Compiler
– 3-2. Context Free Grammar –

JIEUNG KIM

jieungkim@yonsei.ac.kr

Where are we?

Source Code

- Lexical Analysis
- **Syntax Analysis**
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

Machine Code

YONSEI UNIVERSITY

Outlines

- Role of the syntax analysis (parser)
- Context free grammar
- Push down automata
- Top-down parsing
- Bottom-up parsing
- Simple LR
- More powerful LR parsers and other issues in parsers
- Syntactic error handler
- Parser generator

Context free grammar

Context free grammar

- Formal languages
  - An alphabet is a set Σ of symbols that act as letters.
  - A language over Σ is a set of strings made from symbols in Σ.
  - When scanning, our alphabet was ASCII or Unicode characters.
  - We produced tokens.
  - When parsing, our alphabet is the set of tokens produced by the scanner.

Context free grammar

- The limits of regular languages
  - When scanning, we used regular expressions to define each token.
  - Unfortunately, regular expressions are (usually) too weak to define programming languages.
    - Cannot define a regular expression matching all expressions with properly balanced parentheses.
    - Cannot define a regular expression matching all functions with properly nested block structure.
  - We need a more powerful formalism.

YONSEI UNIVERSITY

Context free grammar

- Context free grammar
  - We have seen many languages that can <u>not</u> be regular
  - Context Free Grammars (CFGs) can describe **syntax** in programming languages
  - CFGs are basis of BCF (Backus Naur Form) syntax
  - A strict superset of the the regular languages.
  - CFGs are best explained by the example in the next slide.

Context free grammar

- Arithmetic expression
  - Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
  - Here is one possible CFG:

E → int
E → E Op E
E → (E)
Op → +
Op → −
Op → *
Op → /

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

E
⇒ E Op E
⇒ E Op int
⇒ intOp int
⇒ int /int

Context free grammar

- Context free grammar
  - Formally, a context-free grammar is a collection of four objects:
    - A set of nonterminal symbols (or variables),
    - A set of terminal symbols,
    - A set of production rules (→) saying how each nonterminal can be converted by a string of terminals and nonterminals, and
    - A start symbol (E in the example) that begins the derivation

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

E → int | E Op E | (E)

Op → + | - | * | /

YONSEI UNIVERSITY

Context free grammar

- Context free grammar – for simpler form

E → int
E → E Op E
E → (E)
Op → +
Op → −
Op → *
Op → /

E → int | E Op E | (E)

Op → + | − | * | /

Context free grammar

- Context free grammar – for simpler form
  - The syntax for regular expressions does not carry over to CFGs
  - Cannot use *, |, or parentheses.

S → a*b

⇓

S → Ab

⇓

S → Ab
A → Aa|ε

S → a(b|c*)

⇓

S → aX
X → (b|c*)

⇓

S → aX
X → b|c*

⇓

S → aX
X → b|C

⇓

S → aX
X → b|C
C → Cc|ε

Context free grammar

- Context free grammar
  - A grammar G = (V, T, S, P) is called **Context Free (CFG)** if all the production rules of the form:

$$A \rightarrow \alpha \qquad \text{(where } A \in V,\ \alpha \in \{V \cup T\}^*\text{)}$$

  - I.e., One variable in the left-hand side, no restrictions in the right-hand side
  - Regular grammar is a subset of context free grammar

$$A \rightarrow Bx \mid x \quad \text{or} \quad A \rightarrow xB \mid x \qquad \text{(where } A, B \in V,\ x \in T^*\text{)}$$

Context free grammar – self-study page

- Examples
  - G_1 = ({S}, {0, 1}, S, P) where
    - P : S → 0S1
    - S → ε
  - G_2 = ({S}, {0, 1}, S, P) where
    - P : S → S10
    - S → ε
  - G_3 = ({S, A}, {0, 1}, S, P) where
    - P : S → 0A1
    - 0A → 00A1
    - A → ε

Context free grammar – self-study page

- Examples (CFGs for programming languages)

```
BLOCK      →  STMT
           |  {STMTS }

STMTS      →  ε
           |  STMT STMTS

STMT       →  EXPR;
           |  if (EXPR) BLOCK
           |  while (EXPR) BLOCK
           |  doBLOCK while (EXPR);
           |  BLOCK
           |  …

EXPR       →  identifier
           |  constant EXPR
           |  +EXPR EXPR -
           |  EXPR EXPR *
           |  EXPR
           |  . . .
```

Context free grammar

- Derivations
  - Let $G = (V, T, S, P)$ be a CFG where $A \in V$, $\alpha, \beta \in \{V \cup T\}^*$
  - Derivations ($\Rightarrow$)
    - Suppose $A \rightarrow \gamma \in P$. Then we can write $\alpha A \beta \Rightarrow \alpha \gamma \beta$, then, we say that $\alpha A \beta$ *derives* $\alpha \gamma \beta$
  - We define $\Rightarrow^*$ to be *reflexive* and *transitive* closure of $\Rightarrow$ as:
    - **Basis**: Let $\alpha \in \{V \cup T\}^*$. Then, $\alpha \Rightarrow^* \alpha$
    - **Induction** : If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow^* \gamma$, then $\alpha \Rightarrow^* \gamma$

YONSEI UNIVERSITY

Context free grammar

- Sentential forms
  - A grammar defines a language in a recursive way
  - Sentential form ($\in \{V \cup T\}^*$)
    - S is a sentential form
    - If **αβɣ** is a sentential form and **β → δ** is in P, then **αδɣ** is also a sentential form
  - Sentence ($\in T^*$)

Context free grammar

- Context free languages
  - A language accepted by a grammar G $L(G) = \{ w \mid S \Rightarrow w \text{ and } w \in T^*\}$
    - if $w \in L(G)$, then there is a derivation step:
      $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_k \Rightarrow w$ where each $w_i \in \{V \cup T\}^*$, $w \in T^*$
  - If G is a CFG, then we call L(G) **Context Free Language (CFL)**

Context free grammar

- Examples
  - $L(G) = \{a^{2n}b^n \mid n \geq 0\}$
    - CFG G = ({S}, {a, b}, S, P)
    - P : S → aaSb | ε
  - $L(G) = \{a^m b^n \mid n \leq m \leq 2n\}$
    - CFG G = ({S}, {a, b}, S, P)
    - P : S → aSb | aaSb |ε
  - $L(G) = \{a^{m+n}b^m c^n \mid m \geq 0, n \geq 1\}$
    - CFG G = ({S, A}, {a, b, c}, S, P)
    - P : S → aSc | aAc
      A → aAb | ε

- Examples
  - $L(G) = \{ w \in \{a, b\}^* \mid w = w^R, |w| \text{ is even}\}$
    - CFG G = ({S}, {a, b}, S, P)
    - P : S → aSa | bSb | ε
  - $L(G) = \{ w \in \{a, b\}^* \mid w = w^R, |w| \text{ is odd}\}$
    - CFG G = ({S}, {a, b}, S, P)
    - P : S → aSa | bSb | a | b
  - $L(G) = \{a^m b^n c^n d^m \mid m, n \geq 1\}$
    - CFG G = ({S, A}, {a, b, c, d}, S, P)
    - P : S → aSd | aAd
      A → bAc | bc

Context free grammar

- Examples (CFGs that generate regular languages)
  - L(G) = set of all regular expressions over {a, b}
    - CFG G = ({S}, {a, b, +, *, · , (,)}, S, P)
    - P : S → S+S | S·S | S* | a | b | ε| ∅
  - L(G) = $(a + b)^*ab(ab + b)^+$
    - CFG G = ({S}, {a, b}, S, P)
    - P : S → AabB
    - A → aA | bA | ε
    - B → Bab | Bb |ab | b

