Context-Free Grammars

นำเสนอโดย

ดร.สุธี สุดประเสริฐ

บทนำ

- เราทราบกันแล้วว่า finite automata regular, expressions และ regular grammar ไม่สามารถจดจำภาษาบางอย่างได้

  เช่น $\{0^n1^n \mid n \geq 0\}$ หรือ $\{a^{n!}\}$
- สำหรับภาษา $L = \{0^n1^n \mid n \geq 0\}$ หากเราเปลี่ยน 0 เป็น (และ 1 เป็น) ดังนั้นสตริงในภาษานี้คือ () (()) หรือ ((())) แต่ว่าสตริง (() จะไม่อยู่ในภาษานี้
- ภาษาดังกล่าวเป็นตัวอย่างของโครงสร้างแบบซ้อน (nested structure) อย่างง่าย
- ดังนั้นภาษา regular จึงมีความสามารถไม่เพียงพอในการอธิบายบางคุณสมบัติของภาษาโปรแกรม

บทนำ

- Context-free grammars (CFG) เป็นแบบจำลองการคำนวณหนึ่งที่มีความสามารถในการอธิบายภาษามากกว่าแบบจำลองอื่นๆ ก่อนหน้านี้
- ไวยากรณ์แบบนี้มีลักษณะเด่นคือมีโครงสร้างแบบ ***recursive structure***
- CFG ถูกเริ่มใช้ครั้งแรกในการอธิบายภาษามนุษย์ (ภาษาธรรมชาติ)

บทนำ

- CFG มีบทบาทสำคัญในการเขียนข้อกำหนดและการคอมไพล์โปรแกรมภาษา (programming languages)
- การออกแบบและสร้างตัวแปลภาษา (compilers and interpreters) สำหรับโปรแกรมภาษาหนึ่ง มักจะเริ่มต้นจากไวยากรณ์ของภาษานั้น

ไวยากรณ์บางส่วนของภาษา C

statement → labeled_statement | compound_statement | expression_statement |
selection_statement | iteration_statement | jump_statement

loop → iteration_statement

iteration_statement → while (expression) statement | do statement while (expression) ; |
for_statement

for_statement → for (expression ; expression ; expression) statement

selection_statement → if_statement | switch (expression) statement

if_statement → if (expression) statement | if (expression) statement else statement

ไวยากรณ์บางส่วนของภาษา Java (**Java Syntax Specification**)

Programs

<compilation unit> ::= *<package declaration>*?*<import declarations>*? *<type declarations>*?

Declarations

<package declaration> ::= **package** *<package name>* **;***<import declarations>* ::= *<import declaration>* |*<import declarations>* *<import declaration>*

<import declaration> ::= *<single type import declaration>* |*<type import on demand declaration>*

<single type import declaration> ::= **import** *<type name>* **;**

<type import on demand declaration> ::= **import** *<package name>* **. * ;**

<type declarations> ::= *<type declaration>* | *<type declarations>**<type declaration>*

<type declaration> ::= *<class declaration>* | *<interface declaration>* |**;**

<class declaration> ::= *<class modifiers>*?**class***<identifier>* *<super>*? *<interfaces>*? *<class body>*

<class modifiers> ::= *<class modifier>* | *<class modifiers>* *<class modifier>*

<class modifier> ::= **public** | **abstract** | **final**

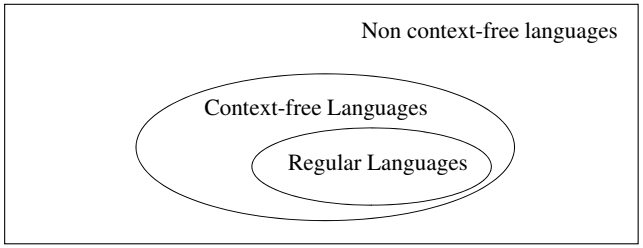
<super> ::=**extends***<class type>*

<interfaces> ::= **implements***<interface type list>*

<interface type list> ::= *<interface type>* | *<interface type list>* **,** *<interface type>*

บทนำ

- สำหรับกลุ่มภาษาที่เกี่ยวข้องกับ CFG จะถูกเรียกว่าภาษาไม่อิงบริบท (context-free languages: CFL)
  - ซึ่งภาษา regular ก็จัดอยู่ในกลุ่มนี้เช่นกัน



ตัวอย่างของ CFG

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

- ไวยากรณ์ประกอบด้วยกฎของแทนที่ (substitution rules) หรืออาจเรียกว่ากฎการสร้าง (production rules)
- ส่วนกฎจะประกอบด้วย ตัวแปร (variables) และ สัญลักษณ์สิ้นสุด (terminal symbols)
- โดยที่จะมีตัวแปรหนึ่งตัวที่ถูกกำหนดให้เป็นตัวแปรเริ่มต้น (start variable) (มักจะตัวแปรที่จะด้านซ้ายมือของกฎที่อยู่บนสุด)

การคำนวณ CFGs

- ในการใช้ CFG สำหรับอธิบายภาษาใดภาษาหนึ่งโดยการสร้างสตริงแต่ละตัวที่อยู่ในภาษานั้น จะมีกระบวนการดังนี้
  - เริ่มต้นจากเขียนตัวแปรเริ่มต้นก่อน
  - จากนั้นหากฎที่มีตัวแปรที่ถูกเขียนไปแล้ว อยู่ทางด้านซ้ายมือของกฎ และจึงแทนตัวแปรนั้นด้วยส่วนขวาของกฎ
  - วนซ้ำในขั้นตอนที่ 2 จึงกว่าจะไม่มีตัวแปรเหลืออยู่

ตัวอย่างการคำนวณ

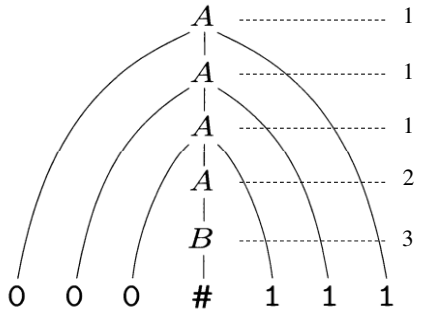
$$A \rightarrow 0A1 \quad (1)$$
$$A \rightarrow B \quad (2)$$
$$B \rightarrow \# \quad (3)$$

- ตัวอย่างการสร้างสตริง 000#111 สำหรับไวยากรณ์นี้

$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$

(1), (1), (1), (2), (3)

ตัวอย่างการคำนวณ



เราสามารถใช้โครงสร้างต้นไม้ในสำหรับแสดงการสร้างสตริงสำหรับไวยากรณ์

CFGs

- สตริงทั้งหมดที่สามารถสร้างจากไวยากรณ์หนึ่ง เราจะเรียกว่าภาษาของไวยากรณ์นั้น (language of the grammar)
- เราเขียน $L(G)$ สำหรับภาษาของไวยากรณ์ G
- จากตัวอย่างที่แล้ว แสดงว่า $L(G)$ คือ $\{0^n\#1^n \mid n \geq 0\}$
- สำหรับภาษาใดๆ ที่สามารถถูกสร้างโดย CFG จะถูกเรียกว่าภาษาไม่อิงบริบท (context-free language: CFL)

CFGs

- เราสามารถเขียนย่อกฎที่มีตัวแปรฝั่งซ้ายเหมือนกัน ไว้เป็นกฎเดียวกันได้ ตัวอย่างเช่น

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$
$$=$$
$$A \rightarrow 0A1 \mid B$$
$$B \rightarrow \#$$

นิยามของ CFG

A context-free grammar is a 4-tuple (V, Σ, S, P), where

1. V is a finite set called the variables,
2. Σ is a finite set, disjoint from V, called the terminals,
3. $S \in V$ is the start variable, and
4. P is a finite set of rules, with each rule being a variable and a string of variables and terminals $(V \rightarrow (\Sigma \cup V)^*)$.

นิยามของ CFG

- ถ้า u, v, และ w เป็นสายลำดับของตัวแปรและตัวสิ้นสุด โดยที่ $A \rightarrow w$ เป็นกฎไวยากรณ์หนึ่ง
  - เราจะพูดว่า uAv ***ผลิต*** uwv โดยเขียนแทนว่า $uAv \Rightarrow uwv$
- ถ้า $u = v$ หรือ ถ้าลำดับ $u_1, u_2, \ldots, u_k$ มีอยู่สำหรับ $k \geq 0$ และ

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

  - เราจะพูดว่า u ***กลายมาเป็น*** v โดยเขียนแทนว่า $u \overset{*}{\Rightarrow} v$
- ดังนั้นภาษาของไวยากรณ์คือ

$$\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$$

ตัวอย่าง

$$A \rightarrow 0A1 \quad (1)$$
$$A \rightarrow B \quad (2)$$
$$B \rightarrow \# \quad (3)$$

- สำหรับไวยากรณ์นี้ $V = \{A,B\}$, $\Sigma = \{0,1,\#\}$, $S = A$, และ P คือกฎไวยากรณ์ทั้งสาม

Derivations

สำหรับ CFG ที่ไม่ได้เป็น linear (ด้านขวาของกฎมีตัวแปรมากกว่าหนึ่งตัว) ในกรณีนี้เราต้องเลือกว่าตัวแปรไหนจะถูกแทนที่ก่อน

ตัวอย่างเช่น $G = (\{A,B,S\}, \{a,b\}, S, P)$ โดยที่ P คือ

1. $S \rightarrow AB$ 2. $A \rightarrow aaA$ 3. $A \rightarrow \varepsilon$
4. $B \rightarrow Bb$ 5. $B \rightarrow \varepsilon$

ซึ่ง $L(G) = \{a^{2n}b^m \mid n \geq 0, m \geq 0\}$ พิจารณาการแทนค่าตัวแปรต่อไปนี้

