บทที่ 2

==========

ไวยากรณ์ของภาษาการทำโปรแกรม
(Syntax of programming languages)

วัตถุประสงค์

1) เพื่อให้ผู้เรียนเข้าใจถึงแนวทางและความสำคัญของการออกแบบภาษาคอมพิวเตอร์
2) เพื่อให้ผู้เรียนรู้จักรูปแบบมาตรฐานที่ใช้ในการอธิบายไวยากรณ์ของภาษา
3) เพื่อให้ผู้เรียนสามารถอ่านและวิเคราะห์ไวยากรณ์ของภาษา
4) เพื่อให้ผู้เรียนสามารถแยกแยะความแตกต่างระหว่างไวยากรณ์ที่กำกวมกับไวยากรณ์ที่ไม่กำกวม
5) เพื่อให้ผู้เรียนรู้จักการจัดลำดับชั้นของภาษาตามรูปแบบของชอมสกี

ไวยากรณ์ทำหน้าที่กำหนดรูปแบบและวิธีการเขียนประโยคต่างๆ ที่ต้องใช้ในการสร้างคำสั่งเพื่อประกอบเป็นโปรแกรม การออกแบบภาษาจึงต้องเริ่มต้นจากการออกแบบรูปแบบของคำไปจนถึงการออกแบบข้อกำหนดโครงสร้างของคำสั่งประเภทต่างๆ ไวยากรณ์ที่ดีจะต้องมีความสอดคล้องระหว่างรูปแบบกับความหมายและจะต้องจดจำทำความเข้าใจได้ง่าย วิธีการอธิบายไวยากรณ์ของภาษาคอมพิวเตอร์นิยมใช้วิธีการที่เป็นทางการด้วยรูปแบบมาตรฐานเช่น BNF, EBNF ผู้ที่ต้องการศึกษาเกี่ยวกับภาษาคอมพิวเตอร์จึงจำเป็นต้องเรียนรู้เกี่ยวกับรูปแบบมาตรฐานที่ใช้อธิบายไวยากรณ์ของภาษา เพื่อให้สามารถอ่านและวิเคราะห์ไวยากรณ์ซึ่งจะนำไปสู่การเขียนโปรแกรมที่ถูกต้อง

2.1 เกณฑ์การออกแบบไวยากรณ์ของภาษา

(Syntactic design criteria)

ภาษาเป็นเครื่องมือที่โปรแกรมเมอร์ใช้ในการเขียนคำสั่งต่างๆ เพื่อประกอบขึ้นเป็นโปรแกรม โดยโปรแกรมจะทำหน้าที่ควบคุมการทำงานของเครื่องคอมพิวเตอร์ ดังนั้นไวยากรณ์ของภาษาจะต้องเอื้อให้โปรแกรมเมอร์สามารถใช้เครื่องมือเพื่อพัฒนาโปรแกรมได้อย่างสะดวก นั่นคือโปรแกรมจะต้องอ่านง่าย เขียนได้ง่าย ตรวจสอบแก้ไขได้ง่าย แปลงเป็นภาษาเครื่องได้ง่ายและไม่กำกวม

- **การอ่านได้ง่าย (readability)**

  โปรแกรมที่อ่านทำความเข้าใจได้โดยง่าย เกิดจากปัจจัยสำคัญสองประการคือสไตล์การเขียนโปรแกรมของโปรแกรมเมอร์และไวยากรณ์ของภาษา โปรแกรมเมอร์ที่ดีจะใช้รูปแบบการเขียนโปรแกรมที่เป็นโครงสร้างชัดเจน และรูปแบบไวยากรณ์ของภาษาที่ดีจะเอื้อต่อการเขียนโปรแกรมให้สามารถอ่านได้ง่าย ตัวอย่างของไวยากรณ์ภาษาที่ดีได้แก่เป็นภาษาที่อนุญาตให้มีการเขียนข้อความหมายเหตุ (comment) เพื่ออธิบายโปรแกรม, อนุญาตให้ตั้งชื่อตัวแปรและชื่อฟังก์ชันที่ยาวเพียงพอที่จะสื่อความหมายได้ชัดเจน, มีรูปแบบคำสั่งที่สื่อความหมาย, ตรรกะของคำสั่งสอดคล้องกับตรรกะการคิดโดยธรรมชาติของมนุษย์ ภาษา COBOL เป็นตัวอย่างภาษาคอมพิวเตอร์ที่ออกแบบมาโดยเน้นความสามารถด้านการอ่านได้ง่าย (แต่โปรแกรมเมอร์จะต้องเขียนคำสั่งที่ยาวมากเพื่อให้สื่อความหมาย) ในขณะที่ภาษา APL มีลักษณะที่ตรงกันข้าม นั่นคือ ไวยากรณ์มีรูปแบบที่สั้นมาก ต้องเป็นผู้เชี่ยวชาญเท่านั้นจึงจะอ่านคำสั่งเข้าใจ (แต่โปรแกรมเมอร์ที่เชี่ยวชาญในภาษาจะเขียนคำสั่งได้เร็ว เพราะรูปแบบคำสั่งสั้นมาก)

- **การเขียนได้ง่าย (writeability)**

  ลักษณะการเขียนโปรแกรมได้ง่าย จะหมายถึงการที่โปรแกรมเมอร์ใช้เวลาสั้นลงในการเขียนชุดคำสั่งเพื่อประกอบขึ้นเป็นโปรแกรม อันเนื่องจากรูปแบบคำสั่งกระทัดรัด และสามารถรวมการทำงานหลายอย่างให้อยู่ในคำสั่งเดียว นอกจากนี้ภาษายังต้องมีคำสั่งพื้นฐานที่จำเป็นเพียงพอต่อการเขียนโปรแกรม ภาษา C และภาษา APL จัดเป็นภาษาที่มีรูปแบบคำสั่งสั้น โปรแกรมเมอร์ที่ชำนาญสามารถเขียน

โปรแกรมได้โดยง่าย แต่ลักษณะการเขียนโปรแกรมได้ง่ายนี้มักจะขัดแย้งกับลักษณะการอ่านได้ง่าย โปรแกรมที่เขียนง่าย สั้นกระทัดรัดมักจะอ่านยาก

- **การตรวจสอบแก้ไขได้ง่าย (ease of verifiability)**

  ภาษาที่ดีควรจะมีไวยากรณ์และโครงสร้างคำสั่ง ที่เอื้อต่อการตรวจสอบความถูกต้องของโปรแกรม นั่นคือคำสั่งจะต้องสื่อความหมาย มีลักษณะบังคับการเกาะกลุ่มที่ชัดเจน ไม่อนุญาตให้ใช้ GOTO ที่สามารถกระโดดไปตำแหน่งใดของโปรแกรมก็ได้

- **การแปลงเป็นภาษาเครื่องได้ง่าย (ease of translation)**

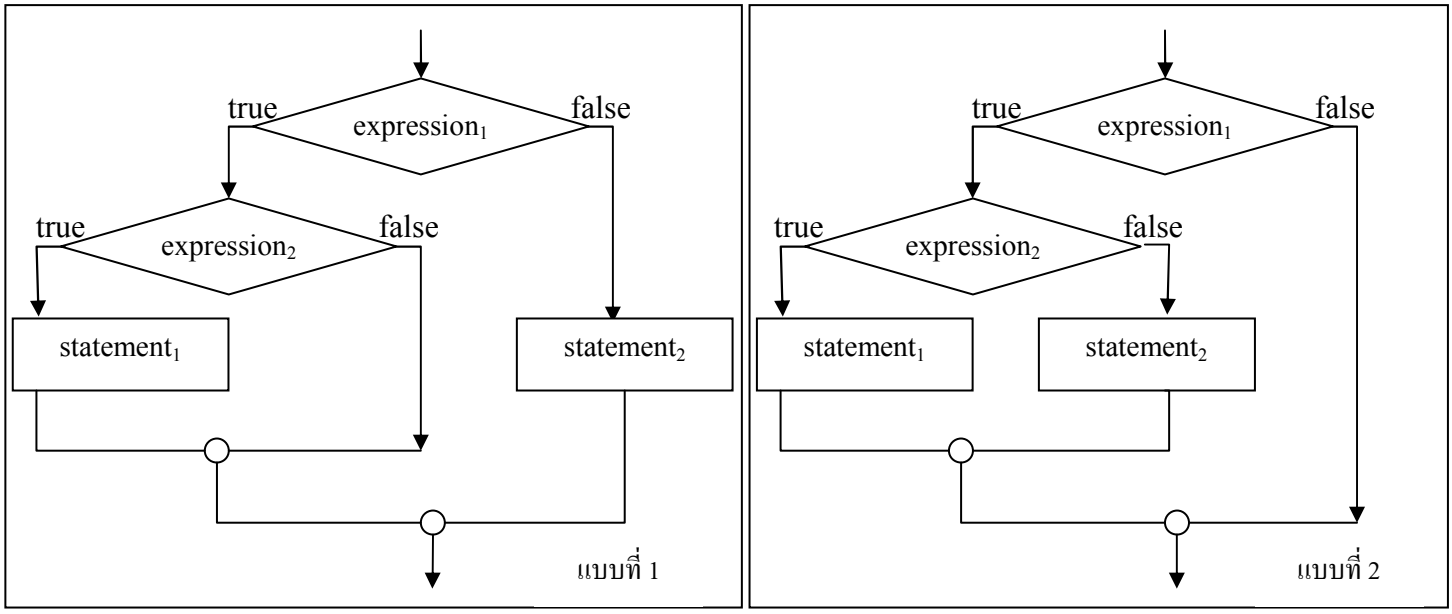
  ลักษณะการอ่านได้ง่าย เขียนได้ง่าย ตรวจสอบแก้ไขได้ง่าย เป็นเกณฑ์ที่ช่วยให้คนสามารถพัฒนาและตรวจสอบแก้ไขซอฟต์แวร์ได้สะดวก ในขณะที่ลักษณะการแปลงเป็นภาษาเครื่องได้ง่ายอำนวยความสะดวกให้เครื่องคอมพิวเตอร์แปลคำสั่งและประมวลผลได้รวดเร็ว ภาษา LISP เป็นตัวอย่างภาษาระดับสูงที่แปลคำสั่งได้ง่าย เนื่องจากโครงสร้างโปรแกรมและข้อมูลเป็นโครงสร้างเดียวกันและไวยากรณ์ของภาษาไม่ซับซ้อน

- **ไม่กำกวม (lack of ambiguity)**

  ความกำกวมของภาษาจะเกิดขึ้นเมื่อการเขียนคำสั่งหนึ่งคำสั่งสามารถแปลความหมายได้มากกว่าหนึ่งแบบ ตัวอย่างเช่น คำสั่งแบบมีเงื่อนไขที่เขียนซ้อนคำสั่งดังต่อไปนี้

```
if expression1 then
     if expression2
          then statement1
               else statement2
```

