Top-down parsing

ต่อไปนี้เราจะพูดถึงการ parse แบบ top-down โดยที่การตัดสินใจว่าจะใช้ production ใดใน CFG นั้น สามารถตัดสินใจได้อย่างถูกต้องโดยดูจาก token ที่อ่านเข้ามาล่วงหน้าเพียงหนึ่งตัวเท่านั้น จุดเริ่มต้นของการนำไปสู่ parsing แบบนี้ เริ่มจาก grammar ที่ไม่มี ambiguity พิจารณา grammar ต่อไปนี้:

S -> E + S | E
E -> num | (S)

ซึ่งเราจะนำมาใช้ในการ parse สตริงต่อไปนี้:

(1) และ

(1) + 2

จะเห็นได้ว่าสตริงทั้งสองนี้สามารถจะถูก accept ได้ด้วย grammar นี้ อย่างไรก็ตาม พิจารณากรณีที่เราอ่าน token (เข้ามา ในกรณีของสตริง (1) เราจะต้องเลือก production

S -> E -> (S)

ขณะที่สำหรับสตริง (1) + 2 เราจะต้องเลือก production

S -> E + S

(เพราะถ้าเราเลือกเหมือนสตริงตัวแรก เราจะไม่สามารถ expand ในส่วนที่เป็น + 2 หลัง (1) ได้)

ดังนั้นสำหรับ grammar นี้ เราตัดสินใจไม่ได้เด็ดขาดว่าเราจะใช้ production ไหนหลังจากที่เราอ่าน token (เข้ามา grammar นี้จึงไม่ใช่ grammar ที่เราเรียกว่า LL(1) ซึ่งเป็น grammar ที่เราต้องการสำหรับการทำ parsing ของคอมไพเลอร์ CSubset ที่เราจะได้ใช้ต่อไปในวิชานี้ LL(1) ย่อมากจาก Left-to-right-scanning Left-most derivation 1 look-ahead symbol การที่จะทำให้ grammar นี้เป็น LL(1) เราจะต้องทำการ left-factor grammar นี้ดังต่อไปนี้

- ดึง common prefix ของ S (คือ E) ออกมา
- เพิ่ม non-termanal S'
- Production ของ S' จะ derive + S และ empty string ε

$$S \rightarrow E + S$$
$$S \rightarrow E$$
$$E \rightarrow \mathbf{num}$$
$$E \rightarrow (S)$$

$$\Downarrow$$

$$S \rightarrow E S'$$
$$S' \rightarrow \varepsilon$$
$$S' \rightarrow + S$$
$$E \rightarrow \mathbf{num}$$
$$E \rightarrow (S)$$

ลองใช้ grammar ใหม่นี้ parse (1) และ (1) + 2 โดยเริ่มต้นจาก start symbol S และสิ้นสุดที่ end-of-file ที่ใช้ symbol $

Derivation	Look-ahead
S	(
-> ES'	(
-> (S)S'	1
-> (ES')S'	1
-> (1S')S'	)
-> (1)S'	$
-> (1)	

Derivation	Look-ahead
S	(
-> ES'	(
-> (S)S'	1
-> (ES')S'	1
-> (1S')S'	)
-> (1)S'	+
-> (1) + S	2
-> (1) + ES'	2
-> (1) + 2S'	$
-> (1) + 2	

การเขียน parser **แบบ recursive descent สำหรับ** LL(1) CFG

เราจะใช้ตารางที่เรียกว่า predictive parsing table เป็นหลัก โดยจาก **grammar** (หลังจาก left-factor แล้ว) จะสร้าง table ได้ต่อไปนี้ โดยในแต่ละแถวเป็น non-terminal แต่ละตัว และแต่ละคอลัมน์เป็น **terminal** ที่เป็นไปได้ทั้งหมด

	num	+	(	)	$
S	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \varepsilon$	$\rightarrow \varepsilon$
E	$\rightarrow$ **num**		$\rightarrow (S)$		