$\mathbf{S} \overset{1}{\Rightarrow} \mathbf{A}B \overset{2}{\Rightarrow} aa\mathbf{A}A \overset{3}{\Rightarrow} aa\mathbf{B} \overset{4}{\Rightarrow} aa\mathbf{B}b \overset{5}{\Rightarrow} aab$

$\mathbf{S} \overset{1}{\Rightarrow} A\mathbf{B} \overset{4}{\Rightarrow} \mathbf{A}Bb \overset{2}{\Rightarrow} aa\mathbf{A}\mathbf{B}b \overset{5}{\Rightarrow} aa\mathbf{A}b \overset{3}{\Rightarrow} aab$

Derivations

- จากตัวอย่างที่แล้ว แสดงให้เห็นว่าสตริงเดียวกัน สามารถสร้างได้จากการใช้กฎชุดเดียวกัน แต่แตกต่างกันที่ลำดับการใช้กฎ
- ตามปกติ เพื่อง่ายต่อการพิจารณา เราจะกำหนดลำดับการแทนที่กฎแบบตายตัว ซึ่งมี 2 แบบคือ
  - Leftmost derivation: แทนตัวแปรซ้ายสุดก่อนเสมอ
  - Rightmost derivation: แทนตัวแปรขวาสุดก่อนเสมอ

ตัวอย่าง

จากกฎต่อไปนี้

$S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A \mid \varepsilon$

Leftmost derivation:

$\mathbf{S} \Rightarrow a\mathbf{A}B \Rightarrow ab\mathbf{B}bB \Rightarrow ab\mathbf{A}bB \Rightarrow abb\mathbf{B}bbB \Rightarrow abbbb\mathbf{B} \Rightarrow abbbb$

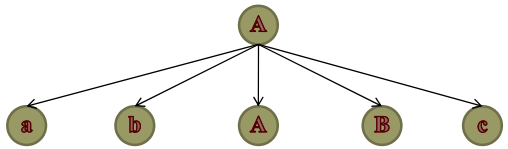
Rightmost derivation:

$\mathbf{S} \Rightarrow a\mathbf{A}\mathbf{B} \Rightarrow a\mathbf{A} \Rightarrow ab\mathbf{B}b \Rightarrow ab\mathbf{A}b \Rightarrow abb\mathbf{B}bb \Rightarrow abbbb$

Derivation Trees

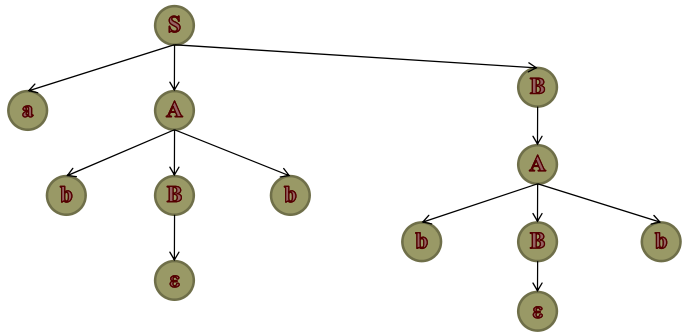
- เราสามารถแสดง derivation ได้โดยการใช้ derivation tree (parse tree) วิธีการนี้จะไม่สนใจในลำดับของการใช้กฎ

$A \rightarrow abABc$



ตัวอย่าง

$S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A \mid \varepsilon$



$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abbB \Rightarrow abbA \Rightarrow abbbBb \Rightarrow abbbb$

การออกแบบ Context-free Grammars

- เหมือนกับ FA คือ การออกแบบ CFG ต้องใช้ความคิดสร้างสรรค์
- ซึ่งแน่นอนว่า CFG ออกแบบยากกว่า FA
- เทคนิกต่อไปนี้อาจจะมีประโยชน์สำหรับใช้ในการออกแบบ CFG

การออกแบบ Context-free Grammars

1. ถ้า CFL เกิดจะการยูเนี่ยนกันระหว่าง CFL หลายตัว การแก้ปัญหาที่ละตัว “มักจะ” ง่ายกว่าการแก้ที่เดียวพร้อมกัน

$$\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$$

$$S_1 = 0S_11 \mid \varepsilon$$
$$S_2 = 1S_20 \mid \varepsilon$$

$$S = S_1 \mid S_2$$

การออกแบบ Context-free Grammars

2. ถ้า CFL ประกอบด้วยสตริงที่มีสตริงย่อยสองส่วนซึ่ง “เชื่อมโยงกัน” เราสามารถสร้าง CFG โดยใช้กฎที่อยู่ในรูป $R \rightarrow uRv$

เช่นในภาษา $\{0^n1^n \mid n \geq 0\}$ สตริงย่อยส่วนที่เป็น 0 กับ ส่วนที่เป็น 1 “เชื่อมโยงกัน” ในแง่ที่ว่าทั้งสองส่วนต้องมีความยาวเท่ากัน

การออกแบบ Context-free Grammars

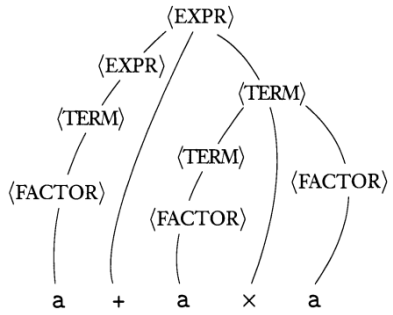
3. ในภาษาที่มีความซับซ้อน สตริงอาจจะประกอบด้วยโครงสร้างหนึ่งที่ปรากฏซ้ำเป็นส่วนหนึ่งของโครงสร้างอื่น หรือ เป็นส่วนหนึ่งของโครงสร้างเดิม

ในกรณีนี้ ให้กำหนดตัวแปรที่แทนโครงสร้างที่ปรากฏซ้ำนั้นก่อน แล้วเขียนตัวแปรนั้นใส่ลงไปในกฎที่สัมพันธ์กับการเกิดซ้ำของโครงสร้างนี้

การออกแบบ Context-free Grammars

a grammar for generating arithmetic expressions

<EXPR> → <EXPR> + <TERM> | <TERM>
<TERM> → <TERM> x <FACTOR> | <FACTOR>
<FACTOR> → (<EXPR>) | a



แบบฝึกหัด

- จงหา CFG สำหรับภาษาต่อไปนี้ ($n \geq 0, m \geq 0$)
  - $L = \{a^n b^m : n \leq m + 3\}$
  - $L = \{a^n b^m : n \neq m - 1\}$

Parsing and Membership

- จากที่แล้วมาเราศึกษาไวยากรณ์ในแง่ของการสร้าง (generative aspect) คือสนใจว่าไวยากรณ์หนึ่งๆ สามารถสร้างสตริงอะไรได้บ้าง
- ในบางกรณี เราต้องการที่จะรู้ว่าสตริง w เป็นสมาชิกของ $L(G)$ หรือไม่ (analytical aspect)
- บางครั้งเราอาจจะต้องรู้ด้วยว่า ถ้า w เป็นสมาชิกของ $L(G)$ แล้วลำดับของการใช้กฎในการสร้าง w คืออะไร ซึ่งวิธีการดังกล่าวนี้จะถูกเรียกว่า “parsing”

Exhaustive Search Parsing

วิธีเบื้องต้นที่สามารถทำได้ สำหรับการหา derivation (leftmost) ของสตริง w ที่เป็นสมาชิกของ $L(G)$ ทำได้โดย

1. หากฎทั้งหมดที่อยู่ในรูปแบบ $S \rightarrow x$ โดยที่ x เป็นลำดับของตัวแปรหรือตัวสิ้นสุด ถ้าไม่ x ที่ตรงกับ w ให้ทำขั้นตอนต่อไป
2. แทนค่าตัวแปรซ้ายสุดใน x ทุกตัวด้วยกฎ ซึ่งผลลัพธ์ที่ได้ในขั้นตอนนี้คือ กลุ่มของ sentential forms ซึ่งอาจจะนำไปสู่ w
3. ถ้ายังไม่ได้ w ให้วนซ้ำในขั้นตอนที่ 2 ไปเรื่อยๆ

ตัวอย่าง

จากไวยากรณ์ S → SS | aSb | bSa | ε จงหา derivation ของ aabb

<u>รอบแรก</u>

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

ตัวอย่าง

S → SS | aSb | bSa | ε

<u>รอบที่สอง</u> (S ⇒ <u>S</u>S)

1. S ⇒ <u>S</u>S ⇒ SS
2. S ⇒ <u>S</u>S ⇒ aSbS
3. S ⇒ <u>S</u>S ⇒ bSaS
4. S ⇒ <u>S</u>S ⇒ S

ตัวอย่าง

S → SS | aSb | bSa | ε

<u>รอบที่สอง</u> (S ⇒ a<u>S</u>b)

1. S ⇒ a<u>S</u>b ⇒ aSSb
2. S ⇒ a<u>S</u>b ⇒ aaSbb
3. S ⇒ a<u>S</u>b ⇒ abSab
4. S ⇒ a<u>S</u>b ⇒ ab

ตัวอย่าง

ตัด sentential form ที่ไม่สามารถสร้าง aabb ได้ออก

<u>รอบที่สอง</u>

1. $S \Rightarrow \underline{S}S \Rightarrow SS$
2. $S \Rightarrow \underline{S}S \Rightarrow aSbS$
3. $S \Rightarrow \underline{S}S \Rightarrow S$
4. $S \Rightarrow a\underline{S}b \Rightarrow aSSb$
5. $S \Rightarrow a\underline{S}b \Rightarrow aaSbb$

