204433 วิชาการแปลภาษาโปรแกรม

Bottom-up parsing

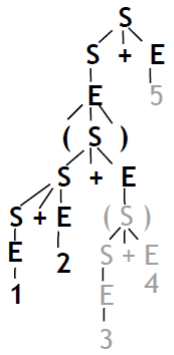
พิจารณา CFG ต่อไปนี้

S → S + E | E
E → number | (S)

และสตริงต่อไปนี้

(1+2+(3+4))+5

ถ้าเรา parse แบบ top-down เราจะได้ parse tree ตามด้านล่างนี้



เรา scan จากซ้ายไปขวา และทุกๆครั้งที่เราพบ terminal เรา predict ว่าจะใช้ production rule ตัวใดที่ในที่สุดจะนำเราไปสู่การสร้าง (derive) สตริงเริ่มต้นได้

แนวคิดใหม่ในการ parse แบบ bottom-up นั้น จะscan จากซ้ายไปขวาเหมือนกัน แต่จะแตกต่างจาก top-down อย่างสิ้นเชิงตรงที่การ parse แบบนี้จะพยายาม reduce ตัวสตริงที่รับเข้ามาทางอินพุทเข้าไปหา start symbol S ดังแสดงด้านล่าง

(1+2+(3+4))+5 ←	(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(1 +2+(3+4))+5
(*S*+2+(3+4))+5 ←	(1 +2+(3+4))+5
(S+*E*+(3+4))+5 ←	(1+2 +(3+4))+5
(*S*+(3+4))+5 ←	(1+2+(3 +4))+5
(S+(*E*+4))+5 ←	(1+2+(3 +4))+5
(S+(*S*+4))+5 ←	(1+2+(3 +4))+5
(S+(S+*E*))+5 ←	(1+2+(3+4))+5
(S+(*S*))+5 ←	(1+2+(3+4))+5
(S+*E*)+5 ←	(1+2+(3+4))+5
(*S*)+5 ←	(1+2+(3+4))+5
E+5 ←	(1+2+(3+4)) +5
S+*E* ←	(1+2+(3+4))+5
S	(1+2+(3+4))+5

การ parse แบบนี้ประกอบไปด้วย

- สองปฏิบัติการคือ shift และ reduce
- มี stack เอาไว้เก็บสถานะของ parser

- ภายใน stack ประกอบด้วยทั้ง terminal และ non-terminal

การ shift คือการอ่าน token เข้ามาและ push ลง stack ไม่มีผลต่อการเปลี่ยนแปลงสัญลักษณ์บน stack ตัวอย่างการทำ shift ของการ parse ด้านบนที่เกิดขึ้นในครั้งแรกที่เราอ่าน token (เข้ามา

stack	input	action
(	1+2+(3+4))+5	*shift* 1
(1	+2+(3+4))+5	

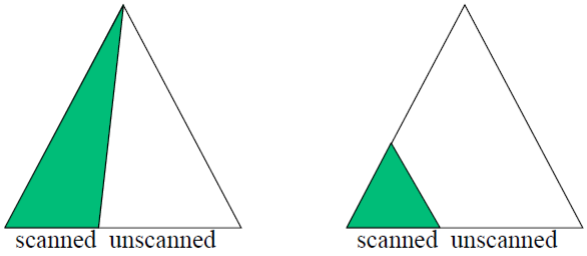
การ reduce คือการแทนที่ส่วน string ที่อยู่ด้านบนของ stack ด้วย non-terminal ที่อยู่ทางด้านซ้ายของ production rule ที่ให้กำเนิด string ดังกล่าว ตัวอย่างด้านล่างแสดงการทำ reduce ในขั้นสุดท้ายของการ parse ด้านบน

stack	input	action
(S+E	+(3+4))+5	*reduce* S→ S+*E*
(*S*	+(3+4))+5	

ด้านล่างแสดงส่วนการ parse แบบ bottom-up ที่ได้แสดงมาด้านบนอีกครั้ง แต่มี action กำกับว่า shift หรือ reduce (และ reduce ด้วย production rule ใด) สังเกตว่าการ parse แบบนี้จะ reduce เข้าหา non-terminal ด้านขวามือที่สุดเสมอ

derivation	stack	input stream	action
(1+2+(3+4))+5 ←		(1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(	1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(1	+2+(3+4))+5	*reduce* E→num
(*E*+2+(3+4))+5 ←	(E	+2+(3+4))+5	*reduce* S → *E*
(*S*+2+(3+4))+5 ←	(S	+2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+	2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+2	+(3+4))+5	*reduce* E→num
(S+*E*+(3+4))+5 ←	(S+E	+(3+4))+5	*reduce* S→ S+*E*
(*S*+(3+4))+5 ←	(S	+(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+	(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(	3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(3	+4))+5	*reduce* E→num