และในแต่ละ cell ของตารางนี้ก็คือ production ที่จะใช้เมื่อกำลังจะ expand non-terminal ที่แถวหนึ่งและ look-ahead ไปพบกับ terminal ที่คอลัมน์หนึ่ง **cell** ที่ไม่มี production แสดงถึง error ในการ parse เนื่องมาจากสตริงที่ป้อนเข้ามาไม่สามารถจะ accept ได้ด้วย grammar นี้

เมื่อได้ตารางนี้แล้ว การเขียนโค๊ด **recursive descent** ทำได้อย่างตรงไปตรงมาดังต่อไปนี้ โดยให้มี **procedure** 3 อันตามจำนวน **non-terminal** ให้ชื่อว่า parse_S parse_S' และ parse_E จะเห็นได้ว่าถ้า **grammar** ของเราเป็น **LL(1)** โค๊ดของเราจะไม่มีการ **backtracking** แต่อย่างใด

```
void parse_S () {                    lookahead token
   switch (token) {
        case num: parse_E(); parse_S'(); return;
        case '(': parse_E(); parse_S'(); return;
        default: throw new ParseError();
   }
}
```

	number	+	(	)	$
→ *S*	→ *ES'*		→ *ES'*		
S'		→ +*S*		→ ε	→ ε
E	→ number		→ (*S*)		

```
void parse_S'() {
   switch (token) {
        case '+': token = input.read(); parse_S(); return;
        case ')': return;
        case EOF: return;
        default: throw new ParseError();
   }
}
```

	number	+	(	)	$
S	→ *ES'*		→ *ES'*		
→ *S'*		→ +*S*		→ ε	→ ε
E	→ number		→ (*S*)		

```
void parse_E() {
   switch (token) {
        case number: token = input.read(); return;
        case '(': token = input.read(); parse_S();
                  if (token != ')') throw new ParseError();
                  token = input.read(); return;
        default: throw new ParseError(); }
}
```

	number	+	(	)	$
S	→ *ES'*		→ *ES'*		
S'		→ +*S*		→ ε	→ ε
→ *E*	→ number		→ (*S*)		

Grammar ที่มี left recursion

การ parse แบบ recursive descent ใช้ไม่ได้กับ grammar ที่มี left recursion เช่น:

$$S \rightarrow S\ \alpha \mid \beta$$

ที่เป็นเช่นนี้ เพราะจะทำให้เกิด infinite recursion ที่ S จะเรียกตัวมันเองไปเรื่อยๆโดยไม่มีจุดที่จะ terminate อย่างไรก็ตาม ปัญหา left recursion ใน grammar นั้นแก้ได้ไม่ยากถ้าเปรียบเทียบกับการแก้ให้ grammar ไม่มี ambiguity นั่นคือมีวิธีที่สามารถเปลี่ยน grammar ที่มี left recursion ให้กลายเป็น grammar ที่เท่าเทียมกันแต่ไม่มี left recursion เช่น grammar ด้านบน สามารถกำจัด left recursion ได้โดยการเขียน grammar ใหม่ดังต่อไปนี้

$$S \rightarrow \beta\ S'$$
$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

ในกรณีทั่วไป ถ้าเรามี grammar ที่มี left recursion ในฟอร์มต่อไปนี้

$$S \rightarrow S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

เราสามารถเขียน grammar ใหม่ที่เท่าเทียมกันได้โดยไม่มี left recursion ดังต่อไปนี้

$$S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$$
$$S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$$

มาถึงจุดนี้เราสามารถบอก necessary conditions ของการเป็น LL(1) grammar ได้ดังต่อไปนี้

1. ต้องเป็น unambiguous grammar
2. ต้องเป็น grammar ที่ผ่านการ left-factor มาแล้ว
3. ต้องเป็น grammar ที่ไม่มี left recursion

อย่าลืมว่า conditions เหล่านี้ไม่ใช่ sufficient conditions ซึ่งหมายถึงว่าอาจมี grammar ที่ผ่าน conditions ทั้งสามที่กล่าวมา แต่ไม่เป็น LL(1) ก็เป็นได้