ตัวอย่าง

จาก sentential form ที่ 5 เราสามารถสร้าง aabb ได้

$$S \Rightarrow a\underline{S}b \Rightarrow aaSbb \Rightarrow \varepsilon$$

แต่ว่า exhaustive search parsing มีข้อเสียคือใช้เวลาในการประมวลผลนาน และ เป็นไปได้ที่การค้นหาจะไม่อยู่ทำงาน (nontermination) ในกรณีที่สตริงไม่ได้อยู่ในภาษาของไวยากรณ์นั้น

ปัญหาการไม่หยุดทำงาน (nontermination)

- ถ้าพิจารณาจากตัวอย่าง เราจะเห็นว่าปัญหานี้เกิดจากที่กฎบางอัน ทำให้ sentential form มีขนาดลดลง นั้นคือกฎ

$$S \rightarrow \varepsilon$$

- จริงๆ แล้วกฎที่ทำให้ปัญหามีอยู่สองแบบคือ (โดยที่ A และ B คือหนึ่งตัวแปร)
  1. $A \rightarrow \varepsilon$
  2. $A \rightarrow B$

Exhaustive Search Parsing

- ถ้าในกฎไวยากรณ์ไม่มีกฎทั้งสองแบบนี้
  1. $A \rightarrow \varepsilon$
  2. $A \rightarrow B$
- exhaustive search parsing จะสามารถหาได้ว่าสตริงใดอยู่ในภาษาของไวยากรณ์หรือไม่ และสามารถหา derivation ของสตริงที่อยู่ในภาษาของไวยากรณ์ได้โดยทำงานไม่เกิน $2 \times |w|$ รอบของการทำงาน
  - ในกรณีเลวร้ายสุดคือในรอบที่ w มี sentential forms ที่เป็นตัวแปรทั้งหมด ดังนั้นต้องทำอีก w รอบ ในการแทนค่าตัวแปรทั้งหมด

Exhaustive Search Parsing

การวิเคราะห์ขอบบนของการคำนวณ (Big-O)

- ถ้า $|P|$ คือจำนวนของกฎไวยากรณ์ ในรอบแรกจะเกิด sentential form ไม่เกิน $|P|$ สำหรับรอบที่สองจะไม่เกิน $|P|^2$
- เราทราบแล้วว่าการคำนวณจะทำไม่เกิน $2|w|$ รอบ ดังนั้นจำนวน sentential form ทั้งหมดจะไม่เกิน

$$|P| + |P|^2 + |P|^3 + \ldots + |P|^{2|w|}$$

- ดังนั้นขอบบนของการคำนวณ คือ $O(|P|^{2|w|+1})$

ความกำกวมของไวยากรณ์ (Ambiguity)

- ในบางครั้งไวยากรณ์หนึ่งสามารถสร้างสตริงหนึ่ง ได้หลายวิธี
  - หนึ่งสตริงสามารถเกิดได้จากหลาย derivation trees
  - หรือเรียกว่า หนึ่งสตริงมีได้หลายความหมาย
- ซึ่งลักษณะเช่นนี้ อาจจะไม่เป็นที่ต้องการสำหรับงานบางอย่าง เช่น โปรแกรมภาษา (programming languages)

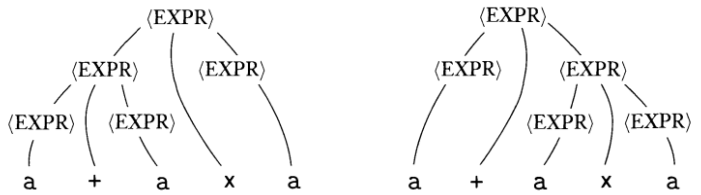
ความกำกวมของไวยากรณ์ (Ambiguity)

- ถ้าไวยากรณ์หนึ่งสามารถสร้างสตริงหนึ่งได้หลายวิธี
  - เราเรียกว่า สตริงนั้นถูกสร้างอย่างกำกวมในไวยากรณ์นี้
  - มีหลาย derivation (parse) trees ที่แตกต่างกัน ไม่ใช่ หลาย derivations ที่แตกต่างกัน
- ถ้าไวยากรณ์หนึ่งสามารถสร้างสตริงอย่างกำกวมได้
  - เราเรียกว่า ไวยากรณ์นั้นกำกวม (ambiguous)

ตัวอย่าง

พิจารณาไวยากรณ์ G ดังนี้

<EXPR> → <EXPR> + <EXPR>
<EXPR> → <EXPR> × <EXPR>
<EXPR> → (<EXPR>) | a



ความกำกวมของไวยากรณ์ (Ambiguity)

เรานิยามความกำกวมได้ดังนี้

- A context-free grammar G is said to be ambiguous if there exists some $w \in L(G)$ that has at least two distinct derivation trees.

แบบฝึกหัด

- จงแสดงว่าไวยากรณ์ต่อไปนี้กำกวม และ จงหาสร้างไวยากรณ์ที่ไม่กำกวมสำหรับภาษาเดียวกัน

  $S \rightarrow AB \mid aaB$

  $A \rightarrow a \mid Aa$

  $B \rightarrow b$

Simplification of Context-Free Grammars and Normal Forms

นำเสนอโดย

ดร.สุธี สุดประเสริฐ

Methods for Transforming Grammars

- เราได้ทราบแล้วว่า สตริงว่าง ทำให้การปัญหาในการคำนวณเพื่อหาว่าสตริงหนึ่งเป็นสมาชิกในภาษาของไวยากรณ์หนึ่งหรือเปล่า

- ในทางทฤษฎี สตริงว่าง มีความสำคัญในการพิสูจน์ทฤษฎีบทต่างๆ แต่ในทางปฏิบัติ เราไม่ต้องการที่จะนำสตริงว่างมาพิจารณา

Methods for Transforming Grammars

- จริงๆ แล้ว การลบสตริงว่างออกจากสมาชิกของภาษาใดภาษาหนึ่ง นั้น ไม่ทำให้สูญเสียคุณสมบัติทั่วไปของภาษานั้นไป เพราะว่า ถ้ามีไวยากรณ์ $G = (V, T, S, P)$ ซึ่ง $L - \{\varepsilon\}$
- เราสามารถไวยากรณ์ใหม่ที่มี สตริงว่างเป็นสมาชิกได้โดยเปลี่ยนให้ S_0 เป็นตัวแปรเริ่มต้นแทน S และเพิ่มกฎ $S_0 \rightarrow S \mid \varepsilon$ เข้าไป
- เพื่อง่ายต้องการทำความเข้าใจ (ถ้าไม่ได้ชี้เฉพาะเจาะจงลงไป) ภาษาที่เราจะพิจารณาต่อไปในบทนี้ จะเป็นภาษาที่ไม่สตริงว่างเป็นสมาชิก (ε-free languages)

การลบกฎที่ไร้ประโยชน์

- กฎที่ไร้ประโยชน์คือกฎที่ไม่มีทางถูกใช้ในสร้างสตริงใดๆ ในภาษาของไวยากรณ์นั้น
- ตัวอย่างเช่น

  $S \rightarrow aSb \mid \varepsilon \mid A$

  $A \rightarrow aA$

กฎ $S \rightarrow A$ และ $A \rightarrow aA$ เป็นกฎที่ไร้ประโยชน์ เพราะถ้า A ปรากฏใน sentential form แล้ว จะไม่มีทางที่จะสร้างสตริงอะไรได้เลย (A ไม่สามารถเปลี่ยนเป็นตัวสิ้นสุดได้)

การลบกฎที่ไร้ประโยชน์

นิยาม

ถ้า $G = (V, T, S, P)$ เป็น CFG และตัวแปร $A \in V$ จะมี **ประโยชน์**ก็ต่อเมื่อ มีอย่างหนึ่งสตริง $w \in L(G)$ ที่

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w$$

เมื่อ x, y in $(V \cup T)^*$

และถ้าตัวแปรใดไม่เป็นตัวแปรที่มีประโยชน์ ตัวแปรนั้น คือ **ไร้ประโยชน์**

การลบกฎที่ไร้ประโยชน์

- กฎที่ไร้ประโยชน์ เกิดจากตัวแปรที่ไร้ประโยชน์ ซึ่งมีสาเหตุเกิดได้ 2 กรณี คือ

1. ตัวแปรนั้นไม่สามารถแปลงเป็นตัวสิ้นสุดได้
2. ตัวแปรนั้นไม่สามารถเข้าถึงได้จากตัวแปรเริ่มต้น

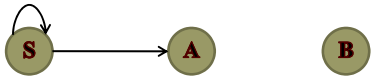
การลบกฎที่ไร้ประโยชน์

- สำหรับกรณีที่ 2 คือตัวแปรที่ไม่สามารถเข้าถึงได้จากตัวแปรเริ่มต้น เราสามารถเขียน dependency graph เพื่อช่วยในการวิเคราะห์ได้ เช่น

$S \rightarrow aS \mid A$

$A \rightarrow a$

$B \rightarrow aa$



การลบกฎที่สร้างสตริงว่าง

ถ้าไวยากรณ์หนึ่งสร้างภาษาที่ไม่มีสตริงว่างเป็นสมาชิก แต่ยังมีกฎที่สร้างสตริงว่างอยู่ เราสามารถลบกฎเหล่านั้นทิ้งได้ โดยวิธีการต่อไปนี้

1. ลบกฎที่อยู่ในรูปแบบ $A \rightarrow \varepsilon$ ทิ้งทั้งหมด
2. หากฎ $A \rightarrow \varepsilon$ ทั้งหมด แล้วเก็บ A ไว้ใน V_N
3. ทำซ้ำขั้นตอนต่อไปนี้ จนกว่าจะไม่มีตัวแปรที่ถูกเพิ่มไปใน V_N
   1. สำหรับกฎ $B \rightarrow A_1A_2 \ldots A_n$ โดย $A_1A_2 \ldots A_n$ อยู่ใน V_N ให้เพิ่ม B เข้าไปใน V_N
4. หากฎ $A \rightarrow x_1x_2 \ldots x_m$, $m \geq 1$ และ $x_i \in V \cup T$ จากนั้นให้แทนตัวแปรที่อยู่ใน V_N ด้วย ε ในทุกๆ แบบ