สามารถแปลความหมายได้สองรูปแบบดังแผนภาพในรูปที่ 2.1



รูปที่ 2.1 ความหมายสองรูปแบบของคำสั่งแบบมีเงื่อนไข

ภาษาที่ดีจะต้องมีการป้องกันกรณีกำกวมดังตัวอย่างข้างต้น ภาษา C และ Pascal แก้ปัญหาด้วยการมีข้อกำหนดว่าคำสั่ง else จะหมายถึง else ของคำสั่ง if..then ที่อยู่ใกล้ที่สุด (นั่นคือมีความหมายเหมือนแบบที่ 2 ในรูปที่ 2.1)

2.2 ส่วนประกอบพื้นฐานของไวยากรณ์ภาษา

(Syntactic elements of a language)

ในการออกแบบภาษาคอมพิวเตอร์ ผู้ออกแบบมักจะต้องสร้างข้อกำหนดเกี่ยวกับส่วนประกอบพื้นฐานต่างๆ ของภาษาดังต่อไปนี้

- เซตของอักขระ (character set)

  ข้อกำหนดเบื้องต้นของภาษามักจะเป็นการกำหนดเซตของอักขระที่สามารถใช้ตั้งชื่อต่างๆ ในโปรแกรมของภาษานั้นๆ โดยทั่วไปอักขระที่ใช้มักจะเป็นอักขระพื้นฐานของแอสกี (ASCII) ดังเช่นที่ปรากฏในภาษา C แต่อาจจะมีบางภาษาที่มีข้อกำหนดต่างออกไป เช่นภาษา APL สามารถใช้อักขระพิเศษเช่น ⊥ ที่ไม่ปรากฏบนแป้นพิมพ์คอมพิวเตอร์ปกติ การเขียนโปรแกรมภาษานี้จึงจำเป็นต้องมีแป้นพิมพ์เป็นการเฉพาะ (รูปที่ 2.2)

รูปที่ 2.2 แป้นพิมพ์พิเศษที่ใช้สำหรับพิมพ์คำสั่งในภาษา APL

- ตัวระบุ (identifiers)

  การตั้งชื่อตัวระบุ เช่น ชื่อตัวแปร มักจะมีข้อกำหนดพื้นฐานคล้ายกันคือ ชื่อจะประกอบด้วยตัวอักษร และ/หรือ ตัวเลข แต่ต้องเริ่มต้นชื่อด้วยตัวอักษร ในหลายๆ ภาษาจะอนุญาตให้ใช้เครื่องหมายขีดล่าง (underscore) ในการเชื่อมคำหลายคำให้เป็นหนึ่งชื่อที่สื่อความหมาย

- สัญลักษณ์ตัวดำเนินการ (operator symbols)

  ผู้ออกแบบภาษาจะต้องกำหนดสัญลักษณ์ที่ใช้ดำเนินการทางคณิตศาสตร์ ทางตรรกะ ทางการเปรียบเทียบ ทางการดำเนินงานกับข้อความ รวมทั้งสร้างข้อกำหนดเกี่ยวกับลำดับความสัมพันธ์ ก่อน-

หลังในการประมวลผล (precedence) ในกรณีที่คำสั่งมีเครื่องหมายดำเนินการหลายชนิดปะปนกัน เช่น

```
X = A + B * C / D
```

หรือในกรณีที่มีการใช้เครื่องหมายเดียวกันหลายครั้ง จะต้องกำหนดการทำงานจากขวาไปซ้าย (right associative) หรือจากซ้ายไปขวา (left associativity) เช่น

```
Y = 10 / 2 / 3
```

- คำหลักและคำสงวน (keywords and reserved words)

  คำหลักเป็นตัวระบุที่กำหนดความหมายไว้เป็นการเฉพาะในภาษานั้นๆ เช่น ภาษา C กำหนดว่า if เป็นคำหลักที่ใช้เริ่มต้นข้อความสั่งแบบมีเงื่อนไข หรือ for ใช้สำหรับสั่งงานแบบวนรอบ เป็นต้น คำหลักจะถูกเรียกว่าคำสงวน เมื่อภาษานั้นๆ ไม่อนุญาตให้โปรแกรมเมอร์ นำคำหลักไปตั้งเป็นชื่อตัวแปรในโปรแกรม ภาษารุ่นใหม่ที่นิยมใช้อยู่ในปัจจุบันคำหลักต่างๆจะถูกกันไว้เป็นคำสงวนไม่สามารถนำไปตั้งเป็นชื่อตัวแปรหรือชื่ออื่นๆ ในโปรแกรมได้ ทั้งนี้เพื่อป้องกันข้อผิดพลาดที่เกี่ยวกับชื่อ แต่ในภาษารุ่นเก่าเช่น ฟอร์แทรน อนุญาตให้ใช้คำหลัก ตั้งเป็นชื่อตัวแปรได้ ตัวอย่างเช่น

```
1     REAL  MYVAR
2     REAL  REAL
3     MYVAR = 0.0
4     REAL = 3.4
```

  คำสั่งในบรรทัดที่ 1 คำว่า REAL เป็นคำหลักหมายถึงชนิดข้อมูลเลขจำนวนจริง ในขณะที่บรรทัดที่ 4 คำว่า REAL เป็นชื่อตัวแปร

- หมายเหตุ (comments)

  ข้อความในส่วนหมายเหตุไม่มีผลต่อการทำงานของโปรแกรม แต่ผู้เขียนโปรแกรมมักจะใส่ข้อความหมายเหตุไว้ในโปรแกรม เพื่ออธิบายส่วนประกอบสำคัญของโปรแกรม หรือเพื่ออธิบายว่าโปรแกรมนั้นๆ เขียนขึ้นเพื่อทำงานอะไร ทั้งนี้เพื่อใช้สื่อสารกับโปรแกรมเมอร์อื่นในทีม หรือเพื่อให้ผู้อื่นที่จะต้องมาปรับปรุงโปรแกรมในภายหลัง ทำความเข้าใจโปรแกรมได้ง่ายขึ้น รูปแบบหมายเหตุมักจะนิยมใช้ใน 3 ลักษณะคือ

  (1) แยกเป็นหนึ่งบรรทัดต่างหาก เช่นข้อความ REM ในภาษา BASIC

```
10    REM   This is a sample program
20    DIM   A(11)
30    FOR   I = 1 TO 11
40    INPUT A(I)
50    NEXT  I
60    END
```

(2) ปรากฏที่ส่วนใดของบรรทัดก็ได้ และจะสิ้นสุดหมายเหตุที่ท้ายบรรทัด เช่น การใช้เครื่องหมาย // ในภาษา C++ และ Java

```
double     A[11];
for  (int i=0; i<11; i++) {   // read the values of A
     cin >> A[i];
}
```

(3) ใช้สัญลักษณ์เริ่มต้นและสิ้นสุดหมายเหตุ เช่นการใช้เครื่องหมาย /* และ */ ในภาษา C และ Pascal

```
double A[11];
int i;

/* read in the values of the array A */
for (i=0; i<11; i++) {
     scanf ("%f", &A[i]);
}
```

- อักขระคั่นและสัญลักษณ์จัดกลุ่ม (delimiters and brackets)

  การออกแบบภาษามักจะต้องกำหนดอักขระพิเศษเพื่อบอกจุดจบหรือบอกจุดเริ่มต้นของข้อความ หรือนิพจน์ เช่นในภาษา C กำหนดจุดจบของคำสั่งด้วยเครื่องหมาย ; อักขระพิเศษนี้เรียกว่าอักขระคั่น ส่วนสัญลักษณ์จัดกลุ่มจะใช้ในการจัดกลุ่มนิพจน์ หรือกลุ่มของคำสั่ง อักขระที่ใช้จะเป็นอักขระคู่ เช่น ใช้วงเล็บ (…), {…}, หรือใช้ข้อความที่เป็นคู่ เช่น `begin…end`

- นิพจน์ (expressions)

  นิพจน์เป็นส่วนประกอบของคำสั่งที่สามารถถูกประมวลผล แล้วส่งผลให้โปรแกรมเปลี่ยนสถานะไปได้ ผู้ออกแบบภาษาจะต้องกำหนดไวยากรณ์ของการเขียนนิพจน์ รวมถึงกำหนดลำดับการทำงานเพื่อประมวลผลนิพจน์

- ข้อความสั่ง (statements)

  ข้อความสั่งเป็นส่วนประกอบสำคัญของภาษาเพื่อใช้ควบคุมลำดับการทำงานของโปรแกรม การออกแบบภาษาจะต้องกำหนดข้อความสั่งประเภทต่างๆ เช่น ข้อความสั่งกำหนดค่า ข้อความสั่งแบบมีเงื่อนไข ข้อความสั่งแบบวนรอบ รวมถึงข้อความเรียกใช้โปรแกรมย่อย ผู้ออกแบบภาษาต้องกำหนดไวยากรณ์ของคำสั่งที่เข้าใจได้ง่าย แต่มีประสิทธิภาพในการทำงาน

2.3 รูปแบบทางการในการกำหนดไวยากรณ์ของภาษา

(Formal methods to syntax definition)

ภาษาที่เราสนใจศึกษาหรือเป็นภาษาที่เรากำลังออกแบบสร้างขึ้น จะเรียกว่า *ภาษาเป้าหมาย* (object language) แต่ในการอธิบายโครงสร้างของภาษาเป้าหมายเราจะใช้ภาษาที่เรียกว่า *ภาษาเมตา* หรือ อภิภาษา (metalanguage) ซึ่งมีความหมายว่าเป็นภาษาที่ใช้อธิบายอีกภาษาหนึ่ง ตัวอย่างเช่น ในการเรียนภาษาญี่ปุ่น ภาษาเป้าหมายที่ผู้เรียนสนใจเรียนคือภาษาญี่ปุ่น แต่เพื่อสอนให้กับผู้เริ่มต้นให้สามารถเข้าใจได้โดยรวดเร็ว อาจารย์ผู้สอนจำเป็นต้องอธิบายโครงสร้างและคำแปลศัพท์ด้วยภาษาไทย ในกรณีนี้ภาษาไทยจะทำหน้าที่เป็น ภาษาเมตา

ในการอธิบายภาษาคอมพิวเตอร์นิยมใช้รูปแบบ *BNF (Backus-Naur Form)* เป็นภาษาเมตา ซึ่งต่อมามีการขยายข้อกำหนดของรูปแบบให้เขียนไวยากรณ์ได้ง่ายขึ้น ทำให้เกิดเป็นรูปแบบที่เรียกว่า *EBNF (Extended BNF)* ในภาษาคอมพิวเตอร์รุ่นเก่าบางภาษา เช่น ภาษา Pascal ใช้แผนภาพเป็นภาษาเมตาเพื่ออธิบายไวยากรณ์ของภาษา Pascal รูปแบบแผนภาพนี้เรียกว่า *แผนภูมิไวยากรณ์ (syntax chart)* หรือ ไดอะแกรมไวยากรณ์ (syntax diagram)

<u>รูปแบบ BNF</u>

รูปแบบบีเอ็นเอฟ พัฒนาขึ้นโดย John Backus และ Peter Naur ในปี ค.ศ. 1960 เพื่อใช้เป็น ภาษาเมตาในการอธิบายไวยากรณ์ของภาษา ALGOL คำว่า BNF จึงหมายถึง Backus-Naur Form หรือในบางครั้งหมายถึง Backus Normal Form รูปแบบ BNF พัฒนาจากทฤษฎีภาษาฟอร์มอล หรือภาษาที่เป็นทางการ ซึ่งเสนอแนวทางไว้โดยนักภาษาศาสตร์ชื่อ Noam Chomsky