จะสังเกตได้ว่าการ parse แบบ bottom-up นั้น ถ้าเปรียบกับการ parse แบบ top-down เราจะต้องจดจำส่วนของ parse tree ไว้น้อยกว่าการทำ top-down parsing มาก (เปรียบเทียบว่าเรา scan จำนวน token เข้ามาเป็นจำนวนเท่ากัน)



Top-down Bottom-up

ปัญหาสำคัญสำหรับการ parse แบบ bottom-up ในตอนนี้คือเราจะรู้ได้อย่างไรว่าเวลาใดจะ shift เวลาใดจะ reduce และถ้า reduce จะทำด้วย production อะไร อย่าลืมว่าการตัดสินใจของเราในแต่ละครั้งจะต้องสามารถนำไปสู่การ reduce จนเหลือเพียง start symbol ได้

เราจะแก้ปัญหานี้ตามแนวคิดดังต่อไปนี้ ณ เวลาหนึ่งที่ parser กำลังทำงาน stack จะเป็นตัวบ่งบอก state ของ parser ณ ขณะนั้น และถ้าเรา scan terminal เข้ามา state ของ stack อาจจะต้องมีการเปลี่ยนแปลงไป ดังนั้นถ้าเราสามารถสร้าง DFA มาแทน state ของ stack ทั้งหมดที่เป็นไปได้ รวมไปถึงการเปลี่ยน state เมื่อพบกับ terminal หรือ non-terminal ใดๆ เราน่าจะสามารถตัดสินใจว่าจะกระทำการ shift หรือ reduce ได้อย่างถูกต้องในแต่ละ state ของ stack อย่าลืมว่า stack จะบรรจุ terminal และ non-terminal ของ grammar ที่เรากำลัง parse ดังนั้น state ของ stack ก็คือสิ่งที่อยู่ภายใน stack จาก top-of-stack ลงไปนั่นเอง

ในการทำความเข้าใจ bottom-up parsing นั้น เราจะเริ่มจากการ parse grammar แบบ LR(0) ซึ่งเป็น grammar ที่มีความสามารถจำกัด แต่จะช่วยให้เราเข้าใจกระบวนการ parse ได้ดี LR(0) ย่อมาจาก: **L**eft-to-right scanning **R**ight-most derivation และ “zero” look-ahead characters เมื่อเปรียบเทียบกับ LL grammar ที่เราได้คุ้นเคยมาแล้ว ความแตกต่างอยู่ที่ LR(0) ทำ right most derivation แต่พวก LL จะทำ left most derivation ที่เป็นเช่นนี้เพราะ bottom-up parsing จะ reduce เข้าหา right most non-terminal เสมอ ถ้าเรายอ้นกลับทิศทางการ parse จาก start symbol ไปยัง string เริ่มต้น (กลับไปดูตัวอย่างการ parse string (1+2+(3+4))+5 ในตอนเริ่มต้นหัวข้อ bottom-up parsing)

เราจะพิจารณา LR(0) grammar ต่อไปนี้
S → (L) | id
L → S | L , S

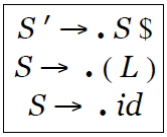
นิยาม item และ state ดังต่อไปนี้
item คือ production ที่ด้านขวามือ (RHS) มีสัญลักษณ์ . (dot) อยู่ด้วย เช่น
S → (. L)