การลบกฎที่สร้างสตริงว่าง

$S \rightarrow ABaC$

$A \rightarrow BC$

$B \rightarrow b \mid \varepsilon$

$C \rightarrow D \mid \varepsilon$

$D \rightarrow d$

การลบกฎที่สร้างสตริงว่าง

$S \rightarrow ABaC$

$A \rightarrow BC$

$B \rightarrow b \mid \varepsilon$

$C \rightarrow D \mid \varepsilon$

$D \rightarrow d$

$V_N = \{B, C, A\}$

การลบกฎที่สร้างสตริงว่าง

$S \rightarrow ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a$

$A \rightarrow BC \mid B \mid C$

$B \rightarrow b$

$C \rightarrow D$

$D \rightarrow d$

$V_N = \{B, C, A\}$

การลบกฎหนึ่งหน่วย (unit-production)

นิยาม

กฎของไวยากรณ์ CFG ที่อยู่ในรูปแบบ

$$A \rightarrow B$$

โดยที่ $A, B \in V$ จะถูกเรียกว่า กฎหนึ่งหน่วย

การลบกฎหนึ่งหน่วย (unit-production)

ถ้าไวยากรณ์หนึ่งสร้างภาษาที่ไม่มีสตริงว่างเป็นสมาชิก แต่ยังมีกฎหนึ่งหน่วยอยู่ เราสามารถลบกฎเหล่านั้นทิ้งได้ โดยวิธีการต่อไปนี้

1. ลบกฎหนึ่งหน่วยทิ้งทั้งหมด
2. สร้าง dependency graph สำหรับกฎหนึ่งหน่วย เพื่อช่วยในการหา $A \overset{*}{\Rightarrow} B$
3. สำหรับทุกๆ $A \overset{*}{\Rightarrow} B$ ให้เพิ่มกฎ $A \rightarrow y_1 | y_2 | \ldots y_n$ โดยที่ $B \rightarrow y_1 | y_2 | \ldots y_n$ (B ไม่รวมส่วนที่เป็นกฎหนึ่งหน่วย)

การลบกฎหนึ่งหน่วย (unit-production)

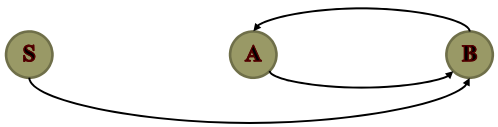
พิจารณากฎการผลิตต่อไปนี้

$S \rightarrow Aa \mid B$

$A \rightarrow a \mid bc \mid B$

$B \rightarrow A \mid bb$

ต้องเพิ่มบทสรุป

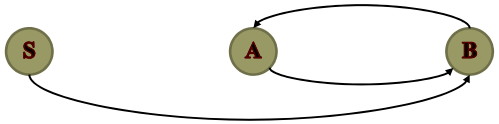


การลบกฎหนึ่งหน่วย (unit-production)

$S \rightarrow Aa$

$A \rightarrow a \mid bc$

$B \rightarrow bb$

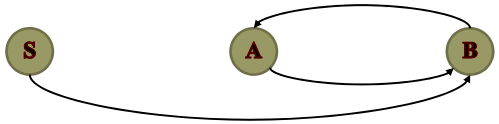


การลบกฎหนึ่งหน่วย (unit-production)

$S \rightarrow Aa \mid a \mid bc \mid bb$

$A \rightarrow a \mid bc$

$B \rightarrow bb$



$S \overset{*}{\Rightarrow} A$

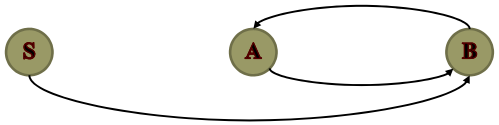
$S \overset{*}{\Rightarrow} B$

การลบกฎหนึ่งหน่วย (unit-production)

$S \rightarrow Aa \mid a \mid bc \mid bb$

$A \rightarrow a \mid bc \mid bb$

$B \rightarrow bb$



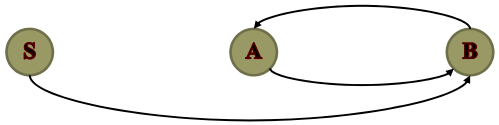
$A \overset{*}{\Rightarrow} B$

การลบกฎหนึ่งหน่วย (unit-production)

$S \rightarrow Aa \mid a \mid bc \mid bb$

$A \rightarrow a \mid bc \mid bb$

$B \rightarrow bb \mid a \mid bc$



$B \overset{*}{\Rightarrow} A$

Two Important Normal Forms

- รูปแบบปกติของ CFG มีอยู่หลายรูปแบบ เนื่องจากมีการนำ CFG ไปประยุกต์และวิจัยกันอย่างกว้างขวาง เช่น
  - Backus-Naur form (BNF)
  - Chomsky normal form (CNF)
  - Kuroda normal form (KNF)
  - Greibach normal form (GNF)
- ในที่นี้เราจะศึกษา 2 ตัว คือ Chomsky normal form และ Greibach normal form

Chomsky Normal Form

- ในการนำ CFG ไปใช้งาน ในบางกรณีหากไวยากรณ์นั้นอยู่ในรูปแบบพิเศษ เราจะสามารถใช้งานไวยากรณ์ในสะดวกขึ้น
- Chomsky normal form เป็นรูปแบบพิเศษหนึ่งที่น่าสนใจและมีประโยชน์ในการนำไปใช้งาน เช่น การหา parse trees โดยใช้ CYK algorithm

Chomsky Normal Form

CFG จะอยู่ในรูปแบบ Chomsky normal form ถ้าทุกๆ กฎไวยากรณ์อยู่ในรูปแบบต่อไปนี้

$$A \rightarrow BC$$
$$A \rightarrow a$$
$$S \rightarrow \varepsilon$$

โดยที่ a คือตัวสิ้นสุดใดๆ และ A, B, และ C คือ ตัวแปรใดๆ ยกเว้นแต่ B และ C ห้ามเป็นตัวแปรเริ่มต้น และ S คือ ตัวแปรเริ่มต้น

Chomsky Normal Form

- วิธีการแปลง CFG ใดๆ ให้อยู่ในรูปแบบ Chomsky normal form สามารถทำได้ดังนี้
  - สร้างตัวแปรเริ่มต้นใหม่ ($S_0 \rightarrow S$)
  - ลบกฎที่สร้างสตริงว่าง ($A \rightarrow \varepsilon$)
  - ลบกฎหนึ่งหน่วย ($A \rightarrow B$)
  - แปลงกฎที่เหลือให้อยู่ในรูปแบบที่เหมาะสม

ตัวอย่าง

$S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S$

$B \rightarrow b \mid \varepsilon$

ตัวอย่าง

สร้างตัวแปรเริ่มต้นใหม่

$\mathbf{S_0 \rightarrow S}$

$S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S$

$B \rightarrow b \mid \varepsilon$

ตัวอย่าง

ลบกฎที่สร้างสตริงว่าง

$\mathbf{S_0 \rightarrow S}$	$S_0 \rightarrow S$
$S \rightarrow ASA \mid aB$	$S \rightarrow ASA \mid aB \mid SA \mid AS \mid S \mid a$
$A \rightarrow B \mid S$	$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$	$B \rightarrow b$

$V_N = \{A, B\}$

ตัวอย่าง

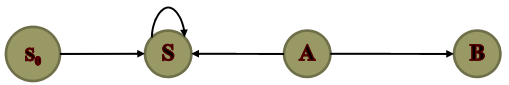
ลบกฎหนึ่งหน่วย

$S_0 \rightarrow S$

$S \rightarrow ASA \mid aB \mid SA \mid AS \mid S \mid a$

$A \rightarrow B \mid S$

$B \rightarrow b$



ตัวอย่าง

ลบกฎหนึ่งหน่วย

$S_0 \rightarrow$

$S \rightarrow ASA \mid aB \mid SA \mid AS \mid a$

$A \rightarrow$

$B \rightarrow b$

ตัวอย่าง

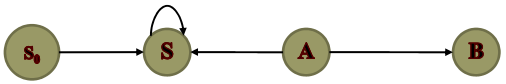
ลบกฎหนึ่งหน่วย

$S_0 \rightarrow ASA \mid aB \mid SA \mid AS \mid a$

$S \rightarrow ASA \mid aB \mid SA \mid AS \mid a$

$A \rightarrow ASA \mid aB \mid SA \mid AS \mid a \mid b$

$B \rightarrow b$



ตัวอย่าง

แปลงกฎที่เหลือให้อยู่ในรูปแบบที่เหมาะสม

$S_0 \rightarrow A\underline{SA} \mid \underline{aB} \mid a \mid SA \mid AS$

$S \rightarrow A\underline{SA} \mid \underline{aB} \mid a \mid SA \mid AS$

$A \rightarrow A\underline{SA} \mid \underline{aB} \mid a \mid SA \mid AS \mid b$

$B \rightarrow b$

ตัวอย่าง

แปลงกฎที่เหลือให้อยู่ในรูปแบบที่เหมาะสม

$S_0 \rightarrow A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS$

$S \rightarrow A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS$

$A \rightarrow A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS \mid b$

$B \rightarrow b$

$\mathbf{A1} \rightarrow \mathbf{SA}$

$\mathbf{U} \rightarrow \mathbf{a}$

Greibach normal form

- การแปลง CFG ใดๆ ให้อยู่ใน GNF ไม่ใช่เรื่องยาก ตรงไปตรงมา
- GNF มีบทบาทสำคัญในการพิสูจน์ทฤษฎีบทบางบท

