Lecture 04

Syntax Analyzer (Parser)

Part 1 (more): Abstract syntax tree

Hyosu Kim

School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

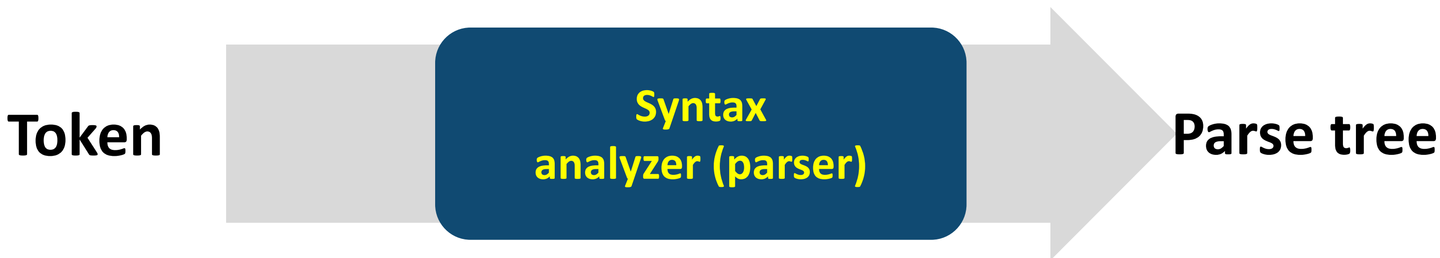
https://sites.google.com/view/hyosukim

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

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**
   - 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

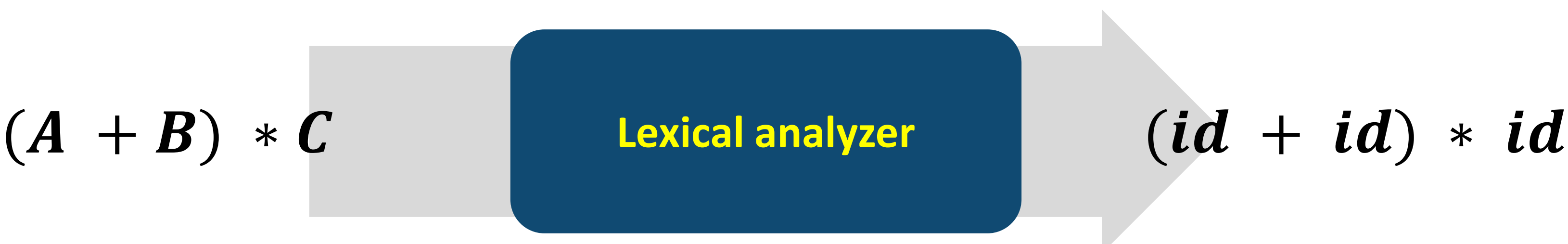
- Step 2: rewrite G by using left factoring

A good output: Abstract Syntax Tree (AST)

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

Example

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



A good output: Abstract Syntax Tree (AST)

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

Example

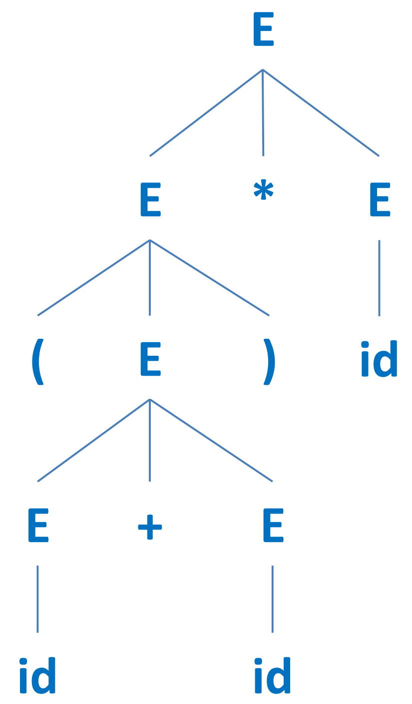
For a token stream $(id\ + id)\ * id$ with **a CFG** $\boldsymbol{G}$: $\boldsymbol{E\ \rightarrow E + E|E * E|(E)|id}$