รูปแบบ BNF เป็นรูปแบบอย่างเป็นทางการที่สร้างขึ้นเพื่อใช้อธิบายไวยากรณ์ของภาษา เรียกว่า ไวยากรณ์บีเอ็นเอฟ (BNF grammar) ไวยากรณ์บีเอ็นเอฟ ประกอบด้วยส่วนประกอบสี่ส่วนคือ <T, N, S, P> โดยที่

T คือ เซตของสัญลักษณ์ปลายทาง หรือเรียกว่าสัญลักษณ์เทอร์มินอล (terminal symbols)

N คือ เซตของสัญลักษณ์นอนเทอร์มินอล (nonterminal symbols)
ที่ไม่ซ้ำกับสัญลักษณ์เทอร์มินอล นั่นคือ $N \cap T = \emptyset$

S คือ สัญลักษณ์นอนเทอร์มินอลที่ใช้เป็นสัญลักษณ์เริ่มต้น (start symbol)

P คือ เซตของกฎที่เรียกว่า โพรดักชัน (production) ใช้อธิบายส่วนประกอบเชิงไวยากรณ์ต่างๆ ของภาษา เช่น ชื่อตัวแปร, จำนวนเลข, นิพจน์, คำสั่งวนรอบ, โปรแกรม

การเขียนโพรดักชันจะใช้รูปแบบ

$$A \rightarrow \omega$$

เมื่อ A เป็นสัญลักษณ์ในเซตของสัญลักษณ์นอนเทอร์มินอล (N)

ω เป็นสัญลักษณ์จากเซตของ N หรือ T หรือ อักขระว่าง

ตัวอย่างเช่น

```
BinaryDigit → 0
BinaryDigit → 1
```

เป็นการเขียนโพรดักชันอธิบายว่าเลขฐานสอง (BinaryDigit) ประกอบด้วยเลข 0 หรือ 1 ในตัวอย่างนี้ BinaryDigit คือสัญลักษณ์นอนเทอร์มินอล ตัวเลข 0 และ 1 คือ สัญลักษณ์เทอร์มินอล โพรดักชันทั้งสองบรรทัดข้างต้นนี้อธิบายความหมายของ BinaryDigit จึงอาจเขียนรวมกันได้ดังนี้

```
BinaryDigit → 0 | 1
```

เครื่องหมาย " | " แทนความหมาย "หรือ" ตัวอย่างข้างต้นจึงเป็นโพรดักชันที่อธิบายว่า BinaryDigit คือ เลข 0 หรือ เลข 1

การกำหนดไวยากรณ์ของจำนวนเลขฐานสิบที่มีได้หลายหลัก สามารถเขียนโพรดักชันได้ดังนี้

```
Integer    → Digit | Integer Digit
Digit      → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

โพรดักชันแรกอธิบายว่า Integer คือ Digit หรือคือ Integer ที่ตามด้วย Digit ซึ่งเป็นการนิยามแบบเรียกตัวเองซ้ำ (recursive) นั่นคือ นิยาม Integer ด้วย Integer เป็นการนิยามที่ใช้ในกรณีต้องการเขียนจำนวนตัวเลขที่ประกอบด้วยเลขหลายๆ หลัก ถ้าเป็นเลขหลักเดียว จะนิยามด้วย Digit ในโพรดักชันที่สอง

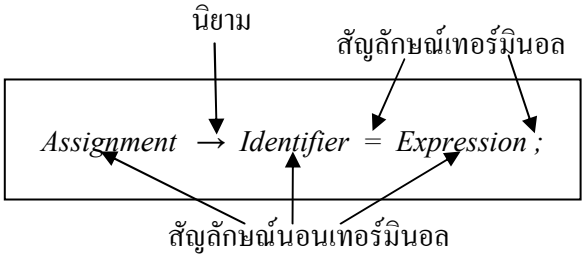
ในการอธิบายโครงสร้างที่ซับซ้อนขึ้น เช่น การอธิบายไวยากรณ์ของการเขียนตัวแปรหรือตัวระบุ (identifier) และคำหลัก (keyword) นิยามในรูปแบบ BNF ได้ดังนี้

```
Identifier → Letter | Identifier Letter | Identifier Digit
Letter     → a | b | ... | z | A | B | ... | Z
Digit      → 0 | 1 | ... | 9
Keyword    → boolean | if | else | void | while | main
```

ตัวอย่างของการนิยามคำสั่งกำหนดค่า (assignment statement) และรูปแบบการเขียนนิพจน์ (expression) แสดงได้ดังต่อ ไปนี้

```
Assignment → Identifier = Expression ;
Expression → Term | Expression + Term |  Expression - Term
Term       → Factor | Term * Factor | Term / Factor
Factor     → Identifier | Literal | (Expression)
Literal    → Boolean | Integer
Boolean    → true | false
```

จะสังเกตได้ว่าการนิยามไวยากรณ์ในรูปแบบ BNF เขียนสัญลักษณ์เทอร์มินอลด้วยตัวพิมพ์เล็ก และเขียนสัญลักษณ์นอนเทอร์มินอล เริ่มต้นคำด้วยตัวพิมพ์ใหญ่ เช่น ในตัวอย่างการนิยามคำสั่งกำหนดค่า



จากนิยามข้างต้นทำให้เราทราบว่าสามารถเขียนคำสั่งกำหนดค่าต่อไปนี้ได้

```
X = 3 * ( 2 + 4 );
SUM = X - 10;
```

การเขียนไวยากรณ์ในรูปแบบ BNF บางครั้งเพื่อให้ไวยากรณ์อ่านง่ายขึ้น ผู้เขียนจะเขียนสัญลักษณ์นอนเทอร์มินอลอยู่ภายในเครื่องหมาย <…> เพื่อแยกความแตกต่างจากสัญลักษณ์เทอร์มินอล และอาจแทนเครื่องหมายนิยาม (→) ด้วยสัญลักษณ์ ::= ดังนั้นไวยากรณ์ในตัวอย่างข้างต้นอาจเขียนได้ใหม่ดังนี้

```
<assignment_statement> ::= <identifier>  =  <expression> ;
<expression>     ::=  <term>
                      | <expression> + <term>
                      |  <expression> - <term>
<term>           ::= <factor>
                      | <term> * <factor>
                      | <term> / <factor>
<factor>         ::= <identifier>
                      | <literal>
                      | (<expression>)
<identifier>     ::= <literal>
                      | <identifier> <literal>
                      |<identifier> <digit>
<literal>        ::= <boolean>
                      | <integer>
<boolean>        ::= <digit>
                      | <integer> <digit>
<digit>          ::= 0 | 1 | … | 9
```

<u>รูปแบบ EBNF</u>

นับตั้งแต่เริ่มมีการใช้รูปแบบ BNF เพื่อนิยามไวยากรณ์ของภาษา ALGOL รูปแบบ BNF ก็ได้รับความนิยมใช้เป็นภาษาเมตา เพื่ออธิบายภาษาคอมพิวเตอร์ระดับสูงอื่นๆ และเริ่มมีการปรับปรุงรูปแบบ BNF ให้สามารถเขียนได้ง่ายขึ้น รูปแบบปรับปรุงนี้เรียกว่า Extended BNF หรือเรียกโดยย่อว่ารูปแบบ EBNF ประเด็นสำคัญของการปรับปรุงคือ หลีกเลี่ยงการนิยามแบบ recursive โดยใช้เครื่องหมายบอกการเกิดซ้ำดังนี้

{…}* หมายถึง สิ่งที่อยู่ภายในวงเล็บเกิดขึ้นซ้ำได้ 0 ครั้งหรือ 1 ครั้งหรือมากกว่า 1 ครั้ง

{…}+ หมายถึง สิ่งที่อยู่ภายในวงเล็บเกิดขึ้นซ้ำได้ 1 ครั้งหรือมากกว่า 1 ครั้ง

นอกจากนี้มีการใช้เครื่องหมายการเลือก (option) ดังนี้

[+ | –] หมายถึง เลือกใช้เครื่องหมายบวก หรือ เครื่องหมายลบ

ตัวอย่างเช่น จากนิยามการเขียนนิพจน์ ที่กำหนดในรูปแบบ BNF

```
Expression → Term | Expression + Term | Expression - Term
Term       → Factor | Term * Factor | Term / Factor
```

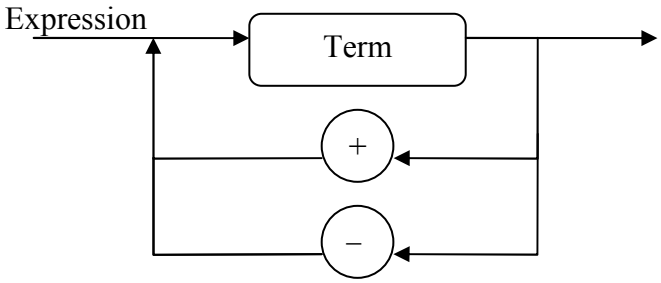
นิยามข้างต้นเป็นการอธิบายว่านิพจน์ คือ เทอมเดี่ยวๆ เพียงเทอมเดียว (เช่น Y, 345) หรือ เป็นเทอมที่ซับซ้อนขึ้นโดยนำหลายเทอมมาบวก ลบ คูณ หาร กันได้ (เช่น Y+345, Z*4/(3+12)-4) ซึ่งนิยามข้างต้นสามารถเขียนในรูปแบบของ EBNF ได้โดยไม่ต้องใช้ recursion ดังนี้

