequence of derivations for $(id + id) * id$

$E \Rightarrow_{lm} E * E \Rightarrow_{lm} (E) * E$

$\Rightarrow_{lm} (E + E) * E$



AST construction

- **Example**

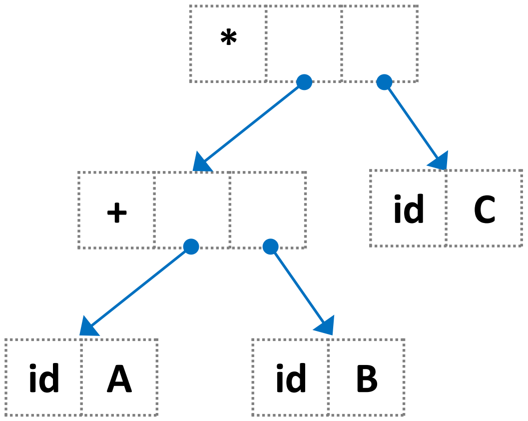
For a CFG G: $E \rightarrow E + E|E * E|(E)|id$

Production	Semantic action
$E \rightarrow E_1 + E_2$	$E.node = new\ Node('+', E_1.node, E_2.node)$
$E \rightarrow E_1 * E_2$	$E.node = new\ Node('*', E_1.node, E_2.node)$
$E \rightarrow (E_1)$	$E.node = E_1.node$
$E \rightarrow id$	$E.node = new\ Leaf(id, id.value)$

An example sequence of derivations for $(id + id) * id$

$E \Rightarrow_{lm} E * E \Rightarrow_{lm} (E) * E$

$\Rightarrow_{lm} (E + E) * E \Rightarrow_{lm}^{*} (id + id) * id$



AST construction

- **Example**

For a CFG $G: S \rightarrow while(C)\{B\}, \quad C \rightarrow id\ comp\ id, \quad B \rightarrow type\ id; \mid id();$

Production	Semantic action
$S \rightarrow while(C)\{B\}$	$S.node = new\ Node(\text{'while'}, C.node, B.node)$
$C \rightarrow id_1\ comp\ id_2$	$C.node = new\ Node('cond', new\ Node('comp', comp.value, new\ Leaf(id_1, id_1.value), new\ Leaf(id_2, id_2.value)))$
$B \rightarrow type\ id;$	$B.node = new\ Node('block', new\ Node('declaration', new\ Leaf(type, type.value), new\ Leaf(id, id.value)))$
$B \rightarrow id();$	$B.node = new\ Node('block', new\ Node('call', new\ Leaf(id, id.value))$

Let's construct AST for $while\ (leftVar < rightVar)\{int\ a;\}$

- After lexical analysis: $while(id\ comp\ id)\{type\ id;\}$
- A sequence of derivations for the input string

  $S \Rightarrow_{lm} while(C)\{B\} \Rightarrow_{lm} while(id\ comp\ id)\{B\} \Rightarrow_{lm} while(id\ comp\ id)\{type\ id;\}$

Summary: AST

Abstract syntax trees look like parse trees, but without some parsing details

We can eliminate the following nodes in parse trees

1. Single-successor nodes
2. Symbols for describing syntactic details
3. Non-terminals with an operator and arguments as their child nodes

AST can be constructed by using semantic actions

The semantic actions can be also used for type checking, code generation, …

Syntax analyzer