บทที่ 4
วากยสัมพันธ์ (Syntax)

4.1 โครงสร้างเชิงศัพท์ ของภาษาโปรแกรม
4.2 ไวยากรณ์ไม่พึ่งบริบท และบีเอ็นเอฟ
4.3 ต้นไม้วิภัช และต้นไม้วากยสัมพันธ์แบบนามธรรม
4.4 ความกำกวม การสลับที่ และการทำก่อน
4.5 อีบีเอ็นเอฟ และ แผนภาพวากยสัมพันธ์
4.6 เทคนิคการวิเคราะห์กระจาย และเครื่องมือที่ใช้
4.7 คำศัพท์ วากยสัมพันธ์ อรรถศาสตร์
แบบฝึกหัด

บทที่ 4
วากยสัมพันธ์ (Syntax)

วากยสัมพันธ์ หมายถึง โครงสร้างของภาษา ในบทที่ 1 เราได้ให้ข้อสังเกตความแตกต่างระหว่าง วากสัมพันธ์ และอรรถศาสตร์ ของภาษาโปรแกรมแล้ว ในอดีตของภาษาโปรแกรม วากยสัมพันธ์ และอรรถศาสตร์ของภาษา ทั้งคู่ อธิบายด้วย ภาษาอังกฤษอย่างยาว และตัวอย่างจำนวนมาก ขณะที่ อรรถศาสตร์ ของภาษายังคงถูกอธิบายด้วยภาษาอังกฤษอยู่ ความรู้ชั้นสูงมากในภาษาโปรแกรม ได้พัฒนา ระบบแบบทางการ สำหรับการอธิบายวากยสัมพันธ์ ซึ่งขณะนี้ ส่วนใหญ่ใช้กันเป็นสากล ในช่วงปี ค.ศ. 1950s Noam Chomsky พัฒนา ความคิดของ ไวยกรณ์ไม่พึ่งบริบท ขึ้น ส่วน John Backus และผู้ร่วมงานของเขา Peter Naur ได้พัฒนาระบบเชิงสัญลักษณ์ (notational system) สำหรับ อธิบาย ไวยากรณ์เหล่านี้ ซึ่งครั้งแรก นำมาใช้อธิบายวากยสัมพันธ์ของ ภาษา Algol60 สิ่งเหล่านี้เรียกว่า Backus - Naur Form หรือเรียกสั้นๆ ว่า BNFs ต่อมาได้นำมาใช้ ใน บทนิยาม ของ ภาษาโปรแกรมอื่นๆ จำนวนมาก เช่น Pascal, Modula-2, C และ Ada จริงๆ แล้ว นักเขียนโปรแกรมสมัยใหม่ และนักวิทยาศาสตร์คอมพิวเตอร์ จำเป็นต้องทราบว่าจะ อ่าน ตีความ และประยุกต์ใช้ การอธิบาย BNF ของวากยสัมพันธ์ภาษาได้อย่างไร

BNFs เหล่านี้ เกิดขึ้น ด้วย ความหลากหลายไม่มากนัก ใน รูปแบบพื้นฐาน สามอย่างคือ : BNF เดิม (original BNF), ส่วนขยายของ BNF (extended BNF (EBNF)) ผู้ที่ทำให้เกิดความนิยมคือ Niklaus Wirth, และรูปแบบที่สามคือ แผนภาพวากยสัมพันธ์ (syntax diagrams)

4 . 1 โครงสร้างศัพท์ ของ ภาษาโปรแกรม
(Lexical structure of programming languages)

โครงสร้างศัพท์ของภาษาโปรแกรม หมายถึง โครงสร้างของคำ หรือ โทเค็น ของมัน (The lexical structure of a programming language is the

structure of its words or tokens.)

โครงสร้างศัพท์ สามารถ พิจารณาแยกต่างหาก จาก โครงสร้างวากยสัมพันธ์ (syntactic structure) แต่มันเกี่ยวข้องกันมาก ในบางกรณี (ขึ้นอยู่กับ การออกแบบภาษา) ไม่สามารถ แยกส่วนต่างหากจากวากยสัมพันธ์ได้ โดยปกติ ขั้น**การกราดตรวจ** (scanning phase) ของตัวแปลภาษา จัดกลุ่ม ลำดับของอักขระต่างๆ จาก โปรแกรม อินพุท ให้เป็น โทเค็นต่างๆ ซึ่งหลังจากนั้น ถูกประมวลผล โดย ขั้น**การวิเคราะห์กระจาย** (parsing phase) ซึ่งจะบอกความถูกต้องของ โครงสร้างวากยสัมพันธ์

โทเค็น แบ่งออกเป็นหลายชนิด ได้แก่

คำสงวน (reserved words) บางครั้ง เรียกว่า **คำหลัก** (keywords) เช่นคำว่า "begin", "if" และ "while"

ค่าคงที่ (constants หรือ literals) เช่น 42 เป็นค่าคงที่ตัวเลข (numeric constant) หรือ "hello" เป็นค่าคงที่สายอักขระ (string constant)

สัญลักษณ์พิเศษ (special symbols) เช่น ";", "<=" หรือ "+"

ไอเดนติไฟเออร์ (identifiers) เช่น x24, average, monthly-balance, หรือ write

คำสงวน มีเป็นจำนวนมาก และเนื่องจากไอเดนติไฟเออร์ จะมีสายอักขระเหมือนกับ คำสงวนไม่ได้ ดังนั้น ในภาษา Pascal การประกาศตัวแปร ข้างล่างนี้ ไม่ถูกต้อง เพราะว่า "if" เป็นคำสงวน

```
var if : integer;
```

ในบางภาษา ไอเดนติไฟเออร์ จะมีขนาดสูงสุดคงที่ (fixed maximum size) ในขณะที่ไอเดนติไฟเออร์ของภาษาใหม่ๆ ส่วนใหญ่มีความยาวเท่าไหร่ก็ได้ (arbitrary length) บางครั้ง เมื่อ ไอเดนติไฟเออร์ มีขนาดอย่างไรก็ได้ แต่เฉพาะอักขระ 6 ตัว หรือ 8 ตัวแรกเท่านั้น ซึ่งมีนัยสำคัญ (significant) สิ่งนี้ ประกันความสับสนให้กับนักเขียนโปรแกรม

ปัญหาอย่างหนึ่งเกิดขึ้น ใน การบอก การจบ (end) ของ โทเค็นความยาว

แปรได้ (variable-length token) เช่น ไอเดนติไฟเออร์ และการแยก ไอเดน-ติไฟเออร์ ออกจาก คำสงวน

ตัวอย่างเช่น ลำดับของตัวอักขระ

doif

ในโปรแกรม อาจจะเป็น คำสงวนสองคำ คือ "do" และ "if" หรืออาจจะเป็น ไอเดนติไฟเออร์ doif ก็ได้

ในทำนองเดียวกัน สายอักขระ x12 อาจจะเป็นไอเดนติไฟเออร์ หนึ่งตัว คือ x12 หรือ อาจจะเป็นไอเดนติไฟเออร์ x และค่าคงที่ตัวเลข 12

เพื่อขจัดความกำกวมนี้ จึงมีข้อตกลงมาตรฐาน (standard convention) ให้ใช้ คือ **หลักของสายอักขระย่อยความยาวสูงสุด** (principle of longest sub-string) ในการหา โทเค็น : ที่แต่ละจุด สายของอักขระที่เป็นไปได้ความยาวสูงสุด รวมเข้าด้วยกันเป็น หนึ่งโทเค็น สิ่งนี้หมายความว่า doif และ x12 เป็นไอเดนติไฟเออร์เสมอ

ตัวอักขระแทรก (intervening characters) เช่น อักขระว่าง (blank) จะทำให้เกิดความแตกต่าง ดังนั้น ในภาษาส่วนใหญ่

do if

จึงไม่ใช่ ไอเดนติไฟเออร์ แต่เป็น คำสงวนสองคำ คือ "do" และ "if"

รูปแบบของโปรแกรม สามารถกระทบกับ วิธีที่จะรู้จำโทเค็น (The format of a program can affect the way tokens are recognized.)
ตัวอย่างเช่น หลักของสายอักขระย่อยความยาวสูงสุด ต้องการให้โทเค็นต่างๆ แยกกันด้วย **ตัวคั่นโทเค็น** (token delimiters) หรือ white space

การจบ ของ หนึ่งบรรทัดของข้อความ (text) อาจมีความหมายได้สองอย่าง คือ : มันอาจเป็น ว่างหนึ่งที่ หรือ อาจหมายถึง การจบของเอนทิตีเชิงโครงสร้าง

การย่อหน้า สามารถนำมาใช้ในภาษาโปรแกรม เพื่อบอกโครงสร้าง
ภาษาซึ่งมีรูปแบบอิสระ หมายถึง ภาษา ซึ่งรูปแบบของมันไม่มีผลกระทบ บน โครงสร้าง โปรแกรม (A **free-format** language is one in which format has no

effect on the program structure.) และยังต้องเป็นไปตามหลักของสายอักขระย่อยความสูงสุด ภาษาโปรแกรมสมัยใหม่ส่วนใหญ่ เป็นภาษารูปแบบอิสระ แต่ภาษาโปรแกรมส่วนน้อย มี ข้อจำกัดรูปแบบอย่างสำคัญ ซึ่งเรียกว่า **รูปแบบคงที่** (fixed format) ซึ่งโทเค็นทั้งหมดต้อง เกิดขึ้น ในตำแหน่งที่กำหนดไว้แล้ว ในหน้ากระดาษ เช่นภาษา RPG, FORTRAN, และ COBOL เป็นต้น

FORTRAN เป็นตัวอย่างที่สำคัญของ ภาษาซึ่ง ฝ่าฝืน (violates) รูปแบบหลายอย่าง และ ข้อตกลงเรื่องโทเค็น

ตัวอย่าง ภาษา FORTRAN

```
DO  99  I = 1.10
```
(1)

ข้อความสั่งนี้ มีความหมายเหมือนกับ ภาษา Pascal

```
DO99I  :=  1.10
```

พูดอีกอย่างหนึ่งคือ เป็นการกำหนดค่า 1.1 ให้กับตัวแปรจำนวนจริง DO99I ในทางตรงกันข้าม ข้อความสั่ง ภาษา FORTRAN ข้างล่างนี้

```
DO 99 I =1, 10
```
(2)

มีความหมายเหมือนกับ ข้อความสั่ง ภาษา Pascal ดังนี้

```
for I := 1 to 10 do
```

นั่นคือ เริ่มการวนซ้ำ (loop) โดยกำหนดขอบเขตให้กับ ตรรชนี I ดังนั้น ข้อความสั่งภาษา FORTRAN ชุดแรกจึงประกอบด้วย โทเค็น 3 ตัว คือ ไอเดนติไฟเออร์, ตัวดำเนินการกำหนดค่า ("=") และ ค่าคงที่จำนวนจริง 1.1 แต่ตรงกันข้าม ส่วนข้อความสั่ง ภาษา FORTRAN ชุดที่สอง ประกอบด้วย โทเค็น 7 ตัว เหตุผลของสิ่งนี้คือ ภาษา FORTRAN ไม่สนใจอักขระว่างอย่างบริบูรณ์ (ignores space completely) - ภาษานี้จะลบอักขระว่าง (space) ออกก่อนเริ่มการประมวลผล นอกจากนี้แล้ว ภาษา FORTRAN ไม่มีคำสงวนใดๆ ทั้งสิ้น ดังนั้น DO, WHILE หรือคำอื่นๆ ซึ่ง อธิบายโครงสร้าง อาจนำมาเป็นไอเดนติไฟเออร์ได้

ตัวอย่าง ข้อความสั่ง ภาษา FORTRAN

```
WHILE = 2.2
```

ถูกต้องอย่างสมบูรณ์ ภาษา FORTRAN โครงสร้างโทเค็น และวากยสัมพันธ์ เป็นสิ่งคู่กัน

ตัวอย่างสุดท้าย ของ โครงสร้างศัพท์ คือ บทนิยาม ของ ข้อตกลงโทเค็น ของภาษา C จาก คู่มือการใช้ใน Kernighan และ Richie (1988) กล่าวดังนี้:

โทเค็น แบ่งออกเป็น หก ประเภท ได้แก่ ไอเดนติไฟเออร์ คำหลัก ค่าคงที่ สายอักขระ ตัวดำเนินการ และตัวคั่นอื่นๆ (: identifiers, keywords, constants, string literals, operators, and other separators.)

