บทที่ 8

วากยสัมพันธ์และการแปลภาษา
(Syntax and Translation)

บทที่ 8
วากยสัมพันธ์และการแปลภาษา
(Syntax and Translation)

ในบทนี้ จะกล่าวถึงโครงสร้างวากยสัมพันธ์ โดยตรง ซึ่งมีสามหัวข้อสำคัญ : สมาชิกวากยสัมพันธ์ โครงสร้างของตัวแปลภาษา ซึ่งประมวลผลวากยสัมพันธ์ และ ข้อกำหนดทางการของวากยสัมพันธ์

(: the syntactic elements, the structure of the translators which process the syntax, and the formal specification of syntax.)

ในเนื้อหาตอนแรกๆ นั้น วากยสัมพันธ์มีบทบาทค่อนข้างน้อย <u>เหตุผลข้อแรก</u> คือ สมาชิกเชิงความหมาย (semantic elements) ที่สำคัญส่วนใหญ่ของโปรแกรม ไม่ได้ถูก แทน ในวากย-สัมพันธ์ ของโปรแกรมโดยตรง แต่ปรากฏ เป็นเพียงโดยนัยเท่านั้น ตัวอย่างเช่น การประกาศโดยนัย โครงสร้างข้อมูลโดยนัย การคำเนินการโดยนัย การควบคุมลำดับโดยนัย และสิ่งแวดล้อมการอ้างถึงโดยนัย ถ้าเราดูเฉพาะวากยสัมพันธ์ ของโปรแกรมแต่เพียงอย่างเดียว เราจะขาดความสำคัญส่วนกลางไป <u>เหตุผลข้อที่สอง</u> การศึกษาวากยสัมพันธ์ของโปรแกรมมีค่อนข้างน้อย อันเนื่องมาจากว่า ความหลากหลาย ของภาษาต่างๆ ใน โครงสร้างเชิงวากยสัมพันธ์ มีมากกว่า ความหลากหลาย ใน ความเข้าใจ โครงสร้างเชิงความหมาย

ความหลากหลาย ในวากยสัมพันธ์เหล่านี้ ส่วนใหญ่แล้ว เป็นความชอบส่วนตัวของนักออกแบบภาษา และไม่ควรจะได้รับการพิจารณามาก ความจริงแล้ว โดยทั่วไป มีกฎเกณฑ์ตรงกัน จำนวนน้อย สำหรับโครงสร้างวากยสัมพันธ์ ในภาษาโปรแกรม กล่าวคือ นักออกแบบภาษาแต่ละคน มีแนวโน้ม ที่จะเลือก โครงสร้างซึ่งดูเป็นธรรมชาติและเหมาะสมกับตนเอง สิ่งนี้ทำให้ขาดความเป็นรูปแบบเดียวกัน (lack of uniformity) กรณีที่เห็นง่ายที่สุด คือ วากยสัมพันธ์ สำหรับการอ้างถึง สมาชิก ของแถวลำดับเชิงเส้น

<u>ตัวอย่าง</u> สมาชิกตัวแรกของแถวลำดับเชิงเส้น A อาจเขียนดังนี้

A(1), A[1], A<1>, (CAR A), FIRST OF A, หรือ A.FIRST ทั้งนี้ขึ้นอยู่กับภาษาโปรแกรม ในโครงสร้างวากยสัมพันธ์ ที่มีขนาดใหญ่ขึ้น เช่น นิพจน์ ข้อความสั่ง การประกาศ และโปรแกรมย่อย จะมีรูปแบบเดียวกันน้อยลง (less uniformity) ไปอีก

เมื่อเรารับเอาภาษาโปรแกรม ภาษาใหม่เข้ามาใช้ ดูเหมือนว่าเราจะต้องเผชิญหน้า กับวากยสัมพันธ์ ใหม่ทั้งหมด ส่วนความเข้าใจความหมาย (semantics) ไม่แตกต่างมากนัก จากภาษา

อื่นๆ ซึ่งเรามีความคุ้นเคยอยู่แล้ว การเลือก วากยสัมพันธ์ ใหม่นี้ อาจจะไม่แตกต่างอย่างสิ้นเชิง นั่นคือ บ่อยครั้ง โครงสร้างวากยสัมพันธ์ใหม่นี้ มีความสวยงามกว่า อ่านง่ายกว่า มีข้อผิดพลาดน้อยกว่า หรือมีข้อดีอื่นๆ เหนือกว่าภาษาโปรแกรมที่เราคุ้นเคย สิ่งที่สำคัญคือ การทำความเข้าใจความหมายของภาษาใหม่นั้น จะกระทำได้อย่างเร็วเท่าที่เป็นไปได้ โครงสร้างความหมาย ดูเหมือนว่ามี ทั้งความคุ้นเคยมากกว่า และเพื่ออธิบายเหตุผลบางอย่างของลักษณะพิเศษเชิงวากยสัมพันธ์

8.1 **เกณฑ์วากยสัมพันธ์โดยทั่วไป** (General syntactic criteria)

วัตถุประสงค์อันแรกของวากยสัมพันธ์ คือ จัดหา สัญกรณ์ สำหรับ การสื่อสาร สารสนเทศ ระหว่าง โปรแกรมเมอร์ กับตัวประมวลผลภาษาโปรแกรม

(The primary purpose of syntax is to provide a notation for communication of information betwen the programmer and the programming language processor.)

อ่านง่าย (Readability)

โปรแกรมจะอ่านง่าย ถ้าความเข้าใจ โครงสร้างของอัลกอริทึมและข้อมูล ซึ่งแทนด้วย โปรแกรม ปรากฏชัด จาก ตัวโปรแกรมเอง

โปรแกรมซึ่งอ่านได้ง่าย บ่อยครั้งเรียกว่า เป็นเอกสารโดยตัวมันเอง หมายถึง โปรแกรมสามารถทำความเข้าใจได้ โดย ไม่ต้องมี เอกสารอธิบายแยกต่างหาก (ถึงแม้ว่า เป้าหมายนี้ จะไม่ค่อยประสบผลสำเร็จในทางปฏิบัติก็ตาม)

(A readable program is often said to be self-documenting - it is understandable without any separate documentation.)

โปรแกรมอ่านง่าย ถูกสนับสนุน โดย คุณสมบัติของภาษา เช่น รูปแบบข้อความสั่งที่เป็นธรรมชาติ (natural statement format) ข้อความสั่งเชิงโครงสร้าง (structured statements) มีอิสระในการใช้คำหลัก และคำรบกวน (liberal use of keywords **and** noise words) รวมคอมเมนต์ไว้ในตัว (provision for embeded comments) การไม่จำกัดความยาวของไอเดนติไฟเออร์ (unrestricted length identifiers), สัญลักษณ์ ตัวดำเนินการช่วยจำ (mnemonic operator symbols) รูปแบบเขตอิสระ (free field formats) และ การประกาศข้อมูลที่สมบูรณ์ (complete data declarations)

การที่โปรแกรมอ่านง่าย ไม่รับประกัน จาก การออกแบบภาษา เพราะว่า การออกแบบแม้ดีที่สุดอาจถูกแวดล้อม โดย การเขียนโปรแกรมที่แย่ (poor programming) ในทางตรงกันข้าม การออกแบบวากยสัมพันธ์ สามารถบังคับ แม้กระทั่ง โปรแกรมเมอร์ ซึ่งมีความตั้งใจดีที่สุด ให้เขียนโปรแกมที่ไม่น่าอ่าน (เช่นภาษา APL) ในบรรดาภาษาต่างๆ การออกแบบภาษา COBOL

เน้นเรื่องการอ่านง่าย มากที่สุด โดยเฉพาะเขียนง่าย และ การแปลภาษาง่าย

การอ่านง่าย ถูกสนับสนุนโดย วากยสัมพันธ์ของโปรแกรม ซึ่ง ความแตกต่างเชิงวากย-สัมพันธ์ สะท้อน บน ความแตกต่างเชิงความหมาย ดังนั้น ตัวสร้างโปแกรม (program constructs) ซึ่งทำสิ่งที่เหมือนกัน จะดูคล้ายกัน และตัวสร้างโปรแกรม ซึ่งทำสิ่งแตกต่างกัน จะดูต่างกัน ตัวอย่างเช่น ความแตกต่าง ระหว่าง การย้ายมีเงื่อนไข (conditional branch) การวนซ้ำ (an iteration) และ โครงสร้างควบคุม **goto** ถูกทำให้ชัดเจนในภาษาส่วนใหญ่ โดยการใช้ ชนิด ข้อความสั่งแตกต่างกัน ด้วยโครงสร้างเชิงวากยสัมพันธ์ที่แตกต่างกัน

โดยทั่วไปแล้ว ความหลากหลายต่างๆ ของ ตัวสร้างวากยสัมพันธ์ ยิ่งมีใช้ มากขึ้น โครงสร้างโปรแกรม ยิ่งง่ายขึ้น ที่จะสะท้อนความแตกต่างบนโครงสร้างความหมาย

(In general the greater the variety of syntactic constructs used, the more easily the program structure may be made to reflect different underlying semantic structures.)

ภาษาโปรแกรม ซึ่ง มีตัวสร้างวากยสัมพันธ์ต่างๆ จำนวนน้อย จะนำไปสู่ โปรแกรมอ่านยาก ตัวอย่างเช่น ภาษา APL มีรูปแบบข้อความสั่งเพียง หนึ่งชนิดเท่านั้น (only one statement format is provided.) ความแตกต่าง ระหว่าง ข้อความสั่งกำหนดค่า การเรียกโปรแกรมย่อย goto อย่างง่าย การส่งคืนโปรแกรมย่อย การย้ายมีเงื่อนไขหลายทาง และโครงสร้างโปรแกรมร่วมอื่นๆ สะท้อนเชิงวากยสัมพันธ์ ก็ต่อเมื่อ สัญลักษณ์ ตัวดำเนินการ หนึ่งตัว หรือ จำนวนไม่กี่ตัว แตกต่างกัน ภายใน นิพจน์ซับซ้อน (a complex expression) บ่อยครั้ง จึงต้องการ การวิเคราะห์โปรแกรมอย่างละเอียด เพื่อหาโครงสร้างควบคุมโดยรวมของมัน นอกจากนี้แล้ว ข้อผิดพลาดวากยสัมพันธ์ง่ายๆ เช่น อักขระหนึ่งตัว ในข้อความสั่ง ไม่ถูกต้อง อาจจะเปลี่ยนความหมายของข้อความสั่งอย่างมาก โดย ไม่มีการบอก ข้อผิดพลาดเชิงวากยสัมพันธ์ของมัน ปัญหาคล้ายกันนี้ เกิดขึ้นเช่นกัน ในภาษา SNOBOL4 ซึ่งมี วากยสัมพันธ์ ข้อความสั่งหลักเพียง หนึ่ง ชนิดเท่านั้น. อักขระว่าง ที่เพิ่ม หนึ่งตัว ภายในข้อความสั่ง ของ SNOBOL4 อาจเปลี่ยนแปลง ข้อความสั่ง จากการเรียกโปรแกรมย่อยอย่างง่าย ไปเป็น ข้อความสั่ง จับคู่รูปแบบ ซึ่งนำไปสู่ การต่อเรียง ของ ข้อผิดพลาดเวลาดำเนินงาน ใน ส่วนอื่นๆ ของโปรแกรม ซึ่ง สามารถถูก ตามรอยย้อนกลับ ไปยัง ข้อผิดพลาดวากยสัมพันธ์ ซึ่งผิดเพิ่มขึ้น โดยเฉพาะ ยากมากขึ้น

ในภาษา LISP ข้อผิดพลาด ในการจับคู่ เครื่องหมายวงเล็บ เป็นเหตุให้เกิดปัญหาคล้ายๆ กัน ระหว่างขั้นตอนการทดสอบโปรแกรม เราอาจตอบได้ดีว่า โปรแกรมซึ่งอ่านง่าย ดูแล้ว ไม่ถูกต้อง เมื่อมัน ไม่ถูกต้องอย่างชัดเจน (a readable program not look correct when it is grossly incorrect.)

เขียนง่าย (Writeability)

คุณสมบัติเชิงวากยสัมพันธ์ ซึ่งทำให้ โปรแกรม เขียนง่าย บ่อยครั้ง จะขัดแย้งกับ คุณสมบัติ ซึ่งทำให้ โปรแกรมอ่านง่าย

การเขียนง่าย ถูกส่งเสริม โดย ใช้โครงสร้างวากยสัมพันธ์ที่รวบรัดและเป็นไปตามกฎเกณฑ์ ในขณะที่ การอ่านง่าย มีโครงสร้างต่างๆ มากมาย ที่นำมาช่วยเหลือ ข้อตกลงวากยสัมพันธ์โดยนัย ซึ่งทำให้ การประกาศ และการดำเนินการต่างๆ ถูกตัดทิ้ง ทำให้โปรแกรมสั้นลง และเขียนง่ายขึ้น แต่อ่านยากขึ้น คุณสมบัติขั้นสูงอื่นๆ มีเป้าหมายทั้งสองอย่าง ตัวอย่างเช่น การใช้ข้อความสั่งเชิงโครงสร้าง รูปแบบข้อความสั่งธรรมชาติอย่างง่าย สัญลักษณ์การดำเนินการช่วยจำ และ ไอเดนติไฟเอร์ ไม่มีข้อจำกัด ปกติ ทำให้การเขียนโปรแกรม ง่ายขึ้นโดย ยอมให้ใช้โครงสร้างธรรมชาติ ของ อัลกอริทึม และข้อมูลปัญหา ถูกแทนได้โดยตรง ในโปรแกรม

วากยสัมพันธ์ จะซ้ำซ้อน (redundant) ถ้ามันสื่อสาร ชิ้นข้อมูลของสารสนเทศ เดียวกันมากกว่า หนึ่งวิธี ความซ้ำซ้อน บางอย่าง เป็นประโยชน์ ในวากยสัมพันธ์ ของภาษาโปรแกรม เพราะว่า มันทำให้ เขียนโปรแกรมง่ายขึ้น และ การตรวจสอบข้อผิดพลาด ซึ่งกระทำ ระหว่างการแปลภาษา มีมากขึ้น ข้อไม่ดีคือ ความซ้ำซ้อน ทำให้โปรแกรมมีขนาดใหญ่ขึ้น และเขียนยากขึ้น กฎอัตโนมัติส่วนใหญ่ สำหรับความหมายของ โครงสร้างภาษา ตั้งใจลดความซ้ำซ้อนโดยการขจัดข้อความสั่งชัดแจ้ง ของ ความหมายซึ่ง สามารถอนุมานได้จาก เนื้อความ (context) ตัวอย่างเช่น ใน ภาษา FORTRAN

