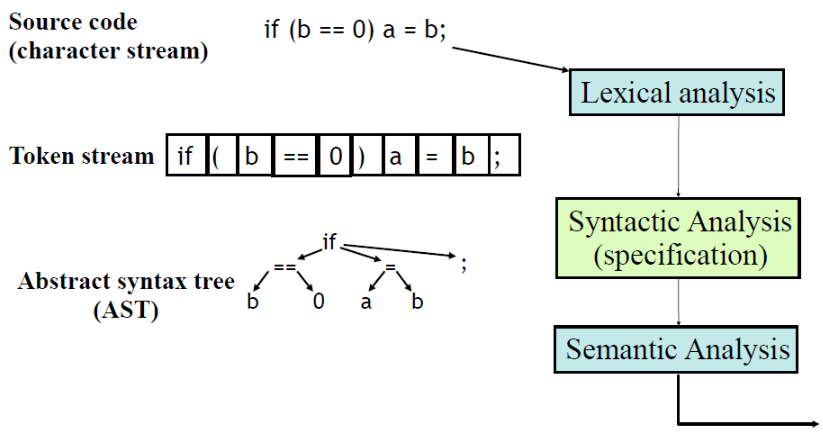


มาถึงจุดนี้ เราได้เข้าใจกระบวนการแปลภาษาในขั้นตอนแรก (phase) ที่สุดคือการทำ lexical analysis เรียบร้อยแล้ว ผลลัพธ์ที่ได้จากการทำ lexical analysis ก็คือ token stream ที่จะนำไปเป็นอินพุทของขั้นตอนต่อไปนั่นคือการทำ syntactic analysis หัวใจของการวิเคราะห์ในขั้นตอนนี้คือการทำ parsing เพื่อให้รู้ว่า token stream ที่ได้จาก lexical analysis ของโปรแกรมนั้น มีโครงสร้างและแบบแผนตามที่ภาษาระดับสูงนั้นได้กำหนดไว้หรือไม่ โดยเราจะใช้ CFG เป็นตัวระบุ (specify) โครงสร้างและแบบแผนของภาษา ผลลัพธ์ที่ได้จากการ parsing คือ Abstract Syntax Tree (AST) ซึ่งเราจะนำไปใช้ในขั้นตอนต่อไปคือการทำ semantic analysis ในการสร้าง AST นั้น เราจะทำในขณะที่เรา parse ตัว CFG ซึ่งเราจะได้พูดกันถึงในรายละเอียดต่อไป

การทำ parsing ตรวจเฉพาะโครงสร้างและแบบแผน แต่จะไม่ได้เช็คในสิ่งที่เกี่ยวข้องกับความสอดคล้องกันของชนิดข้อมูล หรือการใช้ตัวแปรที่ยังไม่ได้ประกาศ เช่น ในภาษาจาวา int x = true เป็นประโยคที่มีโครงสร้างถูกต้อง ซึ่งจะผ่านขั้นตอน parsing แต่จะเห็นได้ว่าประโยคนี้ assign ค่า boolean เข้ากับตัวแปร int ทำให้เกิดความไม่สอดคล้องกัน เราจะตรวจจับกรณีดังกล่าวมาเช่นนี้ในขั้นตอน semantic analysis

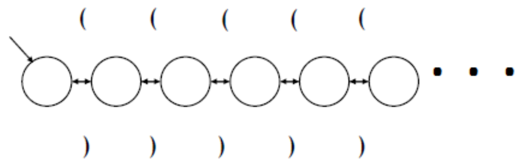
Context Free Grammar (CFG)

ในการทำ lexical analysis เราใช้ regular expression (RE) เป็นตัวระบุ token และ non-token อย่างไรก็ตาม RE มีขีดความสามารถจำกัด และถ้าเราจะ specify สิ่งที่ซับซ้อนไปกว่า token เช่น specify โครงสร้างและแบบแผนของภาษาระดับสูง เราจะต้องหากรอบ (framework) ที่มีความสามารถมากกว่า RE นั่นคือ CFG ลองมาดูตัวอย่างสถานการณ์ที่ RE ทำไม่ได้แต่ CFG ทำได้

ให้ว่าเราต้องการตรวจจับ string ที่ทุกๆวงเล็บเปิดจะมีวงเล็บปิดมารองรับ (balanced parentheses) เช่น