ส่วน blanks, horizontal and vertical tabs, newlines, formfeeds, and comments ตามที่ได้อธิบายนั้น (รวมเรียกว่า "white space") are ignored ยกเว้น เมื่อนำมาใช้แยกโทเค็น

white space บางตัวต้องใช้แยกไอเดนติไฟเออร์, คำหลัก และค่าคงที่ ซึ่งอยู่ติดกัน ถ้า input stream ถูกแยก เป็น โทเค็นต่างๆ ตามจำนวนอักขระซึ่งกำหนดให้โทเค็นถัดไป คือ สายของอักขระ ความยาวสูงสุด ซึ่งประกอบขึ้นเป็นหนึ่งโทเค็น ดังนั้น ภาษา C จึงติดกับ หลักของสายอักขระย่อยความยาวสูงสุด

4.2 ไวยากรณ์ไม่พึ่งบริบท และรปแบบแบกคัส-เนาร์

(Context-freegrammars and BNFs)

เราจะเริ่มต้นบทนิยามของไวยากรณ์ และ BNFs ด้วยตัวอย่าง ในภาษาอังกฤษนั้น ประโยคอย่างง่ายประกอบด้วย a noun phrase และ a verb phrase ตามด้วย a period ซึ่งสามารถแสดงออกดังนี้

1. <sentence> ::= <noun-phrase><verb-phrase>.

ต่อไปอธิบายโครงสร้างของ noun phrase และ verb phrase ดังนี้

2. <noun-phrase> ::= <article><noun>
3. <article> ::= a I the
4. <noun> ::= girl I dog
5. <verb-phrase> ::= <verb><noun-phrase>
6. <verb> ::= sees I pets

กฎไวยากรณ์เหล่านี้ แต่ละข้อประกอบด้วย string หนึ่งตัว อยู่ภายในเครื่องหมาย "< >" (เป็นชื่อของโครงสร้าง ซึ่งกำลังอธิบาย) ตามด้วยสัญลักษณ์ "::=" ซึ่งอ่านว่า "consists of" หรือ "is the same as" หรือ "is defined as" และจากนั้นเป็นลำดับ ของชื่อและสัญลักษณ์อื่นๆ

เครื่องหมาย "< >" แยกความแตกต่าง ชื่อของโครงสร้าง (structure names) ออกจาก คำจริง (actual words) หรือ โทเค็น ซึ่งอาจปรากฏในภาษานั้น ตัวอย่างเช่น ในภาษา Pascal นั้น "<program>" จะหมายถึง โครงสร้างโปรแกรมทั้งหมด (will stand for a whole program structure) ในขณะที่คำว่า "program" หมายถึง โทเค็นตัวแรก ในทุกๆ <program> เขียนดังนี้

<program> ::= program ...

สัญลักษณ์ "::=" เป็น <u>สัญลักษณ์อภิภาษา</u> (metasymbol) ซึ่งใช้แยก ส่วนทางซ้ายมือ ของกฎ ออกจาก ส่วนทางด้านขวามือ ของกฎ

เครื่องหมาย "< >" เป็นสัญลักษณ์อภิภาษาเช่นเดียวกับ vertical bar "|" ซึ่งหมายถึง "or" หรือ ทางเลือก (alternation) ดังนั้น กฎข้อ 6 กล่าวว่า verb อาจจะเป็นคำว่า "sees" หรือคำว่า "pets"

บางครั้ง สัญลักษณ์อภิภาษา อาจจะเป็น สัญลักษณ์จริง (actual symbol) ในภาษานั้นด้วย เช่นกัน ในกรณีเช่นนั้น เราจะใส่เครื่องหมายคำพูดให้กับ สัญลักษณ์จริง เพื่อ แยกมันออกจากสัญลักษณ์อภิภาษา สิ่งนี้เป็นความคิดที่ดี สำหรับ สัญลักษณ์พิเศษ เช่น เครื่องหมายกำกับวรรคตอน เมื่อไม่ได้ใช้เป็นสัญลักษณ์อภิภาษา

ตัวอย่างเช่น กฎข้อ 1 มี period หนึ่งตัว อยู่ท้ายสุด ขณะนั้น period ไม่ใช่ ส่วนของสัญลักษณ์อภิภาษาใดๆ ซึ่งกำลังอธิบาย ดังนั้น จึงอาจผิดพลาดได้ง่าย วิธีที่ดีกว่าคือ เขียนกฎข้อ 1 ดังนี้

<sentence> ::= <noun-phrase><verb-phrase> '.'

(เพราะฉะนั้น ขณะนี้ เครื่องหมายคำพูด เป็นสัญลักษณ์อภิภาษาด้วย)

<u>หมายเหตุ</u> ในตำราบางเล่ม ผู้แต่งอาจใช้ ตัวเอน ตัวใหญ่ เพื่อแยกความแตกต่างระหว่าง สัญลักษณ์ สัญลักษณ์อภิภาษา และชื่อ

ประโยคที่ถูกต้องใดๆ จะต้องเป็นไปตาม ไวยากรณ์ที่กล่าวข้างต้น คือสามารถสร้างขึ้นได้ตามขั้นตอน ดังนี้ :

เริ่มต้นด้วย สัญลักษณ์ <sentence> และทำต่อไป เพื่อแทนที่ ส่วนทางซ้ายโดยทางเลือกต่างๆ ของส่วนทางขวามือ ในกฎที่กล่าวมาแล้ว กรรมวิธีนี้ สร้างการแปลง (derivation) ในภาษา (This process creates a **derivation** in the language.)

ดังนั้น เราสามารถสร้าง (construct) หรือ แปลง (derive) ประโยค "the girl sees a dog." ดังนี้

<sentence> ——> <noun-phrase><verb-phrase>. (กฎข้อ 1)
——> <article><noun><verb-phrase>. (กฎข้อ 2)
---> the <noun>< verb-phrase>. (กฎข้อ 3)
——> the girl <verb-phrase>. (กฎข้อ 4)
——> the girl <verb><noun-phrase>. (กฎข้อ 5)
——> the girl sees <noun-phrase>. (กฎข้อ 6)
——> the girl sees <article><noun>. (กฎข้อ 2)
——> the girl sees a <noun>. (กฎข้อ 3)
——> the girl sees a dog. (กฎข้อ 4)

ในทางตรงกันข้าม เราสามารถเริ่มต้น ด้วยประโยค "the girl sees a dog." และทำงานย้อนหลัง ผ่านการแปลง จนถึง <sentence> เป็นการแสดงให้เห็นว่า ประโยคที่กำหนดให้นั้นเป็น ประโยคถูกต้องในภาษา

ไวยากรณ์อย่างง่ายนี้ แสดงให้เห็นคุณสมบัติส่วนใหญ่ ของไวยากรณ์ ของภาษาโปรแกรม แต่โปรดสังเกตว่า ไม่ใช่ ประโยคถูกต้องทั้งหมด ที่จริงๆ แล้วจะมีความหมาย (Note that not all legal sentences actually make sense.) ตัวอย่างเช่น "the dog pets the girl."

นอกจากนี้แล้ว ยังมีข้อผิดพลาดปลีกย่อยดังนี้ ในภาษาอังกฤษนั้น article ซึ่งปรากฏที่ตอนต้นของประโยค จะต้องเป็นตัวใหญ่ คุณสมบัติของ **ตำแหน่ง** (positional)

เช่นนี้ เป็นเรื่องยากที่จะ ทำงาน โดยใช้ ไวยากรณ์ไม่พึ่งบริบท

ต่อไปนี้ เป็นการสรุปบทนิยามของสิ่งต่างๆ ซึ่งเราได้กล่าวมาแล้ว **ไวยากรณ์ไม่พึ่งบริบท** (A context-free grammar) ประกอบด้วย

1. ชุดของกฎไวยากรณ์ (a series of grammar rules) อธิบายดังนี้ กฎต่างๆ จะประกอบด้วย ส่วนทางซ้ายมือ ซึ่งเป็น ชื่อโครงสร้างหนึ่งชื่อ จากนั้นเป็น เครื่องหมายอภิภาษา "::=" ตามด้วย ส่วนทางขวามือ ซึ่งประกอบด้วย ลำดับ ของ items ซึ่งอาจจะเป็น สัญลักษณ์ หรือ ชื่อโครงสร้างอื่นๆ
2. ชื่อของโครงสร้าง (the names for structures) เรียกว่า **nonterminals** ตัวอย่างเช่น <sentence> เพราะว่า ชื่อโครงสร้างเหล่านี้ สามารถ แบ่งย่อยให้เป็นโครงสร้างต่อไปได้อีก
3. คำ หรือ สัญลักษณ์ โทเค็น (the words or token symbols) เรียกว่า **terminals** เพราะว่า คำเหล่านี้ ไม่สามารถ แบ่งย่อยได้อีก
4. กฎไวยากรณ์ (Grammar rules) เรียกอีกอย่างหนึ่งว่า productions เพราะว่ามัน "produce" strings ของภาษา โดยใช้การแปลง

productions อยู่ในรูปของ Backus - Naur form ถ้ามันถูกกำหนดให้โดยใช้เฉพาะสัญลักษณ์อภิภาษา "::=", "|", "<" และ ">" เท่านั้น (บางครั้งเครื่องหมายวงเล็บเล็ก อนุญาตให้ใช้ได้เพื่อจัดกลุ่ม ของ items เข้าด้วยกันได้)

ในตัวอย่างข้างต้น มี terminal 7 ตัว ได้แก่ "girl", "dog", "sees", "pets", "the", "a", และ "." มี nonterminal 6 ตัว และมี 6 productions โดยทั่วไปแล้ว จำนวน production ใน ไวยากรณ์ไม่พึ่งบริบท จะมีเท่ากับ จำนวน nonterminal

ถึงแม้ว่า เราอาจจะขจัด สัญลักษณ์อภิภาษา "|" ออกไป โดยเขียนทางเลือกแยกต่างหากจากกัน เช่น

```
<noun> ::= girl
<noun> ::= dog
```

ซึ่งจะทำให้ nonterminal แต่ละตัว สมนัยกับ จำนวน production มากเท่ากับจำนวนทางเลือก

ทำไมจึงเรียกว่า ไวยากรณ์ ไม่พึ่งบริบท เหตุผลง่ายๆ คือ nonterminals ซึ่งปรากฏ ทางซ้ายมือ ของ productions มีหนึ่งตัวเท่านั้น สิ่งนี้หมายความว่า nonterminal แต่ละตัว สามารถถูกแทนที่ด้วย ทางเลือกของด้านขวามือ ตัวใดก็ได้ ไม่ว่า nonterminal นั้นจะปรากฏที่ใด พูดอีกอย่างหนึ่งคือ ไม่มี **บริบท** หรือ **อรรถาธิบาย** ภายใต้สิ่งซึ่งเกิดขึ้นเฉพาะการแทนที่เท่านั้น (In other words, there is no **context** under which only certain replacements can occur.)

ตัวอย่างเช่น ในไวยากรณ์ที่เราเพิ่งอภิปราย มันจะรู้เรื่องเมื่อใช้ verb คำว่า "pets" เฉพาะเมื่อ ประธานเป็น "girl" เท่านั้น สิ่งนี้เรียกว่า **การพึ่งบริบท*** (a context-sensitivity)

เราสามารถเขียน ไวยากรณ์พึ่งบริบท โดยยอมให้ "context string" ปรากฏทางซ้ายมือ ของ กฎไวยากรณ์ แต่บางคน อาจถือว่า การพึ่งบริบท เป็นหัวข้อวากยสัมพันธ์ (syntactic issue) เราจะยอมรับสิ่งนี้ อย่างไรก็ตาม สิ่งใดก็ตามซึ่งไม่สามารถแสดงออก ด้วย ไวยากรณ์ไม่พึ่งบริบท สิ่งนั้นคือ อรรถศาสตร์ ไม่ใช่ หัวข้อวากยสัมพันธ์ (We shall adopt the view, however, that anything not expressible using context-free grammars is a semantic, not a syntactic issue.) มีบางสิ่ง ซึ่งสามารถแสดงออกด้วยไวยากรณ์ไม่พึ่งบริบทได้ แต่บ่อยครั้ง เป็นเรื่องดีกว่าที่จะทิ้งให้เป็น การอธิบายความหมาย เพราะว่า มันเกี่ยวข้องกับการเขียน productions พิเศษ จำนวนมาก

ตัวอย่างของ การพึ่งบริบท (context-sensitivity) ตามข้อสังเกตที่ว่าคือ article ซึ่งปรากฏ ตอนต้นของประโยค ใน ไวยากรณ์ข้างต้น ควรจะเป็นตัวใหญ่ วิธีหนึ่งในการทำสิ่งนี้ คือ เขียนกฎข้อ 1 ใหม่ ดังนี้

<sentence> ::= <beginning><noun-phrase><verb-phrase> '.'