แทนที่จะเป็นการประกาศชัดแจ้ง ของชนิดตัวแปรอย่างง่ายทุกตัว ภาษา FORTRAN มีการใช้ข้อตกลงการตั้งชื่อว่า ตัวแปรมีการประกาศโดยนัย คือ ชื่อใดก็ตาม ซึ่งขึ้นต้นด้วยอักษรตัวใดตัวหนึ่ง ใน I-N จะถือว่าเป็น ชนิด integer และตัวแปรอื่นๆ ทั้งหมด เป็น ชนิด real ขณะนี้ การเกิดของ ชื่อตัวแปรใหม่ ใน โปรแกรม พอเพียงแล้ว เพื่อประกาศมัน ความซ้ำซ้อนของการประกาศแยกต่างหาก จะถูกหลีกเลี่ยง โชคไม่ดี การใส่ ความสะดวก ถูกชดเชย โดยผลกระทบเป็นลบที่รุนแรงมากกว่า เช่น การสะกดชื่อตัวแปรผิด ไม่สามารถตรวจพบได้ โดย คอมไพเลอร์ ถ้าโปรแกรมใช้ตัวแปรชื่อ INDEX ซึ่ง ณ จุดที่อ้างถึง ใช้ INDX คอมไพเลอร์ จะถือว่า INDX เป็นตัวแปรตัวใหม่ ชนิด integer (ซึ่ง ไม่มีค่าเริ่มต้น) และข้อผิดพลาดปลีกย่อย จะเกิดขึ้นในโปรแกรม ถ้าตัวแปรแต่ละตัวต้องมี การประกาศชัดแจ้ง เช่นในภาษา Pascal จากนั้น คอมไพเลอร์ จะทำเครื่องหมายว่า ข้อผิดพลาดคือ ชื่อตัวแปรสะกดผิด เพราะว่า ผลกระทบนี้ คือ ข้อผิดพลาด ในโปรแกรมต่าง ๆ ภาษาซึ่งไม่มีความซ้ำซ้อนทั้งหมด บ่อยครั้ง นำไปใช้ยาก

แปลง่าย (Ease of Translation)

เป้าหมายการขัดแย้งข้อที่สาม ได้แก่ การทำให้โปรแกรม ง่ายต่อการแปล ให้เป็นรูปแบบกระทำการได้ (making programs easy to translate into executable form)

การอ่านง่าย และการเขียนง่าย เป็นหลักเกณฑ์ โดยตรงให้กับ ความจำเป็นของโปรแกรมเมอร์ ความง่ายของการแปลเกี่ยวกับความจำเป็นของตัวแปลภาษา ซึ่งจะประมวลผล โปรแกรมที่เขียนขึ้นมา หลักของการง่ายต่อการแปล คือ กฎเกณฑ์ของโครงสร้าง (The key to easy translation is regularity of structure.) วากยสัมพันธ์ ของภาษา LISP เป็นตัวอย่างของโครงสร้างโปรแกรม ซึ่ง อ่านยาก และเขียนยาก แต่ง่ายมากในการแปล โครงสร้างเชิงวากยสัมพันธ์ โดยรวมของโปรแกรม ภาษา LISP ใดๆ อาจถูกอธิบายด้วยกฎง่ายๆ เพียงไม่กี่ข้อ อันเนื่องจากหลักเกณฑ์ของวากยสัมพันธ์ เมื่อจำนวนตัวสร้างวากยสัมพันธ์พิเศษของโปรแกรม เพิ่มขึ้น โปรแกรมจะแปลยากขึ้น ตัวอย่างเช่น การแปลโปรแกรมภาษา COBOL ถูกกระทำยากมาก โดย ข้อความสั่ง และรูปแบบของการประกาศ จำนวนมาก ที่ยอมให้ใช้ ถึงแม้ว่าอรรถศาสตร์ ของภาษา จะไม่ซับซ้อนก็ตาม

ไม่กำกวม (Lack of Ambiguity)

ความกำกวม เป็น ปัญหาหลัก อย่างหนึ่ง ในการออกแบบของทุกภาษา บทนิยาม ของภาษาที่ดีนั้น ตัวสร้างวากยสัมพันธ์ (syntactic construct) แต่ละชนิดที่โปรแกรมเมอร์เขียนขึ้น จะต้องมีความหมายเพียงหนึ่งอย่างเท่านั้น ตัวสร้างกำกวม ทำให้มีการตีความ ที่แตกต่างกันตั้งแต่สองอย่างขึ้นไป ปัญหาของความกำกวม ปกติ จะไม่เกิดขึ้นใน โครงสร้างของ สมาชิกโปรแกรมแต่ละตัว แต่จะเกิดขึ้น ในการใช้ร่วมกันระหว่าง โครงสร้างแตกต่างกัน ตัวอย่างเช่น ภาษา Pascal และ ภาษา ALGOL มีรูปแบบ ของ ข้อความสั่ง มีเงื่อนไข แตกต่างกัน สองชนิดคือ

if <Boolean expression> **then** <statement,> else **<statement$_2$>**

และ

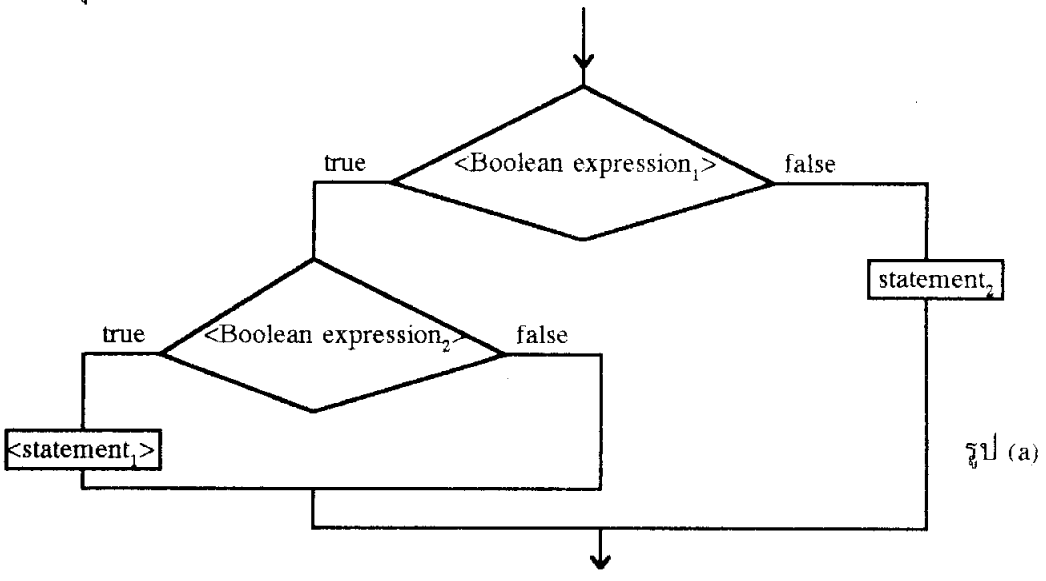
if <Boolean **expression> then** <statement,>

การตีความ (interpretation) ซึ่งกำหนดให้กับ รูปแบบข้อความสั่งแต่ละชนิด นิยามไว้ชัดเจน อย่างไรก็ตาม เมื่อนำสองรูปแบบรวมเข้าด้วยกัน โดยให้ <statement$_1$> เป็นข้อความสั่งมีเงื่อนไข อีกชุดหนึ่ง ดังนั้น โครงสร้างรวม จึงเป็นดังนี้

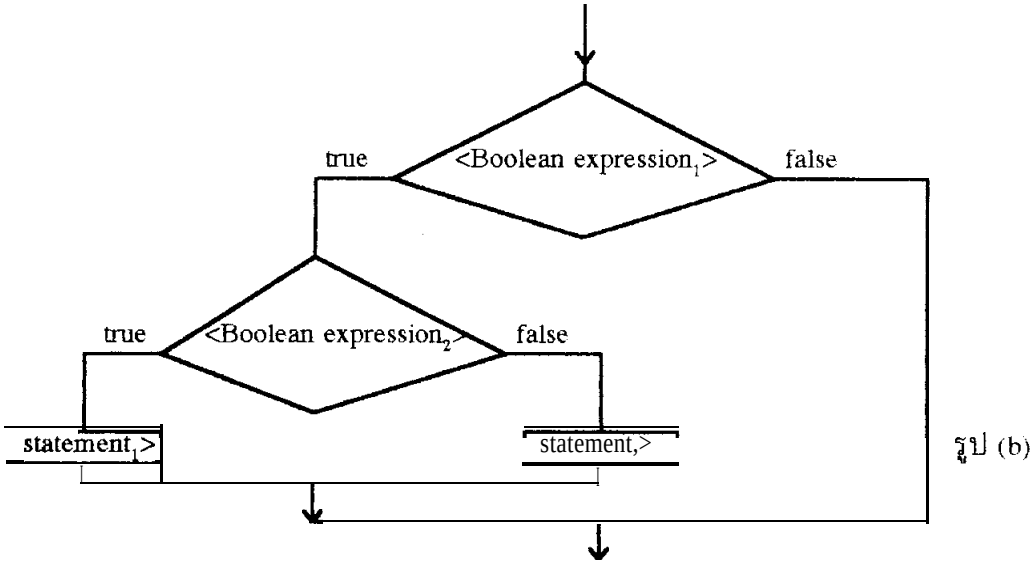
if <Boolean expression> **then** **if** <Boolean expression$_2$> **then** <statement$_1$>

else <statement$_2$>

รูปแบบของ ข้อความสั่งข้างต้นนี้ กำกวม เพราะว่า มันไม่ชัดเจนว่า ผังงาน 2 รูป ในรูป 8-1 เป็นชุดใดที่ต้องการ



รูป (a)



รูป (b)

รูป 8-1 การตีความหมาย สองอย่างของเงื่อนไข ในภาษา ALGOL

(Two interpretations of the ALGOL conditional)

อีกตัวอย่างหนึ่ง ได้แก่ วากยสัมพันธ์ ของภาษา FORTRAN การอ้างถึง A(I, J) อาจหมายถึง การอ้างถึง สมาชิก ของแถวลำดับสองมิติชื่อ A หรือ หมายถึงการเรียก ฟังก์ชันโปรแกรมย่อย ชื่อ A มี I และ J เป็น อาร์กิวเม้นต์ ทั้งนี้เพราะว่า วากยสัมพันธ์ ใน FORTRAN สำหรับการเรียกฟังก์ชัน และ การอ้างถึงแถวลำดับเหมือนกัน ความกำกวมที่คล้ายกันนี้ เกิดขึ้นในภาษาโปรแกรมเกือบทุกภาษา

ความกำกวม ในภาษา FORTRAN และ ALGOL ซึ่งกล่าวถึงข้างต้น ความจริงได้มีการแก้ไขไปแล้ว ในทั้งสองภาษา ข้อความสั่งมีเงื่อนไข ของภาษา ALGOL ความกำกวมทำให้ถูกต้องแล้ว โดย การเปลี่ยน วากยสัมพันธ์ ของภาษา โดย ให้ใส่คู่ อักขระคั่น **begin . . . end** ปิดล้อม ข้อความสั่งมีเงื่อนไข ดังนั้น สิ่งที่เป็นธรรมชาติแต่กำกวมของ การรวมข้อความสั่งมีเงื่อนไขสองชุด ถูกแทนด้วย สิ่งที่เป็นธรรมชาติ น้อยกว่า แต่เป็นโครงสร้างไม่กำกวม เขียนดังนี้

รูป (a)

if <Boolean expression$_1$> **then begin if** <Boolean expression$_2$>
then <statement$_1$> **end else** <statement$_2$>

และรูป (b)

if <Boolean expression$_1$> **then begin if** <Boolean expression$_2$>
then <statement$_1$> **else** <statement$_2$> **end**

ภาษา Ada มีผลเฉลยที่ง่ายกว่า คือ ข้อความสั่ง **if** แต่ละชุด ต้องจบด้วย ตัวคั่น **endif** เสมอ ไม่ว่าจะมีส่วน **else** หรือไม่ก็ตาม

ภาษา Pascal และ PI/1 มีเทคนิคอีกอย่างหนึ่งที่ใช้ แก้ปัญหาความกำกวม กล่าวคือ การตีความใดๆ ถูกเลือกแล้วให้กับโครงสร้างที่กำกวม ในกรณีตัวอย่างข้างต้นนี้ **else** อันสุดท้าย จะคู่กับ **then** ใกล้ที่สุด เพื่อว่า ข้อความสั่งรวม มีความหมายเหมือนกัน คือ รูป (b) ของโครงสร้าง ALGOL ข้างต้น

ความกำกวม ของ ฟังก์ชัน FORTRAN และ การอ้างถึงแถวลำดับ ถูกแก้ไขโดยกฎเกณฑ์คล้ายกัน กล่าวคือ ตัวสร้าง A(I, J) จะหมายถึง การเรียกฟังก์ชัน ถ้าไม่มีการประกาศ ให้กับแถวลำดับ A ทั้งนี้เนื่องจาก แถวลำดับทุกชุด ต้องมีการประกาศ ก่อนนำไปใช้ในโปรแกรม ตัวแปลภาษา อาจตรวจสอบโดยอ่านว่า ความจริง แถวลำดับ A ซึ่งถูกอ้างถึงนั้น มีการประกาศหรือไม่

ถ้าไม่พบ แสดงว่าตัวสร้างนั้น คือ การเรียก ฟังก์ชันภายนอก ชื่อ A การสมมตินี้ จะไม่สามารถตรวจสอบได้ จนกระทั่งเวลาบรรจุ (load time) เมื่อ ฟังก์ชันภายนอกทั้งหมด รวมทั้งฟังก์ชันเฉพาะงานจากคลัง (library function) ถูกโยง (linked) ไปยัง โปรแกรมกระทำการได้ สุดท้าย ถ้าตัวบรรจุ (loader) ไม่พบฟังก์ชัน A ดังนั้นตัวบรรจุ จะให้ (produce) ข้อความผิดพลาด

ภาษา Pascal ใช้เทคนิคแตกต่างกัน เพื่อแยก การเรียกฟังก์ชัน ออกจากการอ้างถึงแถวลำดับ กล่าวคือ การแตกต่างเชิงวากยสัมพันธ์ กระทำขึ้น ดังนี้ วงเล็บใหญ่ [ ] ใช้ปิดรายการดรรชนีล่าง ในการอ้างถึงแถวลำดับ ตัวอย่างเช่น A[I, J] และวงเล็บเล็ก ( ) ใช้ปิด รายการพารามิเตอร์ ของการเรียกฟังก์ชัน ตัวอย่างเช่น A(I, J)

ความกำกวม ปกติ แก้ไขได้โดย เทคนิควิธี หนึ่ง ใน สองวิธีนี้ คือ อาจจะมีการดัดแปรวากยสัมพันธ์บางอย่าง ซึ่งทำให้แยก โครงสร้างกำกวม หรือ การเลือก การตีความคงที่บางอย่าง ซึ่งอาจขึ้นอยู่กับ เนื้อความ และโครงสร้างวากยสัมพันธ์ กำกวม ซึ่งตัดทิ้งไป (เช่น ในเงื่อนไขของ Pascal และการใช้วงเล็บใน FORTRAN) ไม่ว่าจะเป็นเทคนิควิธีไหนก็ตาม ทำให้ความยากเกิดขึ้น