YONSEI UNIVERSITY

Context free grammar

- Is CFG unique?
  - CFG G_1 = ({S, A, B}, {0, 1}, S, P) where
    - P : S → 0B | 1A
    - A → 1AA | 0S | 0
    - B → 0BB | 1S | 1
  - CFG G_2 = ({S, A}, {a, b}, S, P)
    - P : S → 0S1S | 1S0S | ε
  - CFG G_3 = ({S, A}, {a, b}, S, P)
    - P : S → 0S1 | 1S0 | SS | ε
  - Then, $L(G_1) = L(G_2) = L(G_3) = \{w \in \{0, 1\}^* \mid NUM_0(w) = NUM_1(w)\}$

Note: There exist many distinct CFG's for the same CFL

Context free grammar

- Applications of CFGs
  - L_{bal} : a set of all valid strings of balanced parenthesis
    - G = ({S}, {(,)}, S, P) where
    - P : S → (S) | SS | ε
    - Examples of strings :
      - (()), ()(), (()()), ε are accepted, but ((),)(, are not
    - Begin({), End(}) in C program
      - Replace (by { and) by }

YONSEI UNIVERSITY

Context free grammar

- Applications of CFGs
  - L_{ifelse} : a set of all valid strings of **if**'s and **else**'s
    - G = ({S}, {*i*, *e*}, S, P) where
    - P : S → *i* S | *i* S *e* S | SS | ε
    - Examples of strings: *ieie*, *ie* are accepted, but *ei*, *ieeii* are not.
    - **if** can be unbalanced by any **else**

YONSEI UNIVERSITY

Context free grammar

- **Testing a string with CFG**
  - Given a CFL L and a string w, we want test if w ∈ L or not
  - Example:
    - G = ({S}, {0, 1}, S, P) where
    - P : S → 0S1 | 1S0 | SS | ε
    - Is w = 011100 ∈ L(G) or not?
  - There are two methods:
    - Derivations
      - Leftmost Derivations
      - Rightmost Derivations
    - Parsing Trees

Context free grammar

- **Leftmost and rightmost derivations**
  - There are two kinds of derivations:
    - **Leftmost derivation**: Always replace the leftmost variable
    - **Rightmost derivation**: Always replace the rightmost variable
  - Consider

$$\text{CFG } G = (\{E, T, F\}, \{*, +, (,), a\}, E, P)$$
$$P: E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid a$$

  - And when $w = a + a$
    - **Leftmost derivation**: $E \Rightarrow E{+}T \Rightarrow T{+}T \Rightarrow F{+}T \Rightarrow a{+}T \Rightarrow a{+}F \Rightarrow a{+}a$
    - **Rightmost derivation**: $E \Rightarrow E{+}T \Rightarrow E{+}F \Rightarrow E{+}a \Rightarrow T{+}a \Rightarrow F{+}a \Rightarrow a{+}a$

YONSEI UNIVERSITY

Context free grammar

- **Parse trees**
  - Let $G = (V, T, S, P)$ be a CFG. A tree is a **parse tree** if
    - Root is labeled by S
    - Each leaf node is labeled from $T \cup \{\varepsilon\}$
    - Each non-leaf node is labeled from V
    - If a node has a label $A \in V$ and its children (from left to right) are labeled A_1, A_2, ... , A_n, then $A \rightarrow A_1, A_2, \ldots, A_n \in P$
  - **Yield** of a parse tree is a string of *leaves* from *left to right*.
    - Yield consists of only terminal symbols
    - A set of yields the language of the grammar