---

* หรือ ตัวอย่างเช่น ภาษา Pascal กล่าวว่า ตัวแปรทุกตัวในโปรแกรม ต้องมีการประกาศ (declared) หรือ ภาษา FORTRAN กล่าวว่า ชื่อทุกชื่อ ซึ่งขึ้นต้นด้วยตัวอักษรตัวใดตัวหนึ่งใน I ถึง N จะถือว่าเป็น ตัวแปรชนิด integer ส่วนกรณีอื่นๆ ถือว่าเป็น ตัวแปรชนิด real เป็นต้น

จากนั้น ใส่กฎ พึ่งบริบท ดังนี้ :

<beginning><article> ::= The I A

ขณะนี้ การแปลง จะ เป็นดังนี้

<sentence> ——> <beginning><noun-phrase><verb-phrase>.

(กฎข้อ 1 ใหม่)

——> <beginning><article><noun><verb-phrase>.

(กฎข้อ 2)

——> The <noun><verb-phrase>. (กฎพึงบริบท ข้อใหม่)

——> . . .

ไวยากรณ์ไม่พึงบริบท ได้มีการศึกษาอย่างกว้างขวาง โดยนักทฤษฎีภาษาแบบทางการ (formal language theorists) และปัจจุบัน เป็นที่เข้าใจอย่างดีแล้วว่ามันเป็นธรรมชาติที่จะแสดงออก วากยสัมพันธ์ ของ ภาษาโปรแกรมใดๆ ในรูปแบบของ BNF ได้ จริงๆ แล้ว การทำเช่นนี้ ทำให้การเขียนตัวแปลภาษา สำหรับภาษาโปรแกรมง่ายขึ้น เพราะว่าขั้นวิเคราะห์กระจายสามารถเป็นไปอัตโนมัติ (ดูหัวข้อ 4.6)

ตัวอย่างเบื้องต้น ของการใช้ ไวยากรณ์ไม่พึ่งบริบท ในภาษาโปรแกรม ได้แก่ การอธิบาย นิพจน์คำนวณ จำนวนเต็ม อย่างง่าย ด้วย การบวกและการคูณ ซึ่งกำหนดให้ในรูป 4.1

(A typical simple example of the use of a context-free grammar in programming languages is the description of simple integer arithmetic expressions with addition and multiplication given in Figure 4.1.)

```
<exp>    ::= <exp> + <exp> | <exp> * <exp> |
             (<exp>) I <number>
<number> ::= <number><digit> I <digit>
<digit>  ::= 0 1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
```

รูป 4.1 A simple arithmetic expression grammar

โปรดสังเกต ธรรมชาติการเรียกซ้ำ ของกฎ : นิพจน์ หนึ่งชุดอาจจะเป็น ผลบวกของนิพจน์ สองชุด หรือ ผลคูณของนิพจน์ สองชุด ซึ่งแต่ละชุดอาจแบ่งต่อไปอีก เป็นผลบวก หรือ ผลคูณ สุดท้าย กระบวนการนี้ ต้องจบโดยการเลือก ทางเลือก <number> หรือเราอาจจะไม่ได้มาถึงสาย (string) ของ terminals

โปรดสังเกตุว่า การเรียกซ้ำ ในกฎ <number > ใช้เพื่อก่อกำเนิด (generate) ลำดับของเลขโดดซ้ำกัน ตัวอย่างเช่น เลข 2 3 4 ถูกสร้าง ดังนี้

<number> ——> <number><digit>
——> <number><digit><digit>
——> <digit><digit><digit>
——> 2<digit><digit>
——> 23<digit>
——> 234

ต่อไปคือ รูป 4.2 แสดงการเริ่มต้น ของการอธิบาย BNF สำหรับภาษา Pascal

```
<program>              ::=     <program-heading>';'<program-block>'.'
<program-heading>      ::= program  <identifier>  I  program
                           <identifier>   '('<program-parameter>')'
<program-block>        ::= <block>
<program-parameters>   ::=   <identifier-list>
<block>                ::=    <label-declaration-part>
                           <constant-definition-part>
                           <type-definition-part>
                           <variable-declaration-part>
                           <procedure-and-function-declaration-part>
                           <statement-part>
```

```
<label-declaration-part>  ::=  label <label-list> ';' I <empty>
<label-list>              ::=  <label-list> ',' <label> I <label>
                          . . .
                          . . .
```

รูป 4.2 BNFs บางส่วน สำหรับภาษา Pascal

4.2.1 กฎ BNFs คือสมการ

(BNFs rules as Equations)

ทางเลือกวิธีหนึ่งของการอธิบายว่า กฎ BNF สร้างสาย (strings) ของภาษา ทำได้อย่างไรนั้น เป็นดังนี้

กำหนด กฎไวยากรณ์ให้หนึ่งข้อ เช่น

<exp> ::= <exp> + <exp> I <number>

(นามธรรมจาก รูป 4.1) ให้ E เป็นเซตของสาย ซึ่งแทนนิพจน์ ส่วน N หมายถึงเซตของสายซึ่งแทนเลข กฎไวยากรณ์ข้างต้น อาจมองในรูป **สมการเซต** (set equation) ดังนี้

E = E + E U N

เมื่อ E + E หมายถึง เซตสร้างโดยการต่อกัน ของสายทั้งหมดจาก E ด้วยสัญลักษณ์ "+" และจากนั้น สายทั้งหมด จาก E และ "U" หมายถึง ผลผนวกของเซต สมมติว่า เซต N ถูกสร้างเรียบร้อยแล้ว สิ่งนี้ แทน สมการการเรียกซ้ำ สำหรับเซต E เซต E เล็กที่สุด จะเป็นไปตามสมการนี้ สามารถเอามาเป็น เซตนิยามโดยกฎไวยากรณ์ คาดได้ว่า สิ่งนี้ ประกอบด้วยเซต

N U N + N U N + N + N U N + N + N + N U ...

และเราสามารถพิสูจน์ได้ว่า สิ่งนี้เป็นเซตเล็กที่สุด ซึ่งเป็นไปตามสมการที่กำหนดให้ (ดูแบบฝึกหัดข้อ 40) ผลเฉลยของสมการเรียกซ้ำ เกิดขึ้นบ่อยมาก ใน การอธิบาย แบบทางการ ของภาษาโปรแกรม และเป็นหัวข้อหลักของการศึกษา ในวิชาทฤษฎีของภาษาโปรแกรม ซึ่งเรียกว่า **จุดคงที่น้อยที่สุด** (least fixed points)

4.3 ต้นไม้วิภัช และต้นไม้วากยสัมพันธ์แบบนามธรรม

(Parse trees and Abstract syntax trees)

วากยสัมพันธ์ สร้าง โครงสร้าง ไม่ใช่ความหมาย แต่ความหมายของประโยค (หรือ โปรแกรม) ต้องเกี่ยวข้องกับวากยสัมพันธ์

(Syntax establishes structure, not meaning. But the meaning of a sentence (or program) must be related to its syntax.)

ในภาษาอังกฤษ หนึ่งประโยคมี a subject และ a predicate ซึ่งเป็นความคิดเชิงความหมาย เพราะว่า subject (the "actor") และ predicate (the "act ion") บอกความหมายของประโยค

subject ปกติ จะอยู่ตอนต้นของประโยค และกำหนดให้ด้วย a noun phrase ดังนั้น ในวากยสัมพันธ์ ของประโยคภาษาอังกฤษ noun phrase ต้องวางที่ตำแหน่งแรก และสิ่งนั้นคือ subject

ในทำนองเดียวกัน ในไวยากรณ์ของนิพจน์ เมื่อเราเขียน

<exp> ::= <exp> + <exp>

เราหวังว่า ให้บวกค่าของนิพจน์ ทางขวามือ สองชุด เข้าด้วยกัน เพื่อให้ได้ค่าของ นิพจน์ทางซ้ายมือ กรรมวิธีของการผูกติดความหมาย ของ ตัวสร้างหนึ่ง กับโครงสร้างวากยสัมพันธ์ของมัน สิ่งนี้เรียกว่า **วากยสัมพันธ์-ต่อตรงกับ อรรถศาสตร์**

(This process of attaching the semantics of a construct to its syntactic structure is called **syntax-directed semantics.**)

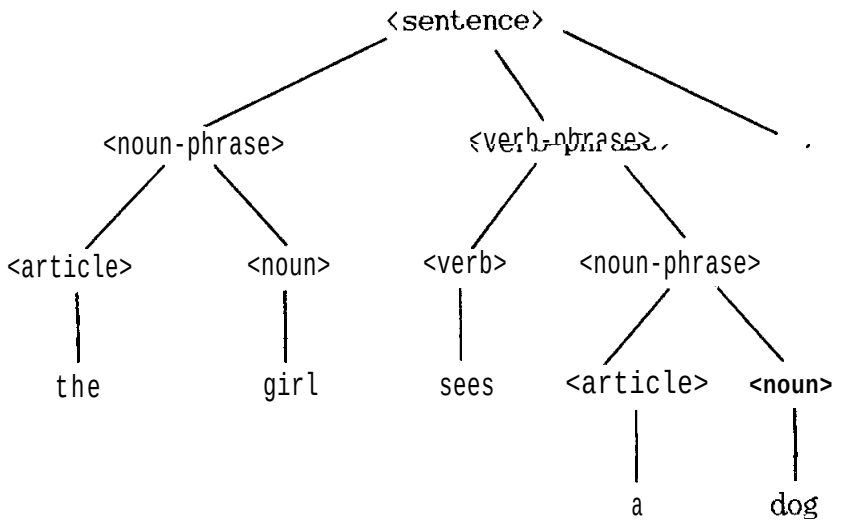
เราต้องสร้างวากยสัมพันธ์ เพื่อให้มันสะท้อน ความหมาย สุดท้ายเราจะผูกติดกับมัน มากที่สุดเท่าที่เป็นไปได้ (วากยสัมพันธ์-ต่อตรงกับ ความหมาย ซึ่งอาจเรียกง่ายๆ ว่า ความหมาย-ต่อตรงกับ วากยสัมพันธ์)

เพื่อใช้ โครงสร้างวากยสัมพันธ์ ของโปรแกรม บอกความหมายของมัน เราต้องมีวิธีแสดงออกของโครงสร้างนี้ เช่น การหาโดยการแปลง

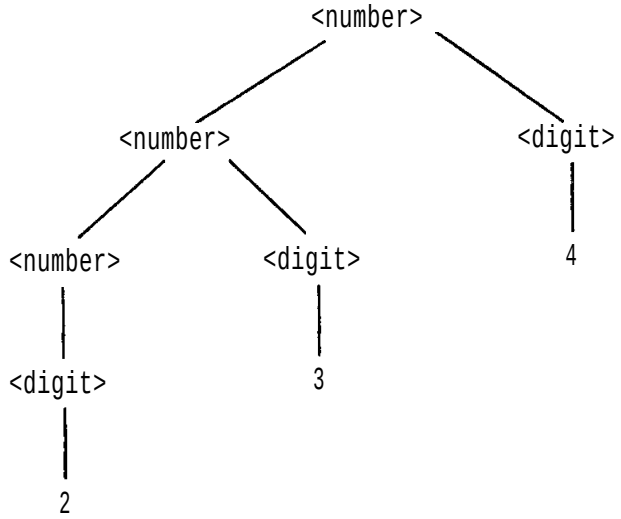
วิธีมาตรฐาน สำหรับทำสิ่งนี้ คือ **ต้นไม้วิภัช หรือต้นไม้วิเคราะห์กระจาย** (parse tree)

ต้นไม้วิภัช อธิบายกรรมวิธี การแทนที่ ของ การแปลงด้วยภาพ (The parse tree describes graphically the replacement process ina derivation.)

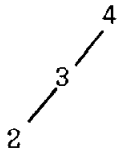
ตัวอย่าง ต้นไม้วิภัช สำหรับ ประโยค "the girl sees a dog."