เป็น item ของ production S → (L)

state คือเซ็ทของ items

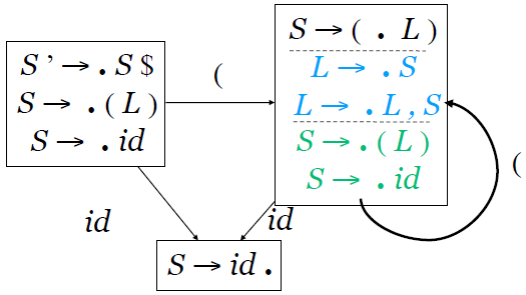
เราจะให้ items จำลองภาพของ stack โดย . (dot) เป็นตัวบอก top-of-stack และ string หลังจาก dot เป็นสิ่งที่ parser กำลังจะต้องประสบในเวลาต่อไป ส่วน state ก็จะบ่งบอกความเป็นไปได้ของรูปแบบ stack ทั้งหมดเวลาที่ parse grammar นี้ เมื่อมีนิยามทั้งสองนี้แล้ว เราพร้อมจะมาสร้าง DFA เพื่อทำ bottom-up parsing ของ grammar นี้แล้ว

การสร้าง DFA เริ่มจาก การแนะนำ production ใหม่คือ S’ → S$ ซึ่งเป็น production เริ่มต้นเข้าหา start symbol S และจบลงด้วย $ (end-of-file) จากนั้นเราหา closure ของ item S’ → .S$ เราจะได้ state เริ่มต้นมาดังต่อไปนี้



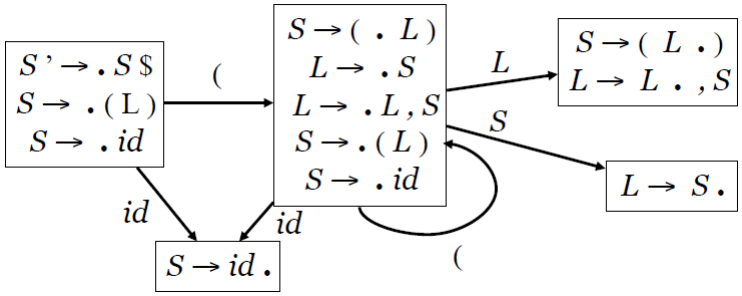
การทำ closure คือการเพิ่ม item เข้าไปใน state โดยเป็น item ที่มาจาก production ที่อยู่หลังจาก . (dot) ซึ่งในกรณีนี้เรามี S ที่อยู่หลังจาก dot และหลังจากที่เราเพิ่ม item ใน S ลงมาแล้ว เราไม่สามารถทำ closure ต่อไปได้อีกเพราะ item จาก production S มีเพียง terminal ที่ตามหลัง dot

จาก state เริ่มต้นนี้ เราดูว่าถ้า scan พบ terminal ใดๆใน grammar เราจะไปยัง state ใดต่อ แน่นอนว่าจาก state เริ่มต้นนี้ถ้าเรา scan terminal ใดๆ ที่ไม่ใช่ (หรือ id ย่อมจะเกิด error เพราะ grammar นี้บ่งว่าถ้าเริ่มจาก S (start symbol) แล้วจะพบได้เพียง (หรือ id เท่านั้น ด้านล่างแสดงการเพิ่ม state ใน DFA เมื่อ scan terminal (หรือ id เข้ามา