การดัดแปรวากยสัมพันธ์ (modifying the syntax) บ่อยครั้งเท่าที่เป็นไปได้ คือ เกิดตัวสร้างวากยสัมพันธ์ ไม่เป็นธรรมชาติ ตัวอย่างเช่น โปรแกรมเมอร์ ภาษา ALGOL มือใหม่จำนวนมาก จำเป็นต้องใช้ตัวคั่น **begin ... end** ในเงื่อนไขซ้อนกัน ซึ่งไม่เป็นธรรมชาติ และโครงสร้างนำไปสู่ข้อผิดพลาดจำนวนมาก แต่ทางเลือกอีกหนึ่งวิธี ของการตีความโดยนัย สำหรับตัวสร้างกำกวมอาจนำไปสู่ ข้อผิดพลาดปลีกย่อย มากขึ้น ตัวอย่างเช่น โปรแกรมเมอร์ ภาษา FORTRAN มือใหม่ อาจจะงงเมื่อพบข้อความผิดพลาด จากตัวบรรจุ ว่า SUBPROGRAM A NOT FOUND เมื่อเขาไม่ได้ทำการประกาศ ให้กับแถวลำดับ A

8.2 สมาชิกเชิงวากยสัมพันธ์ของภาษา

(Syntactic elements of a language)

สไตล์เชิงวากยสัมพันธ์ ของภาษาทั่วไป คือ เซต โดยการเลือก สมาชิกเชิงวากยสัมพันธ์พื้นฐานต่างๆ เราจะพิจารณาโดยย่อถึงสมาชิกที่มีความสำคัญมากที่สุด

ชุดอักขระ (Character Set)

สิ่งแรกที่จะต้องกระทำ ในการออกแบบวากยสัมพันธ์ของภาษา คือ การเลือกชุดอักขระ

ซึ่งมีใช้กันอย่างกว้างขวาง หลายชุด เช่น ชุดแอสกี (ASCII* set) ชุดอักขระแต่ละชุดประกอบด้วย เซตของอักขระพิเศษ (special character) ที่แตกต่างกัน ใส่เพิ่ม ไปในตัวอักษร (letters) และเลขโดด (digits) ปกติ เซตมาตรฐาน เหล่านี้ จะถูกเลือก หนึ่งชุด บางครั้งชุดอักขระ ซึ่งไม่เป็นมาตรฐาน แต่อย่างใด อาจถูกนำมาใช้ ตัวอย่างเช่น ชุดอักขระซึ่งใช้ในภาษา APL

<u>ตัวอย่าง</u> อักขระพิเศษที่ใช้แทน ตัวดำเนินการของ ภาษา APL

< > = ≠ ≤ ≥ ∨ ∧ ~ ⍲ ⍱ + − × ÷ * ? ∈
↑ ⍳ ⌽ ⍴ ⌈ ∇ ⍎ ⊥ ⊤ / . ○ ∘ ⊞ ⍴ ↓ ⌊ ⍋
[⎕ ( ) ⍉ ⍎ , \ ⌸ ⌹ ⍞ ← → ⍕ ∇ ⍙ : ⍠
⍉ ⌽ ⍕ ⍬ ∩ ∪ ⊃ ⊂ ⍺ ⍵ ≠ ⍋

การเลือกชุดอักขระมีความสำคัญ ในการบอกชนิดของอุปกรณ์ อินพุท และ เอ้าพุท ซึ่งจะสามารถนำมาใช้ ในการทำให้เกิดผลในภาษานั้น ตัวอย่างเช่น ชุดอักขระพื้นฐานของภาษา FORTRAN ใช้ได้บนอุปกรณ์ อินพุท และเอ้าพุท เป็นส่วนใหญ่ ตรงกันข้ามกับชุดอักขระของภาษา APL ไม่สามารถนำมาใช้ได้โดยตรงบนอุปกรณ์ I/O ส่วนใหญ่

การเลือกชุดอักขระมีความสำคัญเช่นกัน ในการบอก จำนวน ของอักขระพิเศษที่ใช้ได้ในโปรแกรม และข้อมูล เช่น อักขระคั่น (delimiters) สัญลักษณ์ตัวดำเนินการ (operator symbols) เป็นต้น การเลือกชุดอักขระนี้ เป็นปัจจัยสำคัญ ในการทำให้ วากยสัมพันธ์ของภาษา เป็นธรรมชาติ และไม่กำกวม ตัวอย่างเช่น การใช้เครื่องหมาย semicolon (;) เพื่อคั่น ข้อความสั่ง และการประกาศ ในภาษา Pascal เหมือนกับ การใช้เครื่องหมายกำกับวรรคตอน อย่างธรรมชาติ เช่นที่ใช้ในภาษาอังกฤษ และยังเป็น ตัวจบข้อความสั่ง ที่เป็นเพียงหนึ่งเดียวเท่านั้นด้วย (and also provides a unique statement terminator)

---

* ASCII (American Standard Code for **Information** Interchange) **อ่านว่า แอสกี หมายถึง รหัสมาตรฐาน เพื่อการสับเปลี่ยนสารสนเทศ**

ชุดอักขระที่ใช้ในภาษา FORTRAN ไม่มีเครื่องหมาย semicolon ดังนั้น ภาษานี้ จึงไม่มีตัวจบที่เหมาะสมให้ใช้ (no suitable terminator is available) โดยเฉพาะ ไม่มีเครื่องหมาย commas (,) ไม่มี period (.) ให้ใช้ เนื่องจากความกำกวม ที่อาจเกิดขึ้นได้ เพราะการใช้ตัวอักขระเหล่านี้

ไอเดนติไฟเออร์ **(Identifiers)**

วากยสัมพันธ์พื้นฐาน สำหรับ ไอเดนติไฟเออร์ คือ สายของตัวอักษร และเลขโดด ซึ่งเริ่มต้นด้วยตัวอักษร (a string of letters and digits beginning with a letter.) เป็นวากยสัมพันธ์ที่ยอมรับอย่างกว้างขวาง ความหลากหลายของภาษา อาจมีการใส่อักขระพิเศษ เช่น period(.) หรือ hyphen (-) ซึ่งอาจละเว้น (optional) เพื่อทำให้ อ่านง่ายขึ้น และข้อจำกัดในเรื่องความยาว

ข้อจำกัดเรื่องความยาวของไอเดนติไฟเออร์ เช่น ภาษา FORTRAN มีการจำกัดให้ใช้อักขระไม่เกิน 6 ตัว บังคับการใช้ ไอเดนติไฟเออร์ ด้วย ค่าช่วยจำ (mnemonic value) เล็กน้อย ใน หลายๆ กรณี ดังนั้น สิ่งสำคัญ คือ ข้อจำกัดที่ทำให้โปรแกรม อ่านง่าย

ภาษา COBOL มีการจำกัด ให้ใช้อักขระ ไม่เกิน 30 ตัว เป็นต้น

ภาษา Pascal ให้นิยาม ไอเดนติไฟเออร์ ดังนี้

[1] An identifier is a sequence of one or more letters (a-z, A-Z) and/or digits (0-9), the first of which must be a letter.

[2] A PL/1 variable name may be any sequence of alphabetic (A-Z, @, #, $), numeric (0-9), and/or break (_) characters, the first of which must be alphabetic.

ตัวอย่าง X, GROSS-PAY, #19

---

[1] "Programming languages" by Allen B. Tucker

สำนักพิมพ์ McGraw-Hill, Singapore, 186 หน้า 21

[2] หนังสือเล่มเดียวกัน หน้า 166

สัญลักษณ์ตัวดำเนินการ (Operator Symbols)

ภาษาโปรแกรม ส่วนใหญ่ ใช้อักขระพิเศษ + และ - เพื่อแทนการดำเนินการคำนวณพื้นฐาน สองอย่าง แต่ส่วนใหญ่แล้ว จะไม่เป็นรูปแบบ รูปแบบเดียวกัน (but beyond that there is almost no uniformity.) การดำเนินการดังเดิม (primitive operations) อาจถูกแทนที่ทั้งหมดด้วยอักขระพิเศษ เช่นในภาษา APL ส่วนไอเดนติไฟเออร์ ซึ่งถูกเลือกมา อาจนำไปใช้กับ การดำเนินการดังเดิมทั้งหมด เช่น PLUS, TIMES ในภาษา LISP เป็นต้น

ภาษาส่วนใหญ่ ยอมรับ การรวมกัน บางอย่าง (some combination) เช่น ใช้อักขระพิเศษสำหรับ ตัวดำเนินการบางตัว, ไอเดนติไฟเออร์ สำหรับสิ่งอื่น และบ่อยครั้ง สายอักขระ บางตัว ใช้ไม่เหมือนกับที่กล่าวมาข้างต้นนี้

ตัวอย่างเช่น ภาษา FORTRAN

+, -, *, /, ** (infix operators)

.EQ., .NE., .LT., .GT., .LE., .GE.

.NOT., .AND., .OR.

ภาษา PL/1 การดำเนินการคำนวณพื้นฐาน (basic arithmetic operations) ถูกแทนด้วยตัวดำเนินการเติมกลาง +, -, *, / และ ** (ยกกำลัง) และ prefix - (นิเสธ) นอกจากนี้ มีเซตของ built-in functions ที่ครบถ้วน สำหรับ square root, absolute value, max, min, trigonometric operations เป็นต้น

การดำเนินการสัมพันธ์ (relational operations) อยู่ในรูปของสัญกรณ์เติมกลาง มี 8 ตัว ดังนี้ =, >, <, >=, <=, ¬=, ¬< และ ¬> ตัวดำเนินการทุกตัว ใช้กับข้อมูลชนิด ตัวเลข (numbers), สายบิต (bit strings) หรือ สายอักขระ (character strings)

การดำเนินการแบบบลู (Boolean operation) และการดำเนินการแบบสายอักขระ (string operationsj

and (&), or (|), และ not (¬) เป็นตัวดำเนินการของสายบิต

ส่วนการดำเนินการต่อกัน (concatenation operation) (II) กระทำได้ทั้ง สายอักขระ และสายบิต

ส่วนการคุมแต่งสายอักขระพื้นฐาน ใช้ built-in functions ดังนี้

LENGTH (which returns the length of an argument string)

INDEX (which searchs a string for a given substring and returns its position.)

SUBSTR (which retrieves a specified substring of a given string.)

ตัวดำเนินการกำหนดค่า (assignment operator) ใช้เครื่องหมาย =

ภาษา Pascal

ตัวดำเนินการกำหนดค่า ใช้เครื่องหมาย :=

การดำเนินการคำนวณ และการดำเนินการสัมพันธ์ นิยามโดยใช้สัญกรณ์เติมกลาง (infix notaion) สำหรับข้อมูล ชนิด integer มีดังนี้

+, - , *, div (division), mod (remainder), =, <> (inequality), <, >, <=, และ >=

ส่วนข้อมูลชนิด real มีเซตของการดำเนินการ โดยใช้สัญกรณ์เติมกลาง ดังนี้

+, -, *, =, <>, <, >, <=, >= และ /(division)

ส่วน built-in functions ได้แก่

SIN (sine), COS (cosine) และ ABS (absolute value)

ข้อมูลชนิดบูลีน ซึ่งมีค่า เป็น 0 หรือ 1 เท่านั้น การดำเนินการแบบบลู มีดังนี้

not, and และ or

ภาษา APL

assignment operator ใช้สัญกรณ์ ←

ตัวอย่าง ถ้า A และ B เป็นแถวลำดับ (array) ซึ่งมีรูปร่างและขนาดเท่ากัน ข้อความสั่ง

C ← A + B

หมายถึง แถวลำดับซึ่งเป็นผลรวมของ A และ B เป็นค่าใหม่ของ C ส่วนค่าเก่าของ C จะถูกทำลาย

ส่วนตัวแปรที่มีดรรชนีกำกับ (a subscripted variable) เฉพาะค่าของสมาชิกตัวที่ถูกเลือกของแถวลำดับเท่านั้น ที่ modified

ตัวอย่าง ข้อความสั่งกำหนดค่า

A[2;3] ← 7

หมายถึง ค่าใหม่ของ A[2;3] เท่ากับ 7

ตัวอย่าง A[2;] ← 7

หมายถึง ค่าใหม่ของสมาชิกทุกตัว ใน แถวที่สอง ของ A มีค่าเท่ากับ 7

ตัวอย่าง การกำหนดค่า

A[1 2; 2 4] ← 7

หมายถึง กำหนดให้ สมาชิก $A_{1,2}$, $A_{1,4}$, $A_{2,2}$ และ $A_{2,4}$ ทุกตัวให้มีค่าเท่ากับ 7

คำหลักและคำสงวน (Key Words and Reserved Words)

คำหลัก หมายถึง ไอเดนติไฟเออร์ ซึ่ง ใช้เป็นส่วนคงที่ ของวากยสัมพันธ์ ของข้อความสั่ง

(A **keyword** is an identifier used as a fixed part of the syntax of a statement.)

ตัวอย่างเช่น IF, THEN และ ELSE ในข้อความสั่ง มีเงื่อนไขของภาษา PL/1 หรือคำว่า DO ซึ่งอยู่ตอนต้น ในข้อความสั่งแบบวนซ้ำของ ภาษา FORTRAN

คำหลัก จะเป็น คำสงวน ถ้า นำไปใช้เป็นไอเดนติไฟเออร์ ซึ่งโปรแกรมเมอร์เลือกไม่ได้

(A keyword is a **reserved word** if it may not also be used as a programmer-chosen identifier.)