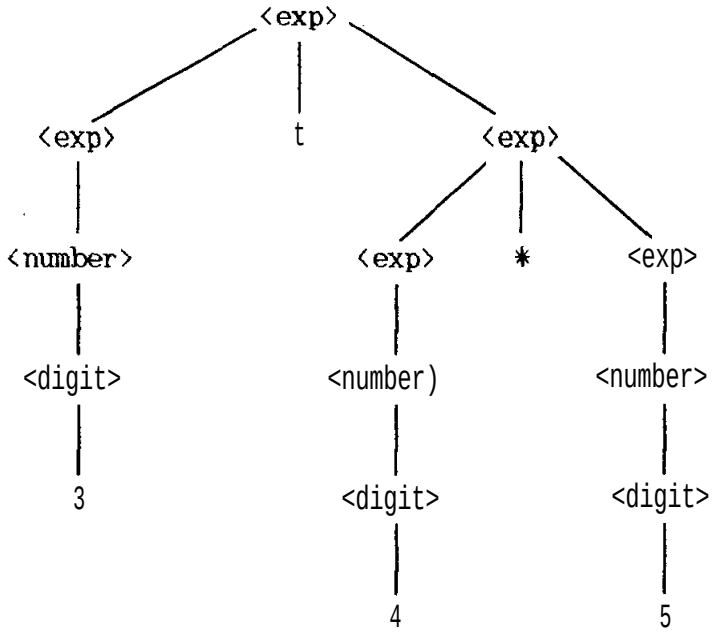
ในทำนองเดียวกัน ต้นไม้วิภัช สำหรับเลข 234 ในไวยากรณ์ของนิพจน์ คือ



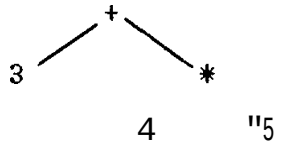
ต้นไม้วิภัช ถูกระบุ โดย nonterminals ที่ โหนดภายใน (interior nodes) และ terminals ที่ **จุดแตกใบ** (leaf) ทั้งนี้ terminals และ nonterminals ทั้งหมดในการแปลง รวมอยู่ในต้นไม้วิภัช แต่ terminals และ nonterminals อาจไม่จำเป็นต้องเป็นทั้งหมด เพื่อบอกความบริบูรณ์ของ โครงสร้างวากยสัมพันธ์ ของ นิพจน์ หรือประโยค ตัวอย่างเช่น โครงสร้างของเลข 234 บอกได้อย่างบริบูรณ์ จากต้นไม้



และต้นไม้วิภัช สำหรับ 3 + 4 * 5 เช่น



อาจทำให้เล็กลงเป็นต้นไม้



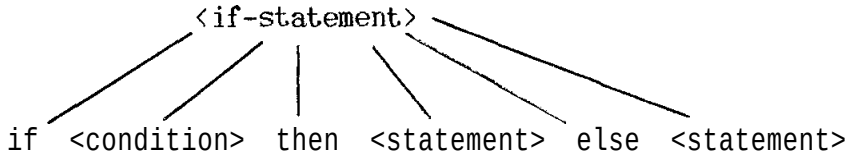
ต้นไม้เช่นนี้เรียกว่า **ต้นไม้วากยสัมพันธ์แบบนามธรรม** (abstract syntax trees) หรือ **ต้นไม้วากยสัมพันธ์** (syntax trees) เพราะว่ามันบอก (abstract) เฉพาะโครงสร้างสำคัญของต้นไม้วิภัช

ต้นไม้วากยสัมพันธ์แบบนามธรรม อาจเอา terminals ซึ่งครั้งหนึ่ง มีซ้ำกันในโครงสร้างของต้นไม้ ออกไป

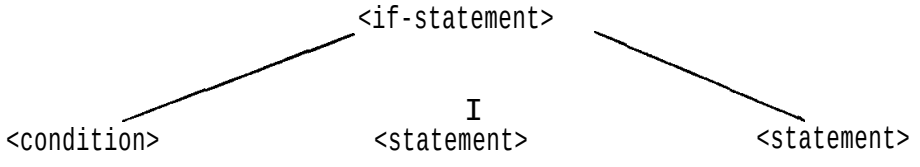
ตัวอย่าง กฎไวยากรณ์

<if-statement> ::= if <condition> then <statement>
else <statement>

จะให้ต้นไม้วิภัช ดังนี้



และต้นไม้วากยสัมพันธ์แบบนามธรรม ดังนี้



มันเป็นไปได้ ที่จะเขียนกฎ สำหรับวากยสัมพันธ์แบบนามธรรม ในวิธีที่คล้ายกับกฎ BNF สำหรับ วากยสัมพันธ์ปกติ แต่ในที่นี้ไม่ได้ทำเช่นนั้น บางครั้ง วากยสัมพันธ์ปกติ ถูกแยกต่างหาก จาก วากยสัมพันธ์นามธรรม ซึ่งเรียกว่า **วากยสัมพันธ์แบบรูปธรรม** (concrete syntax)

ตัวแปลภาษา บ่อยครั้งจะสร้างต้นไม้วากยสัมพันธ์ มากกว่าเป็นต้นไม้วิภัช เต็มรูป เพราะว่า มันกระชับ และแสดงความสำคัญ ของ โครงสร้าง ได้มากกว่า

4.4 ความกำกวม การสลับที่ และ การทำก่อน

(Ambiguity, Associativity and **Precedence**)

การแปลงที่แตกต่างกัน สองชุด สามารถนำไปสู่ ต้นไม้วิภัช ต้นเดียวกันได้ (Two different derivations can lead to the same parse tree.) ตัวอย่างเช่น ในการแปลง เลข 234 ในหัวข้อที่แล้ว เราอาจเลือก เพื่อแทนที่ <digit> เป็นอันดับแรก ดังนี้

<number> ——> <number><digit>

——> <number> 4

——> <number><digit> 4

——> <number> 34

. . .

ซึ่งยังคงได้ ต้นไม้วิภัชเหมือนเดิม อย่างไรก็ตาม การแปลงที่แตกต่างกัน สามารถนำไปสู่ ต้นไม้วิภัชที่แตกต่างกันได้ด้วย ตัวอย่างเช่น ถ้าเราสร้าง 3 + 4 * 5 จากไวยากรณ์ นิพจน์ ของรูป 4.1 เราสามารถใช้การแปลง

<exp> ——> <exp> + <exp>

——> <exp> + <exp> * <exp>

(แทน <exp> ชุดที่สองด้วย <exp> * <exp>)

——> <number> + <exp> * <exp>

——> . . .

หรือ การแปลง

<exp> ——> <exp> * <exp>

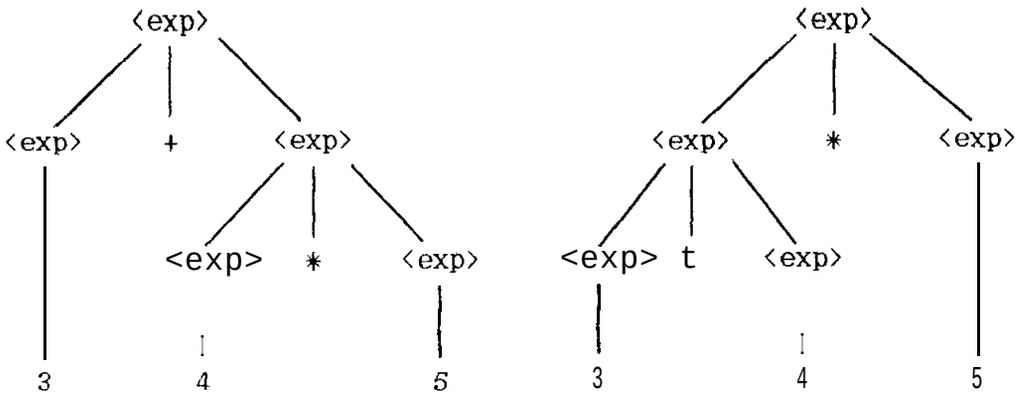
——> <exp> t <exp> * <exp>

(แทน <exp> ชุดแรกด้วย <exp> t <exp>)

——> <number> t <exp> * <exp>

——> . . .

ซึ่งการแปลง สองชุด ข้างต้นนี้ นำไปสู่ต้นไม้วิภัช สองชุด



(ชุดแรก) (ชุดที่สอง)

และ ต้นไม้วากยสัมพันธ์แบบนามธรรม สองชุด

ไวยากรณ์ เช่นที่กำหนดในรูป 4.1 สำหรับ ต้นไม้วิภัช หรือ ต้นไม้วากยสัมพันธ์ที่แตกต่างกันสองชุด อาจเป็นไปได้สำหรับ string ชุดเดียวกัน หมายถึง **ความกำกวม** (ambiguous)

ไวยากรณ์ที่กำกวม นำเสนอสิ่งยาก เพราะว่า โครงสร้าง แสดงออกไม่ชัดเจน เพื่อให้เป็นประโยชน์ ไวยากรณ์นั้นต้องปรับปรุงแก้ไขใหม่ (revised) เพื่อความกำกวม หรือ **กฎความไม่กำกวม** ต้องถูกกล่าวขึ้นเพื่อสร้างว่า โครงสร้างชุดใดที่ต้องการ หมายถึง (a **disambiguating rule** must be stated to establish which structure is meant.)

สำหรับนิพจน์ 3 + 4 * 5 นั้น ต้นไม้วิภัชชุดใดถูกต้อง ถ้าเราคิดถึงอรรถศาสตร์ซึ่งผูกติดกับนิพจน์ เราสามารถเข้าใจความหมายของการตัดสินใจนี้ว่าคืออะไร ต้นไม้วากยสัมพันธ์ต้นแรก แสดงว่า ตัวดำเนินการคูณ "*" ใช้กับ 4 และ 5 (ผลลัพธ์คือ 20)

จากนั้นจึงนำผลลัพธ์นี้ บวกกับ 3 ได้คำตอบ 23 ในทางตรงกันข้าม ต้นไม้วากยสัมพันธ์ชุดที่สอง สิ่งแรก บวก 3 กับ 4 (ผลลัพธ์คือ 7) จากนั้นจึงคูณด้วย 5 คำตอบคือ 35 ดังนั้น การดำเนินการ ซึ่งถูกใช้ในอันดับแตกต่างกัน และความหมายของผลลัพธ์ จะแตกต่างกันด้วย

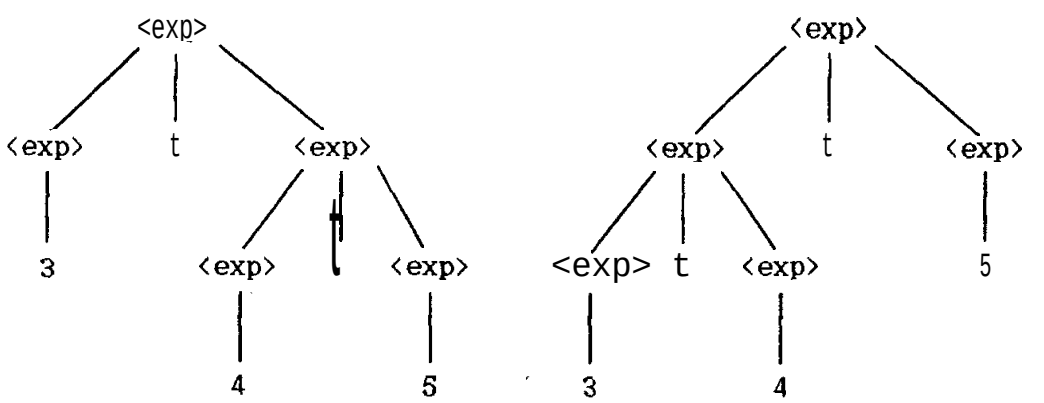
(Thus the operations are applied in a different order, and the resulting semantics are quite different.)