Greibach normal form

CFG จะอยู่ในรูปแบบ Greibach normal form ถ้าทุกๆ กฎไวยากรณ์อยู่ในรูปแบบต่อไปนี้

$$A \rightarrow a x$$

โดยที่ $a \in T$ และ $x \in V^*$

ตัวอย่าง

จงแปลงไวยากรณ์ต่อไปนี้ให้อยู่ในรูปแบบ GNF

$S \rightarrow AB$ (1)

$A \rightarrow aA \mid bB \mid b$ (2)

$B \rightarrow b$ (3)

เราสามารถแปลงได้โดยการแทน (2) ใน (1)

ตัวอย่าง

ผลลัพธ์จะได้

$S \rightarrow \underline{aA}B \mid \underline{bB}B \mid \underline{b}B$

$\underline{A} \rightarrow aA \mid bB \mid b$

$B \rightarrow b$

ตัวอย่าง

จงแปลงไวยากรณ์ต่อไปนี้ให้อยู่ในรูปแบบ GNF

$S \rightarrow abSb \mid aa$

ในกรณีนี้ ให้สร้างตัวแปรใหม่เพื่อ แทน a และ b

$A \rightarrow a$

$B \rightarrow b$

ตัวอย่าง

$S \rightarrow abSb \mid aa$

$A \rightarrow a$

$B \rightarrow b$

ผลลัพธ์จะได้

$S \rightarrow aBSB \mid aA$

$A \rightarrow a$

$B \rightarrow b$

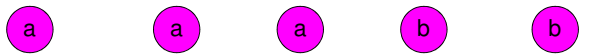
Efficient CFG Parsing Techniques

- Bottom-up parsing
  - CYK algorithm
- Top-down parsing
  - Earley parser

Bottom-up

$S \rightarrow AB, A \rightarrow BB \mid a, B \rightarrow AB \mid b$

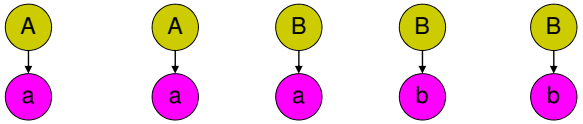
"aabbb"



Bottom-up

S → AB, A → BB | a, B → AB | b

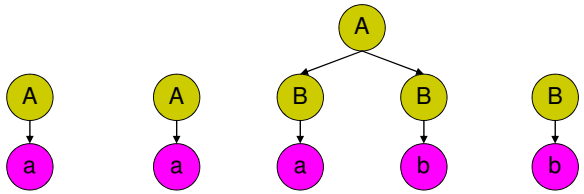
“aabbb”



Bottom-up

S → AB, A → BB | a, B → AB | b

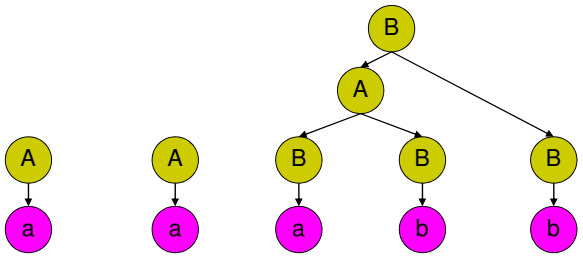
“aabbb”



Bottom-up

S → AB, A → BB | a, B → AB | b

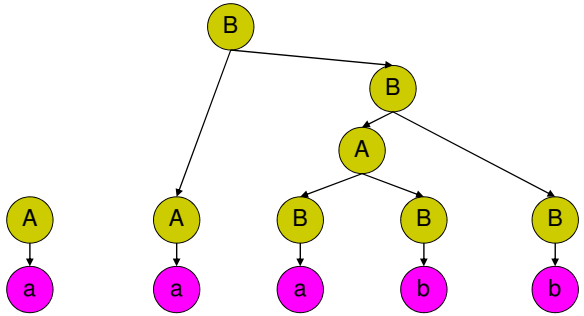
“aabbb”



Bottom-up

S → AB, A → BB | a, B → AB | b

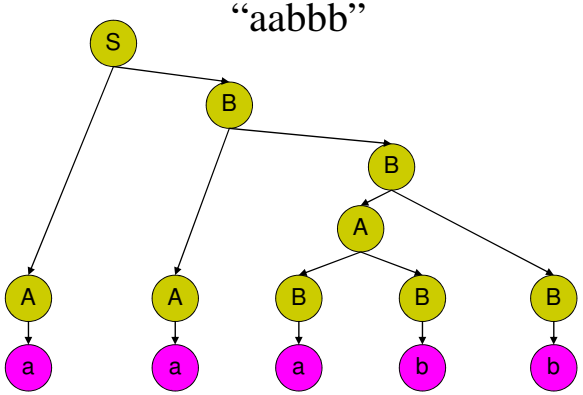
“aabbb”



Bottom-up

$S \rightarrow AB$, $A \rightarrow BB \mid a$, $B \rightarrow AB \mid b$

"aabbb"



Top-down

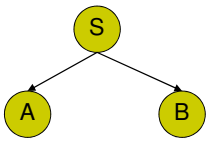
$S \rightarrow AB$, $A \rightarrow BB \mid a$, $B \rightarrow AB \mid b$

"aabbb"

Top-down

$\mathbf{S} \rightarrow \mathbf{AB}$, $A \rightarrow BB \mid a$, $B \rightarrow AB \mid b$

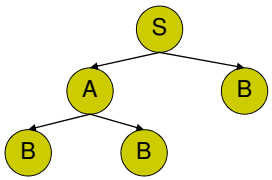
"aabbb"



Top-down

$S \rightarrow AB$, $\mathbf{A} \rightarrow \mathbf{BB} \mid a$, $B \rightarrow AB \mid b$

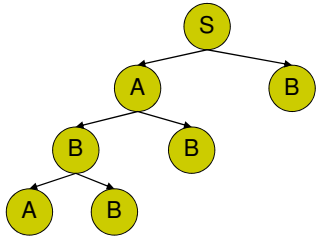
"aabbb"



Top-down

$S \rightarrow AB$, $A \rightarrow BB \mid a$, $\mathbf{B \rightarrow AB} \mid b$

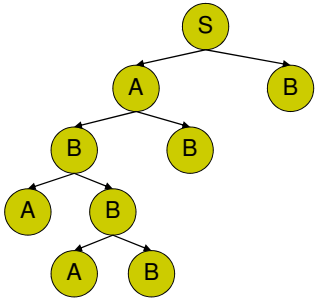
"aabbb"



Top-down

$S \rightarrow AB$, $A \rightarrow BB \mid a$, $\mathbf{B \rightarrow AB} \mid b$

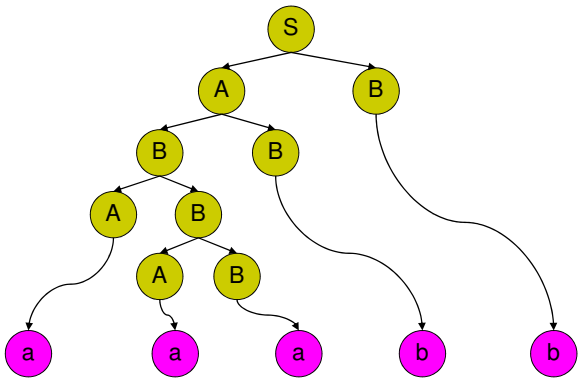
"aabbb"



Top-down

$S \rightarrow AB$, $\mathbf{A \rightarrow BB \mid a}$, $\mathbf{B} \rightarrow AB \mid \mathbf{b}$

"aabbb"



The CYK algorithm

- CYK ย่อมาจาก Cocke–Younger–Kasami
- เป็นวิธีการหา parse trees สำหรับ CFG โดยใช้แนวคิดแบบ bottom-up
- จำเป็นต้องแปลงไวยากรณ์ให้อยู่ในรูปแบบ Chomsky normal form (CNF) ก่อน
- worst case running time = $O(|G| \times n^3)$ โดยที่ n คือความยาวของสตริง และ $|G|$ คือขนาดของ CNF

The CYK algorithm

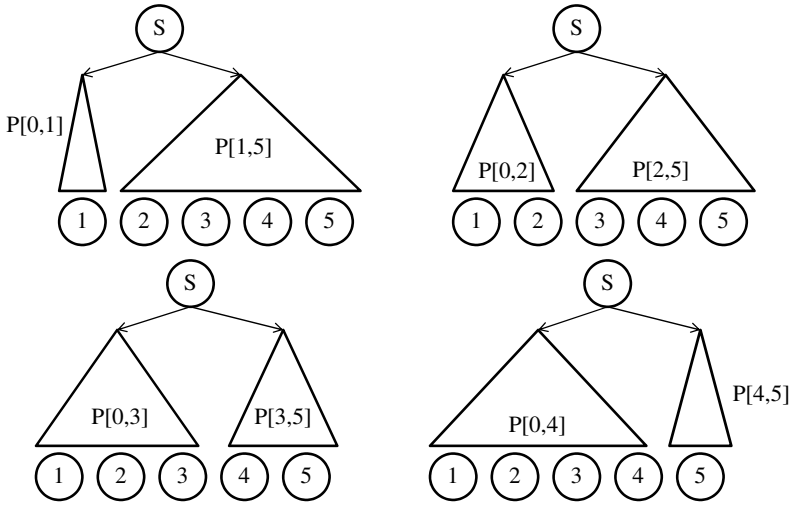
หลักการ

พิจารณาทุกๆ สตริงย่อยของอินพุตว่า แต่ละสตริงย่อยสามารถถูกสร้างขึ้นจากกฎใดได้บ้าง จากนั้นจึงบันทึกข้อมูลไว้ใน P[i,j] โดยที่ i และ j ไว้บอกตำแหน่งของสตริงย่อยในสตริง