```
Expression → Term { [+ | -] Term }*
Term       → Factor { ['*' | /]Factor }*
```

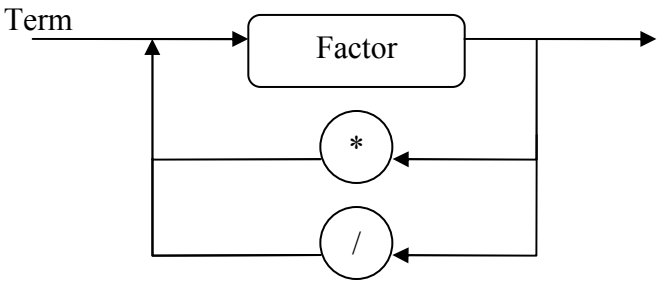
นิยามในบรรทัดแรกอธิบายว่านิพจน์ คือ เทอมหนึ่งเทอม (นั่นคือส่วนภายในวงเล็บ {…} เกิดขึ้นศูนย์ครั้ง) หรือ เทอมหลายเทอมมาบวกหรือลบกัน (นิยามด้วย [+ | -]) บรรทัดที่สองนิยามว่า เทอมคือ แฟคเตอร์หนึ่งแฟคเตอร์ หรือ แฟคเตอร์หลายแฟคเตอร์มาคูณหรือหารกัน สังเกตว่าเครื่องหมายคูณเขียนอยู่ภายในเครื่องหมายคำพูดขีดเดียว ('*') เพื่อบอกให้รู้ว่าเครื่องหมายนี้เป็นสัญลักษณ์เทอร์มินอลที่สามารถปรากฏอยู่ในภาษาเป้าหมาย ทั้งนี้เพื่อแยกความแตกต่างจากเครื่องหมาย {…}* เพื่อบอกการเกิดซ้ำศูนย์ครั้งหรือมากกว่านั้น ซึ่งเครื่องหมาย * ในกรณีหลังนี้เป็นสัญลักษณ์ที่ใช้สื่อสารในภาษาเมตา

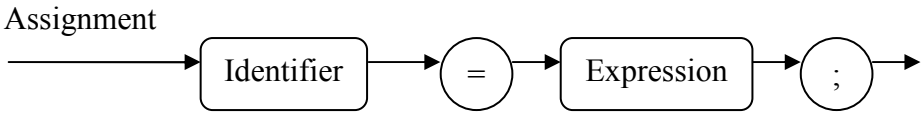
<u>แผนภูมิไวยากรณ์</u>

การอธิบายไวยากรณ์ของภาษาคอมพิวเตอร์ด้วยแผนภูมิไวยากรณ์ (syntax chart) เริ่มใช้ในการอธิบายภาษา Pascal โดยใช้รูปภาพอธิบายโครงสร้างไวยากรณ์ของภาษา ดังตัวอย่างต่อไปนี้



การอ่านแผนภูมิจะเริ่มจากซ้ายไปขวา ซ้ายมือจะมีคำอธิบายกำกับว่าเป็นนิยามของ Expression ทิศทางการอ่านจะกำกับด้วยหัวลูกศร เช่น Expression ประกอบด้วย Term หรือ Term + Term หรือ Term – Term หรือ Term + Term – Term หรือ… สัญลักษณ์นอนเทอร์มินอลจะเขียนอยู่ในกรอบรูปสี่เหลี่ยมผืนผ้า หรือรูปวงรี ส่วนสัญลักษณ์เทอร์มินอลจะเขียนอยู่ภายในรูปวงกลม ตัวอย่างการอธิบายเทอม (Term) และข้อความสั่งกำหนดค่า (Assignment) ด้วยแผนภูมิไวยากรณ์ แสดงได้ดังนี้

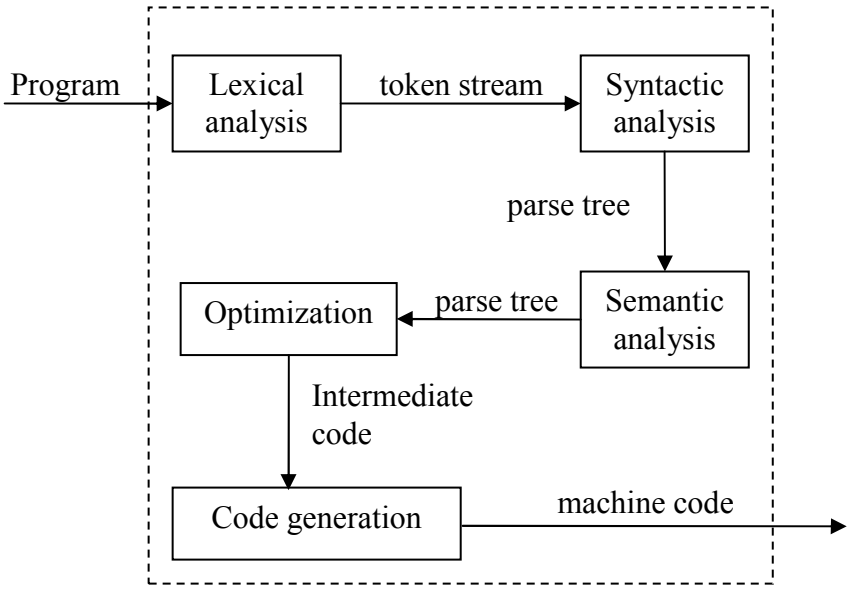




2.4 การวิเคราะห์ไวยากรณ์

(Syntactic analysis)

ไวยากรณ์ที่เขียนอยู่ในรูปแบบ BNF, EBNF หรือ แผนภูมิไวยากรณ์ ทำหน้าที่อธิบายวิธีการเขียนนิพจน์ วิธีการเขียนคำสั่ง และวิธีการเขียนโปรแกรมอย่างถูกต้องตามรูปแบบของภาษานั้นๆ โปรแกรมแปลภาษาหรือคอมไพเลอร์ (compiler) จะทำหน้าที่ตรวจสอบความถูกต้องของการเขียนโครงสร้างโปรแกรมและความถูกต้องของการเขียนคำสั่งทั้งหมดในโปรแกรม เมื่อคำสั่งทั้งหมดถูกต้องตามไวยากรณ์ คอมไพเลอร์จะทำงานขั้นตอนต่อไปคือ แปลโปรแกรมให้อยู่ในรูปแบบของภาษาเครื่องเพื่อให้คอมพิวเตอร์ประมวลผลได้ รูปที่ 2.3 แสดงขั้นตอนการทำงานของคอมไพเลอร์โดยสังเขป (การทำงานโดยละเอียดจะอธิบายในบทที่ 4)



รูปที่ 2.3 ขั้นตอนการทำงานของคอมไพเลอร์

ส่วนประกอบแรกของคอมไพเลอร์คือส่วนวิเคราะห์ศัพท์ (lexical analysis) ทำหน้าที่รับไฟล์โปรแกรมเข้ามาทั้งโปรแกรม แล้วแยกข้อความทั้งหมดในโปรแกรมออกเป็นคำย่อยๆ เรียกว่า โทเค็น (token) พร้อมทั้งทำหน้าที่ตรวจสอบว่าคำศัพท์หรือโทเค็นทั้งหมดนั้นเขียนได้ถูกต้องตามที่กำหนดไว้ในไวยากรณ์ของภาษาหรือไม่ ตัวอย่างเช่น ถ้ามีการออกแบบภาษาที่คล้ายภาษา C ตั้งชื่อว่าภาษา JAY (Tucker and Noonan, 2002) แต่มีโครงสร้างและการใช้คำสั่งที่จำกัดกว่า โดยกำหนดรูปแบบของภาษาในลักษณะ BNF ไว้ดังรูปที่ 2.4

```
InputElement   →   WhiteSpace
                   | Comment | Token
WhiteSpace     →   space | \t | \r | \n | \f | \r\n
Comment        →   // any string ended by \r \n or \r\n

Token          →   Identifier  | Keyword   | Literal
                   | Separator | Operator

Identifier     →   Letter | Identifier Letter
                   | Identifier Digit
Letter         →   a | b | … | z | A | B | … | Z
Digit          →   0 | 1 | … | 9

Keyword        →   boolean | else | if    | int
                   | main  | void | while
Literal        →   Boolean | Integer
Boolean        →   true | false
Integer        →   Digit | Integer Digit
Separator      →   ( | ) | { | } | ; | ,


Operator       →   = | + | - | * | /
                   | < | <= | > | >= | = = | !=
                   | && | || | !
```

รูปที่ 2.4 ไวยากรณ์ของภาษาตัวอย่างโดยแสดงเฉพาะส่วนคำศัพท์

ไวยากรณ์ในรูปที่ 2.4 นิยามคำศัพท์หรือโทเค็น 5 ประเภท คือ Identifier, Keyword, Literal, Separator, Operator การนำคำศัพท์เหล่านี้มาประกอบเป็นข้อความสั่งประเภทต่างๆ กำหนดรูปแบบการเขียนคำสั่งตามไวยากรณ์ BNF ในรูปที่ 2.5

```
Program          →  void main ( ) { Declarations Statements }
Declarations     →  ε | Declarations Declaration
Declaration      →  Type Identifers ;
Type             →  int | boolean
Identifers       →  Identifer | Identifers, Identifer
Statements       →  ε | Statements Statement

Statement        →  ; | Block | Assignment | IfStatement
                    | WhileStatement
Block            →  { Statements }
Assignment       →  Identifer = Expression ;

IfStatement      →  if (Expression) Statement
                    | if (Expression) Statement else Statement
WhileStatement   →  while (Expression) Statement
Expression       →  Conjunction | Expression | Conjunction

Conjunction      →  Relation
                    | Conjunction && Relation
Relation         →  Addition
                    | Relation < Addition
                    | Relation <= Addition
                    | Relation >= Addition
                    | Relation = = Addition
                    | Relation != Addition
Addition         →  Term | Addition + Term | Addition - Term
Term             →  Negation | Term * Negation | Term /
                    Negation
Negation         →  Factor | !Factor
Factor           →  Identifier | Literal | (Expression)
```

รูปที่ 2.5 ไวยากรณ์ของภาษาตัวอย่างส่วนโปรแกรมและข้อความสั่ง
(สัญลักษณ์ ε แทนข้อความว่าง หรือ empty string)

ถ้าโปรแกรมเมอร์ต้องการเขียนโปรแกรมหาเลขไฟโบนาชชีตัวที่ *n* ตามไวยากรณ์ของภาษาตัวอย่างนี้ โปรแกรมจะมีรูปแบบดังรูปที่ 2.6

(<u>หมายเหตุ</u>: เลขไฟโบนาชชี หรือ Fibonacci number เป็นลำดับของตัวเลขที่สองตัวแรกเป็นเลข 1 ตัวเลขลำดับถัดไปจะเป็นผลบวกของเลขสองตัวก่อนหน้านั้น นั่นคือ

Fib(n) = 1 ถ้า n คือ 0 หรือ 1,

Fib(n) = Fib(n - 1) + Fib(n - 2) ในกรณีที่ n มีค่ามากกว่า 1

ตัวอย่างเลขอนุกรมไฟโบนาชชี ได้แก่ 1, 1, 2, 3, 5, 8, 13, 21, 34, …)

```
// Compute result = the nth Fibonacci number
void main ( ) {
      int n, fib0, fib1, temp, result;
      n = 8;
      fib0 = 0;
      fib1 = 1;
      while (n>0) {
            temp = fib0;
            fib0 = fib1;
            fib1 = fib0 + temp;
            n = n-1;
      }
      result = fib0;
}
```

รูปที่ 2.6 โปรแกรมหาเลขไฟโบนาชชีตัวที่ n โดย n = 8

เมื่อโปรแกรมตามรูปที่ 2.6 ถูกส่งเป็นอินพุตเข้ามายังคอมไพเลอร์ ส่วนวิเคราะห์ศัพท์ (lexical analysis) จะทำหน้าที่แยกคำออกเป็นคำย่อยๆ เรียกว่า เล็กซีม (lexeme) พร้อมทั้งจัดประเภทของเล็กซีม ว่าเป็นคำศัพท์ หรือโทเค็น (token) ในกลุ่มใด ดังรูปที่ 2.7

เล็กซีม (lexeme)	**โทเค็น (token)**
`// Compute …`	Comment
`void`	Keyword
`main`	Keyword
`(`	Separator
`)`	Separator
`{`	Separator
`int`	Keyword
`n`	Identifier
`,`	Separator
`fib0`	Identifier
`,`	Separator
`fib1`	Identifier
`,`	Separator
`temp`	Identifier
`,`	Separator
`result`	Identifier
`;`	Separator

รูปที่ 2.7 การจัดกลุ่มโทเค็นจากข้อความสามบรรทัดแรกของโปรแกรมในรูปที่ 2.6

ส่วนวิเคราะห์ศัพท์นอกจากทำหน้าที่จัดกลุ่มโทเค็นแล้ว ยังตรวจสอบความถูกต้องให้ด้วยว่าคำศัพท์นั้นๆ เป็นคำศัพท์ที่นิยามไว้ในภาษา ตัวอย่างเช่น ถ้าในบรรทัดที่สามของโปรแกรมหาเลขไฟโบนาชชี (รูปที่ 2.6) ผู้เขียนโปรแกรมเขียนสัญลักษณ์จบท้ายประโยคผิดเป็นดังนี้