ถ้าเราเอาความหมายปกติ ของนิพจน์ 3 + 4 * 5 จากวิชาคณิตศาสตร์ เราควรเลือก ต้นไม้ต้นแรก มากกว่าที่จะเป็นต้นที่สอง เพราะว่าการคูณ มี การทำก่อน สูงกว่าการบวก (since multiplication has precedence over addition) สิ่งนี้ เป็นการเลือกปกติ ในภาษาโปรแกรม ถึงแม้ว่าบางภาษา (เช่น APL) มีการเลือกที่แตกต่างออกไป เราสามารถแสดงออกความจริงที่ว่า การคูณ มี การทำก่อน สูงกว่าการบวกได้อย่างไร? เราควร กล่าวกฎไม่กำกวม แยกต่างหากจากไวยากรณ์ หรือเราควรปรับปรุงแก้ไข (revise) ไวยากรณ์ วิธีปกติ ของการปรับปรุงแก้ไขไวยากรณ์ คือ เขียนกฎไวยากรณ์ใหม่ (เรียกว่า "<term>") ซึ่งสร้าง ~~การต่อเรียงการทำก่อน~~ (precedence cascade) บังคับให้ การจับคู่ ของ "*" ที่จุดต่ำกว่า ใน ต้นไม้วิภัช

<exp> ::= <exp> + <exp> | <term>

<term> ::= <term> * <term> | (<exp>) | <number>

แต่การแก้ปัญหาความกำกวม ยังไม่สมบูรณ์ เพราะว่า กฎ สำหรับ <exp> ยังคงวิเคราะห์กระจาย นิพจน์ 3 + 4 + 5 เป็น (3 + 4) + 5 หรืออาจเป็น 3 + (4 + 5) พูดอีกอย่างหนึ่งคือ เราสามารถทำการบวก ให้เป็น ~~การสลับที่แบบขวา~~ (right-associative) หรือ ~~การสลับที่แบบซ้าย~~ (left-associative) ได้ :



กรณีของการบวก สิ่งนี้ไม่มีผลกระทบกับผลลัพธ์ แต่ถ้าเรารวมการลบ (subtraction) ไว้ด้วย มีการกระทบแน่นอน ตัวอย่างเช่น นิพจน์ 8 - 4 - 2

กรณี ถ้า "-" เป็น การสลับที่แบบซ้าย ผลลัพธ์คือ 8 - 4 - 2 = 2

แต่ถ้า "-" เป็น การสลับที่แบบขวา ผลลัพธ์คือ 8 - 4 + 2 = 6

ดังนั้นจึงจำเป็นต้องแทนกฎ

<exp> ::= <exp> + <exp>

ด้วยกฎ

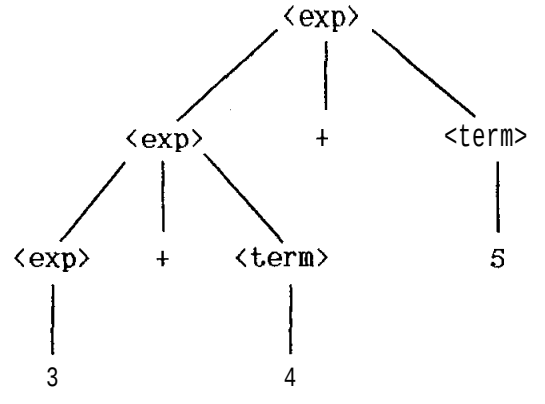
<exp> ::= <exp> + <term> ——— (1)

หรือกฎ

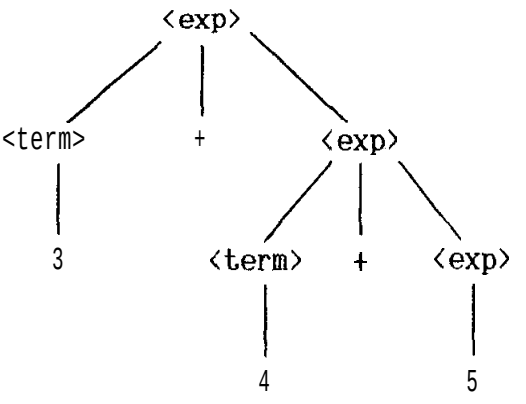
<exp> ::= <term> + <exp> ——— (2)

ชุดไหน ใน กฎสองข้อนี้ เป็นชุดที่ถูกต้อง กฎข้อแรกนั้น เป็น <u>การเรียกซ้ำแบบซ้าย</u> (left-recursive) ในขณะที่กฎข้อสอง เป็น <u>การเรียกซ้ำแบบขวา</u> (right-recursive)

กฎการเรียกซ้ำแบบซ้าย สำหรับการดำเนินการ ทำให้มันเป็นการสลับที่แบบซ้าย เช่น ในต้นไม้วิภัช ข้างล่างนี้



ในขณะที่ กฎการเรียกซ้ำแบบขวา ทำให้ การดำเนินการเป็นการสลับที่แบบขวา



ไวยากรณ์ ชุดแก้ไขปรับปรุงใหม่ สำหรับ นิพจน์คำนวณอย่างง่าย ซึ่งแสดงออกทั้ง การทำก่อน และการสลับที่ กำหนดให้แล้วในรูป 4.3

ขณะนี้ BNF สำหรับ นิพจน์คำนวณอย่างง่าย ไม่กำกวม (การพิสูจน์ ของกฎเหล่านี้ ต้องใช้ เทคนิคชั้นสูงมากขึ้น จากทฤษฎีของการวิเคราะห์กระจาย)

นอกจากนี้แล้ว ต้นไม้วิภัช จะสมนัยกับความหมายของการดำเนินการคำนวณ เช่นที่นิยามใช้ปกติ บางครั้ง กรรมวิธีของ การเขียน ไวยากรณ์ชั้นใหม่ เพื่อขจัดความกำกวม เป็นสาเหตุที่ทำให้ ไวยากรณ์ มีความซับซ้อนอย่างมาก แต่กรณีต่างๆ เช่นนั้น เราชอบที่จะกล่าวไวยากรณ์ที่ไม่กำกวม มากกว่า (ดู การอภิปราย **if-statement** ในบทที่ 7)

```
<exp>    ::= <exp> + <term> | <term>
<term>   ::= <term> * <factor> | <factor>
<factor> ::= (<exp>) I <number>
<number> ::= <number><digit> I <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

รูป 4.3 ไวยากรณ์ชุด แก้ไขปรับปรุง สำหรับนิพจน์คำนวณอย่างง่าย

4.5 อีบีเอ็นเอฟ และแผนภาพวากยสัมพันธ์

(EBNFs and syntax diagrams)

จากกฎไวยากรณ์ ในหัวข้อที่แล้ว

<number> ::= <number><digit> | <digit>

ก่อกำเนิด (generate) เลขหนึ่งจำนวน เป็นลำดับของเลขโดด ดังนี้ :

<number> ——> <number><digit>

——> <number><digit><digit>

——> <number><digit><digit><digit>

. . .

——> <digit> . . . <digit>

(arbitrary repetitions of digits)

ในทำนอง เดียวกัน กฎ

<exp> ::= <exp> + <term> I <term>

ก่อกำเนิด นิพจน์ หนึ่งชุด เป็น ลำดับ ของ term แยกกันด้วยเครื่องหมาย "+" ดังนี้

<exp> ——> <exp> + <term>

——> <exp> + <term> t <term>

——> <exp> + <term> t <term> t <term>

. . .

——> <term> + . . . t <term>

สถานะการณ์เช่นนี้ เกิดขึ้นบ่อยมาก ดังนั้น สัญกรณ์พิเศษหนึ่งตัว สำหรับกฎไวยากรณ์เช่นนี้ ถูกยอมรับว่า แสดงธรรมชาติการทำซ้ำของโครงสร้างของมันได้ชัดเจนมากกว่า

<number> ::= <digit>{<digit>}

และ

<exp> ::= <term>{+ <term>}

สัญกรณ์นี้ (this notation) วงเล็บปีกกา ({ }) หมายถึง "zero or more repetitions of" ดังนั้น กฎนี้ แสดงว่า เลข หมายถึง ลำดับของเลขโดดหนึ่งตัว หรือ เลขโดด มากกว่าหนึ่งตัว และกฎที่แสดงว่า นิพจน์ หมายถึง term ตามด้วยการกระทำซ้ำของ "+" และอีกหนึ่งเทอม อาจจะไม่มี หรือมีหลายชุดได้ (zero or more repetition of a "+" and another term.) สัญกรณีนี้ วงเล็บปีกกา จึงเป็น สัญลักษณ์อภิภาษา ตัวใหม่ (new metasymbols) และเรียกสัญกรณ์นี้ว่า **extended Backus-Naur form** หรือ ตัวย่อว่า EBNF

สัญกรณ์ใหม่นี้ การสลับที่แบบซ้ายของตัวดำเนินการ "+" ซึ่งแสดงออกโดยการเรียกซ้ำแบบซ้าย ของกฎเดิม ใน BNF จะไม่เห็นเลย (obscures) สิ่งนี้เราต้องสมมติว่า ตัวดำเนินการใดๆ ซึ่งเกี่ยวกับ การทำซ้ำ ของวงเล็บปีกกา เป็นการสลับที่แบบซ้าย จริงๆ แล้ว ถ้าตัวดำเนินการ เป็นการสลับที่แบบขวา กฎไวยากรณ์ซึ่งตรงกันควรจะเป็น การเรียกซ้ำแบบขวา และกฎการเรียกซ้ำแบบขวา ปกติ จะไม่เขียน โดยใช้วงเล็บปีกกา ปัญหานี้เป็นช่องโหว่ของ ไวยากรณ์ EBNF ต้นไม้วิภัช และต้นไม้วากยสัมพันธ์ ไม่สามารถเขียนได้โดยตรงจากไวยากรณ์ แต่ ข้อสมมติ (assumptions) ต้องกระทำเกี่ยวกับโครงสร้างของมัน เพราะฉะนั้น ปกติเราจะใช้ BNF เพื่อเขียนต้นไม้วิภัช

สถานะการณ์ที่สอง คือ โครงสร้าง ซึ่งมีส่วนละเว้นได้ (optional part) เช่น ส่วนที่เป็น else ของ ข้อความสั่ง if ในภาษา Pascal แสดงออกด้วย BNF ดังนี้

<if-statement> ::= if <condition> then <ststement> I
if <condition> then <statement> else <statement>

สิ่งนี้เขียนง่ายกว่า เมื่อ แสดงออกด้วย EBNFs ดังนี้

<if-statement> ::= if <condition> then <statement>
Celse <statement>]

10-2

เมื่อ วงเล็บใหญ่ ([ ]) หมายถึง สัญลักษณ์อภิภาษาตัวใหม่ แสดงถึง ส่วนละเว้นได้ ของ โครงสร้าง

สถานะการณ์อีกอย่างหนึ่งซึ่ง โครงสร้าง อาจละเว้นได้ เกิดขึ้นเมื่อ กฎไวยากรณ์ แสดงถึง <u>สายว่าง</u> (empty string) ซึ่งเขียนด้วย สัญลักษณ์อภิภาษาตัวใหม่ $\in$ (ดูแบบฝึกหัด ข้อ 39) หรือ เป็น nonterminal <empty>

<u>ตัวอย่าง</u> กฎไวยากรณ์ในรูป 4.2

<label-declaration-part> ::= label <label-list> ';' |
<empty>

สร้าง label-declaration-part ให้เป็นส่วนละเว้นได้ สิ่งนี้แสดงด้วย EBNF ดังนี้

<label-declaration-part> ::= [label <label-list> ';']

ตัวดำเนินการ (ทวิภาค) สลับที่แบบขวา (Right-associative (binary) operators) สามารถเขียนโดยใช้ สัญลักษณ์อภิภาษาตัวใหม่ เหล่านี้ ตัวอย่างเช่น ถ้า "@" เป็น ตัวดำเนินการสลับที่แบบขวา ดังนี้ เขียนด้วย BNF ดังนี้

<exp> ::= <term> @ <exp> | <term>

ดังนั้น กฎข้อนี้ เขียนใหม่ด้วย EBNF เป็นดังนี้

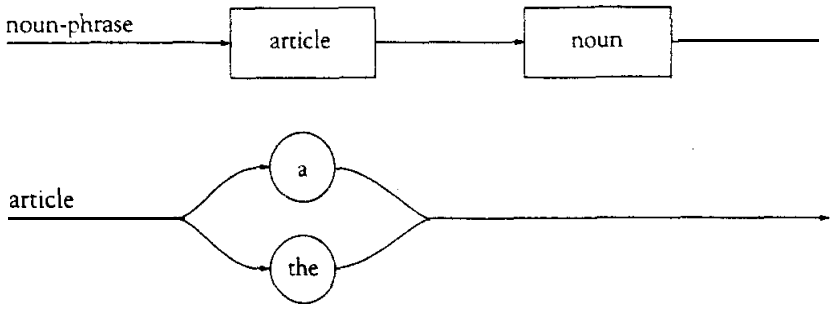
<exp> ::= <term> [@ <exp>]

สำหรับความบริบูรณ์ ของไวยากรณ์ในรูป 4.3 สำหรับนิพจน์คำนวณอย่างง่าย เขียนด้วย EBNF ได้ดังนี้

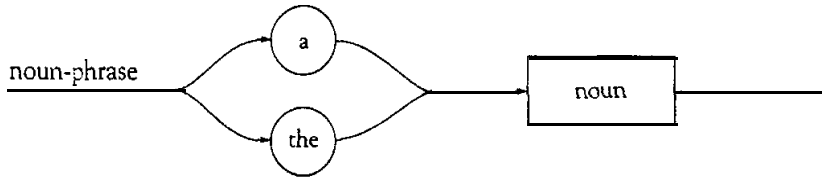
```
<exp>    ::= <term> {+ <term>%
<term>   ::= <factor> {* <factor>}
<factor> ::= (<exp>) I <number>
<number> ::= <digit> {<digit>}
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