การหา FIRST และ FOLLOW เซ็ท

FIRST(A) คือเซ็ทของ terminal ทั้งหมดที่สามารถเป็นตัวเริ่มต้นของสตริงที่ขยายออกมาจาก A ได้ โดย A คือปริมาณที่ประกอบไปด้วย terminal และ/หรือ non-terminal ที่อยู่ทางด้านซ้ายหรือด้านขวาของ grammar ที่เรากำลังพิจารณา

พิจารณา grammar ต่อไปนี้

$T \rightarrow T * F$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

จะได้ว่า FIRST(T*F) = {id, num, (}

ถ้าให้ว่า A = XYZ สิ่งที่เราได้แน่ๆก็คือ FIRST(X) จะเป็นส่วนหนึ่งของ FIRST(A) และถ้า X เข้าหา empty string ได้ (เรียกว่า X นั้น nullable หรือ nullable(X) มีค่าเป็น true) เราก็จะได้ว่า FIRST(Y) ก็เป็นส่วนหนึ่งของ FIRST(A) ด้วย และถ้า Y เข้าหา empty string ได้ เราก็จะได้ว่า FIRST(Z) จะเป็นส่วนหนึ่งของ FIRST(A) ด้วยอีก

FOLLOW(X) คือเซ็ทของ terminal ตัวแรกที่จะตามหลัง X ได้เป็นตัวแรก โดย X คือปริมาณที่ประกอบไปด้วย terminal และ/หรือ non-terminal ที่อยู่ทางด้านซ้ายหรือด้านขวาของ grammar ที่เรากำลังพิจารณา นั่นคือถ้ามีการขยาย production จนได้ Xt โดยที่ t เป็น terminal แล้ว จะได้ว่า t เป็นสมาชิกของ FOLLOW(X) แต่แม้ว่าจะขยายไม่ได้ Xt โดยตรง แต่ได้เป็น XYZt แล้ว Y กับ Z นั้น nullable ก็จะได้ว่า t เป็นสมาชิกของ FOLLOW(X) เช่นกัน

ในการคำนวณหา FIRST และ FOLLOW เซ็ทนั้น เราจะใช้อัลกอริทึมดังต่อไปนี้

Algorithm to compute FIRST, FOLLOW, *and* nullable.
Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.
for each terminal symbol Z
  FIRST$[Z] \leftarrow \{Z\}$
repeat
  for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
      then nullable$[X] \leftarrow$ true
    for each i from 1 to k, each j from $i + 1$ to k
      if $Y_1 \cdots Y_{i-1}$ are all nullable (or if $i = 1$)
        then FIRST$[X] \leftarrow$ FIRST$[X] \cup$ FIRST$[Y_i]$
      if $Y_{i+1} \cdots Y_k$ are all nullable (or if $i = k$)
        then FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FOLLOW$[X]$
      if $Y_{i+1} \cdots Y_{j-1}$ are all nullable (or if $i + 1 = j$)
        then FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FIRST$[Y_j]$
until FIRST, FOLLOW, and nullable did not change in this iteration.

ALGORITHM 3.13. Iterative computation of *FIRST*, *FOLLOW*, and *nullable*.

สังเกตได้ว่าการคำนวณหา FOLLOW(X) นั้นขึ้นอยู่กับตำแหน่งที่ X ไปปรากฏอยู่ที่สตริงทางด้านขวาของ production rule ซึ่งจะแตกต่างจากการคำนวณ FIRST(X) ที่ดู X เป็นสตริงที่เริ่มต้น production rule (อยู่ทางด้านซ้ายของ production rule)

ต่อไปนี้จะแสดงตัวอย่างการหา FIRST และ FOLLOW เซ็ทตามอัลกอริทึมนี้

พิจารณา grammar ต่อไปนี้ ที่มี E เป็น start symbol

E -> TX

T -> (E) | int Y

X -> + E | ε

Y -> * T | ε

จะได้ว่า

nullable(X) = nullable(Y) = true

nullable(E) = nullable(T) = false

เริ่มจากการคำนวณหา FIRST ก่อน

ค่าเริ่มต้นของ FIRST(E) = FIRST(T) = FIRST(X) = FIRST(Y) = {}

FIRST(+) = { + }

FIRST(*) = { * }

FIRST(() = { (}

FIRST()) = {) }

FIRST(int) = { int }

รอบแรก (1st iteration)