() (()) ()()() (())()((()()))
(( )(()) (()()

ถ้าเราคิดว่าเราจะหา DFA ที่มาตรวจจับ string นี้ เราจะต้องใช้ DFA ที่มีจำนวน state เป็น infinity เพราะว่าเราจะต้องจดจำวงเล็บเปิดในทุกๆครั้งที่พบมัน ตัวอย่างโครงสร้างของ DFA เป็นตามรูปด้านล่าง



แต่ตามนิยาม DFA จะต้องมีจำนวน state เป็นค่า finite เท่านั้น จะเห็นได้ว่าในสถานการณ์นี้ DFA ซึ่งมีความสามารถเท่ากับ RE ไม่สามารถตรวจจับ สตริงแบบ balanced parentheses ได้ ลองมาดูว่า CFG จะทำได้หรือไม่ โดยเราจะเริ่มจากนิยามของ CFG กันก่อน

CFG ประกอบไปด้วย

- Terminal ที่เป็น token หรือ empty string ε
- Non-terminal ที่เป็นตัวแปรที่จะต้องถูกขยายความ (expand)
- Start symbol ซึ่งเป็น non-terminal ที่ใช้สัญญลักษณ์ S
- Production บ่งบอกการขยายความของ non-terminal เพื่อประกอบขึ้นเป็น string จะประกอบไปด้วยปริมาณด้านซ้าย (LHS) ซึ่งจะต้องเป็น non-terminal เพียงหนึ่งตัวเท่านั้น และ ปริมาณด้านขวา (RHS) ซึ่งเป็นสตริงของทั้ง terminal และ non-terminal ประกอบกัน

CFG ที่ใช้ในการตรวจจับ balanced parentheses คือ

S -> (S)S

S -> ε

เราบอกว่า CFG ตรวจจับ (accept) สตริงใดๆ เมื่อเราสามารถสร้างสตริงนั้น (เรียกว่าการทำ derivation) โดยใช้ production ของ CFG เช่นให้สตริง (( )) จะเห็นได้ว่า CFG ด้านบน accept สตริงนี้เพราะเราสามารถสตริงนี้ได้ดังต่อไปนี้

$$S = (S)\,\varepsilon = ((S)\,S)\,\varepsilon = ((\varepsilon)\,\varepsilon)\,\varepsilon = (())$$

RE เป็น subset ของ CFG ตัวอย่างเช่น RE (0 | 1)* 1 สามารถเปลี่ยนให้อยู่ในรูปของ CFG ได้ดังนี้

S -> S' digit1

S' -> S' digit | Ɛ

digit -> digit0 | digit1

digit1 -> 1

digit0 -> 0

ซึ่งกฎเกณฑ์ในการแปลง RE ใดๆให้เป็น CFG ที่เทียบเท่านั้นเป็นไปอย่างตรงไปตรงมาดังต่อไปนี้

1. ถ้า RE เป็นเพียง sequence ของ character เช่น xyz หรือ empty string เราสามารถสร้าง CFG production ได้คือ $P \rightarrow \mathrm{xyz}$
2. ถ้า r และ s เป็น RE ที่มี production เป็น R และ S ตามลำดับ จะได้ว่า rs มี production T เป็น $T \rightarrow R\ S$
3. ถ้า r และ s เป็น RE ที่มี production เป็น R และ S ตามลำดับ จะได้ว่า r | s มี production T เป็น $T \rightarrow R \mid S$
4. ถ้า r มี production คือ R จะได้ว่า r* มี production RStar เป็น $RStar \rightarrow R\ RStar \mid \varepsilon$
5. ถ้า r มี production คือ R จะได้ว่า r+ มี production RPlus เป็น $RPlus \rightarrow R\ RPlus \mid R$