รูป 4.4 กฎ EBNFs สำหรับนิพจน์คำนวณอย่างง่าย

แผนภาพวากยสัมพันธ์ เป็นการแทนที่เชิงภาพที่เป็นประโยชน์สำหรับกฎไวยากรณ์ ซึ่งแสดงลำดับของ terminals และ nonterminals ที่พบทางด้านขวามือ ของกฎ (Syntax diagram is a useful graphical representation for a grammar rule, which indicates the sequence of terminals and nonterminals encountered in the right-hand side of the rule.) ตัวอย่างเช่น แผนภาพวากยสัมพันธ์ของ <noun-phrase> และ (article> ของไวยากรณ์ภาษาอังกฤษ ในหัวข้อ 4.2 วาดภาพดังนี้



หรือรวม ทั้งสองรูป ให้เป็น แผนภาพเดียว ดังนี้



ในแผนภาพวากยสัมพันธ์ เราใช้ วงกลม หรือ วงรี สำหรับ terminals และใช้ สี่เหลี่ยมจัตุรัส หรือ สี่เหลี่ยมผืนผ้า สำหรับ nonterminals การต่อสิ่งเหล่านี้เข้าด้วยกันใช้ เส้น (line) และลูกศร (arrows) เพื่อแสดงลำดับที่เหมาะสม

แผนภาพวากยสัมพันธ์ อาจรวมหลาย productions เข้าด้วยกัน ให้เป็นหนึ่ง

แผนภาพได้

แผนภาพวากยสัมพันธ์ สำหรับไวยากรณ์ ในรูป 4.4 กำหนดให้แล้วในรูป 4.5 โปรดสังเกตว่า การใช้ลูป (loops) ในแผนภาพ เพื่อแสดงการทำซ้ำ ซึ่งกำหนดโดยวงเล็บปีกกาใน EBNFs

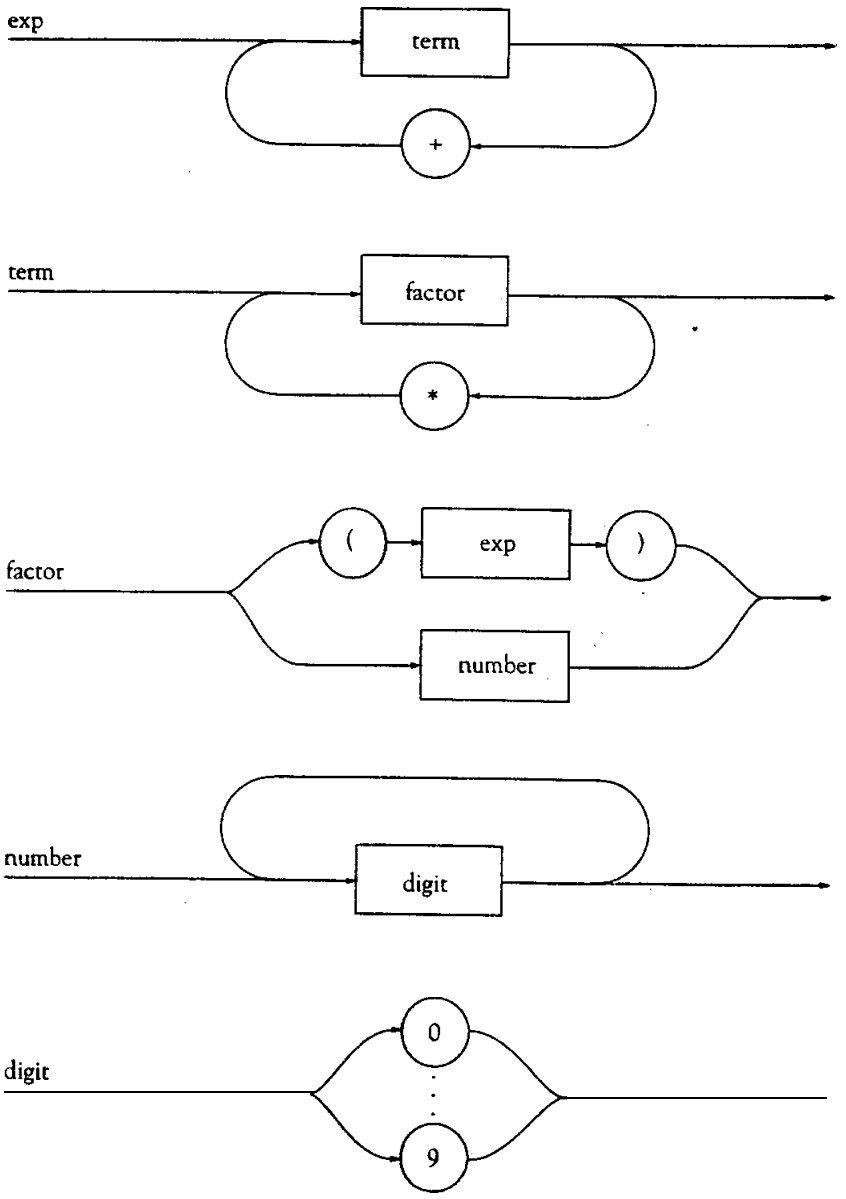
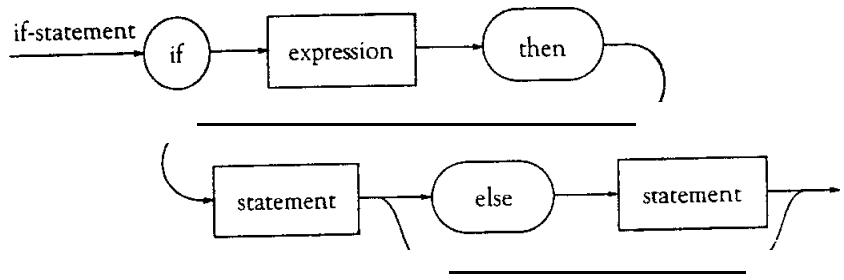
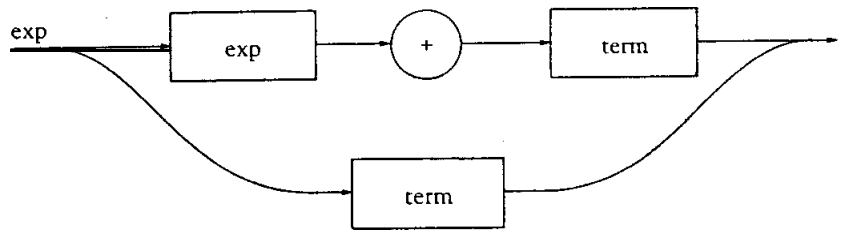


Figure 4-5 Syntax Diagrams for a Simple Expression Grammar

ส่วนการแสดงออกของวงเล็บใหญ่ ([ ]) ในแผนภาพวากยสัมพันธ์ จงดูตัวอย่าง ข้อความสั่ง-if ของภาษา Pascal ข้างล่างนี้



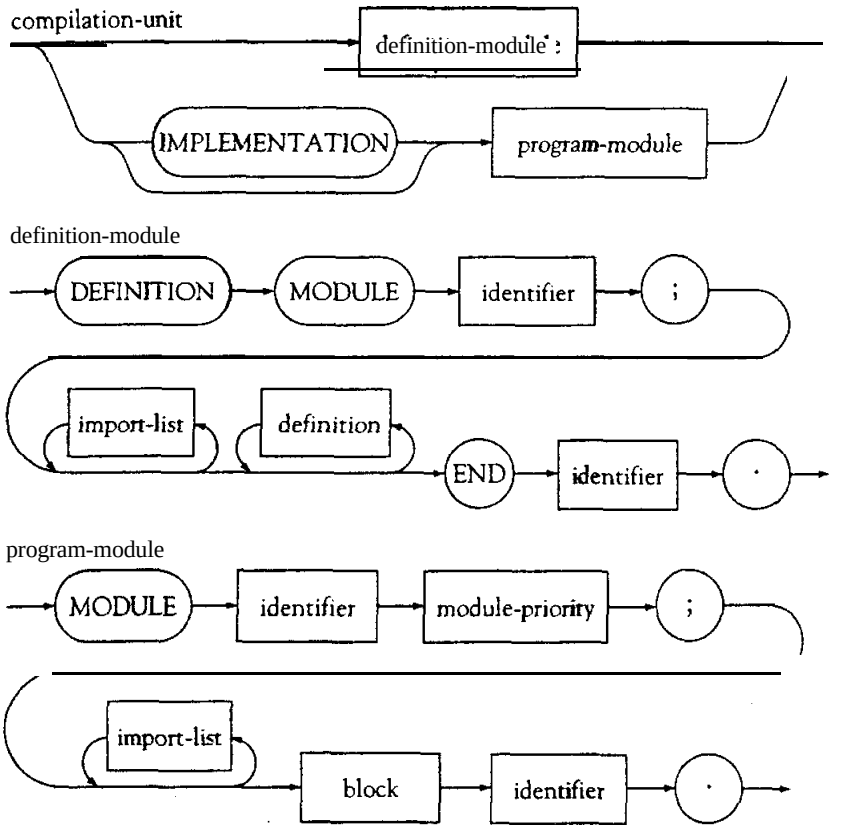
แผนภาพวากยสัมพันธ์ปกติจะเขียนจาก EBNF ไม่ใช่เขียนจาก BNF ดังนั้น แผนภาพของ <exp> ข้างล่างนี้ จึง **ผิด** (incorrect)



ตัวอย่างของ EBNFs และ แผนภาพวากยสัมพันธ์ของกฎไวยากรณ์ ของภาษา Modula-2 กำหนดให้แล้วในรูป 4.6

```
<compilation-unit>      ::=  <definition-module>  I
                             [IMPLEMENTATION]<program-module>
<definition-module>     ::=  DEFINITION  MODULE  <identifier>';'
                             {<import-list>}{<definition>}
                             END  <identifier>'.'
<program-module>        ::=  MODULE  <identifier>
                             [<module-priority>]';'{<input-list>}
                             <block><identifier>'.'
                             . . .
                             . . .
```

รูป 4.6a ตัวอย่าง EBNFs สำหรับภาษา Modula-2



รูป 4.6b ตัวอย่างแผนภาพวากยสัมพันธ์ สำหรับ ภาษา Modula-2

4.6 เทคนิคการวิเคราะห์กระจายและเครื่องมือที่ใช้

(Parsing techniques and tools)

ไวยากรณ์ ซึ่งเขียนด้วยรูปแบบ BNF, EBNF หรือ แผนภาพวากยสัมพันธ์ อธิบาย สายของ โทเค็น ซึ่งถูกต้อง เชิงวากยสัมพันธ์ ในภาษาโปรแกรม

(A grammar written in BNF, EBNF, or a syntax diagrams describes the strings of tokens that are syntactically legal in a programming language.)