คำหลัก สนับสนุน (serve) วัตถุประสงค์ หลายอย่างในภาษาโปรแกรม โดยปกติ ข้อความสั่งส่วนมาก จะเริ่มต้นด้วย คำหลัก เป็นการกำหนด ชนิดของข้อความสั่ง เช่น READ, IF, GOTO เป็นต้น ส่วนคำหลักอื่นๆ อาจอยู่ในข้อความสั่ง ทำหน้าที่เสมือนกับเป็น อักขระคั่น เช่นคำว่า THEN และ ELSE ใน ข้อความสั่งมีเงื่อนไข

การวิเคราะห์เชิงวากยสัมพันธ์ ระหว่างการแปลภาษา จะกระทำง่ายขึ้นโดยการใช้คำสงวน ตัวอย่างเช่น การวิเคราะห์เชิงวากยสัมพันธ์ ของภาษา FORTRAN จะกระทำยาก ด้วยความจริงที่ว่า ข้อความสั่ง ซึ่งขึ้นต้นด้วยคำว่า DO หรือ IF จริงๆ แล้ว อาจจะ**ไม่ใช่** ข้อความสั่งแบบวนซ้ำ (iteration) หรือ ข้อความสั่งมีเงื่อนไข ทั้งนี้ เพราะว่า DO และ IF ไม่ใช่คำสงวน* โปรแกรมเมอร์อาจเลือก คำเหล่านี้ ถูกต้องตามกฎ ให้เป็นชื่อตัวแปร (variable names)

ตัวอย่าง

DO 10 I = 1

---

* **ข้อสังเกต** ภาษา FORTRAN ไม่มีคำสงวน

ข้อความสั่ งข้างต้นนี้ อาจหมายถึง ข้อความสั่ง DO มี I เป็นตัวแปรควบคุม หรือ ข้อความสั่งกำหนดค่า มี DO10I เป็น ชื่อตัวแปร ก็ได้

สำหรับภาษา COBOL ใช้คำสงวนมาก และเนื่องจาก ไอเดนติไฟเออร์ จำนวนมาก เป็นตัวสงวน จึงยากที่จะจำได้ทั้งหมด ดังนั้น บ่อยครั้งที่เราอาจจะเลือก ไอเดนติไฟเออร์ ซึ่งสงวนไว้เป็น ชื่อตัวแปร อย่างไรก็ตาม ความยากอันดับแรก (the primary difficulty) ของคำสงวนเกิดขึ้นเมื่อภาษานั้น มี ความจำเป็นต้อง ขยายเพิ่ม เพื่อรวม ข้อความสั่งใหม่ๆ ซึ่งใช้คำสงวนตัวใหม่ ตัวอย่างเช่น ตลอดช่วงเวลาที่ผ่านมา ภาษา COBOL มีการแก้ไข (revised) เพื่อเตรียมการปรับให้เป็นมาตรฐาน การเพิ่มคำสงวนตัวใหม่ ให้กับภาษาโปรแกรม หมายถึงว่า โปรแกรมเก่าทุกโปรแกรม ซึ่งใช้ ไอเดนติไฟเออร์ ตัวนั้น เป็น ชื่อตัวแปร (หรือ ชื่ออื่น) จะไม่ถูกต้องเชิงวากยสัมพันธ์ อีกต่อไป ถึงแม้ว่า จะไม่มีการดัดแปร (modified) โปรแกรมแต่อย่างใดก็ตาม คอมไพเลอร์ ซึ่ง implements ภาษาซึ่งมีส่วนขยาย จะปฏิเสธ (reject) โปรแกรมเก่าๆ และบังคับ ให้มีการดัดแปรโปรแกรมเหล่านั้น

คำอธิบายและคำรบกวน (Comments and Noise Words)

การรวมคอมเมนต์ เข้าไว้ในตัวโปรแกรม เป็นส่วนที่สำคัญของการทำเอกสาร ของโปรแกรม

(Inclusion of comments in a program is an important part of its documentation.)

ภาษาโปรแกรม อาจยอมให้มี คอมเมนต์ ได้หลายวิธี ดังนี้

1) แยกเป็น บรรทัดคอมเมนต์ ต่างหาก ใน โปรแกรม (as separate comment lines in the program) ตัวอย่างเช่น ในภาษา FORTRAN โดยการใส่อักษร C ในสดมภ์ที่ 1 ของบรรทัดนั้น หรือ ภาษา COBOL โดยใส่เครื่องหมาย * ที่สดมภ์ที่ 7 ของบรรทัดคอมเมนต์

2) คั่นด้วย เครื่องหมายพิเศษ (delimited by special markers) ตัวอย่างเช่น ภาษา Pascal ใช้ { และ) ภาษา PL/1 ใช้ /* และ */ โดยไม่เกี่ยวข้องกับ ขอบเขตบรรทัด (line boundaries)

3) เริ่มต้นที่ใดก็ได้ บนบรรทัดนั้น และสิ้นสุด ด้วยการจบบรรทัด (beginning anywhere on a line but terminated by the end of the line) ตัวอย่างเช่นในภาษา Ada

ทางเลือก วิธีที่สามนั้น รวมกฎข้อแรกไว้ด้วย และยังยอมให้มี คอมเมนต์ขนาดสั้น (short comment) รวมเข้าไปด้วย หลังจากเขียนข้อความสั่ง หรือ หลังจากการประกาศ บนบรรทัด

ทางเลือก วิธีที่สอง มีข้อไม่ดี ตรงที่ ถ้ามีการลืมใส่ ตัวคั่นเพื่อจบ (terminating delimiter)

บนคอมเมนต์ จะทำให้ ข้อความสั่งต่างๆ ที่ตามมา (จนถึง จบคอมเมนต์ถัดไป) เป็น "comments" ดังนั้น ถึงแม้ว่า โปรแกรมจะถูกต้องขณะอ่าน แต่มันจะไม่ถูกต้อง ขณะถูกแปล หรือถูกกระทำการ

ตัวรบกวน หมายถึง คำละเว้นได้ ซึ่งอาจใส่ใน ข้อความสั่ง เพื่อให้ การอ่านง่ายขึ้น

(**Noise words** are optional words which may be inserted in statements to improved redability.)

ภาษา COBOL มีคำเช่นนี้ จำนวนมาก ตัวอย่างเช่น ในข้อความสั่ง GOTO ซึ่งมีรูปแบบดังนี้

GO TO <label>

ในที่นี้ คำหลัก GO ต้องเขียน แต่คำว่า TO เป็นคำรบกวน อาจจะละเว้นได้ แต่ใส่ไว้ในข้อความสั่ง เพื่อให้ อ่านง่ายขึ้นเท่านั้น

ตัวอย่าง ข้อความทั้ง 4 บรรทัดข้างล่างนี้ มีความหมายเหมือนกัน

```
IF A IS GREATER THAN B
IF A IS GREATER B
IF A GREATER THAN B
IF A GREATER B
```

แสดงว่า IS และ THAN เป็นคำซึ่งอาจจะละเว้นได้ ดังนั้น คำว่า IS และ THAN เป็นคำรบกวน

อักขระว่าง (Blanks หรือ spaces)

กฎการใช้อักขระว่าง แปรผันอย่างกว้างขวาง ระหว่างภาษาโปรแกรมต่างๆ (Rules on the use of blanks vary widely between languages.) ตัวอย่างเช่น ภาษา FORTRAN ตัวอักขระว่าง ไม่สำคัญ อยู่ตรงไหนก็ได้ ใน โปรแกรม ยกเว้น ใน ข้อมูลชนิดสายอักขระ

(In FORTRAN, **blanks** are not significant anywhere except in literal character string data.)

ตัวอย่าง ข้อความสั่งกำหนดค่า

DO 10 I = 1

หรือ DO10I = 1

ทั้งสองบรรทัดนี้ มีความหมายเหมือนกัน

ตัวอย่าง

GOTO 100

หรือ GO TO 100

ทั้งสองบรรทัดนี้ มีความหมายเหมือนกัน

ภาษาอื่นๆ ใช้อักขระว่าง เป็น ตัวคั่น (separators) ดังนั้น อักขระว่าง จึงมีบทบาทเชิงวากยสัมพันธ์ ที่สำคัญ

ในภาษา SNOBOL4 การดำเนินการดังเดิมชนิดหนึ่งคือ การต่อเรียง (concatenation) แทนด้วย อักขระว่าง และอักขระว่าง ยังใช้เป็น ตัวคั่น ระหว่าง สมาชิกของ ข้อความสั่งด้วย (สิ่งนี้นำไปสู่ความสับสน อย่างมาก)

อักขระคั่นและการรวมกลุ่ม (Delimiters and Brackets)

อักขระคั่น หมายถึง สมาชิกเชิงวากยสัมพันธ์ ซึ่ง ใช้ทำเครื่องหมาย การเริ่มต้น หรือ การจบ หน่วยวากยสัมพันธ์ บางอย่าง เช่น ข้อความสั่ง หรือ นิพจน์

(A **delimiter** is a syntactic element used simply to mark the beginning or end of some syntactic unit such as a statement or expression.)

การรวมกลุ่ม หมายถึง อักขระคั่นเป็นคู่ ตัวอย่างเช่น คู่ของวงเล็บเล็ก หรือคู่ begin ... end

(Brackets are paired delimeters, e.g. parentheses or **begin ... end** pairs.)

อักขระคั่น อาจนำมาใช้ เพื่อให้ การอ่านง่ายขึ้น หรือ การวิเคราะห์เชิงวากยสัมพันธ์ ทำง่ายขึ้น แต่ส่วนใหญ่แล้ว สมาชิกเชิงวากยสัมพันธ์ เหล่านี้ สนอง (serve) วัตถุประสงค์ที่สำคัญกว่าคือ ขจัดความกำกวม โดยทำให้การนิยาม ขอบเขต ของ ตัวสร้างวากยสัมพันธ์ (syntactic construct) ชัดแจ้งขึ้น

ตัวอย่างเช่น ภาษา Ada กำหนดว่า ข้อความสั่ง if ต้องปิดด้วย อักขระคั่น endif เสมอ

ตัวอย่างเช่น ตัวสร้างเชิงวากยสัมพันธ์ ชนิด compound statement ของภาษา Pascal กำหนดว่า ต้องปิดล้อมด้วย คู่ของ **begin ... end** เสมอ

รูปแบบเขตอิสระ และรูปแบบเขตคงที่

(Free- and Fixed-Field Formts)

วากยสัมพันธ์ จะเป็นรูปแบบเขตอิสระ ถ้าข้อความสั่งของโปรแกรมเขียนที่ใดก็ได้บนบรรทัดอินพุท โดยไม่มีการกำหนดตำแหน่ง บนบรรทัด หรือ การแยกระหว่างบรรทัด

(A **syntax** is **free-field** if program statements may be written anywhere on an input line without regard for positioning on the line or for breaks between lines.)

ตัวอย่าง ภาษาซึ่งมีรูปแบบอิสระเชิงวากยสัมพันธ์

Pascal , PL/1

วากยสัมพันธ์รูปแบบเขตคงที่ จะมีการกำหนดตำแหน่งบนบรรทัดอินพุท ซึ่งจะใส่สารสนเทศ

(A **fixed-free** syntax utilizes the positioning on an input line to convey information.)

วากยสัมพันธ์เขตคงที่ ชนิดเข้มงวดนั้น สมาชิกแต่ละตัวของข้อความสั่ง ต้องอยู่ภายในส่วนซึ่งกำหนดให้ ของบรรทัดอินพุท ซึ่งส่วนใหญ่ จะเห็นได้จาก ภาษาแอสเซมบลี แต่ภาษาจำนวนมากกว่า ใช้รูปแบบเขตคงที่ บางส่วน ตัวอย่างเช่น ภาษา FORTRAN อักขระห้าตัวแรกของแต่ละบรรทัด สำรองไว้สำหรับ เลเบลข้อความสั่ง (statement label) บางครั้ง อักขระตัวแรกของ บรรทัดอินพุท กำหนด ความสำคัญพิเศษไว้ ตัวอย่างเช่น ภาษา SNOBOL4, เลเบลข้อความสั่ง คอมเมนต์ และบรรทัดต่อ ซึ่ง แสดงให้เห็นชัดเจน ด้วยการใส่อักขระ หนึ่งตัว ในตำแหน่งที่ 1 ของบรรทัด

ภาษา COBOL มีวากยสัมพันธ์เป็นรูปแบบเขตคงที่ ซึ่งกำหนดว่า ส่วนของหัวเรื่อง (header), ตัวชี้บอกระดับ (level indicator), ชื่อเซคชัน (section name), ชื่อโปรซีเดอร์ (procedure name) ทั้งหมดนี้ ต้องเขียนที่ margin A (หมายถึงสดมภ์ที่ 8-11) ส่วนอื่นๆ นอกเหนือจากนี้ให้เขียนที่ margin B (หมายถึงสดมภ์ที่ 12-72) และตำแหน่งที่ 7 สำหรับใส่เครื่องหมาย - (hyphen), เพื่อบอก บรรทัดต่อ หรือ ใส่เครื่องหมาย * (asterisk) เพื่อแสดงบรรทัดคอมเมนต์ เหล่านี้เป็นต้น

นิพจน์ (Expressions)

นิพจน์ หมายถึง บล็อกสร้างวากยสัมพันธ์พื้นฐาน ซึ่งประกอบกันเป็น ข้อความสั่ง (และบางครั้งเป็น โปรแกรม)

(Expressions are the basic syntactic buildmg block from which statements (and sometimes programs) are built.)

รูปแบบวากยสัมพันธ์ ของ นิพจน์ มีหลากหลาย ได้แก่

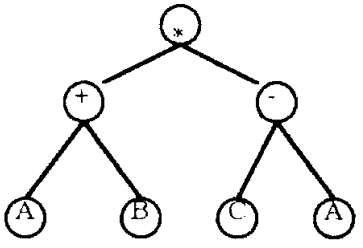
สัญกรณ์เติมหน้า รูปแบบนี้ อันดับแรกเขียนสัญลักษณ์การดำเนินการ จากนั้นตามด้วยตัวถูกดำเนินการ ในลำดับจากซ้ายไปขวา ถ้าตัวถูกดำเนินการหนึ่งตัว เป็นการดำเนินการโดยตัวมันเองด้วย ตัวถูกดำเนินการ ให้ใช้กฎเดียวกัน ในการเขียน

(In **prefix** notation one writes the operation symbol first. followed by the operands in order from left to right. If an operand is itself an operation with operands. then the same rules apply.)

1) **สัญกรณ์เติมหน้าแบบธรรมดา** รูปแบบนี้ ลำดับของตัวถูกดำเนินการ ต้องอยู่ภายในเครื่องหมายวงเล็บ ตัวถูกดำเนินการแต่ละตัว คั่นด้วยเครื่องหมาย comma

(In **ordinary prefix** one simply encloses the sequence of operands in parentheses, separatmg operands by commas.)

ตัวอย่าง



รูป 9.1 โครงสร้างรูปต้นไม้ ของ นิพจน์ อย่างง่าย (A + B) * (C - A)

เขียน สัญกรณ์ เติมหน้าแบบธรรมดา ดังนี้

*(+(A,B),-(C,A))

2) **สัญกรณ์เคมบริจ โพลิช** เริ่มจากสัญกรณ์เติมหน้าแบบธรรมดา แล้วเอาวงเล็บเปิด ซึ่งอยู่ข้างหลัง สัญกรณ์ตัวดำเนินการ มาไว้ข้างหน้า สัญลักษณ์ตัวดำเนินการ จากนั้น ตัดเครื่องหมาย comma ทุกตัวทิ้งให้หมด

(In Cambridge Polish notation the left parenthesis following an operator symbol is moved to immediately precede it, and the commas separating operands are deleted.)