```
int n, fib0, fib1, temp, result:
```

ส่วนวิเคราะห์ศัพท์จะตรวจพบข้อผิดพลาดนี้และแสดงข้อความเตือนให้โปรแกรมเมอร์ได้ทราบ ข้อความเตือนอาจมีลักษณะดังนี้

```
int n, fib0, fib1, temp, result:
                                ^
```

undefined symbol, should be ;

หรือถ้าโปรแกรมเมอร์ตั้งชื่อตัวแปรผิดรูปแบบของภาษา ส่วนวิเคราะห์ศัพท์จะทำหน้าที่แสดงข้อความเตือนเช่นเดียวกันดังตัวอย่างต่อไปนี้

```
int 1n;
    ^
```

incorrect identifier

ถ้าโปรแกรมตั้งต้นเขียนได้อย่างถูกต้องหรือได้รับการแก้ไขให้ถูกต้องตามข้อกำหนดของภาษาแล้ว กลุ่มของโทเค็นจะถูกส่งต่อให้กับส่วนวิเคราะห์ไวยากรณ์ (syntactic analysis) เพื่อวิเคราะห์ไวยากรณ์ของทุกคำสั่งและของทั้งโปรแกรมว่าสอดคล้องตามข้อกำหนดของภาษา (ที่กำหนดไว้ด้วยไวยากรณ์ในรูปแบบ BNF ตามรูปที่ 2.5)

การทำงานของส่วนวิเคราะห์ไวยากรณ์ จะเป็นการนำโทเค็นที่รับมาจากส่วนวิเคราะห์ศัพท์พิจารณาร่วมกับไวยากรณ์ BNF ของภาษาว่าถูกต้องตามไวยากรณ์หรือไม่ กระบวนการตรวจสอบความถูกต้องจะใช้วิธีการสร้างพาสทรี (parse tree) หรือบางครั้งเรียกว่าเดริเวชันทรี (derivation tree) เพราะกระบวนการตรวจสอบจะเป็นการ derive หรือการแปลงตามไวยากรณ์มาตามลำดับของโพรดักชัน โดยมีโพรดักชันเริ่มต้นคือ

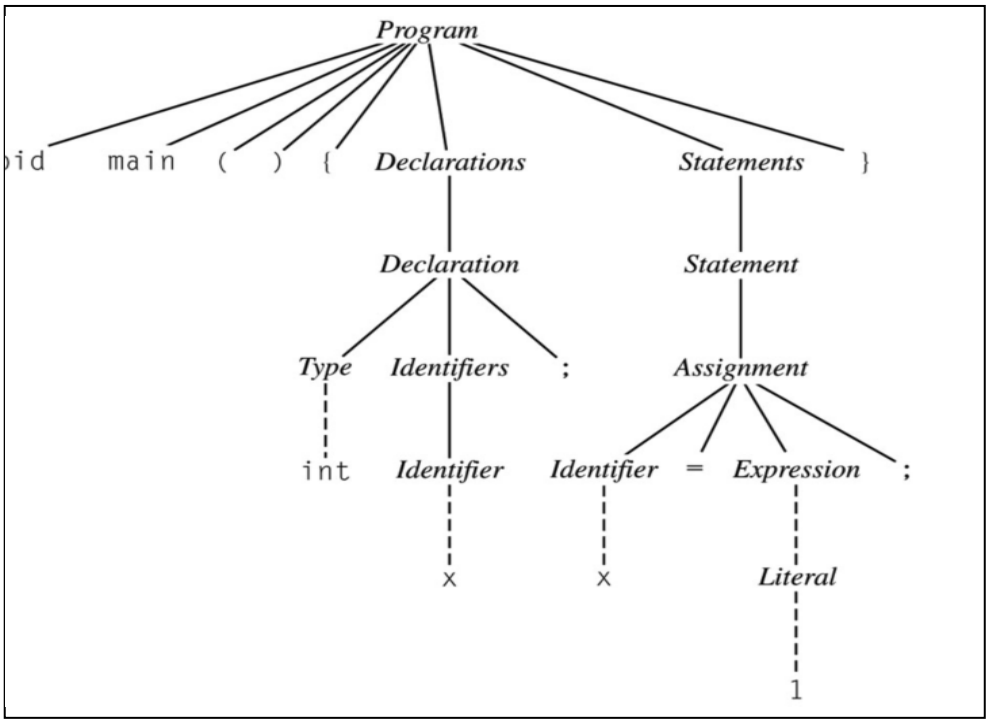
```
Program → void main ( ) { Declarations Statements}
```

นั่นคือ สัญลักษณ์นอนเทอร์มินอล `Program` ทำหน้าที่เป็นสัญลักษณ์เริ่มต้นของไวยากรณ์ BNF การตรวจสอบความถูกต้องจะใช้วิธีแปลง (derive) จากโพรดักชันเริ่มต้นที่มีทั้งสัญลักษณ์นอนเทอร์มินอลและเทอร์มินอลปะปนกัน จนกระทั่งได้รูปแบบที่เป็นสัญลักษณ์เทอร์มินอลทั้งหมด จึงจะถือว่าโปรแกรมนั้นเขียนได้ถูกต้องตามไวยากรณ์ของภาษา หรือถ้าใช้วิธีการสร้างพาสทรี โหนดรากของทรีจะเป็นสัญลักษณ์นอนเทอร์มินอล `Program` โหนดในระดับถัดลงมาจะเป็นส่วนประกอบทั้งหมดของคำว่า `Program` สัญลักษณ์นอนเทอร์มินอลทุกตัวจะถูกแทนด้วยนิยามไล่ลงมาตามลำดับจนถึงที่สุดเมื่อโหนดใบทั้งหมดเป็นสัญลักษณ์เทอร์มินอล

วิธีการวิเคราะห์ไวยากรณ์ของโปรแกรม แสดงด้วยตัวอย่างโปรแกรมขนาดสั้น ต่อไปนี้

```
void main ( ) {
     int x;
     x = 1;
}
```

ถ้าวิเคราะห์ความถูกต้องด้วยการสร้างพาสทรี จะได้แผนภาพต้นไม้ดังรูปที่ 2.8 ซึ่งจะสังเกตได้ว่าที่โหนดใบจะเป็นสัญลักษณ์เทอร์มินอลทั้งหมด



รูปที่ 2.8 พาสทรีของการวิเคราะห์ไวยากรณ์โปรแกรมตัวอย่าง

การวิเคราะห์ความถูกต้องของไวยากรณ์ ด้วยการแปลงจากโพรดักชันเริ่มต้นจนกระทั่งได้รูปแบบที่เป็นสัญลักษณ์เทอร์มินอลทั้งหมดแสดงได้ดังนี้ (แต่ละขั้นของการแปลงจะใช้เครื่องหมาย => และสัญลักษณ์นอนเทอร์มินอลที่ถูกแปลงจะเน้นด้วยการขีดเส้นใต้)

```
Program => void main ( ) { Declarations Statements }
        => void main ( ) { Declarations Declaration Statements }
        => void main ( ) { ε Declaration Statements }
        => void main ( ) { Type Identifiers; Statements }
        => void main ( ) { int Identifiers; Statements }
        => void main ( ) { int Identifier; Statements }
        => void main ( ) { int Letter; Statements }
        => void main ( ) { int x; Statements }
        => void main ( ) { int x; Statements Statement }
        => void main ( ) { int x; ε Statement }
        => void main ( ) { int x; Assignment }
        => void main ( ) { int x; Identifier = Expression; }
        => void main ( ) { int x; Letter = Expression; }
```

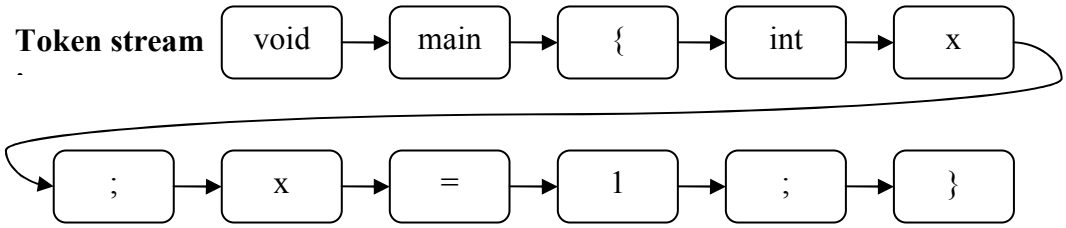
```
=> void main ( ) { int x; x = Expression; }
=> void main ( ) { int x; x = Conjunction; }
=> void main ( ) { int x; x = Relation; }
=> void main ( ) { int x; x = Addition; }
=> void main ( ) { int x; x = Term; }
=> void main ( ) { int x; x = Negation; }
=> void main ( ) { int x; x = Factor; }
=> void main ( ) { int x; x = Literal; }
=> void main ( ) { int x; x = Integer; }
=> void main ( ) { int x; x = Digit; }
=> void main ( ) { int x; x = 1; }
```

การแปลงแบบนี้เรียกว่าการแปลงแบบซ้ายสุด (leftmost derivation) เนื่องจากสัญลักษณ์นอนเทอร์มินอลตัวแรกที่ปรากฏอยู่ซ้ายมือสุดจะถูกแปลงก่อนเสมอ

ถ้าโปรแกรมเขียนผิดไวยากรณ์เช่นหลังคำว่า `main` ไม่มีวงเล็บ

```
void main  {
        int x;
        x = 1;
    }
```

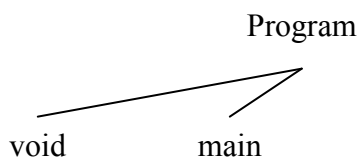
เราจะไม่สามารถสร้างพาสทรีที่สมบูรณ์ หรือไม่สามารถแปลงโพรดักชันจนกระทั่งได้สัญลักษณ์เทอร์มินอลทั้งหมด



BNF grammar :