ดังนั้น ไวยากรณ์ จึงอธิบายการกระทำโดยนัย ซึ่ง **ตัววิเคราะห์กระจาย** (parser) ต้องเอา สายของ โทเค็น มาวิเคราะห์กระจาย อย่างถูกต้อง นั่นคือ การสร้าง (construct) ต้นไม้การแปลง หรือ ต้นไม้วิภัช สำหรับ string ไม่ว่าจะเป็นโดยนัย หรือ ชัดแจ้ง รูปแบบอย่างง่ายที่สุด ของตัววิเคราะห์กระจาย คือ **ตัวรู้จำ** ซึ่งหมายถึง โปรแกรม ยอมรับ หรือ ปฏิเสธ สายของโทเค็น ขึ้นอยู่กับว่า มันเป็น สายโทเค็น ถูกต้องหรือไม่ ในภาษานั้น (The simplest form of a parser is a **recognizer** a program that accepts or rejects strings, based on whether they are legal strings in the language.) ตัววิเคราะห์กระจาย โดยทั่วไป สร้างต้นไม้วิเคราะห์กระจาย (หรือ ต้นไม้วากยสัมพันธ์ แบบนามธรรม) และทำการดำเนินการอื่นๆ ให้เป็นผลสำเร็จ เช่น คำนวณค่าของนิพจน์

กำหนดไวยากรณ์ ให้ ในหนึ่งรูปแบบ จากสามรูปแบบที่อภิปรายมาแล้ว มันสมนัยกับ การกระทำของตัววิเคราะห์กระจายอย่างไร? วิธีหนึ่ง ของ การวิเคราะห์กระจาย คือพยายามจับคู่ (match) อินพุท กับ ส่วนทางขวามือของกฎไวยากรณ์ เมื่อการจับคู่เกิดขึ้น ทางด้านขวามือ ถูกแทนที่ หรือ **ลดทอน** (reduced) โดย nonterminal ทางซ้ายมือ ตัววิเคราะห์กระจายเช่นนี้ เรียกว่า **ตัววิเคราะห์กระจายจากล่างขึ้นบน** (bottom-up parsers) เพราะว่า มันสร้าง การแปลง และ ต้นไม้วิเคราะห์วิภัช จากจุดแตกใบ ไปสู่ ราก บางครั้งเรียกว่า **ตัววิเคราะห์กระจายแบบเลื่อน-ลดทอน** (shift-reduce parser) เพราะว่ามันเลื่อนโทเค็น ไปไว้บนกองซ้อนก่อน จึงลดทอน strings ให้เป็น nonterminals

วิธีวิเคราะห์กระจายอีกวิธีหนึ่ง ที่สำคัญ ได้แก่ **แบบบนลงล่าง** (top-down) วิธีนี้ nonterminals ถูกขยายให้จับคู่กับโทเค็นที่เข้ามา และสร้างการแปลงโดยตรง เทคนิคการวิเคราะห์กระจายทั้งสองวิธีนี้ สามารถทำให้เป็นไปอัตโนมัติ กล่าวคือ เขียนโปรแกรมขึ้นมาเพื่อให้ แปลการอธิบาย BNF ให้เป็นตัววิเคราะห์กระจายอย่างอัตโนมัติ

เนื่องจากการวิเคราะห์กระจายจากล่างขึ้นบน ค่อนข้างดีกว่า (more powerful) การวิเคราะห์กระจายแบบจากบนลงล่าง ซึ่งปกติ เป็นวิธีที่ชอบมากกว่า เช่น **ตัวก่อกำเนิดของตัววิเคราะห์กระจาย** (parser generators) (ในอดีต เรียกว่า compiler compilers)

ตัวก่อกำเนิดของตัววิเคราะห์กระจาย ชนิดหนึ่ง ซึ่งใช้กันอย่างแพร่หลาย คือ YACC (ย่อมาจาก yet another compiler compiler) ซึ่งเราจะได้ศึกษาต่อไปในหัวข้อนี้

อย่างไรก็ตาม มีอีกวิธีหนึ่งซึ่งเป็นวิธีเก่า สำหรับ การสร้างตัววิเคราะห์กระจาย ด้วยมือ จากไวยากรณ์ ซึ่งเป็นวิธีที่มีประสิทธิผลมากและยังคงใช้กันอยู่บ่อยๆ สิ่งที่สำคัญคือ มันดำเนินการโดยการหมุน (turning) nontermonals ให้เป็นกลุ่มของกระบวนงานเรียกซ้ำร่วมกัน ซึ่ง การกระทำ ขึ้นอยู่กับ ทางด้านขวามือ ของ BNFs ดังนั้น จึงมีชื่อว่า **การวิเคราะห์กระจายแบบเรียกซ้ำลงล่าง** (recursive-descent parsing)

ส่วนทางขวามือ ถูก ตีความในกระบวนงาน ดังนี้ โทเค็นถูกจับคู่โดยตรงกับอินพุท โทเค็น ขณะที่สร้างโดย ตัวกราดตรวจ (scanner)

Nonterminals ถูกตีความ เช่น เรียกกระบวนงาน ซึ่งสมนัยกับ nonterminals

ตัวอย่าง ในไวยากรณ์ภาษาอังกฤษอย่างง่าย กระบวนงาน สำหรับ <sentence>, <noun-phrase> และ <article> จะเขียนดังนี้ (ด้วยรหัสเทียมคล้ายภาษา Pascal)

```
procedure    sentence;
begin
          nounphrase;
          verbphrase;
end;

procedure   nounphrase;
begin
          article;
          noun ;
end;

procedure    article;
begin
     if token = 'a' then
          Gettoken
     else if token = 'the' then
          Gettoken
     else Error;
end;
```

ในรหัส ข้างต้นนี้ เราใช้ตัวแปรส่วนกลางชื่อ token เพื่อเก็บโทเค็นปัจจุบัน (current token) ซึ่งถูกสร้างโดยตัวกราดตรวจ (scanner)

กระบวนงาน Gettoken ของตัวกราดตรวจ จะถูกเรียก เมื่อใดก็ตาม ที่ต้องการ โทเค็นตัวใหม่ การจับคู่ของโทเค็น สมนัยกับ การทดสอบที่ประสบผลสำเร็จสำหรับโทเค็นนั้น ตามด้วย การเรียก Gettoken

การวิเคราะห์กระจาย เริ่มต้นด้วย ตัวแปร token เก็บโทเค็นตัวแรกเรียบร้อยแล้ว ดังนั้น การเรียก Gettoken ต้องอยู่ก่อน การเรียกครั้งแรก ของ กระบวนงาน sentence

สมมติว่า มีกระบวนงาน Error อยู่ด้วย ซึ่งยกเลิก (aborts) การวิเคราะห์กระจาย (โปรดสังเกตว่า error จำเป็นเฉพาะถูกตรวจพบ เมื่อ โทเค็นจริง ถูกคาดหวัง ขณะอยู่ใน กระบวนงาน สำหรับ <article> เราควร มีการตรวจสำหรับ "a" หรือ "the" ในกระบวนงาน สำหรับ <sentence> แต่ใน โครงร่างนี้ มันไม่จำเป็น)

ถ้าเราประยุกต์ใช้ กรรมวิธีนี้ กับ การอธิบาย BNF ของรูป 4.3 จะพบปัญหาของกฎ เรียกซ้ำแบบซ้าย สำหรับนิพจน์เช่น

```
<exp> ::= <exp> + <term>
```

ถ้าเราพยายาม เขียนกฎนี้ เป็น กระบวนงาน**เรียกซ้ำแบบลงล่าง** (recursive-descent) จะได้

```
procedure  exp;
begin
     exp;
     if  token = '+'  then  begin
          gettoken;
          term;
     end;
end;
```

โชคไม่ดี เมื่อกระบวนงานนี้ถูกเรียก มันทำให้เกิด **ลูปของการเรียกซ้ำไม่รู้จบ** (infinite recursive loop) ทันที (สิ่งนี้เรียกว่า "head recursion" ซึ่งไม่เหมือนกับ tail recursion) สำหรับเหตุผลนี้ นักทฤษฎีภาษา จึงได้มีการศึกษา เทคนิค **left recursion removal** แต่เทคนิคทั่วไป ไม่จำเป็น ณ ที่นี้ ถ้าเราเขียน กฎใน EBNF หรือเป็น syntax diagrams การเรียกซ้ำแบบซ้าย จะถูกแทนที่อย่างง่าย โดย การวนซ้ำ (สมนัยกับวงเล็บปีกกา ของ EBNF)

ตัวอย่างเช่น กฎ

```
<exp> ::= <term> It {<term>}
```

ใน EBNF สมนัยกับ recursive-descent code ดังนี้

```
procedure exp;
begin
     term;
     while token = '+' do begin
          gettoken;
          term;
     end;
end;
```

ดังนั้น วงเล็บปีกกา ใน EBNF แทน left recursion removal โดยใช้ การวนซ้ำ (loop)

ในทางตรงกันข้าม กฎการเรียกซ้ำแบบขวา ไม่มีปัญหาเช่นนั้น ในการวิเคราะห์กระจายแบบการเรียกซ้ำแบบลงล่าง และกฎ

```
<exp> ::= <term> @ <exp>
```

สมนัยโดยตรง กับ recursive-descent code

```
procedure exp;
begin
     term;
     if token <> '@' then Error
     else  begin
          gettoken
          exp;
     end;
end;
```

มีปัญหาคล้ายกัน ในกฎ BNF ซึงแสดง โครงสร้างส่วนละเว้นได้ เช่น BNF สำหรับข้อความสั่ง - if ดังนี้

```
<if-statement>   ::= if <condition> then <statement> I
                     if <condition> then <statement>
                                    else <statement>
```

สิ่งนี้ไม่สามารถ แปลให้เป็นรหัส (code) ได้โดยตรง เพราะว่าทางเลือกทั้งสอง มี prefix เหมือนกัน อย่างไรก็ตาม ใน EBNF กฎข้อนี้ เขียนด้วย วงเล็บใหญ่ และเอาส่วนของทางเลือก ทีร่วมกันออก ตัวอย่างเช่น

```
<if-statement>  ::= if <condition> then <statement>
                    [else <statement>]
```

กฎข้างต้นนี้ สมนัยโดยตรงกับ recursive-descent code เมือวิเคราะห์กระจาย ข้อความสั่ง - if ข้างล่างนี้

```
procedure  ifstatement;
begin
      if token <> 'if' then Error
      else begin
                gettoken;
                condition;
                if token <> 'then' then Error
                else begin
                     get token;
                     statement;
                     if token = 'else' then begin
                          gettoken;
                          statement;
                     end;
```

```
            end;
        end;
    end;
```

กรรมวิธีของการเขียน ส่วนละเว้นได้ (optional parts) ของ BNF rules ใน EBNFs โดย เอา prefix ร่วมออก และการใช้วง เล็บ [ ] เรียกว่า **left** factoring และเ ป็นเรื่องจำเ ป็นสำหรับ recursive-descent parsing ดังนั้น กฎ EBNF หรือ syntax diagrams สมนัยอย่างธรรมชาติ กับ recursive-descent parser และสิ่งนี้ เป็น หนึ่ง ในเหตุผลหลัก ที่เลือกใช้ ในรูป 4.7 จะเป็นโครงร่าง ของ recursive descent parser ที่สมบูรณ์ สำหรับ ไวยากรณ์ของนิพจน์ ในรูป 4.4

```
procedure exp;
begin
  term;
  while Token = '+' do begin
    GetToken;
    term;
  end;
end;

procedure term;
begin
  factor;
  while Token = '*' do begin
    GetToken;
    factor;
  end;
end;

procedure factor;
begin
  if Token = '(' then begin
    GetToken;
    exp;
    if Token = ')' then GetToken
    else Error
  end else number;
end;

procedure number;
begin
  digit;
  while Token in ['0'..'9'] do digit;
end;

procedure digit:
begin
  if Token In ['0'..'9'] then GetToken
  else Error;
end;

procedure parse:
beg In
  GetToken;
  exp;
end;
```

Figure 4-7 Sketch of a Recursive-Descent Parser for Simple Arithmetic Expressions

ตัวก่อกำเนิดของตัววิเคราะห์กระจาย ซึ่งเป็นที่นิยมกันมากตัวหนึ่ง คือ YACC ซึ่งมีอยู่บนระบบ UNIX ส่วนใหญ่ ผู้เขียนคือ Steve Johnson ในช่วงกลางปี ค.ศ. 1970s มัน generates ก่อกำเนิด โปรแกรม ภาษา C ซึ่งใช้อัลกอริทึม แบบล่างขึ้นบน เพื่อวิเคราะห์กระจาย ไวยากรณ์ ไวยากรณ์ที่กำหนดให้ อยู่ในรูปแบบคล้าย BNF และเกี่ยวข้องกับ การกระทำด้วย ภาษา C

YACC มีประโยชน์ ไม่ใช่เฉพาะผู้เขียนตัวแปลภาษาเท่านั้น แต่ยังเป็นประโยชน์สำหรับนักออกแบบภาษาด้วย (YACC is useful not only to the translator writer, it is also useful to the langauge designer) : กำหนดไวยากรณ์ให้ มันจัดหา การอธิบายของปัญหาและความกำกวมที่เป็นไปได้ อย่างไรก็ตาม เพื่อให้สามารถอ่านสารสนเทศนี้ได้ เราต้องมีความรู้ เฉพาะด้าน ของเทคนิค การวิเคราะห์กระจาย แบบ จากล่างขึ้นบน มากกว่านี้

11-2

```
%{
#include <stdio.h>
#include  ●  cfyps.h>
%}

Ktoksn  NUMBER

%%
command : exp {printf("%d\n",$1);}
        ; /* allows printing of the result ● /

exp : exp '+' term {$$ = $ + $3;}
    | term {$$ = $1;}


term : term '*' factor {$$ = $1 * $3;}
     | factor {$$ = $1;}


factor : NUMBER {$$ = $1;}
       | '(' rxp ')' {$$ = $2;}

%%

void main(void)
  {yyparse();}

int yylex(void)
{int c ;
  while((c = getchar())-   -    '');
  /* eliminates blanks */
  if (isdigit(c))
      {yylval = 0 ;
      while (isdigit(c))
            {yylval = 10*yylval + c - '0';
            c = getchar() ;}
      ungetc(c,stdin);
      return(NUMBER);}
  if (c - '\n')return 0;
  /* makes the parse stop */
  return(c);
}

void yyerror(char ●  s) /* allows for printing
  error message */
  {printf("%s\n",s);}
```

Figure 4-8 YACC Input for Simple Arithmetic Expressions

4.7 โครงสร้างศัพท์ วากยสัมพันธ์ อรรถศาสตร์

(**Lexics** versus Syntax versus Semantics)

ไวยากรณ์ ไม่พึงบริบท รวม การอธิบาย คำศัพท์ต่างๆ ของภาษา โดยรวมสาย ของตัวอักขระ ซึ่งประกอบ เป็นโทเค็น ในกฎไวยากรณ์

(A context-free grammar includes a description of the tokens of a language by including the strings of characters that form the tokens in the grammar rules.)