ดังนั้น รูปแบบของนิพจน์ จึงดูเหมือนกับ เซตของรายการต่างๆ ที่ซ้อนกัน แต่ละรายการขึ้นต้นด้วย สัญกรณ์ตัวดำเนินการหนึ่งตัว แล้วตามด้วย รายการ ซึ่งแทนตัวถูกดำเนินการ จากตัวอย่างข้างต้น รูปแบบสัญกรณ์เคมบริจ โพลิช เขียนดังนี้

(*(+AB)(-CA)

นิพจน์ในภาษา LISP ใช้รูปแบบ สัญกรณ์เคมบริจ โพลิช

3) **สัญกรณ์โพลิช หรือ สัญกรณ์ไม่มีวงเล็บกำกับ หรือ สัญกรณ์เติมหน้า** (Polish or parenthesis-free or prefix notation) จากรูปแบบ ในข้อ 2 ตัดวงเล็บทิ้งให้หมด สมมติว่า ถ้าเราทราบตัวถูกดำเนินการ ซึ่งเป็นค่าคงที่ ของตัวดำเนินการแต่ละตัว จะเห็นว่า ไม่มีความจำเป็น ต้องใช้เครื่องหมายวงเล็บ กำกับ ดังนั้น โพลิช จากตัวอย่างข้างต้น เขียนดังนี้

*+AB-CA

สัญกรณ์รูปแบบนี้ คิดค้น โดย นักคณิตศาสตร์ชาวโปแลนด์ ชื่อ Lukasiewiez ดังนั้น คำว่า "Polish" จึงถูกนำมาใช้เป็นชื่อ ของนิพจน์รูปแบบนี้

ตัวอย่าง จากสูตร ในการคำนวณหาราก ตัวหนึ่ง ของ สมการกำลังสอง ข้างล่างนี้

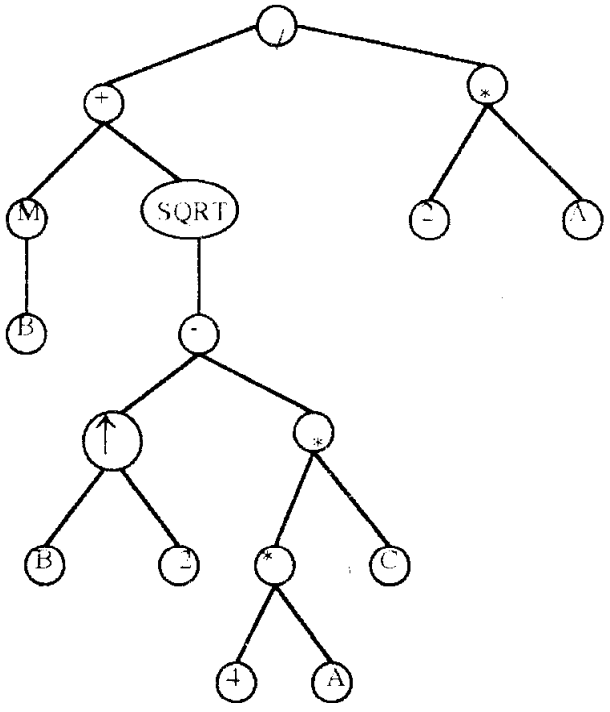
$$root = (-B + \sqrt{B^2 - 4AC})/2A$$

ภาษา FORTRAN เขียนข้อความสั่งกำหนดค่า ดังนี้

ROOT = (-B + SQRT(B**2 - 4*A*C))/(2*A)

เมื่อให้ ↑ เป็นเครื่องหมายยกกำลัง, M เป็น unary minus

โครงสร้างรูปต้นไม้ของ นิพจน์คำนวณทางขวามือของเครื่องหมาย = วาดรูปดังนี้



สัญกรณ์เติมหน้า แบบธรรมดา คือ

/ (+ (M (B) , SQRT (- (↑ (B , 2) , * (* (4 , A) , C)))) , * (2 , A))

สัญกรณ์เคมบริจโพลิช เขียนดังนี้

(/ (+ (M B) (SQRT (- (↑ B 2) (* (* 4 A) C)))) (* 2 A))

สัญกรณ์โพลิช เขียนดังนี้

/ + M B SQRT - ↑ B 2 * * 4 A C * 2 A

ความจริงซึ่งเห็นชัดเจนมากที่สุด เกี่ยวกับ นิพจน์เติมหน้าเหล่านี้ คือ สัญกรณ์เหล่านี้ ถอดรหัสยาก (difficult to decipher) นั่นคือ ถ้าเราไม่ทราบจำนวน ตัวถูกดำเนินการ ของ สัญลักษณ์ตัวดำเนินการแต่ละตัว เราจะไม่สามารถ ถอดรหัส รูปแบบโพลิช ได้เลย

ส่วนนิพจน์รูปแบบ สัญกรณ์เติมหน้าแบบธรรมดา และนิพจน์แบบโพลิชเคมบริจ ใช้เครื่องหมายวงเล็บ จำนวนมาก ซึ่ง เป็นสัญลักษณ์ซึ่งเราไม่คุ้นเคยมาก เท่ากับ สัญกรณ์เติมกลาง (infix notation)

อย่างไรก็ตาม ไม่ใช่ว่าสัญกรณ์เติมหน้าไม่มีค่า จริงๆ แล้ว สัญกรณ์เติมหน้าแบบธรรมดา เป็น สัญกรณ์คณิตศาสตร์มาตรฐาน (standard mathematical notation) สำหรับการดำเนินการ

ส่วนใหญ่มากกว่า การคำเนินการคณิตศาสตร์แบบทวิภาค และการคำเนินการเชิงตรรกะ ตัวอย่าง เช่น f(x,y,z) เขียนด้วยรูปแบบสัญกรณ์เติมหน้า สิ่งที่สำคัญมากกว่านั้นคือ สัญกรณ์เติมหน้า ใช้แทนการคำเนินการ ซึ่งมีตัวถูกคำเนนการ จำนวนกี่ตัวก็ได้ ดังนั้น สัญกรณ์ ชนิดนี้ จึงใช้กฎวากย-สัมพันธ์เพียงข้อเดียวเท่านั้น ที่สมบูรณ์ ในการเรียนรู้ เพื่อที่จะเขียนนิพจน์ใดๆ

ตัวอย่างเช่น ภาษา LISP ตัวโปรแกรมคือนิพจน์ต่างๆ โดยใช้ สัญกรณ์ เคมบริจโพลิช เท่านั้น เป็นหลักสำหรับเขียนนิพจน์ต่าง ๆ สัญกรณ์เติมหน้า เป็นรูปแบบที่ค่อนข้างง่าย ต่อกลไก การถอดรหัส และด้วยเหตุผลนี้ การแปล นิพจน์เติมหน้า ให้เป็น ลำดับรหัสอย่างง่าย จึงทำให้เป็นผลสำเร็จง่าย ภาษา SNOBOL4 ใช้รูปแบบ prefix โดยตรง ระหว่างการกระทำการ ให้เป็นนิพจน์รูปแบบกระทำการได้

สัญกรณ์เติมหลัง หรือ สัญกรณ์โพลิชผันกลับ (Postfix notation or Suffix notation or Reverse Polish Notation)

นิพจน์รูปแบบนี้ คล้ายกับ สัญกรณ์เติมหน้า ยกเว้นเฉพาะ สัญลักษณ์ตัวดำเนินการ อยู่ข้างหลังรายการของตัวถูกคำเนินการ เท่านั้น

(Postfix notation is similar to prefix notation except that the operation symbol follows the list of operands.)

ตัวอย่าง นิพจน์ของต้นไม้ ในรูป -1 เขียนสัญกรณ์เติมหลัง ดังนี้

((A , B) + , (C , A) -) *

หรือ A B + C A - *

สัญกรณ์เติมหลัง ไม่ใช่การแทนที่วากยสัมพันธ์โดยทั่วไป ของนิพจน์ในภาษาโปรแกรมแต่ความสำคัญอยู่ที่ มันเป็นมูลฐาน (basis) สำหรับ การแทนที่ ของนิพจน์ ณ เวลากระทำการโดยเฉพาะ

สัญกรณ์เติมกลาง (Infix notation)

นิพจน์รูปแบบนี้ เหมาะสม เฉพาะกับ การคำเนินการ แบบทวิภาค (binary operation) ได้แก่ การคำเนินการซึ่งมีตัวถูกคำเนินการ สองตัว การเขียนนิพจน์แบบสัญกรณ์เติมกลางนั้น สัญกรณ์ ตัวคำเนินการ จะอยู่ระหว่างตัวถูกคำเนินการ สองตัว เนื่องจากสัญกรณ์เติมกลาง สำหรับการคำเนินการคำนวณ, เปรียบเทียบ และเชิงตรรกะ พื้นฐาน ใช้กันปกติ ในวิชาคณิตศาสตร์

ดังนั้น สัญกรณ์ สำหรับ การคำเนินการเหล่านี้ จึงมีการยอมรับกัน อย่างกว้างขวาง ในภาษาโปรแกรม และ **ในบางกรณี ได้ขยายไปยัง** การคำเนินการ อื่นๆ เช่นเดียวกัน

ตัวอย่าง นิพจน์ของต้นไม้ ในรูป 9-1 เขียนสัญกรณ์เติมกลาง ดังนี้

(A + B) * (C - A)

ถึงแม้ว่า สัญกรณ์เติมกลาง จะใช้ ในภาษาโปรแกรมทั่วไป แต่ รูปแบบนี้ นำไปสู่ปัญหา ดังนี้

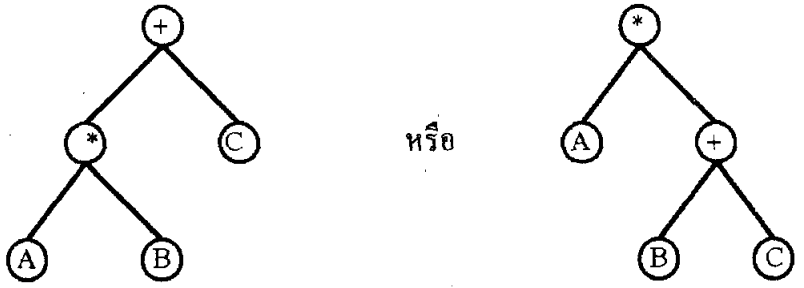
1. เนื่องจาก สัญกรณ์เติมกลาง เหมาะสม เฉพาะกับ ตัวคำเนินการแบบทวิภาคเท่านั้น ภาษาโปรแกรม ไม่สามารถใช้ เฉพาะสัญกรณ์เติมกลางเท่านั้น แต่จำเป็นต้องรวม (combine) สัญกรณ์เติมกลาง และ สัญกรณ์เติมหน้า หรือ สัญกรณ์เติมหลัง เข้าด้วยกัน การผสมกัน ทำให้ การแปลภาษา ที่สมนัยกัน มีความซับซ้อนมากขึ้น

2. เมื่อนิพจน์นั้น มี ตัวคำเนินการ เติมกลาง มากกว่า หนึ่งตัว สัญกรณ์นั้น จะกำกวม ถ้าไม่มีวงเล็บกำกับ

ตัวอย่าง นิพจน์เติมกลาง

A * B + C

อาจแทนด้วย ต้นไม้ รูป 9-2 รูปใดรูปหนึ่ง



รูป 9-2

เครื่องหมายวงเล็บ ถูกนำมาใช้ เพื่อระบุ ความชัดแจ้ง การจัดกลุ่ม ของ ตัวคำเนินการ และ ตัวถูกคำเนินการ เช่น (A * B) + C หรือ A * (B + C) แต่ใน นิพจน์ ซับซ้อน (complex expression) การซ้อนกัน หลายๆ ชั้น ของเครื่องหมายวงเล็บ ผลลัพธ์ อาจสับสนได้ ด้วยเหตุผล

เช่นนี้ โดยปกติ ภาษาโปรแกรมต่างๆ จะมีกฎควบคุมโดยนัย (implicit control rules) สองข้อ ซึ่งทำให้ ส่วนใหญ่ ไม่จำเป็นต้องใช้วงเล็บ ดังนี้

a) **ลำดับชั้นของการดำเนินการ (กฎการทำก่อน)**

(Hierarchy of operations (precedence rules))

ตัวดำเนินการ ในนิพจน์ ถูกวาง ใน ลำดับชั้น หรือ อันดับ การทำก่อน ตัวอย่างเช่น ลำดับชั้นของตัวดำเนินการใน ภาษา Ada (ดูตารางข้างล่างนี้)

Ada Hierarchy of Operations

Highest precedence level	**	(exponent&ion)
	*, /	(multiplication, diision)
	+, -, not	(unary operations)
	+, -	(addition, subtraction,!
	=, <, ≤, >, ≥	(relational operations)
I Lowest precedence level	and, or, xor	(Boolean operations)

ในนิพจน์ซึ่งมี ตัวดำเนินการต่างๆ ในลำดับชั้นมากกว่า หนึ่งระดับ จากกฎ โดยนัย ที่ว่า ตัวดำเนินการ ซึ่งมีการทำก่อนสูงกว่า จะถูกกระทำเป็นอันดับแรก ดังนั้น ใน A * B + C, ตัวดำเนินการ * มีลำดับชั้น สูงกว่า + จึงต้องกระทำการ (execute) การดำเนินการ * เป็นอันดับแรก

b) **การเปลี่ยนกลุ่ม** (Associativity)

ในนิพจน์ ซึ่งมี การดำเนินการต่างๆ ระดับเดียวกัน ในลำดับชั้น มีการเพิ่มกฎ โดยนัย สำหรับการเปลี่ยนกลุ่ม ที่จำเป็น เพื่อนิยาม อันดับของ การดำเนินการที่สมบูรณ์ ตัวอย่างเช่น นิพจน์ A - B - C สิ่งแรก จะทำการลบ ตัวที่หนึ่ง หรือ จะทำการลบตัวที่สอง? กฎโดยนัยทั่วไป ส่วนใหญ่ เป็น left-to-right associativity ดังนั้น นิพจน์ A - B - C หมายถึง (A - B) - C

อย่างไรก็ตาม ภาษา APL ใช้กฎ right-to-left associativity ดังนั้น A - B - C หมายถึง A - (B - C) คือ A - B + C

สัญกรณ์ ของนิพจน์ แต่ละรูปแบบ มี ความยาก ของตัวมันเอง

สัญกรณ์เติมกลาง ใช้กับกฎ การทำก่อนโดยนัย และกฎการเปลี่ยนกลุ่ม และใช้เครื่องหมายวงเล็บชัดแจ้ง ทำให้การแทนที่ ค่อนข้างเป็นธรรมชาติ สำหรับ นิพจน์ คำนวณ, เปรียบเทียบ

และเชิงตรรกะ มากที่สุด อย่างไรก็ตาม ความจำเป็น สำหรับ กฎโดยนัยที่ซับซ้อน และความจำเป็น ที่ต้องใช้ สัญกรณ์เติมหน้า หรือ สัญกรณ์รูปแบบอื่น สำหรับการดำเนินการซึ่งไม่ใช่แบบทวิภาค ทำให้การแปลนิพจน์นั้น ซับซ้อน สัญกรณ์เติมกลาง ซึ่งไม่มีกฎโดยนัย (ได้แก่ รูปแบบ full parenthesization) ค่อนข้างรุงรัง เพราะว่าต้องใช้เครื่องหมายวงเล็บกำกับ จำนวนมาก

ส่วนสัญกรณ์เติมหน้าแบบธรรมดา และสัญกรณ์เคมบริจโพลิช ทั้งคู่มีปัญหาเรื่องวงเล็บเช่นเดียวกัน สัญลักษณ์โพลิช ไม่ใช้วงเล็บ แต่เราต้องทราบล่วงหน้าว่า ตัวดำเนินการแต่ละตัว ต้องการตัวถูกดำเนินการกี่ตัว เป็นเงื่อนไข ที่ยาก เมื่อเกี่ยวข้องกับ การดำเนินการ ซึ่งโปรแกรมเมอร์นิยามขึ้นเอง นอกจากนี้แล้ว การขาดหลักของโครงสร้าง ทำให้การอ่าน นิจน์โพลิช ที่ซับซ้อนยาก สัญกรณ์เติมหน้า และสัญกรณ์เติมหลัง มี ข้อดีของการประยุกต์ ให้กับการดำเนินการ ด้วยตัวถูกดำเนินการ จำนวนต่างๆ กัน