FIRST(E) = FIRST(E) U FIRST(T) = {}

FIRST(T) = FIRST(T) U FIRST('(') = { (}

FIRST(T) = FIRST(T) U FIRST(int) = { (, int }

FIRST(X) = { + }

FIRST(Y) = { * }

รอบสอง (2nd iteration) โดยแสดงเฉพาะตัวที่เปลี่ยน

FIRST(E) = FIRST(E) U FIRST(T) = { (, int }

ผลลัพธ์สุดท้ายของ FIRST สำหรับ terminal และ non-terminal ของ grammar นี้ได้แสดงไว้ด้วยตัวหนา

ต่อไปเราจะแสดงการคำนวณหา FOLLOW บ้างโดยที่ เราจะเพิ่มกฎใน grammar อีกหนึ่งข้อ

S' -> E$

เพื่อระบุว่าหลังจากที่เราได้ขยาย production โดยเริ่มจาก start symbol E แล้ว เราจะจบด้วยตัว EOF (End of File) ซึ่งจะแทนด้วย $

ค่าเริ่มต้นของ FOLLOW(E) = FOLLOW(T) = FOLLOW(X) = FOLLOW(Y) = {}

รอบแรก (1st iteration)

FOLLOW(E) = FIRST($) = {$}

FOLLOW(T) = FOLLOW(T) U FOLLOW(E) = {$}

FOLLOW(T) = FOLLOW(T) U FIRST(X) = { + , $}

FOLLOW(X) = FOLLOW(X) U FOLLOW(E) = {$}

FOLLOW(E) = FOLLOW(E) U FIRST(')') = {) , $ }

FOLLOW(Y) = FOLLOW(Y) U FOLLOW(T) = { + , $ }

FOLLOW(E) = FOLLOW(E) U FOLLOW(X) = {) , $ }

FOLLOW(T) = FOLLOW(T) U FOLLOW(Y) = { + , $ }

รอบสอง (2nd iteration) โดยแสดงเฉพาะตัวที่เปลี่ยน

FOLLOW(T) = FOLLOW(T) U FOLLOW(E) = { + ,) , $}

FOLLOW(X) = FOLLOW(X) U FOLLOW(E) = {) , $ }

FOLLOW(Y) = FOLLOW(Y) U FOLLOW(T) = { + ,), $}

ในขั้นตอนเดียวกัน สามารถหา FOLLOW ของ terminal ต่างๆได้ดังต่อไปนี้

FOLLOW(() = FIRST(E) = { (, int }

FOLLOW())= FOLLOW(T) = { +,) , $ }

FOLLOW(int) = FIRST(Y) U FOLLOW(T) = { * , + ,) , $ }

FOLLOW(+) = FIRST(E) = { (, int }

FOLLOW(*) = FIRST(T) = { (, int }

การสร้าง predictive parsing table

เมื่อเราหา FIRST และ FOLLOW ได้แล้ว ต่อไปเราจะมาสร้างตาราง predictive parsing table T สำหรับ CFG G ใดๆ โดยใช้กฎดังนี้

สำหรับ production X -> A ใน G

1. ทุกๆ terminal t **∈** FIRST(A) ให้ใส่ production ->A เข้าไปในช่องของตาราง T[X, t] โดย T[X, t] คือช่องในตารางที่ตรงกับแถวของ non-terminal X และคอลัมน์ของ terminal t
2. ในกรณีที่ nullable(A) เป็น true ทุกๆ t **∈** FOLLOW(X) ให้ใส่ production ->A เข้าไปในช่องของตาราง T[X, t]
3. ในกรณีที่ nullable(A) เป็น true และ $ **∈** FOLLOW(X) ให้ใส่ production ->A เข้าไปในช่องของตาราง T[X, $]