การพิจารณาจะเริ่มจากสตริงย่อยขนาดเล็กที่สุดก่อน แล้วจะค่อยๆ เพิ่มขนาดของสตริงย่อยขึ้นไป โดยที่จะนำส่วนของที่ถูกพิจารณาไว้ก่อนแล้วมาใช้พิจารณาร่วมด้วยดังนี้

$$P[i,j] \bullet P[j,k] = P[i,k]$$

The CYK algorithm



The CYK algorithm

ให้ไวยากรณ์ G_1 คือ

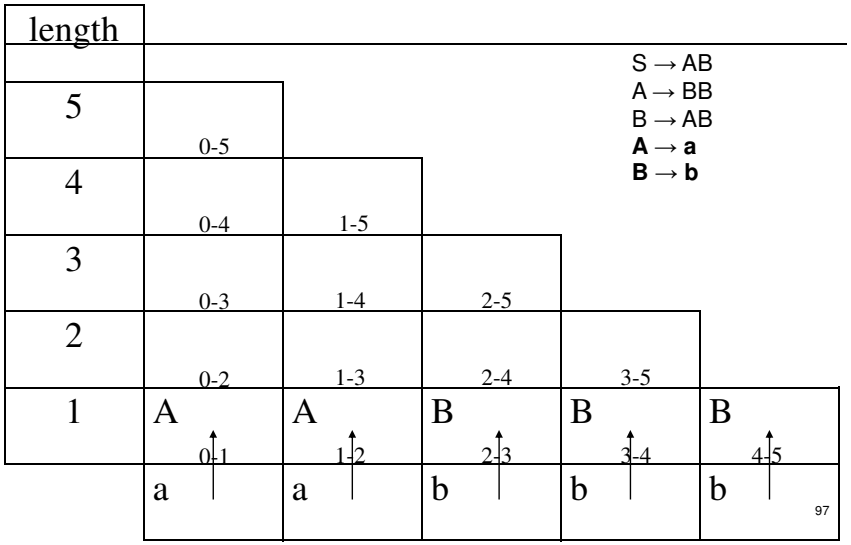
$$S \rightarrow AB$$
$$A \rightarrow BB \mid a$$
$$B \rightarrow AB \mid b$$

"aabbb" สามารถสร้างจาก G_1 ได้หรือไม่

The CYK algorithm

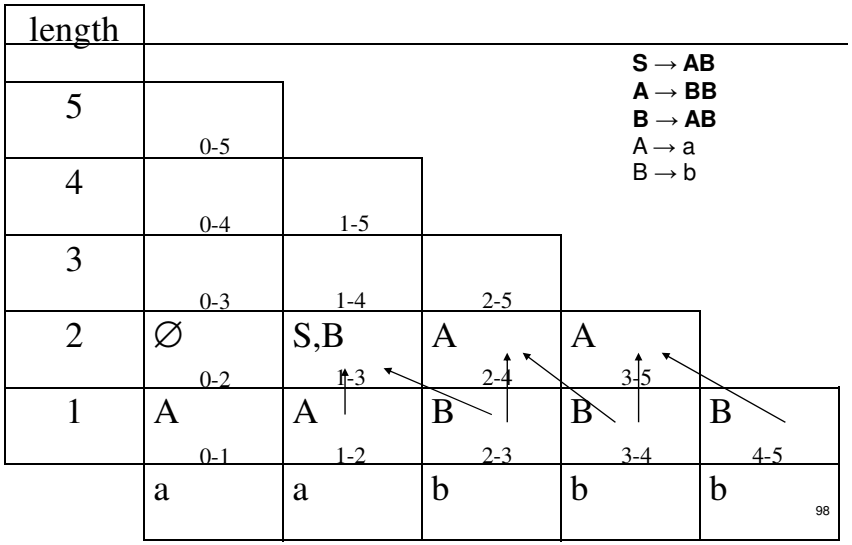
length					
5	0-5				
4	0-4	1-5			
3	0-3	1-4	2-5		
2	0-2	1-3	2-4	3-5	
1	0-1	1-2	2-3	3-4	4-5
	a	a	b	b	b