YONSEI UNIVERSITY

Context free grammar – self-study page

- Parse trees

E

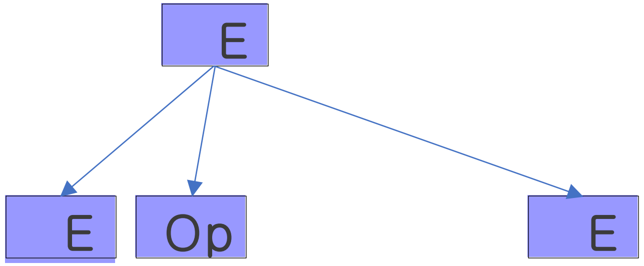
E

Context free grammar – self-study page

- Parse trees

E

$\Rightarrow E \; Op \; E$



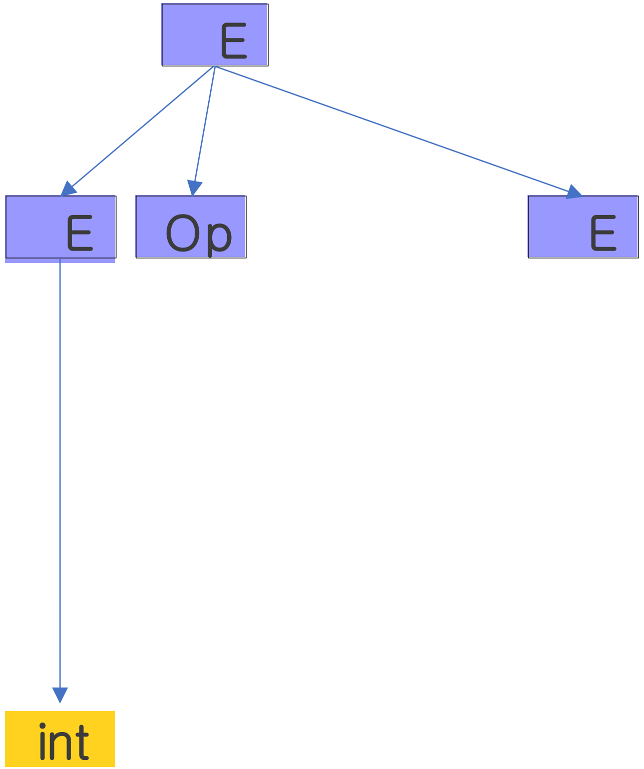
Context free grammar – self-study page

- Parse trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ intOp E



Context free grammar – self-study page

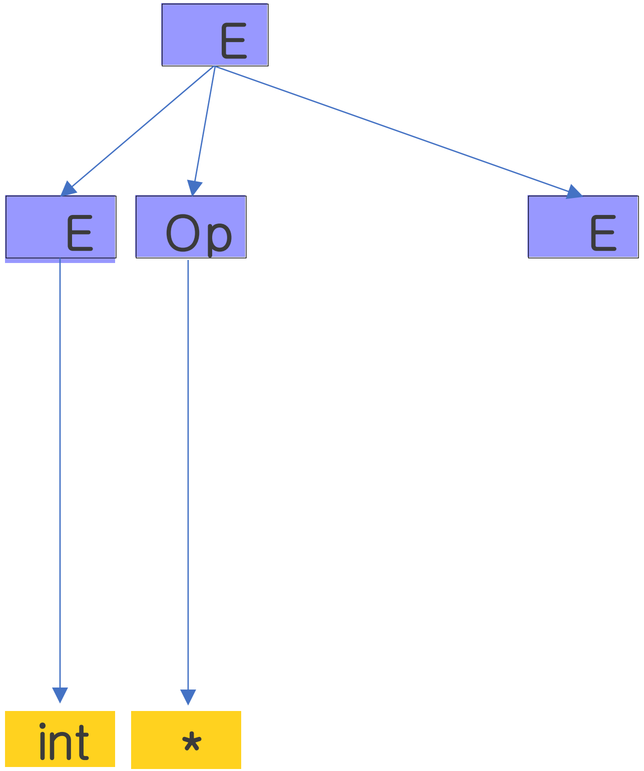
- Parse trees

E

⇒ E Op E

⇒ intOp E

⇒ int *E



Context free grammar – self-study page

- Parse trees

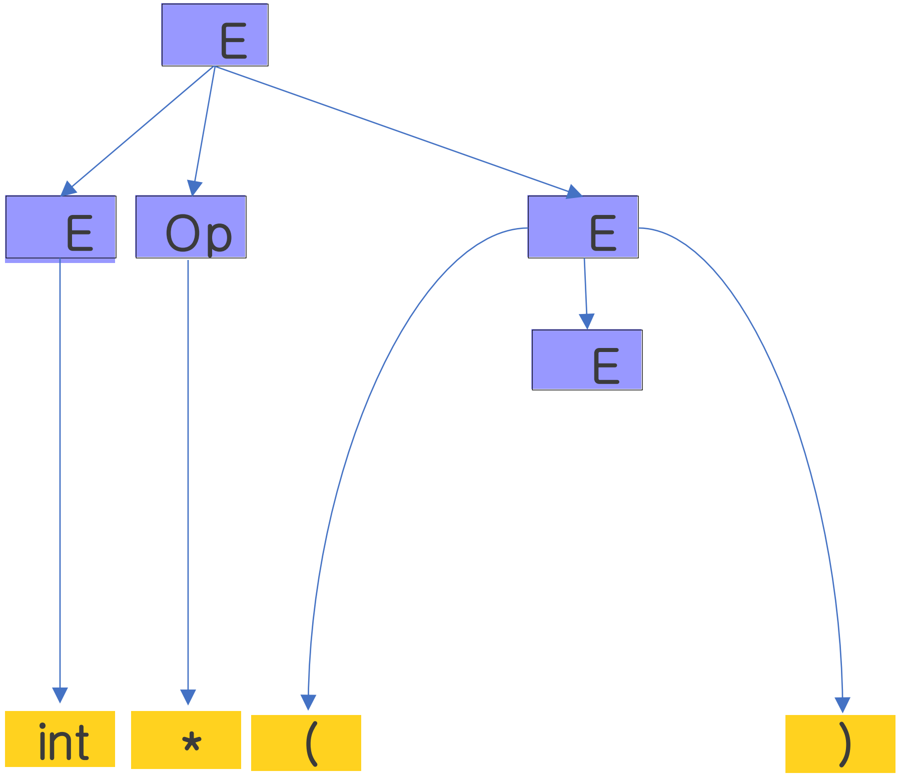
E

$\Rightarrow$ E Op E

$\Rightarrow$ intOp E

$\Rightarrow$ int *E

$\Rightarrow$ int * (E)



Context free grammar
– self-study page

- Parse trees

E

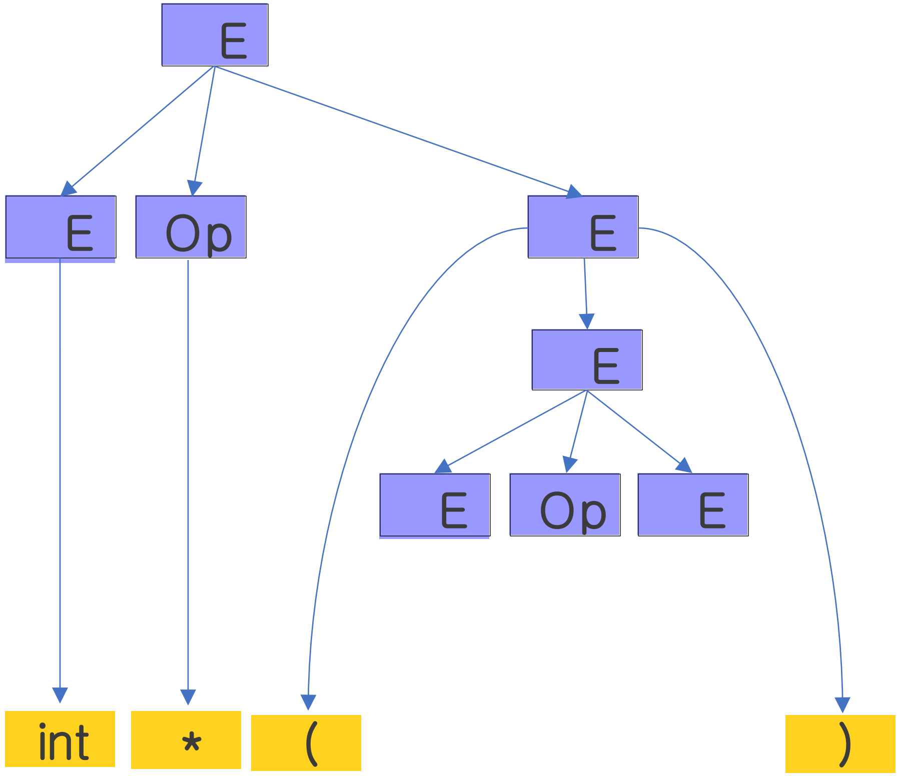
⇒ E Op E

⇒ intOp E

⇒ int *E

⇒ int * (E)

⇒ int * (E Op E)



Context free grammar – self-study page

- Parse trees

E

⇒ E Op E

⇒ intOp E

⇒ int *E

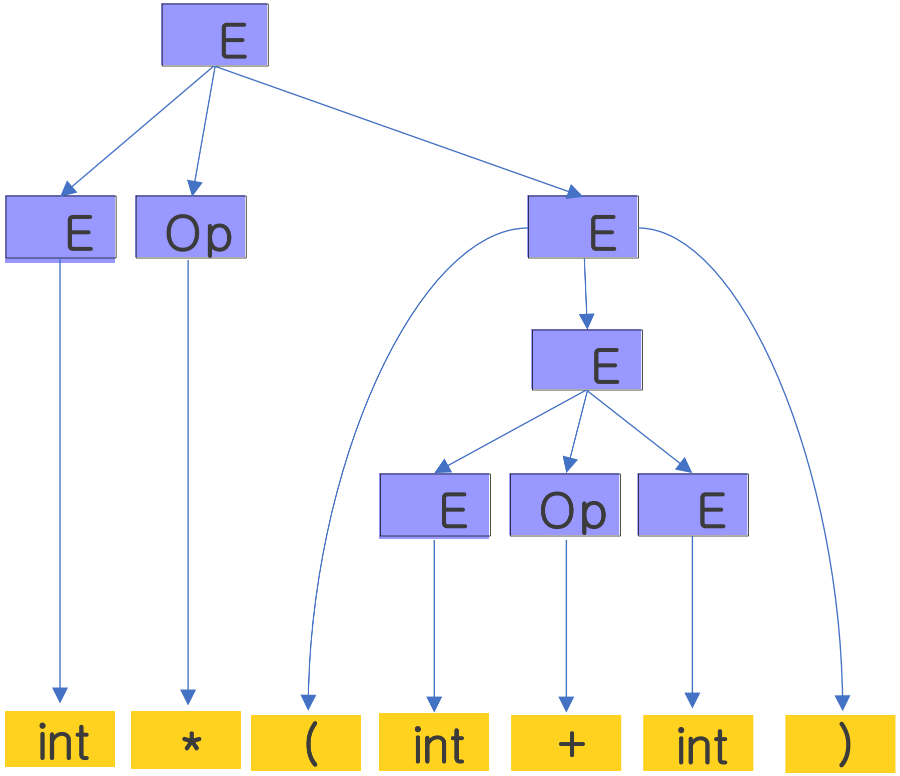
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (intOp E)

⇒ int * (int+E)