```
Program → void main ( ) { Declarations Statements }
```

Parse Tree:



การสร้างพาสทรีไม่สามารถทำต่อจนสมบูรณ์ได้ เพราะตามไวยากรณ์ต้องการเครื่องหมาย '(' แต่โทเค็นที่ได้มาเป็นเครื่องหมาย '{' เมื่อสัญลักษณ์เทอร์มินอลไม่ตรงกันคอมไพเลอร์จะแจ้งข้อความเตือนเหตุผิดพลาดและจะไม่ทำงานในขั้นตอนต่อไป จนกว่าโปรแกรมเมอร์จะแก้ไขโปรแกรมให้ถูกต้องตามหลักไวยากรณ์ของภาษา

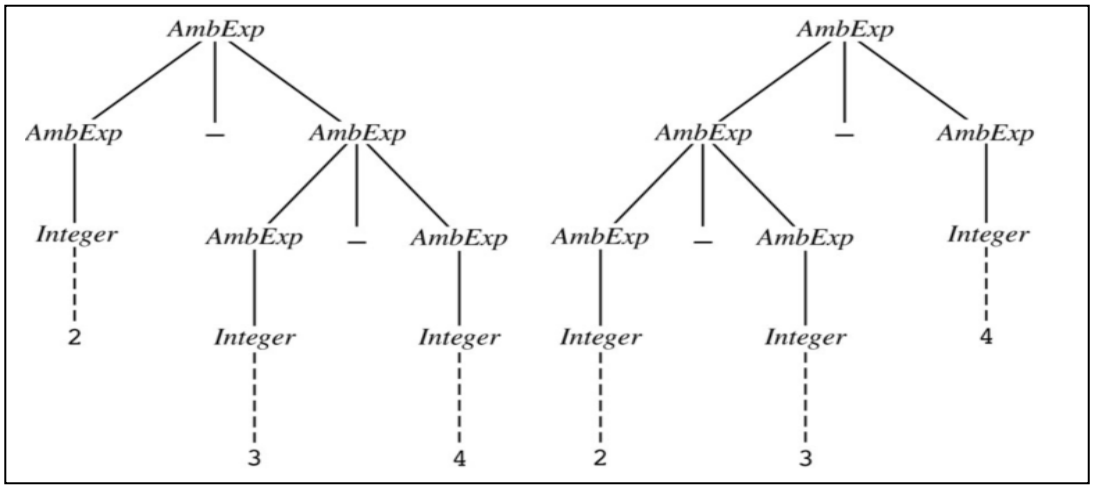
2.5 ไวยากรณ์กำกวม

(Ambiguous grammar)

การนิยามไวยากรณ์ของภาษา จะเกิดกรณีกำกวมขึ้นเมื่อรูปแบบไวยากรณ์กำหนดไว้ไม่ชัดเจนเพียงพอ ทำให้สามารถสร้างพาสทรีหรือสามารถแปลงไวยากรณ์ไปได้หลายรูปแบบ ไวยากรณ์ที่ดีควรจะมีวิธีการแปลงที่เป็นไปได้เพียงรูปแบบเดียว ตัวอย่างของไวยากรณ์ที่กำกวมแสดงได้ดังนี้

```
AmbExp → Integer | AmbExp - AmbExp
```

นิยาม `AmbExp` เป็นการนิยามนิพจน์ที่สามารถเป็นตัวเลขจำนวนเดียวหรือเป็นตัวเลขหลายจำนวนสัมพันธ์กันด้วยเครื่องหมายลบ และถ้าในโปรแกรมมีการเขียนนิพจน์ 2-3-4 การตรวจสอบความถูกต้องในด้านไวยากรณ์ของนิพจน์นี้ด้วยพาสทรี จะได้พาสทรีที่เป็นได้สองรูปแบบ ดังรูปที่ 2.9



รูปที่ 2.9 พาสทรีสองรูปแบบที่ใช้ตรวจสอบนิพจน์ *2-3-4*

การประมวลผลตามพาสทรีซ้ายมือจะได้ลำดับการทำงานเป็น 2-(3-4) ผลลัพธ์คือ 3 ในขณะที่พาสทรีขวามือจะมีลำดับการทำงานเป็น (2-3)-4 ผลลัพธ์คือ -5 จะเห็นว่าพาสทรีทั้งสองรูปแบบสามารถใช้ตรวจสอบได้ผลเหมือนกันว่ารูปแบบการเขียนนิพจน์ 2-3-4 นั้นถูกต้อง แต่ในขั้นประมวลผลจะให้ผลลัพธ์ที่แตกต่างกัน

ในการออกแบบภาษาผู้กำหนดรูปแบบภาษาควรหลีกเลี่ยงกรณีกำกวม จากตัวอย่างข้างต้น สามารถปรับปรุงไวยากรณ์ให้หมดความกำกวมได้ดังนี้

```
UnambExp → Integer | UnambExp - Integer
```

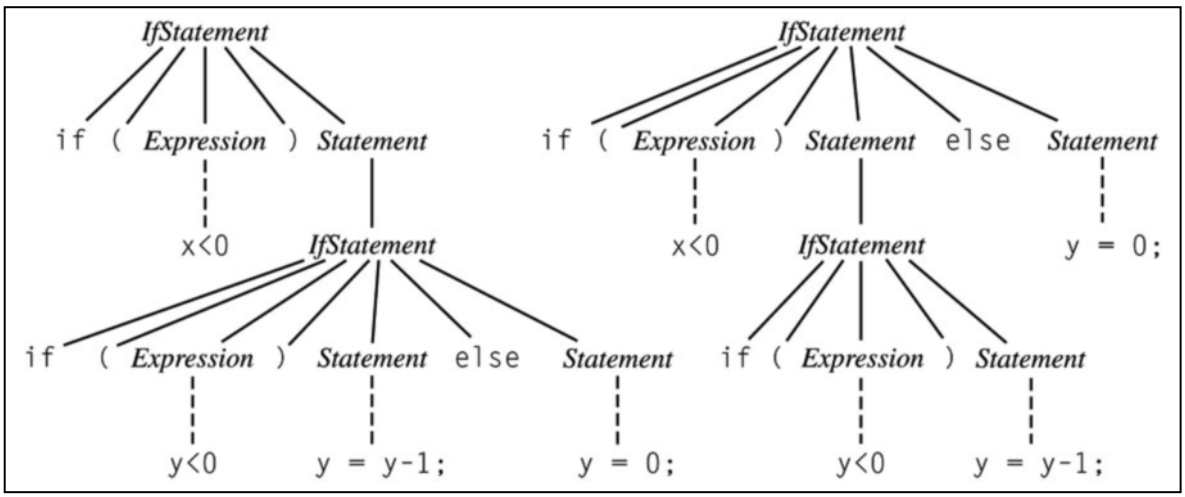
การกำหนดรูปแบบของคำสั่งแบบมีเงื่อนไขสามารถเกิดกรณีกำกวมได้เช่นเดียวกัน

```
IfStatement → if (Expression) Statement
                | if (Expression) Statement else Statement
Statement   → Assignment
                | IfStatement
```

ถ้าโปรแกรมเมอร์เขียนคำสั่งต่อไปนี้

```
if (x<0)
      if (y<0) y = y - 1;
      else y = 0;
```

คำสั่งนี้ถูกต้องตามไวยากรณ์ แต่ปัญหาคือสามารถสร้างพาสทรีได้สองรูปแบบ ดังรูปที่ 2.10



รูปที่ 2.10 พาสทรีสองรูปแบบของคำสั่ง IF_THEN_ELSE

ภาษา C และ C++ แก้ปัญหานี้ (เรียกว่าปัญหา dangling else) ด้วยการตีความว่า else จะคู่กับ if ที่อยู่ใกล้ที่สุด นั่นคือ ตีความแบบพาสทรีซ้ายมือตามรูปที่ 2.10 แต่ถ้าโปรแกรมเมอร์ต้องการให้มีความหมายเหมือนรูปพาสทรีขวามือ สามารถบังคับด้วยเครื่องหมายจัดกลุ่มคำสั่ง ({…}) ดังนี้

```
if (x<0)
      { if (y<0) y = y - 1; }
      else y = 0;
```

2.6 การจัดกลุ่มไวยากรณ์ตามลำดับชั้นชอมสกี

(Class of grammars by Chomsky hierarchy)

ไวยากรณ์ของภาษานิยามขึ้นโดยการกำหนดเซตของสัญลักษณ์นอนเทอร์มินอล, เซตของสัญลักษณ์เทอร์มินอล, สัญลักษณ์เริ่มต้น (เป็นสัญลักษณ์นอนเทอร์มินอล), และเซตของโพรดักชัน ในปี ค.ศ. 1959 Noam Chomsky ได้จัดกลุ่มไวยากรณ์ของภาษาออกเป็น 4 กลุ่ม เรียกว่า Type 0, Type 1, Type 2, Type 3 โดยแยกความแตกต่างตรงที่รูปแบบการเขียนโพรดักชัน

<u>Type 3</u> ไวยากรณ์ปรกติ (regular grammars)

ไวยากรณ์ปรกติเป็นกรณีเฉพาะของไวยากรณ์ BNF ตรงที่มีข้อจำกัดเพิ่มขึ้นในการเขียนโพรดักชัน นั่นคือด้านขวามือของโพรดักชันจะต้องเป็นสัญลักษณ์เทอร์มินอลเท่านั้น หรือเป็นสัญลักษณ์เทอร์มินอลที่ตามด้วยสัญลักษณ์นอนเทอร์มินอล

```
Nonterminal → terminal Nonterminal | terminal
```

ตัวอย่างการเขียนไวยากรณ์ปรกติที่นิยามตัวระบุ (identifier) ว่าประกอบด้วยตัวอักษร หรือตัวอักษรที่อาจตามด้วยตัวอักษรหรือตัวเลข แสดงได้ดังนี้

```
Identifier → aX | bX | … | zX | a | … | z
X          → aX | bX | … | zX | 0X |1X |…| 9X | a | … | z | 0
```

ไวยากรณ์ปรกตินิยมใช้ในการอธิบายคำศัพท์ที่เป็นโครงสร้างพื้นฐานของภาษา เช่น ตัวระบุ คำหลัก และคำสงวน แต่ไม่สามารถใช้แสดงไวยากรณ์ที่มีความซับซ้อนมากขึ้น เช่น ไม่สามารถเขียนไวยากรณ์ที่ระบุว่า จำนวนวงเล็บ “(“ มีเท่ากับจำนวนวงเล็บ “)” ลักษณะเช่นนี้ต้องอธิบายด้วยไวยากรณ์ที่มีความสามารถสูงขึ้น นั่นคือไวยากรณ์ BNF

<u>Type 2</u> ไวยากรณ์ไม่พึ่งบริบท (context-free grammars)

การเขียนโพรดักชันในไวยากรณ์ไม่พึ่งบริบท หรือไวยากรณ์ BNF มีรูปแบบดังนี้

$$\text{Nonterminal} \rightarrow \beta$$

เมื่อ β สามารถเป็นสัญลักษณ์เทอร์มินอล หรือสัญลักษณ์นอนเทอร์มินอลจำนวนเท่าใดก็ได้ ตัวอย่างเช่น