ภาษา APL ใช้สัญกรณ์เติมกลาง ใน การดำเนินการดังเดิม และการดำเนินการซึ่งโปรแกรมเมอร์นิยามเอง แต่ไม่มีลำดับชั้นของการดำเนินการ และใช้กฎ right-to-left associativity

ภาษา LISP ใช้เฉพาะสัญกรณ์รูปแบบ เคมบริจโพลิช

ภาษาส่วนใหญ่ ยอมรับ สัญกรณ์เติมกลาง สำหรับ การดำเนินการคำนวณ, ตรรกะ และสัมพันธ์

สัญกรณ์โพลิชเติมหน้า สำหรับ ตัวดำเนินการ ชนิดเอกภาพชนิด built-in เหมือนกับ negation และ logical not

สัญกรณ์เติมหน้าแบบธรรมดา สำหรับ สิ่งอื่นๆ รวมทั้งการดำนินการชนิดโปรแกรมเมอร์นิยามเอง และ built-in ฟังก์ชัน เช่น sine และ cosine

ภาษา SNOBOL4 แทน การดำเนินการสัมพันธ์ (relational operations) ด้วย สัญกรณ์เติมหน้าแบบธรรมดา แต่เพิ่มตัวดำเนินการแบบเอกภาพ และแบบทวิภาคใหม่ จำนวนหนึ่ง ในรูปแบบสัญกรณ์ โพลิชเติมหน้า และสัญกรณ์เติมกลาง ตามลำดับ

จะเห็นว่า ยังไม่มีข้อตกลงทั่วไป ที่ว่า สัญกรณ์รูปแบบไหน ดีที่สุด สำหรับ นิพจน์ ในภาษาโปรแกรม

โปรดสังเกตว่า รูปแบบเติมกลาง เติมหน้า และเติมท้าย และความหลากหลาย ในบางภาษา เช่น LISP และ APL ยอมรับ วากยสัมพันธ์ หนึ่งอย่าง สำหรับ การสร้างนิพจน์ ซึ่งใช้เป็นแบบเดียวกัน (uniformly)

แต่ที่ใช้ในภาษาโปรแกรมส่วนใหญ่ จะเป็นรูปแบบผสม ตัวอย่างเช่น arithemetic primitives ใช้รูปแบบเติมกลาง, function calls บางอย่างใช้ รูปแบบเติมหน้า เป็นต้น การแปลภาษา ซึ่งเป็นรูปแบบผสมนี้ มีความยากมากขึ้น แต่ข้อดีคือ ปกติแล้ว นิพจน์เหล่านี้ อ่านง่ายกว่า

นอกจาก ความแตกต่างวากยสัมพันธ์ที่เห็นชัดเจน ใน รูปแบบนิพจน์ เหล่านี้แล้ว ยังมีความแตกต่าง ระหว่าง ภาษาโปรแกรม ใน การกำหนดความสำคัญ ให้กับนิพจน์ ตัวอย่างเช่น ใน ภาษา LISP และ APL นิพจน์ หมายถึง โครงสร้างวากยสัมพันธ์ส่วนกลาง (central syntactic structure) โดยรวมแล้ว ถ้ามีการใช้ข้อความสั่งต่างๆ จะประกอบด้วย นิพจน์รูปแบบเดียว ดังนั้น โปรแกรมภาษา APL จึงเป็นลำดับของนิพจน์ (in APL a program is simply a sequence of expressions.)

ส่วนภาษา FORTRAN และ COBOL, กับตรงกันข้าม นิพจน์ มีความสำคัญ น้อยกว่ามาก คือข้อความสั่งต่างๆ จะเป็นรูปแบบเชิงวากยสัมพันธ์หลัก (primary syntactic form) และ นิพจน์เหล่านี้ กลับนำมาใช้ภายใน ข้อความสั่ง เฉพาะเมื่อมีค่าต้องทำการคำนวณ

ข้อความสั่ง (Statements)

ข้อความสั่ง หมายถึง ส่วนประกอบเชิงวากยสัมพันธ์ที่สำคัญมากที่สุดในภาษา ส่วนใหญ่ วากยสัมพันธ์ของ ข้อความสั่ง มีผลกระทบรุนแรง บนกฎเกณฑ์ การอ่านง่าย และการเขียนง่าย โดยรวม ของภาษาโปรแกรม

(Statements are the most prominent syntactic component in most languages, and their syntax has a critical effect on the overall regularity, readability, and writeability of the language.)

บางภาษา มีรูปแบบข้อความสั่งหลักเพียง หนึ่งชนิด เท่านั้น ในขณะที่ภาษาโปรแกรมอื่นๆ มีวากยสัมพันธ์ต่างๆ กัน สำหรับชนิดข้อความสั่งต่างๆ กัน ภาษาโปรแกรมชนิดแรกนั้น เน้นที่กฎเกณฑ์ ส่วนภาษาโปรแกรมรูปแบบหลัง เน้น การอ่านง่าย

ตัวอย่างเช่น ภาษา SNOBOL4 มี วากยสัมพันธ์ ข้อความสั่งพื้นฐาน เพียง หนึ่งชนิดเท่านั้นคือ **ข้อความสั่ง แบบรูป-การจับคู่-แทนที่** (the pattern-matching-replacement statement) ดังนั้น ข้อความสั่ง ชนิดอื่นๆ จึงอาจได้มาจากการประกอบกัน โดย ตัด สมาชิก (elements) ของ ข้อความสั่งพื้นฐาน

ภาษาส่วนใหญ่ มี โครงสร้างวากยสัมพันธ์ต่างๆ มากมาย สำหรับ ชนิด ข้อความสั่ง แต่

ละแบบ ตามข้อสังเกตนี้ ที่เห็นชัดเจนมากที่สุด คือ ภาษา COBOL

ข้อความสั่งแต่ละชนิด ของ ภาษา COBOL มี โครงสร้างเป็นหนึ่งเดียว (unique structure) เกี่ยวข้องกับ คำหลักเฉพาะ, คำรบกวน ตัวสร้างเลือก (alternative constructions) สมาชิกละเว้นได้ เป็นต้น **ข้อดี** ของการใช้ โครงสร้างเชิงวากยสัมพันธ์ หลากหลาย คือ โครงสร้าง แต่ละชนิด ทำให้การแสดงออก ของ การดำเนินการที่เกี่ยวข้อง เป็นวิธีธรรมชาติ

(The advantage of using a variety of syntactic structures, of course, is that each may be made to express in a natural way the operations involved.)

ความแตกต่างที่สำคัญมากกว่า ใน โครงสร้างข้อความสั่ง คือ ระหว่าง ข้อความสั่งเชิงโครงสร้าง หรือ ข้อความสั่งซ้อนกัน (structured or nested statements) กับ ข้อความสั่งอย่างง่าย

ข้อความสั่งอย่างง่าย หมายถึง ข้อความสั่ง ซึ่ง ไม่มีข้อความสั่งอื่นๆ ฝังอยู่ภายใน (A simple statement in one which contains no other embeded statements.) ตัวอย่างเช่น ภาษา APL และภาษา SNOBOL4 ใช้ได้เฉพาะ ข้อความสั่งอย่างง่าย เท่านั้น

ข้อความสั่งเชิงโครงสร้าง หมายถึง ข้อความสั่งหนึ่งอย่าง ซึ่งอาจจะมีข้อความสั่งอื่นๆ ฝังอยู่ภายใน (A structured statement is one which may contain embeded statements.)

ตัวอย่าง ภาษา Pascal

simple statements	ได้แก่	assignment statement
		procedure statement
		goto statement
structured statements	ได้แก่	compound statement
		conditional statement
		repetitive statement
		with statement

โครงสร้างโดยรวมของโปรแกรม และโครงสร้างของโปรแกรมย่อย

(Overall Program - Subpogram Structure)

การจัดองค์กรเชิงวากยสัมพันธ์โดยรวม ของบทนิยาม โปรแกรมหลัก และบทนิยาม ของโปรแกรมย่อย หลากหลาย เช่นเดียวกับ วากยสัมพันธ์ด้านอื่น ๆ ของภาษา

1) **บทนิยามโปรแกรมย่อยแยกออกจากกัน** (Separate subprogram definitions)

ภาษา FORTRAN และภาษา APL แสดงให้เห็น การจัดองค์กรโดยรวมที่ว่า บทนิยามโปรแกรมย่อยแต่ละชุด ถือว่าเป็น หน่วยวากยสัมพันธ์แยกต่างหาก หนึ่งหน่วย (each subprogram definition is treated as a separate syntactic unit.)

ภาษา FORTRAN โปรแกรมย่อยแต่ละชุด ถูกคอมไพล์ แยกต่างหากจากกัน และโปรแกรม ซึ่งคอมไพล์แล้ว จะเชื่อมกัน ณ เวลาบรรจุโปรแกรม (In FORTRAN each subprogram is compiled separately and the compiled programs linked at load time.)

ภาษา APL โปรแกรมต่างๆ ถูกแปล แยกต่างหากจากกัน และเชื่อมด้วยกัน เฉพาะเมื่อโปรแกรมหนึ่ง เรียก อีกโปรแกรมหนึ่ง ระหว่างการกระทำการ (In APL, programs are separately translated and are linked only when one calls another during execution.)

ผล (effect) ของการจัดองค์กรเช่นนี้ จะเห็นชัดเจน โดยเฉพาะ ในภาษา FORTRAN เมื่อโปรแกรมย่อยแต่ละชุด ต้องมีการประกาศครบถ้วน ให้กับสมาชิกข้อมูล ทั้งหมด แม้กระทั่งสมาชิก ใน COMMON blocks ซึ่งจะใช้ร่วมกัน กับ โปรแกรมย่อยอื่นๆ การประกาศเหล่านี้เป็นสิ่งจำเป็น เนื่องจาก ข้อสมมติ (assumption) ของการคอมไพล์ แยกต่างหากกัน

2) **บทนิยามโปรแกรมย่อยซ้อนกัน** (Nested subprogram definitions)

ภาษา Pascal แสดงให้เห็น โครงสร้างโปรแกรมซ้อนกัน ซึ่งบทนิยามโปรแกรมย่อย จะปรากฏ เป็น การประกาศ ภายใน โปรแกรมหลัก และ ตัวโปรแกรมย่อยเอง อาจจะประกอบด้วยบทนิยามโปรแกรมย่อย ซ้อนอยู่ภายใน บทนิยามของมัน ลึกเท่าใดก็ได้

การจัดองค์กรโดยรวม ของโปรแกรมเช่นนี้ สัมพันธ์กับภาษา Pascal ซึ่งเน้น เรื่องข้อความสั่งเชิงโครงสร้าง แต่ความจริงแล้ว สิ่งนี้ สนับสนุน (serve) เป้าหมายต่างๆ

ข้อความสั่งเชิงโครงสร้าง แต่แรกที่แนะนำขึ้นมานั้น เพื่อให้เป็นสัญกรณ์อย่างธรรมชาติสำหรับการแบ่งย่อยลำดับชั้นปกติ ใน โครงสร้างอัลกอริทึม

แต่บทนิยาม โปรแกรมย่อยซ้อนใน จัดให้เพื่อสนับสนุน (serve) การอ้างถึงสิ่งแวดล้อมไม่เฉพาะที่ (nonlocal) สำหรับโปรแกรมย่อย ซึ่งนิยาม ณ เวลาคอมไพล์ และยอมให้มีการตรวจสอบชนิดแบบคงที่ (static type checking) และการแปลโปรแกรม ของ รหัสกระทำการได้อย่างมีประสิทธิภาพ สำหรับ โปรแกรมย่อย ซึ่งมีการอ้างถึงไม่เฉพาะที่

ถ้าไม่มี การซ้อนใน ของบทนิยามโปรแกรมย่อย สิ่งที่จำเป็นคืออาจจะต้องมีการประกาศให้กับ ตัวแปรไม่เฉพาะที่ (nonlocal variables) ภายใน บทนิยามโปรแกรมย่อย แต่ละชุด (เหมือนกับที่กระทำในภาษา FORTRAN) หรือต้องเลื่อน การตรวจสอบชนิดข้อมูลทั้งหมด ของ การอ้าง

ถึง ไม่เฉพาะที่ไปก่อน จนกระทั่งถึงเวลาดำเนินงาน (until run time)

การซ้อนใน (the nesting) ยังสนับสนุน หน้าที่ ของการยอมให้ ชื่อโปรแกรมย่อย มีความสำคัญ น้อยกว่า สโคบส่วนกลาง (global scope)

3) **การอธิบายข้อมูลแยกต่างหากจากข้อความสั่งกระทำการได้**

(Data descriptions separated from executable statements)

การจัดองค์กร ที่ แตกต่างเช่นนี้ พบในภาษา COBOL ในภาษาโปรแกรมส่วนใหญ่ บทนิยามโปรแกรมย่อยแต่ละชุด จะประกอบด้วย การประกาศของข้อมูลเฉพาะที่ (และบางครั้ง ข้อมูลส่วนกลาง) และเซตของข้อความสั่งกระทำการได้ ในโปรแกรม ภาษา COBOL การประกาศข้อมูล และข้อความสั่งกระทำการได้ สำหรับ โปรแกรมย่อยทั้งหมด ถูกแบ่งออกเป็น สองส่วน (divisions) ของโปรแกรมแยกต่างหากจากกัน คือ data division และ procedure division ส่วนที่สาม เรียกว่า environment division ประกอบด้วยการประกาศ ซึ่งเกี่ยวข้องกับ สิ่งแวดล้อมการคำเนินการ ภายนอก (external operating environment)

ส่วนที่เป็น โปรซีเดอร์ ดิวิชัน (procedure division) ของโปรแกรม จัดเป็น หน่วยย่อยๆ (subunits) สมนัยกับ ตัวโปรแกรมย่อย แต่ข้อมูลทั้งหมด เป็นส่วนกลาง ให้กับ ทุกโปรแกรมย่อย **ข้อดี** ของ data division แบบรวมศูนย์ (centralized) ประกอบด้วย การประกาศข้อมูลทั้งหมด คือ มันบังคับ (enforces) ความเป็นอิสระเชิงตรรกะ ของรูปแบบข้อมูล และอัลกอริทึม ใน procedure division การเปลี่ยนแปลงเล็กน้อย ใน โครงสร้างข้อมูล กระทำ ด้วยการดัดแปร ส่วนของ data division โดยไม่ต้อง ดัดแปร ส่วนของ procedure division ข้อดีอีกอย่างหนึ่งคือ เป็นเป็นการรวมการอธิบายข้อมูล ในที่แห่งเดียว ไม่ใช่ กระจัดกระจาย ทั่วไปในโปรแกรมย่อยต่างๆ

4) **บทนิยามโปรแกรมย่อยไม่แยกจากกัน**

(Unseparated subprogram definitions)

การจัดองค์กรโดยรวมแบบที่สี่ (หรือ การไม่จัดองค์กร) แสดงให้เห็นในภาษา SNOBOL4 ซึ่งภาษานี้ ไม่มีความแตกต่างเชิงวากยสัมพันธ์ใดๆ ระหว่าง ข้อความสั่ง ของโปรแกรมหลัก และ ข้อความสั่ง ของโปรแกรมย่อย