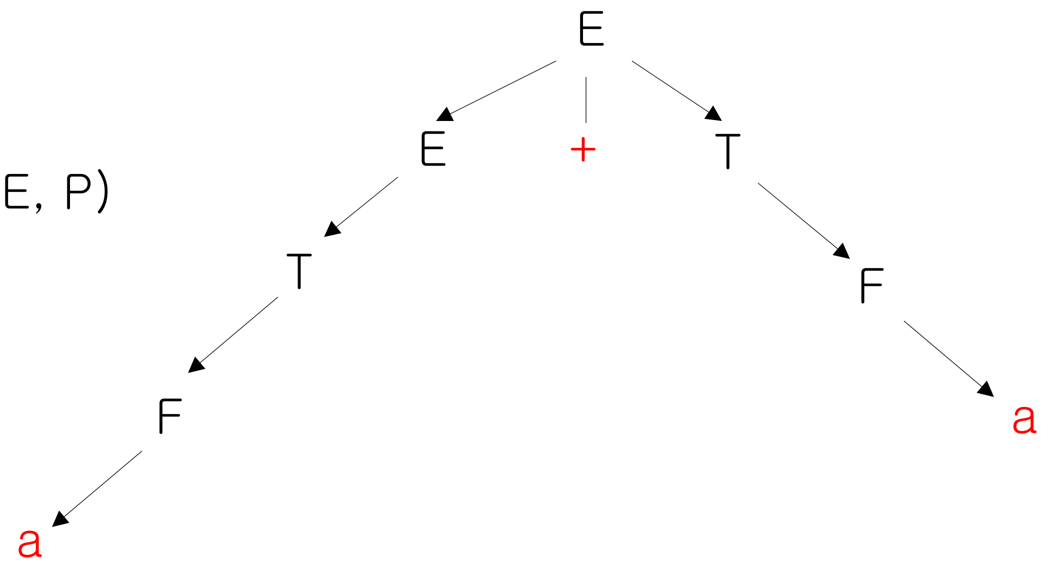
⇒ int * (int + int)



Context free grammar

- Parse trees

CFG G = ({E, T, F}, {*, +, (,), a}, E, P)
  P : E → E + T | T
  T → T * F | F
  F → (E) | a



Parse Tree

Is w = a + a ∈ L(G) ? Yes, because we have a yield a + a

Context free grammar

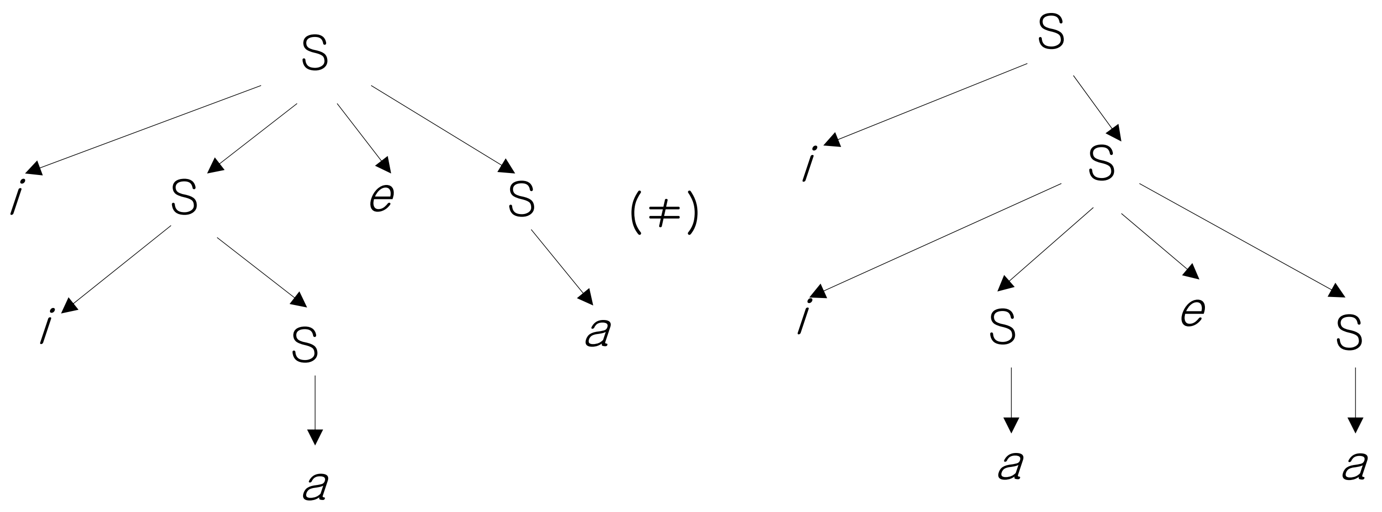
- **Parse tree vs. derivations**
  - If a string w ∈ L(G) for some CFG G, then
    - there exists a **parse tree** for w
    - there exists a **leftmost derivation** for w
    - there exists a **rightmost derivation** for w
  - Let G = (V, T, S, P) be a CFG and A ∈ V, then the followings are all equivalent
    - $A \overset{*}{\Rightarrow} w$
    - $A \overset{*}{\underset{lm}{\Rightarrow}} w$
    - $A \overset{*}{\underset{rm}{\Rightarrow}} w$
    - There is a parse tree of G with root A and yield w

Context free grammar

- Parse tree vs. derivations
  - Parse trees are alternative representation to derivations
    - This gives us syntactic structure of w
  - Given string w, is a parse tree always unique? No!
    - There can be **several** parse trees for the same string
  - Ideally, there should be only one parse tree (unambiguous structure) for each string

Context free grammar

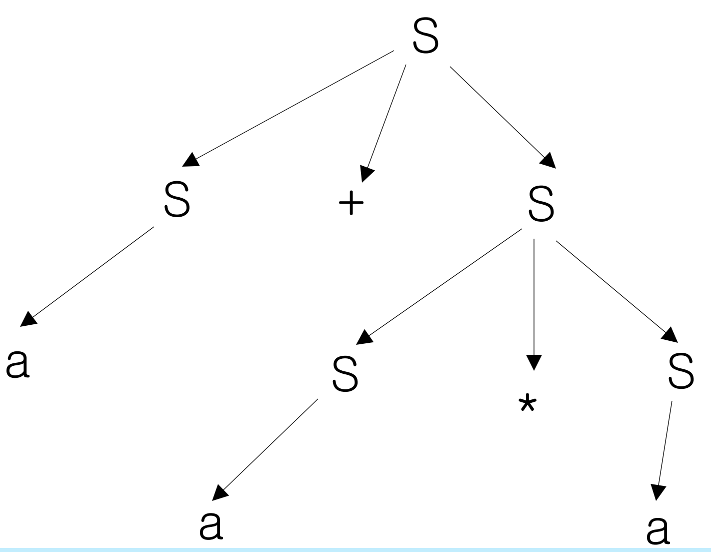
- Ambiguity in grammars
  - G = ({S}, {*i*, *e*, *a*}, S, P) where P : S → *i* S *e* S | *i* S | *a*
  - Is w = *iiaea* accepted? Yes, it has two leftmost derivations
    - S ⇒ *i* S *e* S ⇒ *ii* S *e* S ⇒ *iiae* S ⇒ *iiaea*
    - S ⇒ *i* S ⇒ *ii* S *e* S ⇒ *iiae* S ⇒ *iiaea*
  - This also gives us two parse trees:



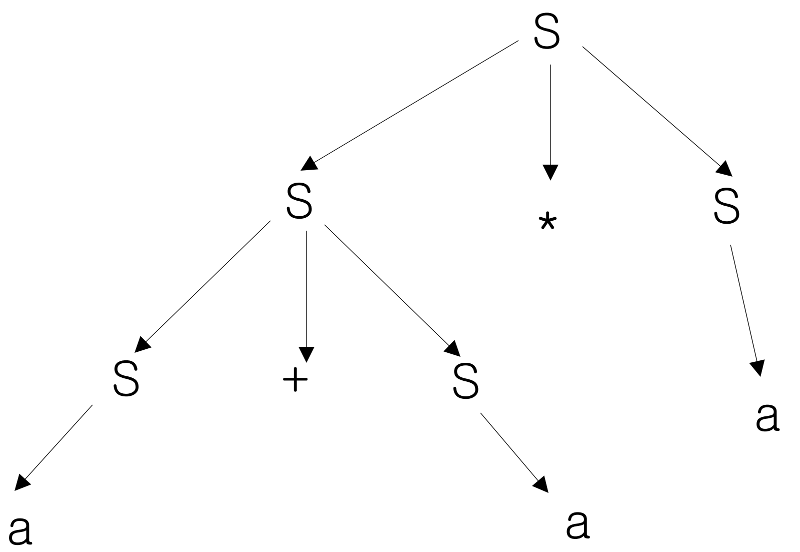
Context free grammar

- Ambiguity in grammars
  - G = ({S}, {a, +, ⋆, (,)}, S, P) where P : S → S + S | S ⋆ S | (S) | a
  - w = a+a⋆a has two leftmost derivations:
    - S ⇒ S+S ⇒ a+S ⇒ a+S⋆S ⇒ a+a⋆S ⇒ a+a⋆a
    - S ⇒ S⋆S ⇒ S+S⋆S ⇒ a+S⋆S ⇒ a+a⋆S ⇒ a+a⋆a
  - This also gives us two parse trees:

(a + (a ⋆ a))



(≠)



((a + a) ⋆ a)

Context free grammar

- **Ambiguity definition**
  - A **sentence** (or string) w ($\in T^*$) is ambiguous if it has **more than one** parse tree
  - A **CFG G** is ambiguous if it has an ambiguous sentence
    - I.e., If there are more than 2 parse trees for the sentence, the G is ambiguous
  - A **CFL L** is ambiguous if it has an ambiguous CFG

Context free grammar

- **Deciding ambiguity**
  - Given CFG G, we want know if G is ambiguous?
    - Guess first
    - If ambiguous, find any string w that has 2 parse trees
      - I.e., find at least one ambiguous sentence
    - If not, you must show that every string has only 1 parse tree
      - Show that all strings have unique parse trees
  - Given CFG G, can we decide whether G is ambiguous or not? NO!
    - Bad News: there is no algorithm to do it
  - **Deciding ambiguity is undecidable (=Unsolvable)**

Context free grammar

- Example – deciding ambiguity
  - G = ({S, A, B}, {a, b}, S, P) where
    P : $S \rightarrow A \mid B$
    $A \rightarrow a$
    $B \rightarrow a$
  - G = ({S}, {a, b}, S, P) where P : $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Context free grammar

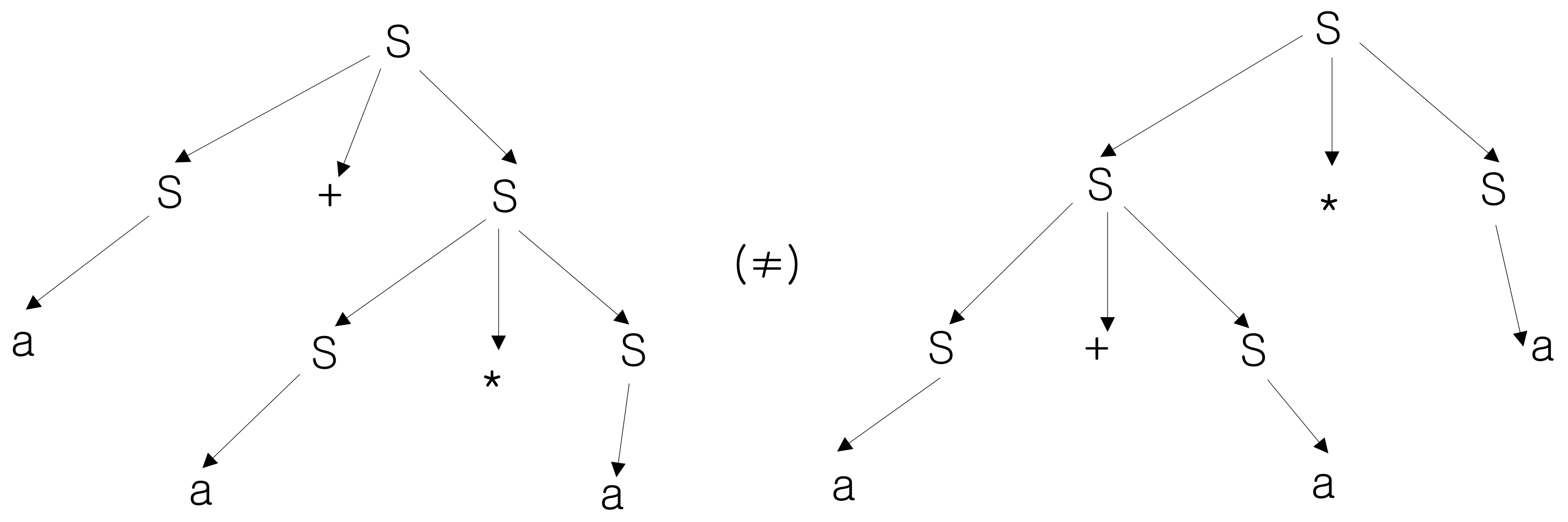
- **Deciding ambiguity**
  - Typical ambiguous patterns (with $A, B \in V$, $\alpha, \beta \in \{V \cup T\}^*$)
    - $A \rightarrow A\alpha A$
    - $A \rightarrow \alpha A \mid \beta A$
    - $A \rightarrow AA \mid \alpha$
    - $A \rightarrow \alpha A \mid \alpha A \beta A$
  - Example
    - $G = (\{S\}, \{a, b\}, S, P)$ where P : $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Context free grammar

- Removing ambiguity
  - Given ambiguous CFG G, can we remove ambiguity? No!
    - Bad News: There is no algorithm it
    - More Bad News: Some CFL's have only ambiguous grammars
    - Good News: We can do it by hand
  - Again, removing ambiguity is undecidable (=Unsolvable)
  - But, there is a hope
    - 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.

Context free grammar

- Removing ambiguity
  - G = ({S}, {a, +, *, (,)}, S, P) where P : S → S + S | S * S | (S) | a
  - This is ambiguous because it has two parse trees: w = a + a * a



(a + (a * a))   ((a + a) * a)

Context free grammar

- Removing ambiguity
  - Grammar: G = ({S}, {a, +, *, (,)}, S, P) where P : S → S + S | S * S | (S) | a
  - Two problems
    - No operator precedence
      - For w = a + a * a, we can accept both (a + (a * a)) and ((a + a) * a) without considering precedence
      - Solution: set priority with * > +
    - Grouping is available for operators with the same precedence
      - We can accept both (S + S) + S and S + (S + S)
      - We need to select one of them to remove ambiguity
      - In general, we select left → right rule (exception: exponential operator)