ตัวอย่างเช่น ในไวยากรณ์ภาษาอังกฤษ ของหัวข้อ 4 . 2 โทเค็น คือ คำในภาษาอังกฤษ ได้แก่ "a", "the", "girl", "dog", "sees", "pet" และตัวอักขระ "." ส่วนใน ไวยากรณ์ นิพจน์ จำนวนเต็ม อย่างง่าย ใน หัวข้อ 4 . 3 โทเค็น หมายถึง สัญลักษณ์คำนวณ "+" และ "*" วงเล็บ "(" และ ")" และเลข โดด 0 ถึง 9 รายละเอียดเฉพาะ ของ การจัดรูปแบบ เช่น ช้อตกลงของ white-space ที่กล่าวถึง ใน หัวข้อ 4.1 ถูกทิ้งให้กับ ตัวกราดตรวจ (scanner) และจำเป็นต้องกล่าวถึง ข้อตกลง แยกต่างหากจากไวยากรณ์

ประเภทของโทเค็น บางอย่าง เช่น ค่าคงที่ ไอเดนติไฟเออร์ หมายถึง ไม่ใช่ ลำดับคงที่ ของ ตัวอักขระ ในตัวมันเอง แต่ถูกสร้าง ให้เป็นเซตคงที่ ของ ตัวอักขระ เช่น เลขโดด 0 . . 9

(Some typical token categories, such as constants and identifiers, are not fixed sequences of characters in themselves, but are built up out of a fixed set of characters, such as the digits 0 . . 9)

ประเภทของโทเค็นเหล่านี้ บ่อยครั้ง มี โครงสร้างของมัน นิยามโดยไวยากรณ์ เช่น ตัวอย่าง กฎไวยากรณ์ สำหรับ <number> และ <digit> ใน ไวยากรณ์ของนิพจน์ อย่างไรก็ตาม มันเป็นไปได้ ที่ จะใช้ ตัวกราดตรวจ (scanner) เพื่อรู้จำ (to recognise) โครงสร้างเหล่านี้ เนื่องจากกำลังเรียกซ้ำแบบเต็ม ของ ตัววิเคราะห์กระจาย ไม่มีความจำเป็น และ ตัวกราดตรวจสามารถ รู้จำได้ โดยการดำเนินการ ทำซ้ำอย่างง่าย

สิ่งนี้มีประสิทธิภาพมากกว่า - มันทำการรู้จำ เลข และ ไอเดนติไฟเออร์ เร็วกว่า และง่ายกว่า และ มันลดขนาด ของ ตัวกราดตรวจ และ จำนวน กฎ BNFs

เพื่อแสดงให้เห็นความจริงที่ว่า เลข (a number) ใน ไวยากรณ์ของนิพจน์ควรจะเป็น โทเค็น ไม่ใช่แทนด้วย nonterminal เราจะเขียน ไวยากรณ์ ของรูป 4-4 ใหม่ ในรูป 4-9 ดังนี้

```
<exp>    ::=   <term>{+ <term>}
<term>   ::=   <factor>{* <factor>3
<factor> ::=   (<exp>) I NUMBER
```

รูป 4-9 เลข เป็น โทเค็น ใน การคำนวณอย่างง่าย

การทำให้ สาย NUMBER เป็น อักษรตัวใหญ่ ใน ไวยากรณ์ใหม่นี้ เรากำลังพูดว่า มัน ไม่ใช่ โทเค็น ซึ่ง รู้จำได้ โดยภาษา แต่ มันเป็น โครงสร้างอย่างหนึ่ง ซึ่งบอกได้ โดยตัวกราดตรวจ โครงสร้างของ โทเค็น ประเภทนี้ ต้องระบุให้เป็นส่วนของ ข้อตกลงคำศัพท์ ของภาษา

สัญกรณ์ ซึ่ง ใช้บ่อยมาก สำหรับ ข้อกำหนดนี้ คือ **นิพจน์ปกติ** (regular expressions) อย่างไรก็ตาม มีนักออกแบบภาษา จำนวนมาก ซึ่งเลือก ที่จะ รวมโครงสร้างของ โทเค็น เช่นนี้ ให้เป็นส่วนหนึ่งของ ไวยากรณ์ ด้วย ความเข้าใจว่า นักทำภาษาให้เกิดผล (implementor) อาจจะ รวมสิ่งเหล่านี้ ใน ตัวกราดตรวจ แทนที่จะเป็น ตัววิเคราะห์กระจาย ดังนั้น การอธิบาย ภาษา โดยใช้ BNF, EBNF หรือแผนภาพวากยสัมพันธ์ ไม่เพียงแต่ รวม วากยสัมพันธ์ไว้ แต่ยังรวม โครงสร้างศัพท์ (หรือ lexics) ส่วนใหญ่ ของ ภาษาโปรแกรมไว้ด้วย ถึงแม้ว่า ขอบเขต (boundary) ระหว่าง โครงสร้างวากยสัมพันธ์ และ โครงสร้างศัพท์ ปกติ ไม่เขียนชัดเจน แต่ขึ้นอยู่กับจุดของการมอง ของ นักออกแบบภาษา และ นักทำภาษาให้เกิดผล

เช่นเดียวกัน มันเป็นจริงด้วย สำหรับวากยสัมพันธ์ และอรรถศาสตร์ เราได้

มีการเข้าถึง มาแล้วว่า วากยสัมพันธ์ คือ สิ่งใดก็ตาม ซึ่งถูกนิยาม ได้ด้วย ไวยากรณ์ไม่พึ่งบริบท และอรรถศาสตร์ คือ สิ่งใดก็ตาม ซึ่งนิยามไม่ได้ ด้วย ไวยากรณ์ ไม่พึ่งบริบท อย่างไรก็ตาม มีผู้เขียนหนังสือ จำนวนมาก ได้รวมคุณสมบัติ ซึ่งเราเรียกว่า อรรถศาสตร์ เป็น คุณสมบัติเชิงวากยสัมพันธ์ ของภาษา ตัวอย่างเช่น กฎต่างๆ ของการประกาศ **ก่อน** ใช้ตัวแปร และ การไม่มีการประกาศใหม่ ของ ไอเดนติไฟเออร์ ภายใน โปรซีเดอร์ สิ่งเหล่านี้เป็นกฎ ซึ่งเรียกว่า **ไวยากรณ์ไม่พึ่งบริบท** (context-sensitive grammar) และไม่สามารถเขียนด้วยกฎ พึ่งบริบท ดังนั้น เราจึงชอบมากกว่า ที่จะคิดว่า มันเป็น ความหมาย ไม่ใช่กฎ วากยสัมพันธ์

ข้อขัดแย้งอีกประการหนึ่งระหว่างวากยสัมพันธ์ และอรรถศาสตร์ เกิดขึ้น เมื่อภาษานั้น ต้องการ สายอักขระบางอย่าง ให้เป็น **ไอเดนติไฟเออร์นิยามมาแล้ว** (predefined identifiers) ไม่ใช่ **คำสงวน** (reserved words) จากหัวข้อ 4.1 ซึ่งกล่าวว่า คำสงวน หมายถึ สายคงที่ของ อักขระ ซึ่งเป็นโทเค็นโดยตัวมันเอง และนำไปใช้ เป็น ไอเดนติไฟเออร์ ไม่ได้

(Recall that reserved words are fixed strings of characters that are tokens themselves and that cannot be used as identifiers.)

ตัวอย่างเช่น "begin", "end", "while", "do", และ "procedure" strings ทั้งหมดนี้ เป็นคำสงวน ในภาษา Pascal

อย่างไรก็ตาม string "true" และ "false" ไม่ใช่คำสงวน แต่เป็น predefined identifiers นั่นคือ มันเป็น ไอเดนติไฟเออร์ ซึ่ง มี ความหมายคงที่ในภาษา แต่ ความหมายนี้ สามารถเปลี่ยนแปลงได้ โดยการ ประกาศให้ใหม่ ภายในโปรแกรม

(**Predefined identifiers** are identifiers that have a fixed meaning in the language, but this meaning can be changed be **redeclaring** them within a program.)

อย่างไรก็ตาม การทำเช่นนี้ เป็นแนวปฏิบัติของการเขียนโปรแกรมที่แย่มาก เพราะว่า มันจะ ไม่มีความหมายปกติ เช่น ค่าคงที่แบบบลู ด้วย ค่าปกติ วิธีที่ดีกว่า คือ ทำให้ "true" และ "false" เป็น syntactic entities คือ เป็นคำสงวน ซึ่งสมนัย กับ อรรถศาสตร์ คงที่

แบบฝึกหัด

1. Add subtraction and division to the

   (a) BNF

   (b) EBNF

   and (c) syntax diagrams of simple arithmetic expression (Figures 4-3, 4-4, and 4–5) Be sure to give them the appropriate precedence.

2. Add the mod and power operation to

   (a) the arithmetic BNF or

   (b) EBNF.

   Use % for the mod operation and ^ for the power operation. Recall that mod is left-associative, like division, but that power is right-associative. (Thus 2^2^3 = 256, not 64)

3. Unary minuses can be added in several ways to the arithmetic expression grammar of Figure 4-3 or the grammar from Exercise 1. Revise the BNF and EBNF for each of the cases that follow so that it satisfies the stated rule:

   (a) At most one unary minus is allowed in each expression, and it must come at the beginning of an expression, so -2 - 3 is legal (and equals -5) and -2 - (-3) is legal, but -2 - -3 is not.

   (b) At most one unary minus is allowed before a number or left parenthesis, so -2 - -3 is legal but - -2 and -2 - - -3 are not.

(c) Arbitrarily many unary minuses are allowed before numbers and left parentheses, so everything above is legal, but, for example, 2 - t3 is not.

4. จงวาดรูปต้นไม้วิภัช และต้นไม้วากยสัมพันธ์แบบนามธรรม ของนิพจน์คำนวณข้างล่างนี้ (Draw parse trees and abstract syntax trees for the arithmetic expressions) :

(a) ((2))

(b) 3 + 4 * 5 + 6 * 7

(c) 3 * 4 + 5 * 6 + 7

(d) 3 * (4 + 5) * (6 + 7)

(e) (2 + (3 + (4 + 5)))

5. เป็นไปได้หรือไม่ ที่จะมีภาษา ซึ่งไม่มีคำสงวนใดๆ ให้อภิปราย (Is it possible to have a language without any reserved words? Discuss.)

6. A number is defined in the grammar of Figure 4-3 using a left-recursive rule. However, it could also be defined using a right-recursive rule :

<number> ::= <digit><number>|<digit>

which is better, or does it matter? Why?

7. Given the following BNF :

<exp> ::= (<list>) | a

<list> ::= <list>, <exp> I <exp>

(a) Write EBNF rules and syntax diagrams for the language.

(b) Draw the parse tree for ((a, a), a, (a)).

(c) Write a recursive-descent recognizer for the language.