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



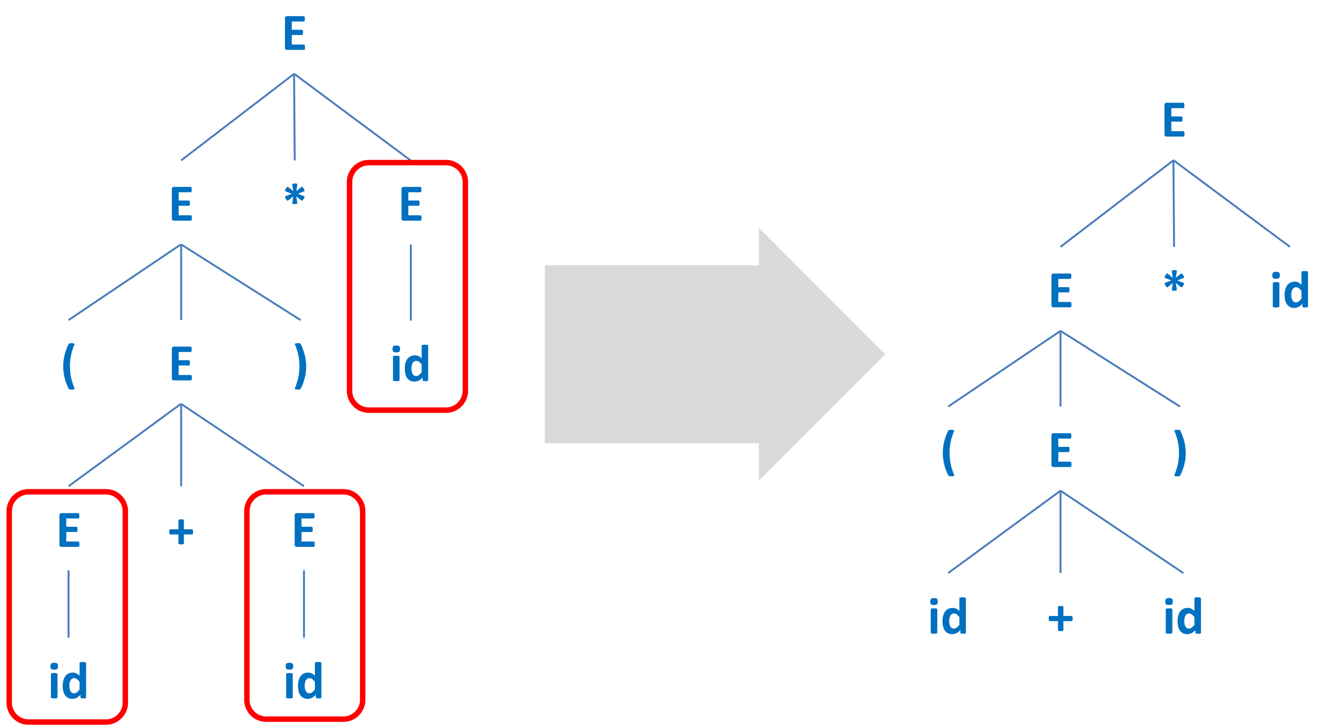
A good output: Abstract Syntax Tree (AST)

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

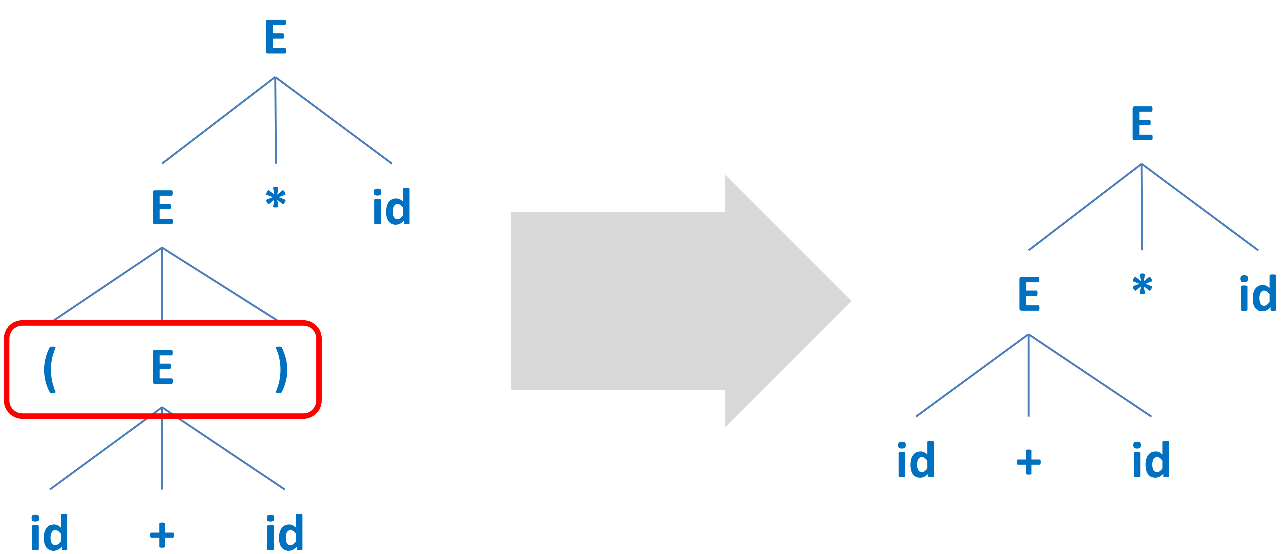
Q) Which nodes can be reduced?

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?