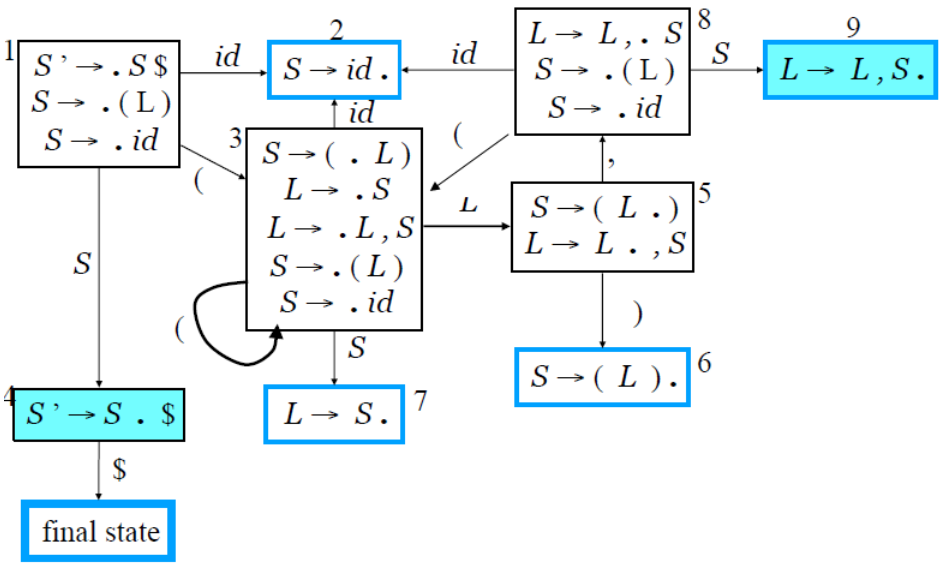


จะเห็นได้ว่าสำหรับ state ที่มี item S → (. L) คือ state ที่ระบุว่าขณะนี้ (อยู่ที่ top-of-stack และจะเห็นได้ว่าเรามีการทำ closure ของ state นี้ด้วยดังแสดง การเลื่อน dot ไปทางด้านขวาแสดงถึงการเติบโตของ stack

DFA ของเรานอกจากจะต้องดู input ที่เป็น terminal แล้ว เรายังจะต้องพิจารณาตัว non-terminal อีกด้วย ภาพด้านล่างแสดงการเพิ่ม state ใน DFA เมื่อประสบกับ non-terminal L และ S จาก state ที่มี item S → (. L) อยู่



เมื่อเราค่อยๆไล่พิจารณาแต่ละ state ว่าเมื่อรับ input ที่เป็น terminal หรือ non-terminal เข้ามา จะไปอยู่ที่ state ใด แล้วเราทำ closure ของ state นั้น ย้อนกลับไปทำเช่นนี้เรื่อยๆจนกระทั่งสุดท้ายเราจะได้ DFA ของการ parse grammar LR(0) ตัวนี้มาดังแสดงต่อไปนี้



state ที่ dot ไปอยู่ท้ายสุดของ production ทางด้านขวา เป็น state ที่เราจะทำการ reduce กลับไปที่ non-terminal ทางด้านซ้ายมือ (แทนที่ string จาก top-of-stack ลงไป ที่ match (เทียบเท่ากับ) RHS ของ production นี้ด้วย non-terminal ทางด้านซ้าย) เรา label แต่ละ state ด้วยตัวเลขจาก 1 ไปถึง 9 state ที่จะทำการ reduce คือ 2 4 6 7 และ 9 ซึ่งใส่กรอบสีฟ้าไว้ state พิเศษอีกอันหนึ่งคือ accept หรือ final state ที่เข้าถึงได้เมื่อ scan $ เข้ามาหลังจากที่ได้ parse S เรียบร้อยแล้ว

จาก DFA ที่ได้ เราสามารถนำมาสร้าง parsing table ของ LR(0) grammar นี้ได้ดังแสดงด้านล่างนี้

	(	)	*id*	,	$	*S*	*L*
1	s3		s2			g4	
2	*S→id*	*S→id*	*S→id*	*S→id*	*S→id*		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	*S→(L)*	*S→(L)*	*S→(L)*	*S→(L)*	*S→(L)*		
7	*L→S*	*L→S*	*L→S*	*L→S*	*L→S*		
8	s3		s2			g9	
9	*L→L,S*	*L→L,S*	*L→L,S*	*L→L,S*	*L→L,S*		

อัลกอริทึมอย่างเป็นทางการในการหา DFA ของ grammar แบบ LR(0) เป็นดังนี้

ให้ T และ E เป็นเซ็ทของ state และ edge ที่แสดงการเปลี่ยนแปลง state

Initialize T to {**Closure**($\{S' \rightarrow .S\$\}$)}
Initialize E to empty.
repeat
  for each state I in T
    for each item $A \rightarrow \alpha.X\beta$ in I
      let J be **Goto**(I, X)
      $T \leftarrow T \cup \{J\}$
      $E \leftarrow E \cup \{I \xrightarrow{X} J\}$
until E and T did not change in this iteration

และการหา Closure และ Goto ตามอัลกอริทึมด้านบนเป็นดังต่อไปนี้

Closure(I) =
  repeat
    for any item $A \rightarrow \alpha.X\beta$ in I
      for any production $X \rightarrow \gamma$
        $I \leftarrow I \cup \{X \rightarrow .\gamma\}$
  until I does not change.
  return I

Goto(I, X) =
  set J to the empty set
  for any item $A \rightarrow \alpha.X\beta$ in I
    add $A \rightarrow \alpha X.\beta$ to J
  return Closure(J)