ตาราง predictive parsing สำหรับ grammar ที่เราใช้เป็นตัวอย่างในการคำนวณ FIRST และ FOLLOW มีลักษณะดังต่อไปนี้

	(	)	+	*	int	$
E	->TX				->TX	
T	->(E)				->intY	
X		->**ε**	->+E			->**ε**
Y		->**ε**	->**ε**	->*T		->**ε**

ถ้าเรามี grammar ที่เป็น LL(1) เราสามารถที่จะตัดสินใจการใช้ production rule ที่ถูกต้องในการทำ derivation ของสตริงใดๆ ได้โดยดูจาก token ที่อ่านเข้ามาล่วงหน้าเพียงหนึ่งตัวเท่านั้น ดังนั้น entry ใน predictive parsing table แต่ละ entry นั้นจะมี production อยู่ได้อย่างมากที่สุดหนึ่งตัวเท่านั้น ถ้า predictive parsing table มี entry ที่มี production rule มากกว่าหนึ่ง จะบอกได้ทันทีว่า grammar นั้นไม่ใช่ LL(1)

พิจารณา grammar ต่อไปนี้:

S -> Sa | b

โดย a และ b เป็น terminal และ S เป็น non-terminal จะเห็นได้ว่าเรามี production rule อยู่สองตัวคือ:

S -> Sa และ

S -> b

ตาราง predictive parsing ที่เราจะสร้าง จะมีจำนวนแถวเท่ากับหนึ่งซึ่งแทน non-terminal S ที่เรามีอยู่ตัวเดียว และจะมีจำนวนคอลัมน์เท่ากับสามซึ่งแทน terminal ทั้งสามคือ a และ b และ $ (สัญลักษณ์ end of file) ในการเติมตารางนี้ เราจะต้องหา FIRST(Sa) และ FIRST(b) โดยเราจะได้ว่า

- b เป็นสมาชิกของ FIRST(S)

- FIRST(S) ⊆ FIRST(Sa)
- ดังนั้น FIRST(b) = FIRST(Sa) = { b }

เราไม่จำเป็นต้องหา FOLLOW เพราะเราไม่จำเป็นต้องใช้ข้อมูลจาก FOLLOW ในการเติมตาราง predictive parsing ที่จะสร้างขึ้นเนื่องจากไม่มี production rule ทางด้านขวามือใดๆใน grammar นี้ที่ derive ไปหา empty string **ε** ได้

ตาราง predictive parsing table สำหรับ grammar นี้เป็นดังต่อไปนี้

	a	b	$
S		->b	
		->Sa	

ดังนั้น grammar นี้ไม่ใช่ LL(1) เพราะใน entry T[S, b] มี production rule มากกว่าหนึ่ง

เรามีวิธีดู grammar อย่างเร็วๆ โดยไม่ต้องลงทุนสร้าง parsing table เพื่อตัดสินว่า grammar นี้ *ไม่* เป็น grammar ชนิด LL(1) โดยใช้ necessary conditions ของการเป็น LL(1) grammar นั่นคือ ถ้า grammar G เป็น LL(1) แล้ว G:

- เป็น unambiguous grammar
- ไม่มี left-recursion
- เป็น grammar ที่ได้ทำการ left-factor แล้ว

ดังนั้นถ้า grammar G ไม่เป็นไปตามหนึ่งในสาม condition ดังกล่าวมา เราจะบอกได้ทันทีว่า G ไม่ใช่ LL(1) ขอให้นิสิตระลึกไว้ว่า condition ทั้งสามนี้เป็น necessary ไม่ใช่ sufficient condition ดังนั้นถ้าในทางกลับกัน G เป็น grammar ที่เป็นไปตาม condition ทั้งสามนี้ จะยังตัดสินไม่ได้ทันทีว่า G เป็น LL(1) จะให้แน่ใจได้จริงๆว่า G เป็น LL(1) เราจะต้องกลับไปสร้างตาราง predictive parsing table ของ G และต้องให้แน่ใจว่าทุกๆ entry ในตารางที่สร้างขึ้นมี production rule อยู่อย่างมากที่สุดหนึ่งตัว