```
Expression → Expression + Term | Term
Term       → Term * Factor | Factor
Factor     → Identifier | ( Expression )
```

จากไวยากรณ์ข้างต้นจะเห็นได้ว่ามีความสามารถตรวจสอบกรณีการใช้วงเล็บที่คู่กันในนิพจน์ ว่าจำนวนวงเล็บเปิด “(“ จะต้องเท่ากับจำนวนวงเล็บปิด “)” แต่ไวยากรณ์ประเภทนี้ไม่ตรวจสอบบริบท นั่นคือไม่สนใจข้อความที่อยู่โดยรอบ เช่น ในการเขียนนิพจน์ $x*y$ ไวยากรณ์ไม่สามารถกำหนดได้ว่า ตัวระบุ x

และ y ต้องเป็นชนิดจำนวนเลขเท่านั้น และไม่สามารถกำหนดว่า ถ้า x เป็นชนิดเลขจำนวนเต็ม y ต้องเป็นชนิดเลขจำนวนเต็มด้วย

<u>Type 1</u> ไวยากรณ์พึ่งบริบท (context-sensitive grammars)

ไวยากรณ์ประเภทนี้มีรูปแบบในการเขียนโพรดักชัน

$$\alpha \rightarrow \beta$$

เมื่อ α เป็นสัญลักษณ์นอนเทอร์มินอลจำนวนเท่าใดก็ได้ β เป็นสัญลักษณ์เทอร์มินอลหรือสัญลักษณ์นอนเทอร์มินอลจำนวนเท่าใดก็ได้ แต่จำนวนสัญลักษณ์รวมใน α ต้องน้อยกว่าหรือเท่ากับจำนวนสัญลักษณ์รวมใน β

ตัวอย่างไวยากรณ์ต่อไปนี้เป็นไวยากรณ์พึ่งบริบท ที่กำหนดการเขียนข้อความให้เริ่มต้นด้วย x แล้วตามด้วยตัวอักษร a, b, c ที่มีจำนวนเท่าๆ กัน ($xa^nb^nc^n$)

```
 x  →  A
CB  →  BC
CA  →  AC
BA  →  AB
CCY →  CYc
BCY →  BYc
BBY →  BYb
ABY →  AYb
AAY →  AYa
 AY →  xa
```

ไวยากรณ์พึ่งบริบทไม่ได้ถูกนำมาใช้ในการเขียนไวยากรณ์ของภาษาคอมพิวเตอร์ เนื่องจากมีความซับซ้อนมากเกินไป

<u>Type 0</u> ไวยากรณ์ไม่มีข้อจำกัด (unrestricted grammar)

รูปแบบโพรดักชันในไวยากรณ์ประเภทนี้ คือ

$$\alpha \rightarrow \beta$$

เมื่อ α เป็นสัญลักษณ์นอนเทอร์มินอลจำนวนเท่าใดก็ได้ β เป็นสัญลักษณ์เทอร์มินอลหรือสัญลักษณ์นอนเทอร์มินอลจำนวนเท่าใดก็ได้ โดยไม่มีข้อจำกัดเรื่องขนาดของ α และ β

2.7 เครื่องเชิงนามธรรม
(Abstract machines)

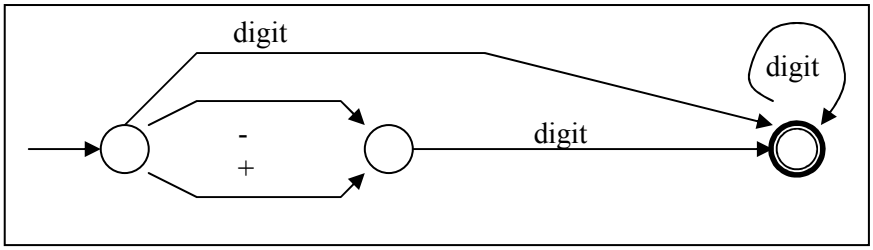
ไวยากรณ์ถูกสร้างขึ้นมาเพื่อทำหน้าที่ระบุข้อกำหนดในการเขียนข้อความให้ถูกต้องตามรูปแบบของภาษานั้นๆ ไวยากรณ์จึงเป็นเพียงคู่มือหรือแนวทางในการอ่านหรือเขียนข้อความ แต่การตรวจสอบความถูกต้องของข้อความจะใช้อุปกรณ์อัตโนมัติที่เรียกว่า *เครื่องเชิงนามธรรม* หรือ *เครื่องจักรสมมุติ* (abstract machine) เครื่องจักรเหล่านี้ไม่ได้มีอยู่จริง แต่สามารถจำลองขึ้นมาได้ด้วยการเขียนโปรแกรมสร้างขึ้น

เครื่องเชิงนามธรรมที่ทำหน้าที่ตรวจสอบไวยากรณ์ในกลุ่มต่างๆ สรุปได้ดังนี้

ลำดับชั้นชอมสกี	ชนิดของไวยากรณ์	ชนิดของเครื่องเชิงนามธรรม
0	unrestricted	Turing machine
1	context sensitive	Linear-bounded automaton
2	context free	Pushdown automaton
3	regular	Finite-state automaton

เครื่องอัตโนมัติสถานะจำกัด (finite-state automaton) ประกอบด้วยกราฟที่มีจำนวนโหนดหรือจุดยอดที่แน่นอน จุดเหล่านี้แทนสถานะต่างๆ ของเครื่อง มีจุดที่บอกสถานะเริ่มต้นและจุดที่เป็นสถานะสิ้นสุด (สถานะสิ้นสุดเขียนแทนด้วยวงกลมซ้อนกันสองวง) นอกจากกราฟแล้วยังมีแถบบันทึกหรือเทปที่บรรจุรหัสข้อความ แถบบันทึกนี้จะถูกอ่านไปได้ในทิศทางเดียว เมื่อเริ่มต้นอ่านอักขระแรก สถานะของเครื่องจะเปลี่ยนสถานะจากเริ่มต้นไปสู่สถานะถัดไป การอ่านรหัสข้อความแต่ละอักขระจะมีผลให้เครื่องเปลี่ยนสถานะไปตามลำดับ จนกระทั่งเมื่อหมดข้อความและสถานะสุดท้ายเป็นสถานะสิ้นสุด (final state) แสดงว่าข้อความนั้นถูกต้องตามไวยากรณ์ และเครื่องอัตโนมัติสถานะจำกัดจะรายงานว่ายอมรับข้อความ (accept) แต่ถ้าหมดการอ่านข้อความแล้วเครื่องไปหยุดที่สถานะอื่นซึ่งไม่ใช่สถานะสิ้นสุด แสดงว่าข้อความไม่ถูกต้องตามไวยากรณ์ เครื่องจะรายงานว่าปฏิเสธข้อความ (reject) รูปที่ 2.11 แสดงเครื่องอัตโนมัติสถานะจำกัดที่ทำหน้าที่ตรวจจับข้อความที่เป็นจำนวนเลขที่สามารถมีเครื่องหมายบวกหรือลบกำกับหน้าจำนวนเลขซึ่งเขียนด้วยไวยากรณ์ EBNF ได้ดังนี้

```
SignedIntegers → [ +| - ] digit {digit}*
```



รูปที่ 2.11 เครื่องอัตโนมัติสถานะจำกัดที่ทำหน้าที่ตรวจสอบ SignedIntegers

เครื่องอัตโนมัติแบบกดลง (pushdown automaton) มีส่วนประกอบพื้นฐานที่เหมือนกับเครื่องอัตโนมัติสถานะจำกัด คือมีกราฟแสดงการเปลี่ยนสถานะและมีแถบบันทึกข้อความที่อ่านเดินหน้าไปได้ทิศทางเดียว แต่เพื่อเพิ่มขีดความสามารถในการตรวจจับไวยากรณ์ที่ซับซ้อนขึ้นจึงได้มีการเพิ่มสแตก (stack) เข้ามาช่วยในการทำงานให้สามารถตรวจสอบข้อความที่ต้องมีอักขระปรากฏเป็นคู่ในจำนวนที่เท่ากัน เช่นนิพจน์ที่ใช้วงเล็บซ้อนกันหลายชั้น จะต้องมีจำนวนวงเล็บเปิดเท่ากับจำนวนวงเล็บปิด ดังตัวอย่างต่อไปนี้

```
( x + ( 2 - 3 / Z ) - ( 4 * ( 2 / y ) * 10 ))
```

หรือในโปรแกรมภาษา C ที่มีการใช้วงเล็บ {…} ซ้อนกันหลายๆ ชั้น

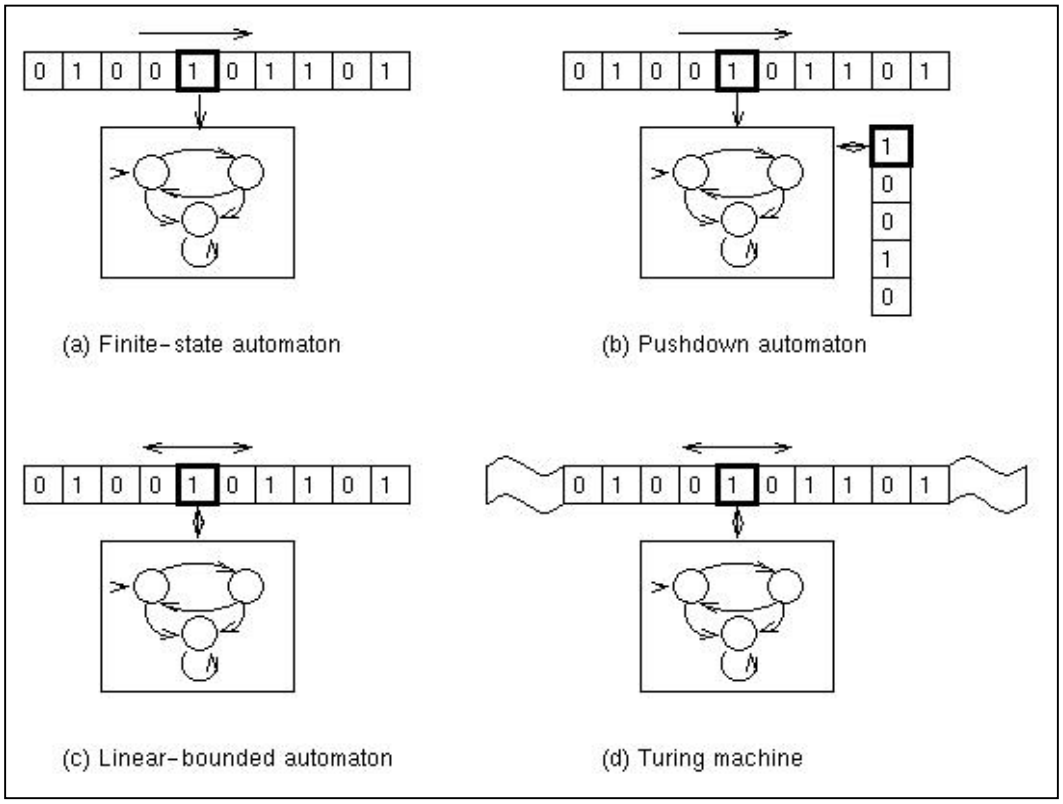
เครื่องอัตโนมัติขอบเขตจำกัดเชิงเส้น (linear-bounded automaton) มีส่วนประกอบเช่นเดียวกับเครื่องอัตโนมัติสถานะจำกัด คือมีกราฟแสดงการเปลี่ยนสถานะและมีแถบบันทึกข้อความ เพียงแต่การเคลื่อนที่ของหัวอ่านแถบบันทึก สามารถเคลื่อนที่ไปได้ทั้งสองทิศทางคือเดินหน้าและถอยหลัง ทำให้สามารถย้อนกลับไปตรวจสอบข้อความก่อนหน้าว่าสอดคล้องกับข้อความปัจจุบันหรือไม่ ตัวอย่างเช่น