การเขียนโปรแกรม ไม่จำเป็นต้องระวังจำนวน โปรแกรมย่อย ซึ่งเป็นเพียงรายการของ ข้อความสั่ง จุดซึ่งเริ่มต้น โปรแกรมย่อย และจุดจบ โปรแกรมย่อย ไม่แตกต่างกัน จริงๆ แล้ว ข้อความสั่งใดๆ อาจจะเป็นส่วนของโปรแกรมหลัก และเป็นส่วนของโปรแกรมย่อย จำนวนเท่าใดก็ได้ ด้วย ณ เวลาเดียวกัน ในแง่ที่ว่า ณ จุดหนึ่งระหว่างการกระทำการของโปรแกรมหลัก มันอาจ

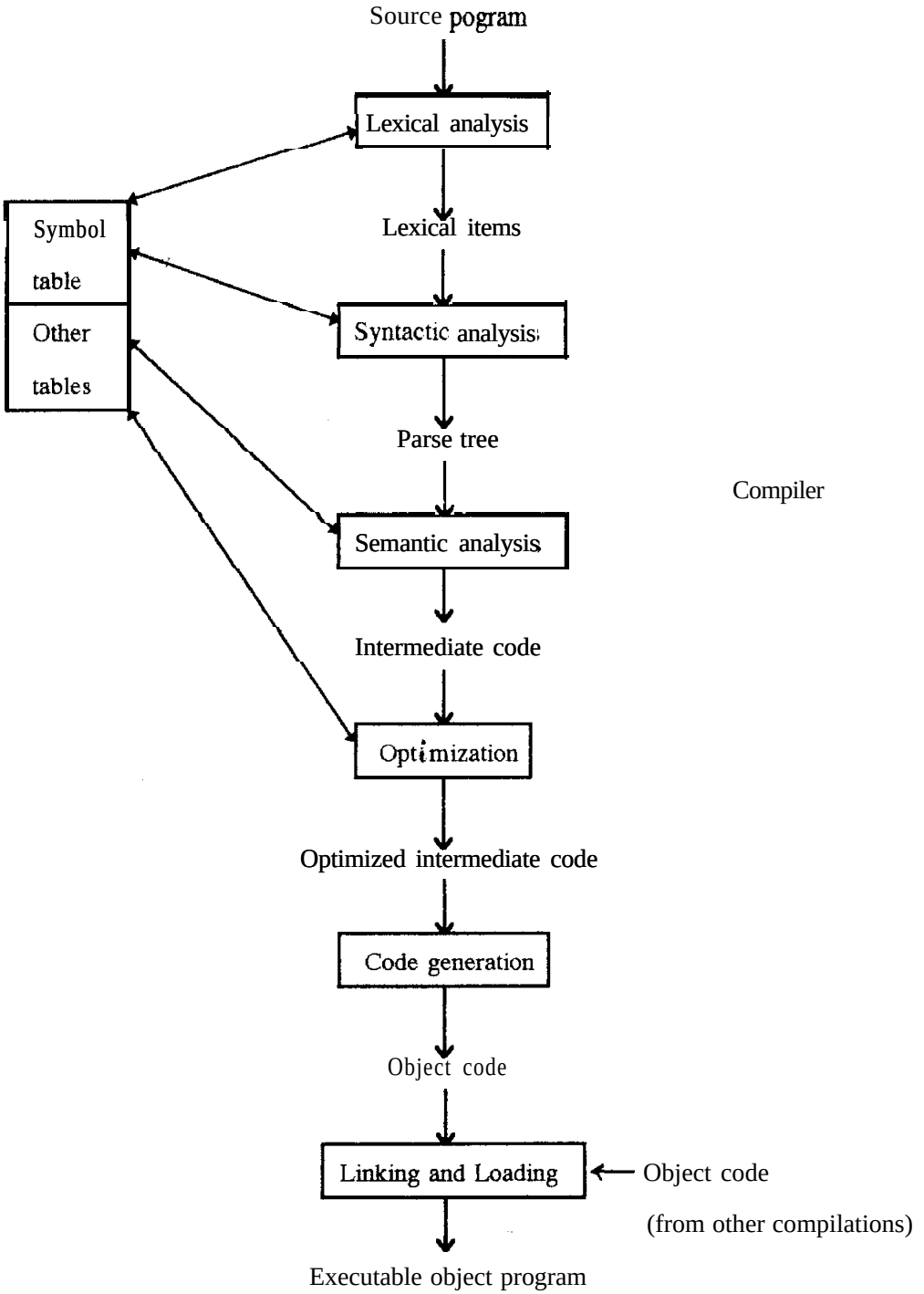
ถูก executed และต่อมา ถูก executed อีกครั้งหนึ่ง เป็นส่วนกระทำการของโปรแกรมย่อย การจัดองค์กรเช่นนี้ ค่อนข้างสับสน มีค่าเฉพาะเมื่อยอมให้ การแปล ณ เวลาดำเนินงาน กับ การกระทำการของ ข้อความสั่งใหม่ และโปรแกรมย่อย ด้วยกลไกอย่างง่าย ที่เกี่ยวข้องกัน

โปรแกรมเมอร์ ภาษา SNOBOL4 ส่วนใหญ่ แนะนำ ให้มีข้อแตกต่างเทียม ระหว่าง ตัวโปรแกรมย่อย โดยการใส่คอมเมนต์ หรือ ตัวดั่นวากยสัมพันธ์อื่นๆ

8.3 ขั้นตอนในการแปลภาษา (Stage in Translations)

กรรมวิธีของการแปลโปรแกรม จากวากยสัมพันธ์เดิม ของมัน ไปยัง รูปแบบกระทำการได้ หมายถึง ศูนย์กลาง การทำให้เกิดผลของ ภาษาโปรแกรม ทุกภาษา การแปลภาษาอาจง่ายมาก เช่น ในกรณีของโปรแกรมภาษา APL หรือ LISP แต่ภาษาส่วนมาก เป็นกระบวนที่ซับซ้อน และต้องมีส่วนร่วมสำคัญ ของความพยายาม ในการทำภาษาให้เกิดผล ภาษาส่วนใหญ่ สามารถทำให้เกิดผลด้วยการแปลเล็กน้อยเท่านั้น ถ้าเราตั้งใจเขียน ตัวแปลคำสั่งซอฟท์แวร์ (software interpreter หรือ software simulated virtual computer) และถ้าเรายอมรับ ความเร็วการกระทำการ ที่ช้าอย่างไรก็ตาม ในกรณีส่วนใหญ่ การกระทำการที่มีประสิทธิภาพ เป็นเป้าหมายที่ต้องการ ซึ่งความพยายามหลัก ถูกกระทำขึ้น เพื่อแปล โปรแกรม ให้เป็น โครงสร้างกระทำการได้ อย่างมีประสิทธิภาพ โดยเฉพาะ hardware-interpretable machine code กรรมวิธีของการแปลภาษายิ่งก้าวหน้า ยิ่งมีความซับซ้อนมากขึ้น เป็น รูปแบบโปรแกรมกระทำการได้ มี โครงสร้าง ห่างไกลจาก โปรแกรมเดิม ยิ่งขึ้น (The translation process becomes progressively more complex as the executable program form becomes further removed in structure from the original program.) ตัวอย่างเช่น optimizing compiler สำหรับภาษาซับซ้อน เช่น PL/1 อาจเปลี่ยน โครงสร้างโปรแกรม มากมาย เพื่อให้ได้ การกระทำการที่มีประสิทธิภาพมากกว่า คอมไพเลอร์เช่นนี้ มีอยู่ท่ามกลาง โปรแกรมซับซ้อนส่วนใหญ่

ในเชิงตรรกะ เราแบ่ง การแปลภาษา (translation) ออกเป็น สองส่วนที่สำคัญ คือ **การวิเคราะห์** (analysis) โปรแกรมต้นฉบับ ซึ่งเป็นอินพุท และ**การสังเคราะห์** (synthesis) โปรแกรมจุดหมาย ซึ่งกระทำการได้ ภายในของแต่ละส่วนนี้ ยังแบ่งออกเป็นส่วนย่อยอีก ซึ่งจะได้กล่าวถึงต่อไป ในตัวแปลภาษาส่วนใหญ่ ขั้นตอนเชิงตรรกะเหล่านี้ จะแยกกันไม่ชัดเจน แต่เป็นการผสมกัน (mixed) เพื่อให้การวิเคราะห์ และการสังเคราะห์ สลับกัน บ่อยครั้ง อยู่บนหลักการ การทำทีละข้อความสั่ง (often on a statement-by-statement basis.) รูป 8-2 แสดงให้เห็นโครงสร้างของคอมไพเลอร์



รูป 9-2 โครงสร้างของตัวแปล โปรแกรม

(Structure of a compiler)

การวิเคราะห์โปรแกรมต้นฉบับ (Analysis of the Source Program)

สำหรับตัวแปลภาษา, โปรแกรมต้นฉบับ เริ่มต้นจากเป็น สายอักขระ หนึ่งชุด ความยาว ประกอบด้วย อักขระ นับจำนวน พันๆ ตัว หรือ จำนวนหมื่นๆ ตัว

(To a translator, the source program appears initially as one long undifferentiated character string composed of thousands or tens of thousands of characters.)

สำหรับโปรแกรมเมอร์มอง โปรแกรม ที่ โครงสร้างของมัน ซึ่งแบ่งเป็น โปรแกรมย่อย ข้อความสั่ง การประกาศ และอื่นๆ แต่สำหรับตัวแปลภาษา ไม่มีสิ่งนี้ ปรากฏให้เห็น การวิเคราะห์โครงสร้างของโปรแกรม ต้องเป็นงานของ การสร้างทีละตัวอักขระ ระหว่างการแปลภาษา

การวิเคราะห์ศัพท์ (Lexical analysis)

ระยะที่สำคัญที่สุด ของการแปลภาษา คือ โปรแกรมอินพุท (input program) ถูกแบ่งย่อย ให้เป็นส่วนประกอบพื้นฐานของมัน ได้แก่ ไอเดนติไฟเออร์ ตัวดั่น สัญลักษณ์ตัวคำเนินการ เลข คำหลัก คำรบกวน อักขระว่าง คอมเมนต์ (comments) เป็นต้น ระยะนี้เรียกว่า **การวิเคราะห์ศัพท์** และหน่วยโปรแกรมหลัก ซึ่งเป็นผลลัพธ์ จากการวิเคราะห์ศัพท์ เรียกว่า **ชิ้นศัพท์** (หรือ โทเค็น)

(The basic program units which result from lexical analysis are termed **lexical items** (or **tokens**)

โดยปกติ ตัววิเคราะห์ศัพท์ หรือตัวกราดตรวจ (lexical anlyzer (or scanner)) หมายถึง อินพุท รูทีน (input routine) สำหรับตัวแปลภาษา ซึ่งอ่านบรรทัดของอินพุท โปรแกรมอย่างสืบเนื่อง แบ่งย่อยบรรทัดเหล่านั้น ให้เป็นชิ้นศัพท์แยกเฉพาะแต่ละตัว และ ป้อนชิ้นศัพท์เหล่านี้ ให้กับขั้นตอนต่อไป ของตัวแปลภาษา ซึ่งจะนำไปใช้ในการวิเคราะห์ ระดับที่สูงขึ้น

ตัววิเคราะห์ศัพท์ ต้อง จำแนก (identify) ชนิด ของชิ้นศัพท์แต่ละตัว (เลข ไอเดนติไฟ-เออร์ ตัวดั่น ตัวคำเนินการ เป็นต้น) และติดเครื่องหมายชนิดไว้ นอกจากนี้แล้ว การแปลงผัน (conversion) ให้เป็น การแทนที่ภายใน บ่อยครั้ง ถูกกระทำให้กับ ชิ้นข้อมูล เช่น เลข (แปลงผันเป็นรูปแบบ internal binary fixed หรือรูปแบบ floating-point) และไอเดนติไฟเออร์ (เก็บในตารางสัญลักษณ์ และเลขที่อยู่ของข้อมูล ใน ตารางสัญลักษณ์ แทน สายอักขระ)

ขณะที่ การวิเคราะห์ศัพท์ เป็นแนวคิดอย่างง่าย ขั้นตอนนี้ ของการแปลภาษา บ่อยครั้ง

ต้องการ เวลาของการแปล ซึ่งมากกว่า ขั้นตอนอื่นๆ ความจริงคือ เกี่ยวกับความจำเป็น ที่ใช้กราดตรวจ (scan) และวิเคราะห์ (analyze) โปรแกรมต้นฉบับ ทีละตัวอักขระ และเป็นจริงที่ว่าในทางปฏิบัติ บางครั้ง เป็นการยากที่จะหาขอบเขต (boundaries) ระหว่างชิ้นศัพท์ ถ้าไม่มีอัลกอริทึม ซึ่งขึ้นอยู่กับเนื้อหาที่ซับซ้อน (complex context-dependent algorithms)

ตัวอย่างเช่น ข้อความสั่ง ภาษา FORTRAN สองชุด

DO 10 I = 1, 5

และ

DO 10 I = 1.5

มีโครงสร้างเชิงศัพท์ แตกต่างกัน อย่างสิ้นเชิง นั่นคือ บรรทัดแรก เป็นข้อความสั่ง DO ส่วนบรรทัดที่สอง เป็น ข้อความสั่งกำหนดค่า แต่ความจริง จะไม่สามารถค้นพบได้ ถ้าไม่มีการวิเคราะห์อย่างถูกต้อง

การวิเคราะห์วากยสัมพันธ์ (Syntactic analysis หรือ parsing)

ขั้นตอนที่สองในการแปลภาษาคือ การวิเคราะห์วากยสัมพันธ์ หรือ การวิเคราะห์กระจาย (syntactic analysis or parsing) ในที่นี้ โครงสร้างโปรแกรม ซึ่งมี ขนาดใหญ่กว่า ชิ้นศัพท์ จะถูกจำแนก (are identified) ได้แก่ ข้อความสั่ง การประกาศ นิพจน์ โดยใช้ ชิ้นศัพท์ ซึ่งได้จากตัววิเคราะห์ศัพท์

การวิเคราะห์วากยสัมพันธ์ ปกติ ทำสลับกับ การวิเคราะห์ความหมาย **ขั้นแรก** ตัววิเคราะห์วากยสัมพันธ์ จำแนก (identifies) ลำดับของ ชิ้นศัพท์ ซึ่งประกอบขึ้นเป็น หน่วยวากยสัมพันธ์ (syntactic unit) เช่น นิพจน์ ข้อความสั่ง การเรียกโปรแกรมย่อย หรือ การประกาศ จากนั้น ตัววิเคราะห์ความหมาย (semantic analyzer) ถูกเรียกให้ไป ประมวลผล หน่วยวากยสัมพันธ์นี้ ปกติ ตัววิเคราะห์วากยสัมพันธ์ และตัววิเคราะห์ความหมาย สื่อสารถึงกัน โดยใช้ **กองซ้อน** (stack) ตัววิเคราะห์วากยสัมพันธ์ ใส่สมาชิกต่างๆ ของ หน่วยวากยสัมพันธ์ ที่พบในกองซ้อน และสมาชิกเหล่านี้ ถูกค้นคืน (are retrieved) และประมวลผลด้วยตัววิเคราะห์ความหมาย มีการทำวิจัยมากมาย ซึ่ง มีศูนย์กลาง บนการค้นหา เทคนิคการวิเคราะห์วากยสัมพันธ์ ที่มีประสิทธิภาพ โดยเฉพาะ เทคนิคซึ่งมีหลักการบน การใช้ไวยากรณ์แบบทางการ (formal grammars)