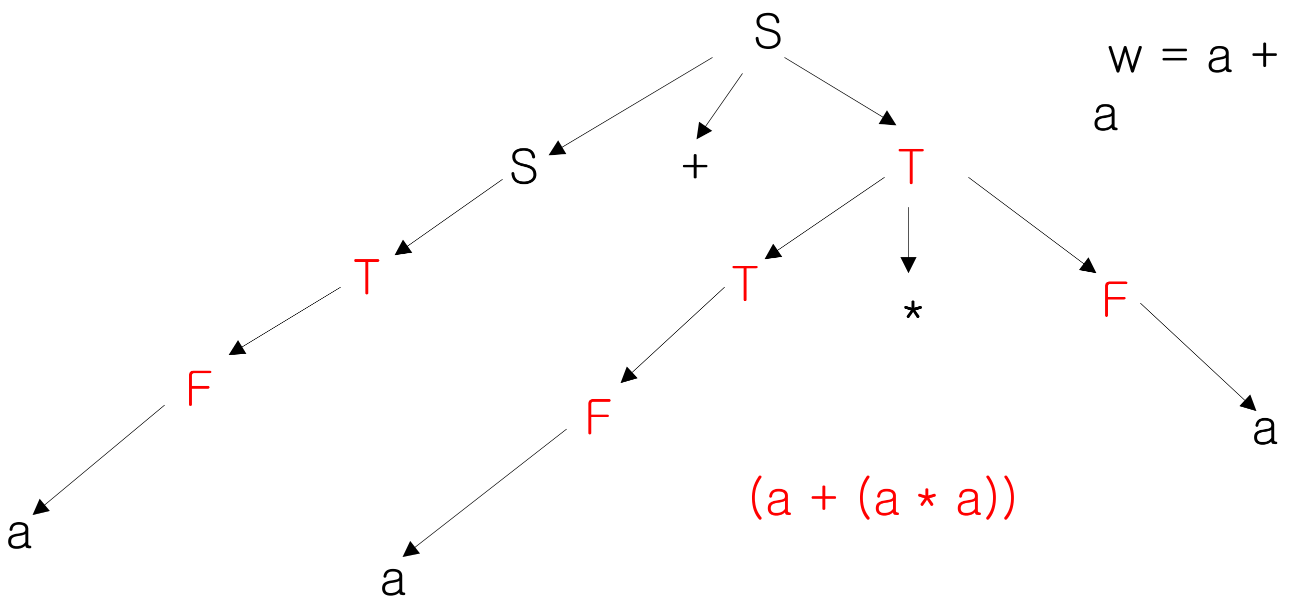
Context free grammar

- Example – removing ambiguity
  - G = ({S}, {a, +, *, (,)}, S, P) where P : S → S + S | S * S | (S) | a
  - Solution: We introduce **new** variables: {T, F}
  - G' = ({S, T, F}, {a, +, *, (,)}, S, P) where

    P : S → S + T | T

    T → T * F | F

    F → (S) | a



w = a + a * a

(a + (a * a))

Context free grammar

- Examples
  - CFG G = ({S}, {0, 1}, S, P) where P : S → 0S1 | 00S1 | ε
    - Equivalent unambiguous CFG G':
      P : S → 0S1 | T
      T → 00T1 | ε
  - CFG G = ({S}, {0, 1}, S, P) where P : S → 0S1 | 01S | ε
    - Equivalent unambiguous CFG G':
      P : S → T | ε
      T → 0T1 | 01T | 01

Context free grammar

- Inherent ambiguity
  - A CFL L is **inherently ambiguous** if *all* grammars for L are ambiguous
  - $L = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous
    - Is L CFL? Yes, because of the following reasons
      - $L = L_1 \cup L_2 = \{a^i b^i c^j\} \cup \{a^i b^j c^j\}$
      - $G_1 : S_1 \rightarrow S_1 c \mid A$ $\qquad$ $G_2 : S_2 \rightarrow aS_2 \mid B$
      - $A \rightarrow aAb \mid \varepsilon$ $\qquad$ $B \rightarrow bBc \mid \varepsilon$
    - Now, L is accepted by combining G_1 and G_2, and adding $G : S \rightarrow S_1 \mid S_2$
    - G is ambiguous since $w = a^i b^i c^i$ ($\in L$) has two leftmost derivations
    - Example: w = aabbcc
      - $S \Rightarrow S_1 \Rightarrow S_1 c \Rightarrow S_1 cc \Rightarrow Acc \Rightarrow aAbcc \Rightarrow aaAbbcc \Rightarrow aabbcc$
      - $S \Rightarrow S_2 \Rightarrow aS_2 \Rightarrow aaS_2 \Rightarrow aaB \Rightarrow aabBc \Rightarrow aabbBcc \Rightarrow aabbcc$

CFG & parsing

- **The goal of parsing**
  - 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 the input string.
  - Usually, we do some simplifications on the tree; more on that later.
  - We'll discuss how to do this next week.

CFG & parsing

- **Parsing**
  - The test if $w \in L$ or not given a CFL L and a string w
  - Example
    - $G = (\{S\}, \{0, 1\}, S, P)$ where $P : S \rightarrow 0S1 \mid 1S0 \mid SS \mid \varepsilon$
    - Is $w = 011100 \in L(G)$ or not?
  - There are two methods:
    - Parsing Trees
    - Derivations
      - Leftmost Derivations
      - Rightmost Derivations

CFG & parsing

- **Parsing**
  - If a string $w \in L(G)$ for CFG G, then
    - there exists a **parse tree** for w
    - there exists a **leftmost derivation** for w
    - there exists a **rightmost derivation** for w
  - If a string $w \notin L(G)$ for CFG G, then
    - there does not exist a **parse tree** for w
    - there does not exist a **leftmost derivation** for w
    - there does not exist a **rightmost derivation** for w

CFG & parsing

- Exhaustiveness and non-termination
  - CFG G = ({S}, {a, b}, S, P) where P : S → SS | aSb | bSa | ε
  - Question: Is w = aabb ∈ L(G)?

Round 1

1. S ⇒ SS
2. S ⇒ aSb
3. S ⇒ bSa
4. S ⇒ ε

Round 2

rule 1:

S ⇒ SS ⇒ SSS
S ⇒ SS ⇒ aSbS
S ⇒ SS ⇒ bSaS
S ⇒ SS ⇒ S

rule 2:

S ⇒ aSb ⇒ aSSb
S ⇒ aSb ⇒ asSbb
S ⇒ aSb ⇒ abSab
S ⇒ aSb ⇒ ab

Round 3

. . . . . .

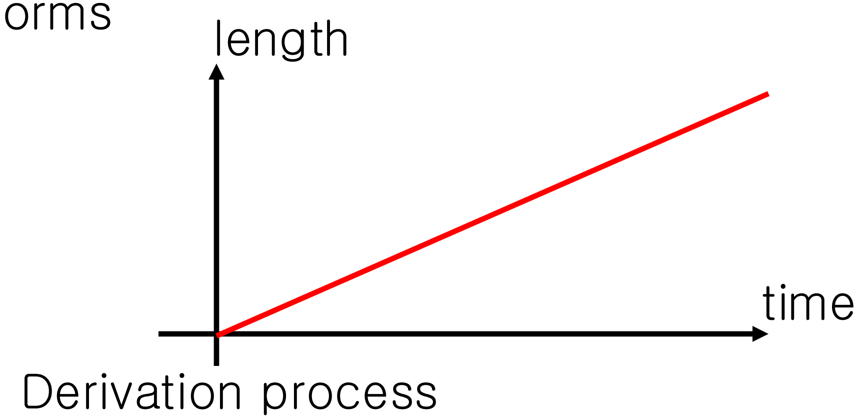
YONSEI UNIVERSITY

CFG & parsing

- Exhaustiveness and non-termination
  - Case 1: $w \in L(G)$: Always terminate
  - Case 2: $w \notin L(G)$:
    - It may not terminate for $w \notin L(G)$
    - Example : Is $w = abb \notin L(G)$?