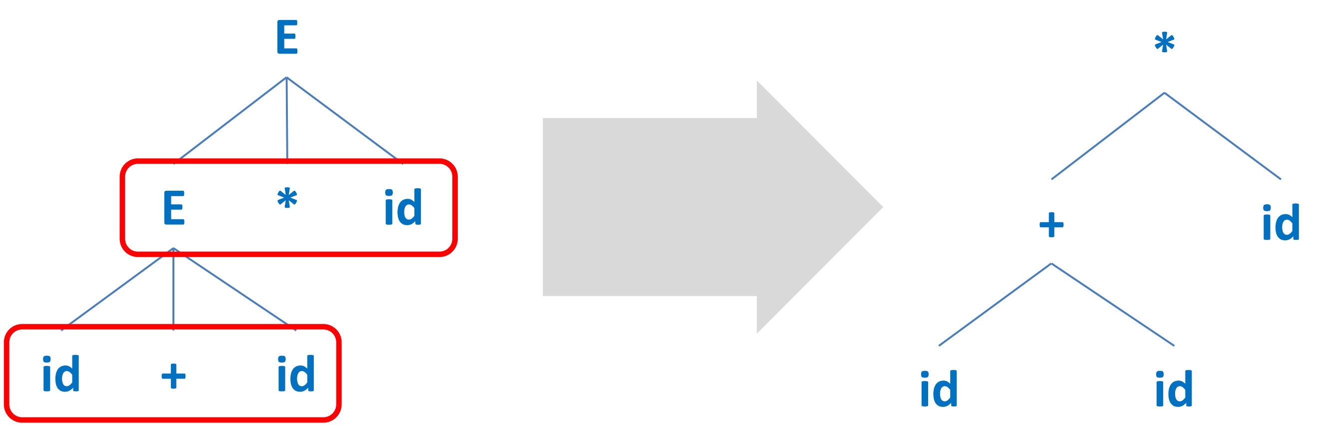
1. **Single-successor nodes** which have exactly one child node

   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)

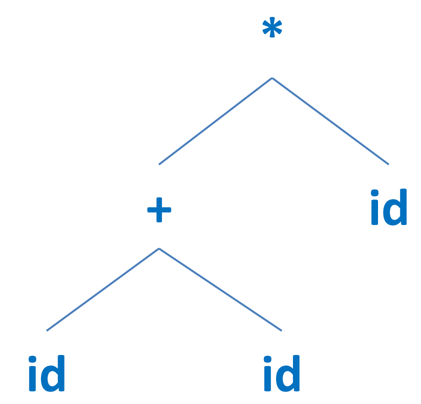
Abstract syntax trees look like parse trees, but without some parsing details

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

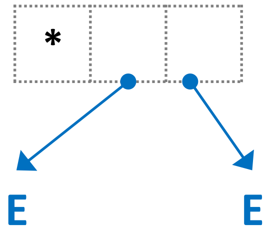
- **Example**

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

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

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

$E \Rightarrow_{lm} E * E$



AST construction

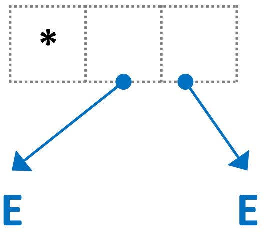
- **Example**

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 $(id + id) * id$

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



AST construction

- **Example**

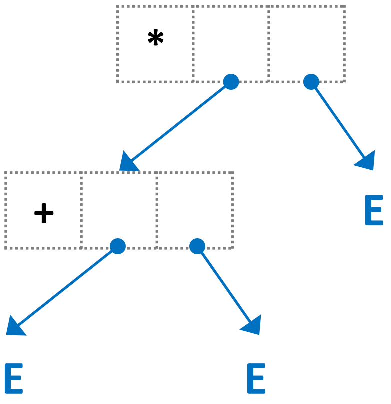
For a CFG G: $E \rightarrow E + E | E * E | (E) | id$

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

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

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



AST construction

- **Example**

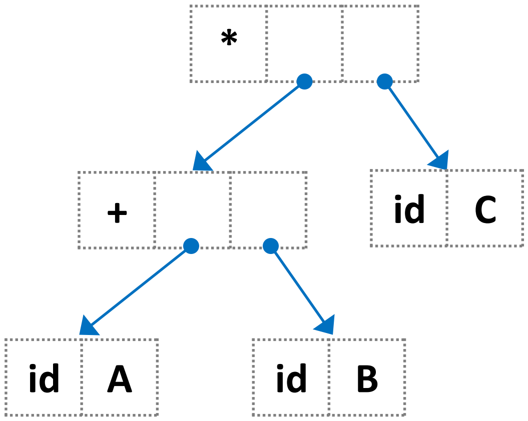
For a CFG G: $E \rightarrow E + E | E * E | (E) | id$

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: $S \rightarrow while(C)\{B\}$, $C \rightarrow id\ comp\ id$, $B \rightarrow type\ id; \mid id();$

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