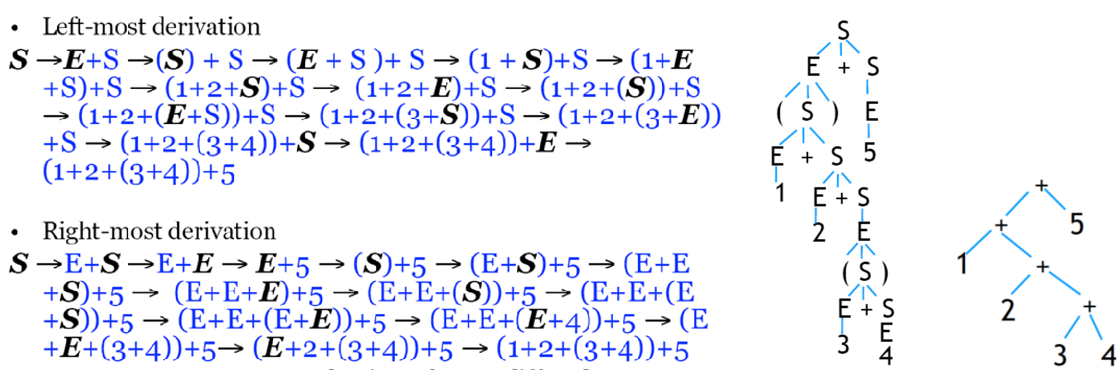
Unambiguous Grammar

พิจารณา grammar ต่อไปนี้

S -> E + S | E

E -> number | (S)

ดูว่าเราสามารถทำ derivation ให้เกิดสตริง (1 + 2 + (3 + 4)) + 5 ได้หรือไม่ จะเห็นได้ว่าเราสามารถทำได้ทั้งจากการให้ลำดับของการ expand เริ่มจากด้านซ้ายหรือจากด้านขวาดังแสดงต่อไปนี้



ทั้งสองแบบให้ผลลัพธ์ที่เมื่อแปลงเป็น parse tree แล้ว มีรูปแบบเดียวกันตามที่แสดงในรูปกลาง และมีรูปแบบ abstract tree ที่เหมือนกันตามรูปขวามือสุด เรียก grammar ประเภทนี้ว่า unambiguous grammar โดยในการ specify ภาษาระดับสูงนั้นเราจะต้องใช้ grammar ในลักษณะนี้

Ambiguous Grammar

พิจารณา grammar ต่อไปนี้ :

S -> S + S | S * S | number

และลองพิจารณาการ derive expression ต่อไปนี้ 1 + 2 * 3

- Derivation 1: $\boldsymbol{S} \rightarrow \boldsymbol{S} + S \rightarrow 1 + \boldsymbol{S} \rightarrow 1 + \boldsymbol{S} * S \rightarrow 1 + 2 * \boldsymbol{S} \rightarrow 1 + 2 * 3$
- Derivation 2: $\boldsymbol{S} \rightarrow S * \boldsymbol{S} \rightarrow \boldsymbol{S} * 3 \rightarrow S + \boldsymbol{S} * 3 \rightarrow \boldsymbol{S} + 2 * 3 \rightarrow 1 + 2 * 3$

จะเห็นได้ว่าการทำ derivation ทั้งสองแบบจะให้ผลลัพธ์ที่เป็น parse tree และ abstract tree ที่แตกต่างกัน โดย abstract tree ที่ได้จาก derivation ที่ 1 เป็นไปตามรูปซ้ายมือด้านล่าง ส่วน abstract tree ที่ได้จากการทำ derivation ที่ 2 เป็นไปตามรูปขวามือด้านล่าง

ซึ่งถ้าเราจะนำ abstract tree ทั้งสองนี้ไปหาค่า เราจะได้ค่าที่แตกต่างกัน grammar แบบ ambiguous เป็น grammar ที่เราต้องหลีกเลี่ยงในการ specify ภาษาระดับสูง เพราะจะทำให้เกิดการตีความความหมายของโปรแกรมเป็นไปได้มากกว่าหนึ่งแบบ ขาดคุณสมบัติของ formal language ไป

โดยทั่วไปการกำจัด ambiguity ใน grammar ไม่มีเทคนิคที่ทำให้เราสามารถกระทำการนี้ได้โดยอัติโนมัติ ต้องอาศัยการพิจารณาโดยใช้คน (เช่นโปรแกรมเมอร์ หรือ นักคณิตศาสตร์) เขียน ambiguous grammar ในรูปแบบใหม่ที่ตัด ambiguity ออกไป ซึ่งโดยส่วนใหญ่สามารถทำได้โดยการเพิ่ม non-terminal เข้ามาและอนุญาตให้มี recursion ได้เฉพาะด้านซ้ายหรือด้านขวาเท่านั้น ตัวอย่างเช่น grammar ในรูปแบบ