The CYK algorithm



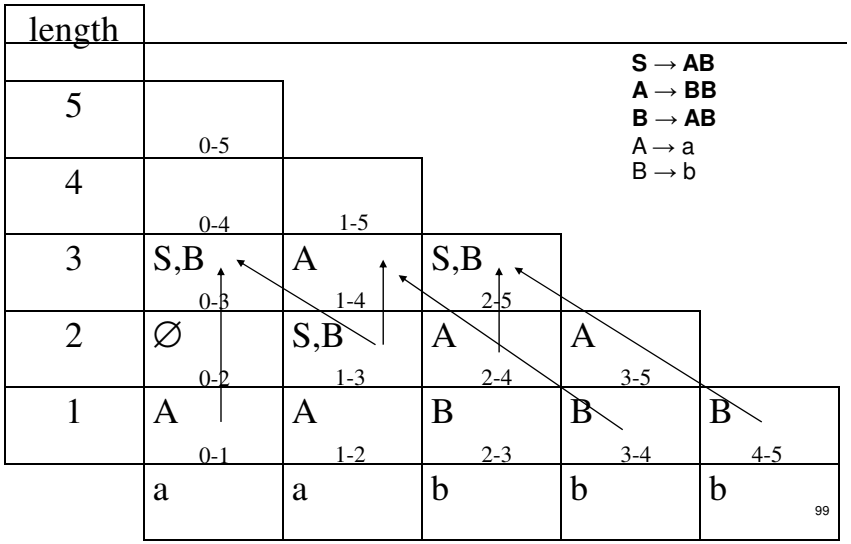
97

The CYK algorithm



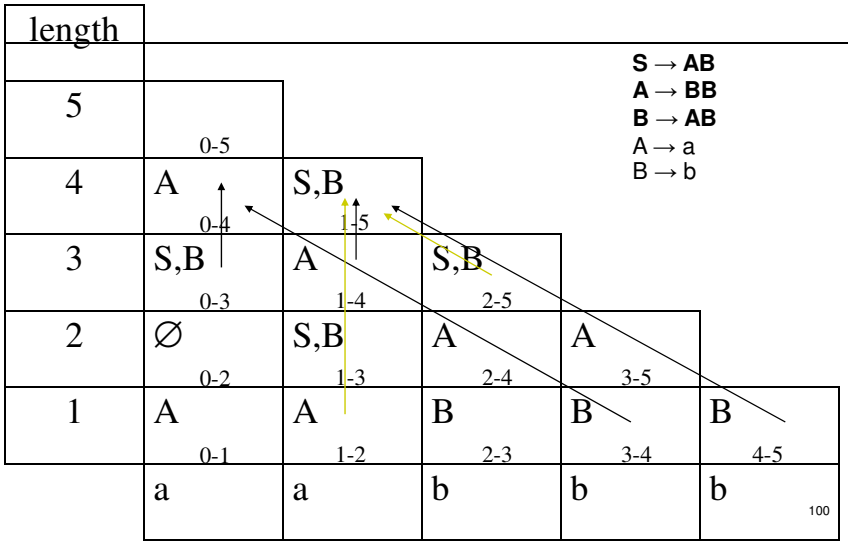
98

The CYK algorithm



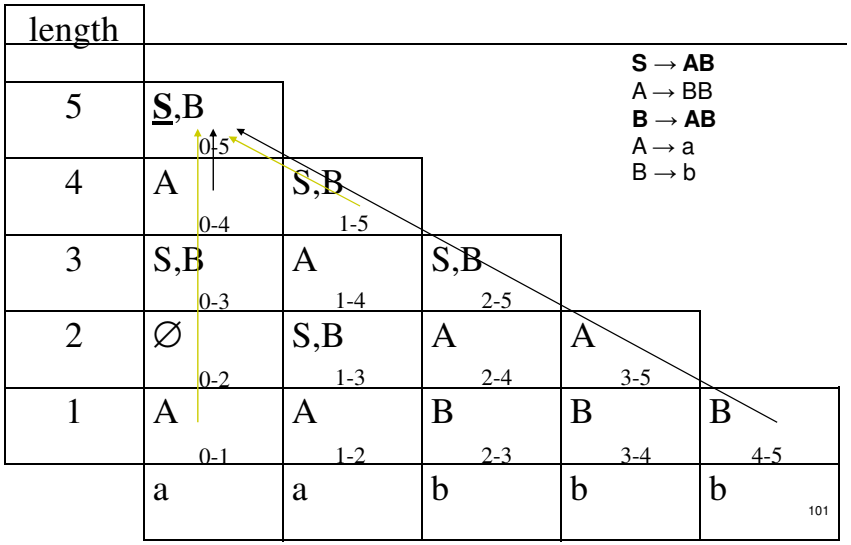
99

The CYK algorithm



100

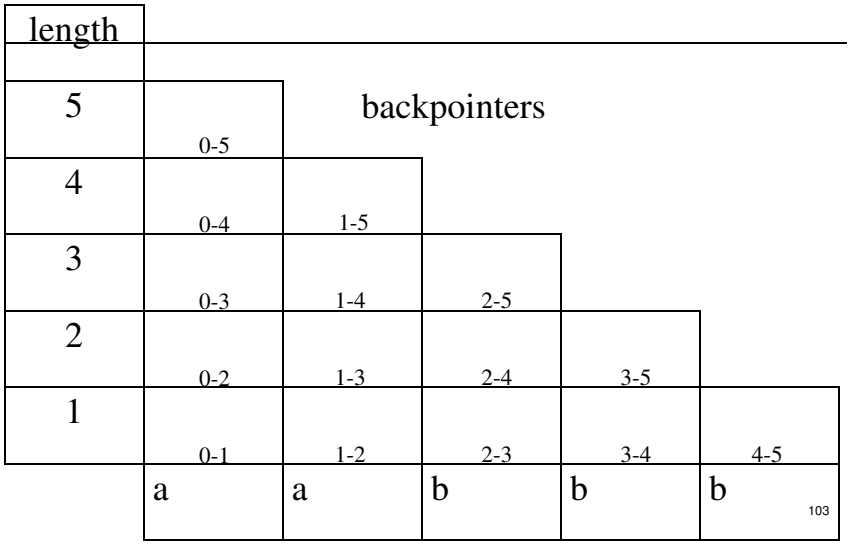
The CYK algorithm



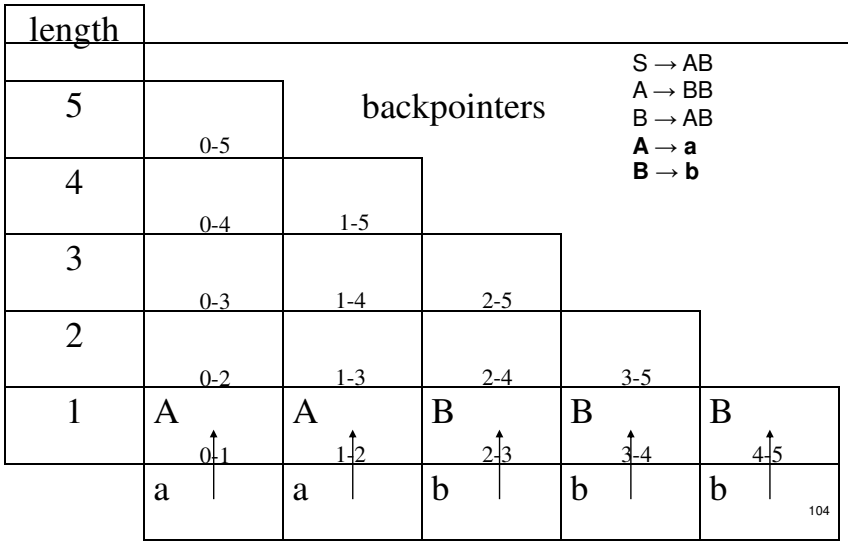
The CYK algorithm

- หาก S ปรากฏใน P[1,n] โดยที่ n คือความยาวของอินพุต แสดงว่าอินพุตนั้นสามารถถูกสร้างได้โดยไวยากรณ์นั้น
- ดังนั้นจึงสรุปได้ว่า aabbb สามารถสร้างได้โดย G_1
- หากต้องการหา parse trees สำหรับการสร้างสตริง สามารถทำได้โดยเพิ่มตาราง B[i,j] (backpointers) เพื่อบันทึกวิธีการสร้างข้อมูลในตำแหน่ง i,j ของตาราง P[i,j]

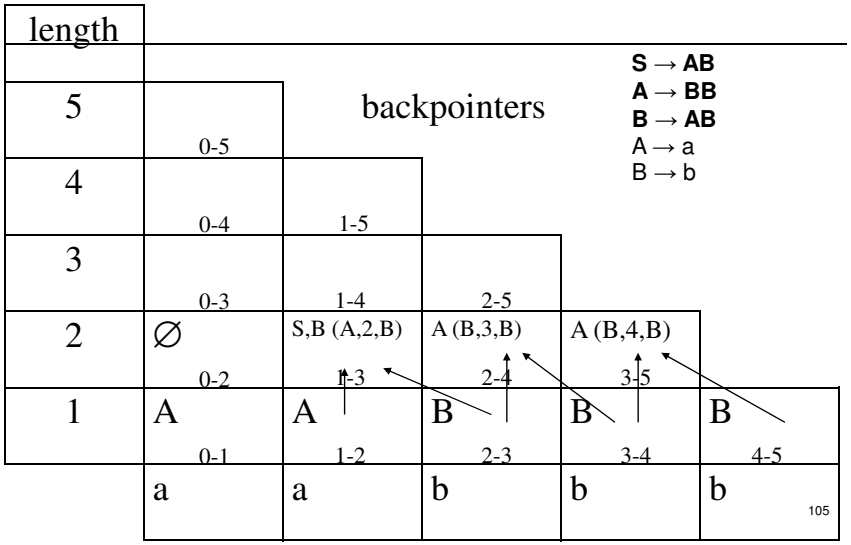
The CYK algorithm



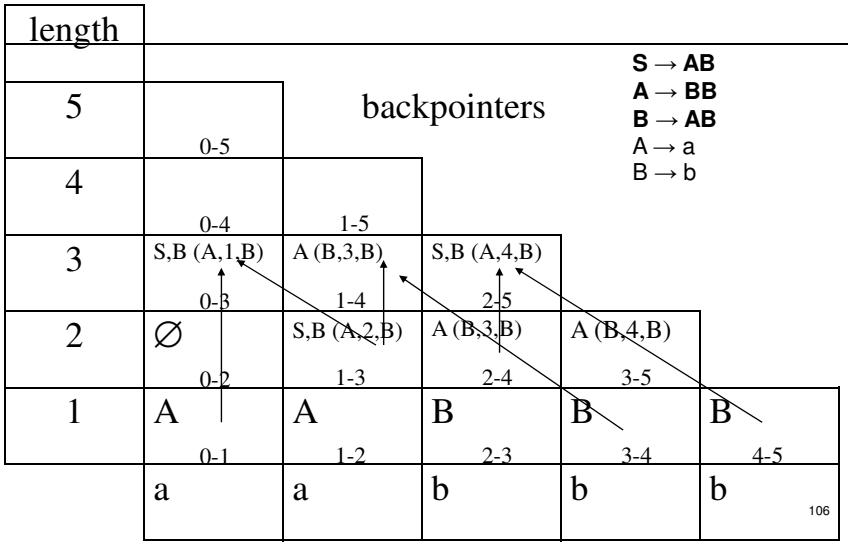
The CYK algorithm



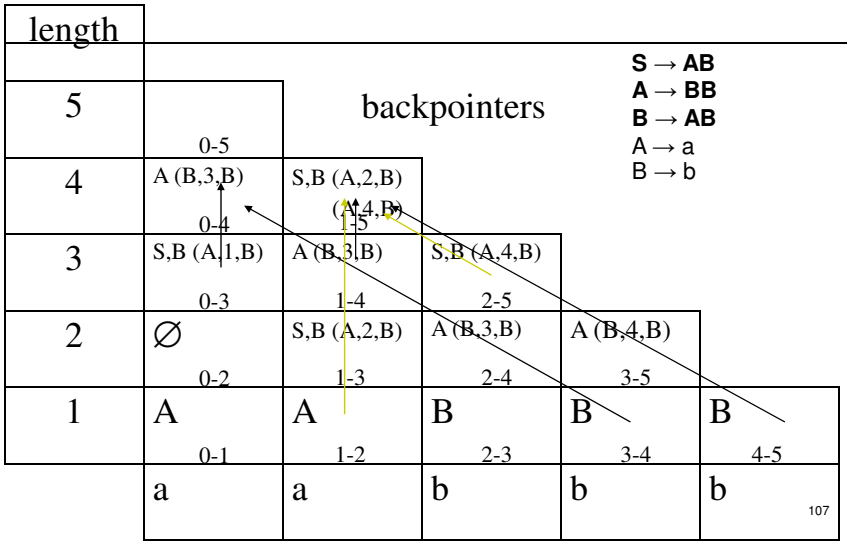
The CYK algorithm



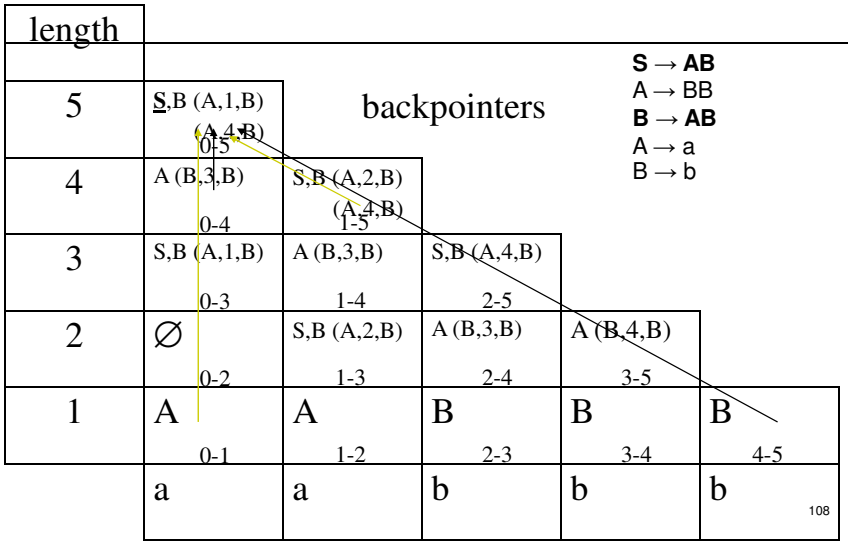
The CYK algorithm



The CYK algorithm



The CYK algorithm



The CYK algorithm

length					
5	S,B (A,1,B) (A,4,B) 0-5	backpointers			
4	A (B,3,B) 0-4	S,B (A,2,B) (A,4,B) 1-5			
3	S,B (A,1,B) 0-3	A (B,3,B) 1-4	S,B (A,4,B) 2-5		
2	∅ 0-2	S,B (A,2,B) 1-3	A (B,3,B) 2-4	A (B,4,B) 3-5	
1	A 0-1	A 1-2	B 2-3	B 3-4	B 4-5
	a	a	b	b	b