CFG & parsing

- Exhaustiveness and non-termination – why?
  - ε– production: A → ε
    - It decrease the length of sentential forms
    - Example: S ⇒ SS ⇒ S ⇒ SS ⇒ S ⇒ S . . . .
  - Unit production: A → B
    - It may never increase the length of sentential forms. (S → A, A → B, B → C, . . .)
    - It may create a *cycle*; thus, loop forever!
    - Example: S → A, A → B, B → C, C → S
  - To always terminate parsing, length of sentential forms (during derivation) must be increased
    ➔ we call it "**monotonic**"



CFG & parsing

- Exhaustiveness and non-termination – why?
  - Suppose CFG G = (V, T, S, P) does not have any of the follow two forms: A → ε or A → B
  - Then, exhaustive search parsing always terminate successfully for any $w \in T^*$
  - It always tells us whether $w \in L(G)$ or $w \notin L(G)$
  - Length of each sentential form during derivation increases
  - Since length of sentential form cannot exceed $|w|$, derivation cannot involve more than $2|w|$ rounds

YONSEI UNIVERSITY

CFG & parsing

- Example

CFG G = ({S}, {a, b}, S, P) where
 P: S → SS | aSb | bSa | ε

CFG G' = ({S', S}, {a, b}, S, P') where
 P': S' → S | ε
 S → SS | aSb | bSa | ab | ba

- Note that L(G) = L(G')
- G' guarantees that parsing algorithm will always terminate successfully
- After |w| rounds, it tells us whether w ∈ L(G') or w ∉ L(G')

Simplify CFG

- Simple grammar
  - A CFG G = (V, T, S, P) is called a **s (simple) – grammar** if
    - Every production is of the form: $A \rightarrow ax$ where $A \in V$, $a \in T$, $x \in V^*$
    - Any pair (A, a) occurs at most once in P
    - Example 1 : G is s-grammar ➔ G : $S \rightarrow aS \mid bSS \mid c$
    - Example 2 : G' is not s-grammar ➔ G' : $S \rightarrow aS \mid bSS \mid aSS \mid c$
  - Properties of Simple grammar
    - s-grammar is very restrictive, but many syntax of programming languages can be described by s-grammars
    - If G is s-grammar, then any string w in L(G) can be parsed by linear time O(|w|)
    - Every regular grammar is s-grammar, but not every s-grammar is regular expression
    - Every s-grammar (and regular grammar also) is deterministic and unambiguous

Simplify CFG

- **Simplification of CFG's**
  - Why simplify CFG's?
    - CFG has no restriction on the right-hand side of production ➔ We need to "clean up" it
    - We must guarantee parsing always **terminate**
    - We also want parsing can be done **efficiently**
  - How to simplify CFG's?
    - Remove useless symbols
    - Remove **ε**-productions
    - Remove unit productions

YONSEI UNIVERSITY

Simplify CFG

- Substitution rules
  - Let $G = (V, T, S, P)$ be a CFG where $A, B \in V$, and suppose P has
    - $A \rightarrow \alpha_1 B \alpha_2$
    - $B \rightarrow \beta_1 \mid \beta_2 \mid . . . \mid \beta_n$
  - We can construct CFG $G' = (V, T, S, P')$ which satisfies $L(G) = L(G')$ by performing the following two in G
    - Deleting $A \rightarrow \alpha_1 B \alpha_2$
    - Adding $A \rightarrow \alpha_1 \beta_1 \alpha_2 \mid \alpha_1 \beta_2 \alpha_2 \mid . . . \mid \alpha_1 \beta_n \alpha_2$

YONSEI UNIVERSITY

Simplify CFG

- Example – substitution rules
  - Let G = ({A, B}, {a, b, c}, A, P) be a CFG
    P : A → a | aaA | abBc
    B → abbA | b
  - Then, we can construct CFG G' = (V, T, S, P')
    P' : A → a | aaA | ababbAc | abbc
    B → abbA | b
  - We substitute B as follows:
    - Add A → ababbAc | abbc and delete A → abBC such that L(G) = L(G')

Simplify CFG

- Useless symbols
  - Let $G = (V, T, S, P)$ be a CFG. A symbol X is called **useful** if there is a derivation $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ where $w \in T^*$
    - A symbol X *derives* a terminal string if $X \Rightarrow^* w$ for some $w \in T^*$
    - A symbol X is *reachable* (from S) if $S \Rightarrow^* \alpha X \beta$ for some $\alpha, \beta \in \{V \cup T\}^*$
  - Symbols that are not useful are called **useless** when they are either of the following two:
    - X does <u>not derive</u> any terminal string w
    - X is <u>unreachable</u> from start symbol S

YONSEI UNIVERSITY

Simplify CFG

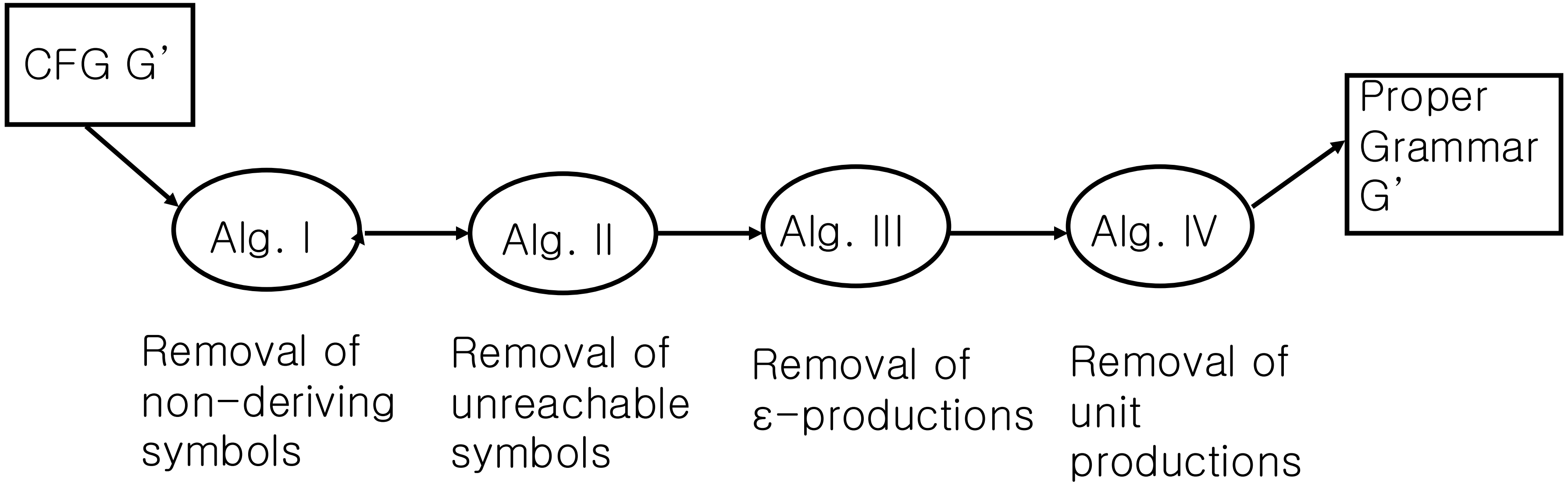
- **Example – useless symbols**
    - CFG G = ({S, A, B, C}, {a, b, c}, S, P)
      P : S → aS | bB | bA | ε
      A → c
      B → bB
      C → b
        - Symbol B is reachable from S, but it does not derive any terminal string
        - Symbol C derives terminal string, but it is not reachable from S
        - Thus, {B, C} are useless symbols. We must remove them
    - After removing {B, C}, simplified G will be as follows:
        - P : S → aS | bA | ε
          A → c