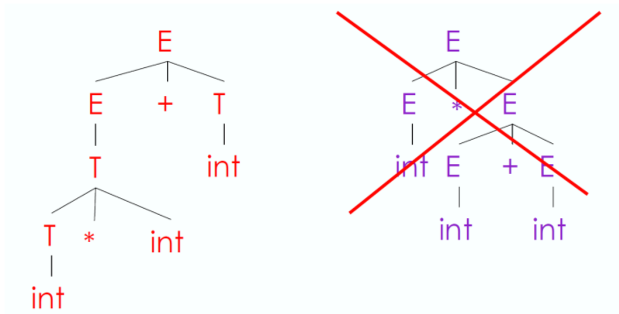
E -> E + E | E * E | number

ด้านบน เราสามารถกำจัด ambiguity ได้โดยการเขียน grammar ใหม่ดังนี้:

E -> E + T | T

T -> T *int | int

grammar ที่เขียนใหม่นี้จะบังคับให้ * มี precedence มากกว่า + และบังคับ associativity ทางด้านซ้ายของทั้ง * และ + ในการทำ derivation ของสตริงในรูปแบบ int * int + int นั้น เราสามารถสร้าง parse tree ได้เพียงรูปแบบเดียวดังต่อไปนี้



ในกรณีที่เราต้องการบังคับ right-associativity เราจะให้ recursion เกิดขึ้นทางขวาแทนดัง grammar ที่เขียนใหม่ต่อไปนี้:

E -> T + E | T

T -> int * T | int

CFG ของ expression ที่เราใช้มีรูปแบบสัญลักษณ์แบบ EBNF ซึ่งใช้สัญลักษณ์ที่มีใช้ใน RE เข้าไปร่วมด้วย ตัวอย่าง grammar:

E -> E + T | T

T -> T * int | int

เราสามารถเขียนในแบบ EBNF ได้เป็น

E -> T (+ T)*

T -> F (* F)*

F-> int

โดยถ้าเราใช้สัญลักษณ์ตามตารางในเลคเชอร์สัปดาห์ที่สองเราจะเขียนได้ดังนี้

E -> T, {+ T };

T -> F, {* F};

F-> int;

โดยรูปแบบ EBNF มีความสามารถเท่ากับรูปแบบเดิม โดยจะเห็นได้ว่าถ้าเรากระจายกฎ E -> E + T | T ไปอย่างต่อเนื่องเราจะได้ E -> E + T + T + T ... + T โดยสุดท้ายเราจะ terminate ที่ T และได้แบบเหมือนในรูป EBNF

ประวัติย่อของการนำ CFG มาใช้ในการระบุโครงสร้างภาษาโปรแกรม

1952 – Grace Hopper เขียนคอมไพเลอร์ตัวแรกสำหรับภาษา A-0

1957 – กำเนิดภาษา FORTRAN โดยทีมของ IBM มี John Backus เป็นผู้นำ มีคอมไพเลอร์ที่สมบูรณ์มากกว่าของ Grace แต่โครงสร้างภาษา FORTRAN ยังคงอธิบายโดยใช้ภาษาอังกฤษเป็นหลัก

1960 – ภาษา ALGOL 60 เป็นภาษาแรกที่ระบุโครงสร้างภาษาโดยใช้ CFG โดย John Backus เป็นผู้บุกเบิก specification ของภาษาในรูปแบบนี้ ตัวทฤษฎีของ CFG จริงๆถูกพัฒนาโดย Noam Chomsky ในช่วงประมาณปี 1954-55

1960 – 1963 Peter Naur มีส่วนร่วมในการเขียน specification ของภาษา ALGOL 60 โดยที่รูปแบบของ CFG ที่ใช้ถูกเรียกว่า BNF หรือ Backus Normal Form

1964 – Donal Knuth เขียนจดหมายถึง ACM (Association of Computing Machinery) บอกว่า BNF น่าจะแทน Backus Naur Form เพราะไม่ได้มีความเป็น “Normal” ที่แท้จริงอยู่ และการให้ N แทน Naur จะได้เป็นการให้การยอมรับ Naur ว่าเป็นผู้มีส่วนร่วมพอๆกับ Backus