Earley Parser

- วิธีการหา parse trees จากไวยากรณ์ CFG ซึ่งจัดอยู่ในรูปแบบ chart parser ซึ่งคิดค้นโดย Jay Earley (1970)
- ใช้แนวคิดแบบ top-down ในการสร้าง parse trees (การคำนวณจะทำเฉพาะรูปแบบที่สามารถสร้างได้จาก S เท่านั้น)
- สามารถใช้ได้สำหรับไวยากรณ์ CFG ทุกรูปแบบ ไม่ต้องแปลงให้อยู่ในรูปแบบพิเศษก่อน
- Worst case running time = $O(n^3)$
- ถ้าสำหรับไวยากรณ์ที่ไม่กำกวม จะเป็น $O(n^2)$

Earley Parser

หลักการ

การทำงานจะแบ่งเป็นสถานะ โดยที่แต่ละสถานะจะแสดงถึงจำนวนอินพุตที่อ่านไปแล้ว เช่น อินพุตคือ aab สถานะการทำงานจะมี 4 สถานะ คือ

1. •aab (ยังไม่เริ่มอ่าน)
2. a•ab (อ่านไปแล้วหนึ่งตัว)
3. aa•b (อ่านไปแล้วสองตัว)
4. aab• (อ่านไปแล้วสามตัว)

Earley Parser

หลักการ (ต่อ)

ในแต่ละสถานะจะประกอบด้วยข้อมูลคู่ลำดับของ item และ origin position ซึ่งเรียกว่า suffix

- item คือ กฎไวยากรณ์ที่เพิ่มสัญลักษณ์ • เข้าไป (หนึ่งตัวในตำแหน่งใดก็ได้)ในฝั่งขวาของกฎ เช่น $B \rightarrow \bullet AB$ หรือ $B \rightarrow A \bullet B$ หรือ $B \rightarrow AB \bullet$
- origin position คือ เลขที่แทนค่าสถานะ จริงๆแล้วจะให้เลขอะไรก็ได้ที่ไม่ซ้ำกัน

Earley Parser

<u>หลักการ</u> (ต่อ)

มีฟังก์ชั่น 3 อัน ที่ใช้ในการเพิ่ม suffix เข้าไปในสถานะ คือ

- Prediction
- Scanner
- Completion

Earley Parser

- Prediction จะถูกใช้งานเมื่อมี item ของ suffix ใดๆ ที่มี • อยู่หน้าตัวแปร โดยการเพิ่ม suffix ใหม่เข้าไปในสถานะปัจจุบัน

  ณ. สถานะ t ใดๆ ถ้ามี suffix ดังนี้

  $M \rightarrow x \bullet Ny$,n : เมื่อ predict จะเพิ่ม suffix ใหม่คือ

  $N \rightarrow \bullet z$,t เข้าไปในสถานะ t

  $M, N \in$ V, $x,y \in$ (V $\cup$ T)* และ $z \in$ (V $\cup$ T)$^+$

Earley Parser

- Scanner จะถูกใช้งานเมื่อมี item ของ suffix ใดๆ ที่มี • อยู่หน้าตัวสิ้นสุด (terminal) และตัวสิ้นสุดนั้น สอดคล้องกับอินพุตของสถานะปัจจุบัน โดยจะเพิ่ม suffix ใหม่ในสถานะถัดไป

  ณ. สถานะ t : $a \bullet sb$

  มี suffix คือ $M \rightarrow x \bullet sy$,n เมื่อ scan จะเพิ่ม

  $M \rightarrow xs \bullet y$,n ในสถานะ t+1

  $M \in$ V, $s \in$ T, $a,b \in$ T*, และ $x,y \in$ (V $\cup$ T)*

Earley Parser

- Completion จะถูกใช้งานเมื่อมี item ของ suffix ใดๆ ที่มี • อยู่ในตำแหน่งสุดท้ายของกฎ

  ณ. สถานะ t ใดๆ ถ้ามี suffix ดังนี้

  $M \rightarrow z \bullet$,n และ

  ใน สถานะ n มี suffix ดังนี้

  $L \rightarrow x \bullet My$,s ให้เพิ่ม $L \rightarrow xM \bullet y$,s ในสถานะ t

  $L, M, N \in$ V, $x,y,z \in$ (V $\cup$ T)*

Earley Parser

ให้ไวยากรณ์ G_1 คือ

$$S \rightarrow AB$$
$$A \rightarrow BB \mid a$$
$$B \rightarrow AB \mid b$$

"aabbb" สามารถสร้างจาก G_1 ได้หรือไม่

Earley Parser

เริ่มต้นที่สถานะ 0 คือ •aabbb

S(0) : •aabbb

1. P → •S,0 #suffix เริ่มต้น
2. S → •AB,0 #predict จาก 1.
3. A → •BB,0 #predict จาก 2.
4. A → •a,0 #predict จาก 2.
5. B → •AB,0 #predict จาก 3.
6. B → •b,0 #predict จาก 3.

S(0,4) ตรงกับเงื่อนไขของ scanner

Earley Parser

S(1) : a•abbb

1. A → a•,0 #scan จาก S(0,4)

S(0) : •aabbb
2. S → •AB,0 #predict จาก 1.
5. B → •AB,0 #predict จาก 3.

เมื่อ • ไปอยู่ตำแหน่งท้ายสุด เข้าเงื่อนไข completion

2. S → A•B,0 # complete จาก S(0,2)
3. B → A•B,0 # complete จาก S(0,5)
4. B → •AB,1 # predict จาก 2,3.
5. B → •b,1 # predict จาก 2,3.
6. A → •BB,1 # predict จาก 4.
7. A → •a,1 # predict จาก 4.

B → AB | b

A → BB | a

S(1,7) ตรงกับเงื่อนไขของ scanner

Earley Parser

S(2) : aa•bbb

1. A → a•,1 # scan จาก S(1,7)

S(1)
4. B → •AB,1 # predict จาก 2,3

2. B → A•B,1 # complete จาก S(1,4)
3. B → •AB,2 # predict จาก 2.
4. B → •b,2 # predict จาก 2.
5. A → •BB,2 # predict จาก 3.
6. A → •a,2 # predict จาก 3.

B → AB | b

A → BB | a

S(2,4) ตรงกับเงื่อนไขของ scanner

Earley Parser

S(3) : aab•bb

1. B -> b•,2 # scan จาก S(2,4)
2. B → AB•,1 # complete จาก S(2,2)
3. A → B•B,2 # complete จาก S(2,5)
4. S → AB•,0 # complete จาก S(1,2)
5. B → AB•,0 # complete จาก S(1,3)
6. A → B•B,1 # complete จาก S(1,6)
7. P → S•,0 # complete จาก S(0,1)
8. A → B•B,0 # complete จาก S(0,3)
9. B → •AB,3 # predict จาก 3,6,8.
10. B → •b,3 # predict จาก 3,6,8.
11. A → •BB,3 # predict จาก 9.
12. A → •a,3 # predict จาก 9.

S(2)
2. B → A•B,1 # complete จาก S(1,4)
5. A → •BB,2 # predict จาก 3.

S(1)
2. S → A•B,0 # complete จาก S(0,2)
3. B → A•B,0 # complete จาก S(0,5)
6. A → •BB,1 # predict จาก 4.

S(0)
1. P → •S,0 #suffix เริ่มต้น
3. A → •BB,0 #predict จาก 2.

B → AB | b

A → BB | a

Earley Parser

S(4) : aabb•b

1. B → b•,3 # scan จาก S(3,10)
2. A → BB•,2 # complete จาก S(3,3)
3. A → BB•,1 # complete จาก S(3,6)
4. A → BB•,0 # complete จาก S(3,8)
5. A → B•B,3 # complete จาก S(3,11)
6. B → A•B,2 # complete จาก S(2,3)
7. B → A•B,1 # complete จาก S(1,4)
8. S → A•B,0 # complete จาก S(0,2)
9. B → A•B,0 # complete จาก S(0,5)
10. B → •AB,4 # predict จาก 5,6,7,8,9.
11. B → •b,4 # predict จาก 5,6,7,8,9.
12. A → •BB,4 # predict จาก 10
13. A → •a,4 # predict จาก 10

S(3)
3. A → B•B,2 # complete จาก S(2,5)
6. A → B•B,1 # complete จาก S(1,6)
8. A → B•B,0 # complete จาก S(0,3)
11. A → •BB,3 # predict จาก 9.

S(2)
3. B → •AB,2 # predict จาก 2

S(1)
4. B → •AB,1 # predict จาก 2,3

S(0)
2. S → •AB,0 #predict จาก 1.
5. B → •AB,0 #predict จาก 3.

Earley Parser

S(5) : aabbb•

1. B → b•,4 # scan จาก S(4,11)
2. A → BB•,3 # complete จาก S(4,5)
3. B → AB•,2 # complete จาก S(4,6)
4. B → AB•,1 # complete จาก S(4,7)
5. S → AB•,0 # complete จาก S(4,8)
6. B → AB•,0 # complete จาก S(4,9)
7. A → B•B,4 # complete จาก S(4,12)
8. P → S•,0 # complete จาก S(0,1)

S(4)
5. A → B•B,3 # complete จาก S(3,11)
6. B → A•B,2 # complete จาก S(2,3)
7. B → A•B,1 # complete จาก S(1,4)
8. S → A•B,0 # complete จาก S(0,2)
9. B → A•B,0 # complete จาก S(0,5)
12. A → •BB,4 # predict จาก 10

S(0)
1. P → •S,0 #suffix เริ่มต้น

เนื่องจาก P → S•,0 ปรากฏอยู่ในสถานะสุดท้าย (อ่านสตริงหมดแล้ว) จึงสรุปได้ว่า G_1 สามารถสร้างสตริง aabbb ได้