```
y = "Hello" ; x = 10*y ;
```

เมื่อหัวอ่านเคลื่อนที่ไปถึงตำแหน่งของ y ที่ปรากฏเป็นครั้งที่สอง สามารถถอยกลับมาที่ต้นข้อความเพื่อตรวจสอบชนิดข้อมูลของ y ว่าสามารถนำมาคูณกับค่า 10 ได้หรือไม่ ซึ่งในกรณีนี้ไม่สามารถทำได้ (ตามไวยากรณ์ของภาษาคอมพิวเตอร์โดยทั่วไป) เครื่องจะรายงานปฏิเสธข้อความ

เครื่องทัวริง (turing machine) มีส่วนประกอบเหมือนกับเครื่องอัตโนมัติขอบเขตจำกัดเชิงเส้น เพียงแต่แถบบันทึกข้อความมีความยาวไม่จำกัด ทำให้มีขีดความสามารถสูงขึ้นในการตรวจสอบไวยากรณ์

รูปที่ 2.12 แสดงภาพเปรียบเทียบเครื่องเชิงนามธรรมทั้ง 4 ประเภทที่ใช้ตรวจสอบข้อความว่าถูกต้องตามไวยากรณ์ของภาษาหรือไม่



รูปที่ 2.12 ภาพเปรียบเทียบเครื่องเชิงนามธรรมทั้งสี่ประเภท

2.7 สรุป

ไวยากรณ์ คือรูปแบบการจัดเรียงคำและอักขระให้ถูกต้องตามข้อกำหนดของภาษา วิธีการที่นักออกแบบภาษาคอมพิวเตอร์นิยมใช้ในการเขียนไวยากรณ์ คือการอธิบายด้วยรูปแบบอย่างเป็นทางการที่เรียกว่ารูปแบบบีเอ็นเอฟ (Backus- Naur Form--BNF) ซึ่งเริ่มใช้เป็นครั้งแรกในช่วงทศวรรษที่ 1950 เพื่อใช้อธิบายไวยากรณ์ของภาษา ALGOL ในระยะหลังมีการดัดแปลงรูปแบบไปเป็น Extended BNF (เรียกว่ารูปแบบ EBNF) และมีการอธิบายไวยากรณ์ด้วยภาพ เรียกว่า แผนภูมิไวยากรณ์ (Syntax chart or syntax diagram)

การใช้รูปแบบอย่างเป็นทางการอธิบายไวยากรณ์ของภาษา มีวัตถุประสงค์เพื่อให้คำอธิบายสั้นกระทัดรัด เข้าใจได้ง่ายและเป็นมาตรฐานสากล สามารถเข้าใจได้ตรงกันในกลุ่มนักคอมพิวเตอร์ทั่วโลก นอกจากนี้ไวยากรณ์ในรูปแบบบีเอ็นเอฟ ยังเป็นคู่มือสำคัญที่คอมไพเลอร์ใช้ในการตรวจสอบความถูกต้องในเชิงไวยากรณ์ของโปรแกรมที่โปรแกรมเมอร์เขียนขึ้น

การตรวจสอบความถูกต้องของไวยากรณ์ด้วยคอมไพเลอร์ เป็นการตรวจสอบด้วยเครื่องจักรจริง นั่นคือใช้คอมพิวเตอร์ดำเนินงานจริง นอกจากวิธีการตรวจสอบจริงแบบนี้แล้วเรายังมีทางเลือกอื่นในการตรวจสอบเชิงไวยากรณ์ นั่นคือใช้เครื่องจักรสมมุติที่เรียกว่า เครื่องเชิงนามธรรม (abstract machine) เครื่องเชิงนามธรรมที่เป็นที่รู้จักกันอย่างกว้างขวางในกลุ่มนักคอมพิวเตอร์มีด้วยกัน 4 ประเภท เรียงลำดับตามขีดความสามารถในการตรวจสอบไวยากรณ์ตั้งแต่ความสามารถต่ำไปจนถึงความสามารถสูงได้ดังนี้ เครื่องอัตโนมัติสถานะจำกัด (finite-state automaton) ใช้ตรวจสอบไวยากรณ์ปรกติ หรือไวยากรณ์ลำดับชั้นที่ 3 ตามการจัดลำดับของชอมสกี, เครื่องอัตโนมัติแบบกดลง (pushdown automaton) ใช้ตรวจสอบไวยากรณ์ไม่พึ่งบริบท หรือไวยากรณ์ลำดับชั้นที่ 2, เครื่องอัตโนมัติขอบเขตจำกัดเชิงเส้น (linear-bounded automaton) ใช้ตรวจสอบไวยากรณ์พึ่งบริบท หรือไวยากรณ์ลำดับชั้นที่ 1 และเครื่องทัวริง (turing machine) ใช้ตรวจสอบไวยากรณ์ไม่มีข้อจำกัด หรือไวยากรณ์ลำดับชั้นที่ 0

แบบฝึกหัดท้ายบทที่ 2

คำถามปรนัย: ให้เลือกคำตอบที่ถูกต้องที่สุด

1. ภาษาคอมพิวเตอร์ใดที่ใช้อธิบายภาษาคอมพิวเตอร์อื่นๆ ?

ก. meta-language ข. free-language

ค. context-language ง. high-language

2. grammar G1 ต่อไปนี้ กำหนดรูปแบบการเขียนโปรแกรม โดยมี <program> เป็น start symbol

```
<program>        → begin <statement_list> end
<statement_list> → <statement> ;
                   | <statement> ; <statement_list>
<statement>      → <variable> := <expression>
<variable>       → A | B | C
<expression>     → ( <expression> )
                   | ( )
                   | < expression > + < expression >
                   | < expression > – < expression >
                   | <variable>
```

ข้อใดต่อไปนี้เป็นการเขียนโปรแกรมที่ถูกต้องตาม grammar G1 ?

ก. A=B+C ;C:=(A) ข. begin A:=A+((((())))); C:=(A+B)+A;end

ค. begin A=(B+C()-A;end ง. begin A=A+C end

3. ถ้าต้องการแก้ไข grammar G1 ในข้อ 26 เพื่อให้รูปแบบการเขียนโปรแกรมต่อไปนี้ <u>ไม่สามารถทำได้</u>

begin A := (B + C); end

จะต้องแก้ไขอย่างไร ?

ก. เปลี่ยนนิยาม <expression> เป็น

```
<expression>     → ( <expression> )
                   | < expression > + < expression >
                   | < expression > – < expression >
                   | <variable>
```

ข. เปลี่ยนนิยาม <expression> เป็น

```
<expression>     → <expression>
                   | < expression > + < expression >
                   | < expression > – < expression >
                   | <variable>
```

ค. เปลี่ยนนิยาม <expression> เป็น

```
<expression>     → ( <expression> )
                   | ( )
                   | < expression > + < expression >
                   | < expression > – < expression >
                   | X | Y | Z
```

ง. เปลี่ยนนิยาม <expression> เป็น

<expression> → (<expression>)
| ( )
| < expression > + < expression >
| < expression > – < expression >

4. grammar ในข้อใดมีลักษณะ ambiguous (กำหนดให้ <S> เป็น start symbol) ?

ก. <S> → <S> <A> | <A> < S> | <A>
<A> → a

ข. <S> → <S> <A> | <A>
<A> → a

ค. <S> → <A> | < C >
<A> → a
<C> → c

ง. <S> → <A> <C> | <B>
<A> → a
<B> → b
<C> → c

5. กำหนด grammar G2 ให้ดังต่อไปนี้

< stmt_list> → < stmt > ;
| < stmt > ; < stmt_list >
< stmt > → < var > := < expr >
< expr > → < expr > – < expr >
| < expr > * < expr >
| < var >
< var > → X | Y

ถ้า **<expr>** เป็น start symbol ข้อความใดต่อไปนี้ถูกต้องตามข้อกำหนดของไวยากรณ์และข้อกำหนด start symbol ?

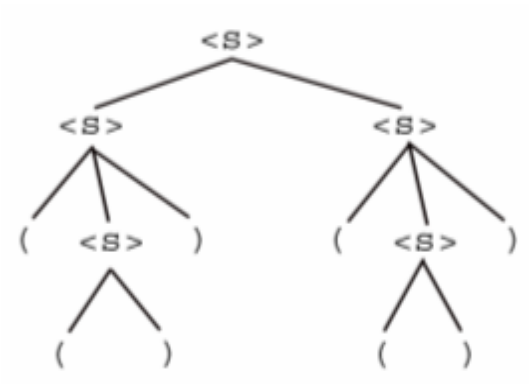
ก. X := Y * Y;

ข. Y := Y – Y;

ค. X := Y + Y; X := Y + Y;

ง. Y - Y

6. กำหนด parse tree ให้ดังรูปต่อไปนี้ ข้อความที่สร้างจาก parse tree นี้คืออะไร ?



ก. ()()(())

ข. (())()()

ค. ((()))

ง. (())(())

7. กำหนด grammar G3 ให้ดังต่อไปนี้

<expr> → <expr> # <expr> | <expr> $ <expr> | <id>
<id> → a | b | c | d | e

เมื่อ start symbol คือ <expr> ซึ่งหมายถึงนิพจน์คณิตศาสตร์ สัญลักษณ์ # และ $ หมายถึงโอเปอเรเตอร์ หรือเครื่องหมายกระทำการทางคณิตศาสตร์ (operator)

ถ้ากำหนดข้อความ a # b $ c การคำนวณนิพจน์ส่วนใดจะเกิดขึ้นก่อน?

ก. a # b เกิดขึ้นก่อน
ข. b $ c เกิดขึ้นก่อน
ค. grammar ไม่ได้ระบุไว้
ง. a # b และ b $ c เกิดขึ้นได้พร้อมกัน

8. ถ้าเปลี่ยน grammar G3 ในข้อ 31 เป็น G3’ ดังต่อไปนี้

<expr> → <factor> | <expr> $ <factor>
<factor> → <id> | <id> # <factor>
<id> → a | b | c | d | e

ถ้ากำหนดข้อความ a # b $ c การคำนวณนิพจน์ส่วนใดจะเกิดขึ้นก่อน?

ก. a # b เกิดขึ้นก่อน
ข. b $ c เกิดขึ้นก่อน
ค. grammar ไม่ได้ระบุไว้
ง. a # b และ b $ c เกิดขึ้นได้พร้อมกัน

9. พิจารณา grammar G3 และ G3’ grammar ใดเป็น ambiguous grammar ?

ก. G3 เป็น ambiguous grammar
ข. G3’ เป็น ambiguous grammar
ค. ทั้ง G3 และ G3’ เป็น ambiguous grammar
ง. ทั้ง G3 และ G3’ ไม่เป็น ambiguous grammar

10. ถ้าต้องการตรวจสอบข้อความตาม grammar G3 ด้วยเครื่องเชิงนามธรรม ควรใช้เครื่องชนิดใด ?

ก. Turing machine
ข. push-down automata
ค. finite-state automata
ง. linear-bounded automata