Simplify CFG

- **Proper grammar**
  - For every CFG G, there exists a CFG G' with
    - L(G) = L(G')
    - No useless symbols
    - No ε-productions (= ε-free)
    - No unit productions (= cycle free)
  - We call G' **proper grammar**

Algorithm to simplify CFG

- Algorithm to simplify CFG
  - Algorithm I: removing non-deriving symbols
  - Algorithm II: removing unreachable symbols
  - Algorithm III: removing ε-productions
  - Algorithm IV: removing unit productions



Algorithm to simplify CFG

- Algorithm I: removing non-deriving symbols
  - Input : CFG G = (V, T, S, P)
  - Output : CFG G' = (V, T, S, P) with no non-deriving symbols
  - Method : Construct N_0, N_1, . . . as follows:
    - (1) $N_0 = \emptyset$; i = 1;
    - (2) $N_i = N_{i-1} \cup \{X \mid X \rightarrow \alpha \text{ and } \alpha \in \{N_{i-1} \cup T^*\}\}$
    - (3) If N_i != N_{i-1} then $i \leftarrow i + 1$; go to (2) else $N_d = N_i$

Algorithm to simplify CFG

- Example – algorithm I: removing non-deriving symbols

CFG $G = (\{S, A, B, C, D, E\}, \{a, b, c\}, S, P)$

$P : S \rightarrow AB \mid B \mid aE$

$A \rightarrow AA \mid a$

$B \rightarrow c$

$C \rightarrow \varepsilon$

$D \rightarrow bB \mid aC$

$E \rightarrow bE$

- $N_0 = \emptyset$
- $N_1 = N_0 \cup \{A, B, C\} = \{A, B, C\}$
- $N_2 = N_1 \cup \{D, S\} = \{A, B, C, D, S\}$
- $N_3 = N_2 \cup \emptyset = \{A, B, C, D, S\} = N_d$
- Thus, Since $E \notin N_d$, E is useless

YONSEI UNIVERSITY

Algorithm to simplify CFG

- **Algorithm II: removing unreachable symbols**
  - Input : CFG G = (V, T, S, P)
  - Output : CFG G' = (V', T', S, P') with only reachable symbols
  - Method : Construct N_0, N_1, . . . as follows:
    - (1) $N_0 = \{S\}$; i = 1;
    - (2) $N_i = N_{i-1} \cup \{X \mid A \rightarrow \alpha X \beta \text{ and } A \in N_{i-1}\}$
    - (3) If N_i != N_{i-1} then $i \leftarrow i + 1$; go to (2) else
      $V' = N_i \cap V$;
      $T' = N_i \cap T$;
      P' : productions containing only symbols in V'

Algorithm to simplify CFG

- Example – algorithm II: removing unreachable symbols

CFG G = ({S, A, B, C, D}, {a, b, c}, S, P)

P : S → AB | B
A → AA | a
B → c
C → ε
D → bB | aC

- $N_0 = \{S\}$
- $N_1 = N_0 \cup \{A, B\} = \{S, A, B\}$
- $N_2 = N_1 \cup \{a, c\} = \{S, A, B, a, c\}$
- $N_3 = N_2 \cup \emptyset = \{S, A, B, a, c\}$.

After removing {C, D, b}, we will get:

P' : S → AB | B
A → AA | a
B → c

Algorithm to simplify CFG

- **Background for removing ε-productions**
  - A CFG G = (V, T, S, P) is called **ε-free** if
    - There is no any ε-production
    - There is no any **nullable** variables
    - Suppose ε ∈ L(G);
      - Then, there is only one production S → ε and S does not appear on right had side of any production

Algorithm to simplify CFG

- Example – background for removing ε-productions
  - CFG G_1 = ({S, A}, {0, 1}, S, P) where
    P : S → 0A1
    A → 00A1
    A → ε
  - CFG G_2 = ({S}, {0, 1}, S, P) where
    P : S → ε
    S → 0S1
  - CFG G_3 = ({S, A}, {0, 1}, S, P) where
    P : S → ε | A
    A → 0A1 | 01

Algorithm to simplify CFG

- **Algorithm III: removing ε-productions**
  - Input: CFG G with no useless symbols
  - Output: CFG G' with ε-free
  - Method:
    - (1) Find all nullable variables: $N_\varepsilon = \{A \mid A \Rightarrow \varepsilon\}$
    - (2) If $A \rightarrow \alpha_0 X_1 \alpha_1 X_2 \ldots X_k \alpha_k \in P$ and $X_1, X_2, \ldots, X_k \in N_\varepsilon$, then add $A \rightarrow \alpha_0 Y_1 \alpha_1 Y_2 \ldots Y_k \alpha_k$ where $Y_i = X_i$ or $Y_i = \varepsilon$, for $i = 1, 2, \ldots k$, but remove $A \rightarrow \varepsilon$
    - (3) If $S \in N_\varepsilon$ then add $S' \rightarrow \varepsilon \mid S$, when S' is a new start variable

Algorithm to simplify CFG

- Example – Algorithm III: removing ε-productions

CFG $G = (\{S, B, C, D\}, \{a, b, c, d\}, S, P)$

$P : S \rightarrow aBcD \mid C$

$B \rightarrow b \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

$D \rightarrow BC \mid BBC \mid d$

(1) $N_{\varepsilon} = \{B, C, D, S\}$

(2) $P' : S \rightarrow aBcD \mid acD \mid aBc \mid ac \mid C$

$B \rightarrow b$

$C \rightarrow c$

$D \rightarrow BC \mid B \mid C \mid BBC \mid BB \mid d$

(3) Since $S \in N_{\varepsilon}$, we add

$S' \rightarrow S \mid \varepsilon$ (S' : new start symbol)

Algorithm to simplify CFG

- **Background for removing unit productions**
  - Any production rule of the form A → + B where A, B ∈ V is called a **unit production**
  - A CFG G is called **cycle-free** if there is no derivation of the form A ⇒ A where A ∈ V
  - Example:
    P : S → S | A | ε
    A → A | B | ε
    B → B | C | ε
    C → C | S | ε
    There is a cycle : S ⇒ A ⇒ B ⇒ C ⇒ S
    Therefore, S ⇒+ S

Algorithm to simplify CFG

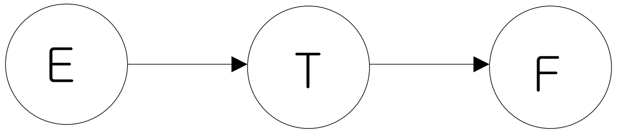
- **Algorithm IV: removing unit productions**
  - Input : CFG G
  - Output : CFG G' with no unit productions
    - (1) For each $A \in N$, construct $N_A = \{B \mid A \Rightarrow B\}$ as follows:
      - a) $N_0 = \{A\}$; $i = 1$;
      - b) $N_i = N_{i-1} \cup \{C \mid B \rightarrow C \text{ and } B \in N_{i-1}\}$
      - c) if $N_i \neq N_{i-1}$ then $i \leftarrow i + 1$; go to b) else $N_A = N_i$;
    - (2) If $B \rightarrow \alpha$ is not a unit production, place $A \rightarrow \alpha$ for all A such that $B \in N$

YONSEI UNIVERSITY

Algorithm to simplify CFG

- Example – Algorithm IV: removing unit productions

CFG G = ({E, T, F}, {*, +, (,), a}, E, P)
P : E → E + T | T
T → T * F | F
F → (E) | a



(1) N_E = {E, T, F},
N_T = {T, F},
N_F = {F}
(2) P’ : E → E + T | T * F | (E) | a
T → T * F | (E) | a
F → (E) | a

Questions?