การวิเคราะห์ความหมาย (Semantic analysis)

การวิเคราะห์ความหมาย เป็น ระยะกลาง ของการแปลภาษา ในที่นี้ โครงสร้างวากยสัมพันธ์ รู้จำได้ (recognized) ด้วย **ตัววิเคราะห์วากยสัมพันธ์** จะถูกประมวลผล และ โครงสร้าง

ของ รหัสจุดหมายซึ่งกระทำการได้ (executable object code)) เริ่มชัดขึ้น

ดังนั้น การวิเคราะห์ความหมาย จึงเป็นสะพานระหว่าง ขั้นการวิเคราะห์ และ ขั้นการสังเคราะห์ ของการแปลภาษา หน้าที่ปลีกย่อยที่สำคัญอื่นๆ อีกจำนวนหนึ่ง เกิดขึ้น ในขั้นตอนนี้ด้วย ได้แก่ การบำรุงรักษา ตารางสัญลักษณ์ (symbol-table maintenance) การตรวจพบข้อผิดพลาดส่วนใหญ่ (most error detection) การขยายของแมโคร (the expansion of macros) และการกระทำการของ ข้อความสั่ง เวลาแปล (the execution of compile-time statements) ตัววิเคราะห์ความหมาย อาจจะให้รหัสจุดหมาย กระทำการได้ ในการแปลเบื้องต้น แต่โดยปกติแล้วเอ้าพุทจาก ขั้นตอนนี้ เป็นรูปแบบภายในบางอย่าง ของ โปรแกรมกระทำได้สุดท้าย มากกว่า ซึ่งจากนั้น จะถูกคลุมแต่ง (manipulated) ด้วยขั้น optimization ของ ตัวแปลภาษา ก่อนที่จะก่อกำเนิดรหัสกระทำการ ได้จริง

ตัววิเคราะห์ความหมาย ปกติ จะถูกแบ่งออกเป็น เซตของตัววิเคราะห์ความหมายขนาดเล็กๆ โดยที่แต่ละตัว จัดกระทำ (handle) ตัวสร้างโปรแกรมเฉพาะ หนึ่งชนิดเท่านั้น ตัวอย่างเช่น การประกาศ ของแถวลำดับ จัดกระทำด้วย ตัววิเคราะห์หนึ่งตัว นิพจน์คำนวณ ถูกจัดกระทำโดยตัววิเคราะห์อีกตัวหนึ่ง และ ข้อความสั่ง goto ถูกจัดกระทำ โดยตัววิเคราะห์อีกตัวหนึ่ง ตัววิเคราะห์ความหมายที่เหมาะสม ถูกเรียกโดย ตัววิเคราะห์วากยสัมพันธ์ เมื่อใดก็ตามที่มันรู้จำหน่วยวากยสัมพันธ์ ซึ่งจะถูกประมวลผล

ตัววิเคราะห์ความหมายต่างๆ โต้ตอบระหว่างกัน ผ่านสารสนเทศ ซึ่งเก็บใน โครงสร้างข้อมูลต่างๆ โดยเฉพาะใน ตารางสัญลักษณ์ ส่วนกลาง ตัวอย่างเช่น ตัววิเคราะห์ความหมาย ซึ่งประมวลผล ชนิด การประกาศ สำหรับตัวแปรอย่างง่าย บ่อยครั้ง อาจทำงานเล็กน้อย เพียงใส่ชนิดการประกาศ (declared types) เข้าไปใน ตารางสัญลักษณ์ จากนั้นตัววิเคราะห์ความหมายถัดไป ซึ่ง ประมวลผล นิพจน์คำนวณ อาจจะใช้ ชนิดการประกาศ เพื่อ ก่อกำเนิด การดำเนินการคำนวณเฉพาะชนิดที่เหมาะสม สำหรับรหัสจุดหมาย หน้าที่แน่นอน ของตัววิเคราะห์ความหมายต่างๆ ผันแปรอย่างมาก ขึ้นอยู่กับภาษาและการจัดองค์กรเชิงตรรกะของตัวแปลภาษา

หน้าที่บางอย่าง ของตัววิเคราะห์ความหมาย ซึ่งสำคัญที่สุด อธิบายได้ดังนี้

1. **การบำรุงรักษาตารางสัญลักษณ์** (Symbol-table maintenance)

ตารางสัญลักษณ์ เป็น โครงสร้างข้อมูลหลักอย่างหนึ่ง ในตัวแปลทุกภาษา ปกติ ตารางสัญลักษณ์ จะประกอบด้วย **หน่วยข้อมูล** (entry) สำหรับไอเดนติไฟเออร์ แต่ละตัวที่แตกต่างกันที่พบ ในโปรแกรมต้นฉบับ เริ่มจาก ตัววิเคราะห์ศัพท์ ทำให้มีข้อมูล ขณะที่มันกราดตรวจ อินพุท

โปรแกรม หลังจากนั้น เป็นความรับผิดชอบ ของตัววิเคราะห์ความหมาย โดยทั่วไป หน่วย ข้อมูล ในตารางสัญลักษณ์ ๆ ไม่ใช่มี ไอเดนติไฟเออร์ อย่างเดียว แต่ประกอบด้วย ข้อมูลเพิ่มเติม เกี่ยวกับลักษณะประจำ (attributes) ของไอเดนติไฟเออร์ ตัวนั้นด้วย เช่น ชนิดของมัน (ตัวแปร อย่างง่าย ชื่อแถวลำดับ ชื่อโปรแกรมย่อย พารามิเตอร์ทางการ เป็นต้น) ชนิดของค่า (integer real เป็นต้น) สิ่งแวดล้อมของการอ้างถึง (referencing environment) และสารสนเทศอื่นๆ ซึ่ง ได้มาจาก โปรแกรมอินพุท ผ่านทางการประกาศ และการใช้

ตัววิเคราะห์ความหมาย ใส่ (enter) สารสนเทศเหล่านี้ เข้าไปในตารางสัญลักษณ์ ขณะที่ มัน ประมวลผล (process) การประกาศ หัวเรื่องโปรแกรมย่อย และข้อความสั่งโปรแกรม

ตัววิเคราะห์ความหมายตัวอื่นๆ เช่น optimizer ใน ขั้นสังเคราะห์ของตัวแปลภาษา ใช้ สารสนเทศหล่านี้ เพื่อสร้าง รหัสกระทำการได้ ที่มีประสิทธิภาพ

ตารางสัญลักษณ์ ใน ตัวแปลภาษา สำหรับภาษาแปลความ ปกติ จะถูกตัดทิ้งไป เมื่อสิ้น สุดการแปล (The symbol table in translators for compiled language is usually discarded at the end of translation.) อย่างไรก็ตาม ตารางสัญลักษณ์ อาจจะยังเก็บไว้ ระหว่าง การกระทำการ ตัวอย่างเช่น ในภาษาโปรแกรม ซึ่งอนุญาต ให้ ไอเดนติไฟเออร์ตัวใหม่ ถูกสร้างขึ้นได้ ณ เวลา ดำเนินงาน

การทำให้เกิดผลของภาษา APL, ANOBOL4 และ LISP ทั้งหมดนี้ ใช้ตารางสัญลักษณ์ ซึ่งสร้างขึ้น ระหว่าง การแปลภาษาให้เป็น โครงสร้างข้อมูลส่วนกลางซึ่งระบบ นิยาม ณ เวลา ดำเนินงาน (a central run-time system-defined data structure)

2. **การใส่สารสนเทศโดยนัย** (Insertion of implicit information) บ่อยครั้งที่ในโปรแกรม ต้นฉบับ, สารสนเทศ ซึ่งเป็นข้อมูล ชนิดโดยนัย ต้องถูกทำให้เป็นข้อมูลชัดแจ้ง ใน โปรแกรม ภาษาจุดหมายระดับต่ำกว่า สารสนเทศ โดยนัยนี้ ส่วนใหญ่เป็นไป ภายใต้ หัวเรื่องทั่วไป ของ **ข้อตกลงโดยปริยาย** (default conventions) หมายถึง การแปลความ จะถูกจัดให้ เมื่อโปรแกรมเมอร์ ไม่ได้ให้ข้อกำหนดชัดแจ้ง ตัวอย่างเช่น ตัวแปร PL/1 ซึ่งนำมาใช้ แต่ไม่มีการประกาศ จะถูกจัด โดยอัตโนมัติ ด้วย รายการของคุณสมบัติ โดยปริยาย ดังนี้ ถ้าชื่อตัวแปร ขึ้นต้นด้วยตัวอักษรตัวใด ตัวหนึ่งใน I-N จะเป็นตัวแปรชนิด FIXED BINARY และเป็น AUTOMATIC มีสโคป เท่ากับ บล็อก ซึ่งมันถูกประกาศเช่นนี้ เป็นต้น ข้อกำหนดโดยปริยาย (default specifications) เหล่านี้ ทั้งหมด จะอยู่ต่ำกว่า การประกาศของโปรแกรมเมอร์โดยชัดแจ้ง งานของตัววิเคราะห์ความหมาย รวมทั้ง การใส่ ข้อกำหนดโดยปริยาย เหล่านี้ ไว้ใน ตารางสัญลักษณ์ หรือ รหัสจุดหมาย

3. **การตรวจหาข้อผิดพลาด** (Error detection)

ตัววิเคราะห์วากยสัมพันธ์ และตัววิเคราะห์ความหมาย ต้องถูกเตรียมไว้เพื่อจัดกระทำ (to handle) โปรแกรม ไม่ถูกต้อง เช่นเดียวกับ จัดกระทำกับ โปรแกรมถูกต้อง ณ จุดใดๆ ก็ตาม ตัววิเคราะห์ศัพท์ อาจส่ง ชิ้นศัพท์ ซึ่ง ไม่เหมาะสมกับ เนื้อหาล้อมรอบ ไปยังตัววิเคราะห์วากยสัมพันธ์ ตัวอย่างเช่น ตัวคั่นข้อความสั่ง อยู่ตรงกลางของนิพจน์, การประกาศอยู่ตรงกลาง ของลำดับข้อความสั่ง หรือ สัญลักษณ์ตัวดำเนินการ อยู่ที่ตำแหน่ง ซึ่งคาดว่าเป็นตำแหน่งของ ไอเดนติไฟเออร์

ข้อผิดพลาดอาจเป็นเรื่องปลีกย่อยต่างๆ เช่น เป็นตัวแปร real ซึ่ง ตำแหน่งนั้น ต้องเป็นตัวแปร integer, การอ้างถึงตัวแปรดรรชนีล่าง ที่มี ดรรชนีล่าง สามตัว ในขณะที่ การประกาศแถวลำดับ มีเพียงสองมิติ, หรือลาเบลข้อความสั่ง ในข้อความสั่ง goto อ้างถึงชื่อข้อความสั่ง ภายในข้อความสั่งวนซ้ำ (iteration statement) ซึ่ง ไม่อนุญาต ให้กระโดดข้าม

แต่ละขั้นตอน ในการแปล ข้อผิดพลาดมากมายเช่นนี้ อาจเกิดขึ้นได้ ตัววิเคราะห์ความหมาย ไม่เพียงแต่รู้จำ ข้อผิดพลาดที่เกิดขึ้น เท่านั้น แต่ยังต้อง ให้ ข้อความระบุความผิดพลาด (error message) ที่เหมาะสม ด้วย แต่ทั้งหมด กรณีสำคัญที่สุด คือ หาวิธีเหมาะสม เพื่อให้การวิเคราะห์วากยสัมพันธ์ ของ ส่วนที่เหลือของโปรแกรมดำเนินต่อไป การจัดหา การตรวจหาความผิดพลาด และการจัดกระทำ ในระยะวิเคราะห์วากยสัมพันธ์ และวิเคราะห์ความหมาย ต้องการความพยายามมากกว่า การวิเคราะห์พื้นฐานของมันเอง

4. **การประมวลผลแมโคร และการดำเนินการเวลาแปล**

(Macro processing and compile-time operations)

ไม่ใช่ภาษาโปรแกรม ทุกภาษา ที่มีคุณสมบัติแมโคร หรือการจัดหา สำหรับ การดำเนินการ เวลาแปล เมื่อมีสิ่งเหล่านี้ การประมวลผล ปกติจัดกระทำระหว่างวิเคราะห์ความหมาย

แมโคร (macro) ในรูปแบบง่ายที่สุดของมัน หมายถึง ชิ้นส่วนของเนื้อหาโปรแกรม ซึ่งนิยามแยกต่างหากจากกัน และจะถูกนำไปใส่ในโปรแกรม ระหว่างการแปล เมื่อใดก็ตาม ที่มีการเรียกแมโคร (macro call) ใน โปรแกรมต้นฉบับ ดังนั้น แมโคร เหมือนกับ โปรแกรมย่อยมาก **ยกเว้น** มีการแปลแยกต่างหาก และถูกเรียก ณ เวลาดำเนินงาน ตัว body ของมัน ถูกแทนที่ ทุกครั้งเมื่อมีการเรียก ระหว่าง การแปลโปรแกรม แมโคร อาจจะเป็นเพียงสายอักขระอย่างง่าย (simple string) ซึ่งจะถูกนำมาแทนที่ ตัวอย่างเช่น การนำมาแทนที่ ของ 3.1416 ให้ PI เมื่อใดก็ตาม เมื่อมีการอ้างถึง PI สิ่งซึ่งทำให้แมโครดูเหมือน โปรแกรมย่อยมากยิ่งขึ้นคือ พารามิเตอร์

ต้อง ถูกประมวลผล ก่อนการนำไปแทนที่ ของ การเรียกแมโคร

เมื่อยอมให้มีการใช้แมโคร ตัววิเคราะห์ความหมาย ต้องจำแนก macro calls ภายใน โปรแกรมต้นฉบับ และจัด (set up) การแทนที่ อย่างเหมาะสม ของ ตัวแมโคร ให้กับ การเรียก บ่อยครั้ง งานนี้ เกี่ยวข้องกับ การขัดจังหวะ (interrupting) ตัววิเคราะห์ศัพท์ และ ตัววิเคราะห์วากยสัมพันธ์ และจัดมันให้ทำงานการวิเคราะห์ การแทนที่สายอักขระ ตัวแมโคร ก่อนการประมวลผล ด้วย ส่วนที่เหลือ ของ สายต้นฉบับ (source string) อีกทางเลือกหนึ่งคือ ตัวแมโครอาจจะแปลเรียบร้อยแล้วบางส่วน ดังนั้น ตัววิเคราะห์ความหมาย สามารถ ประมวลผล โดยตรง การใส่ รหัสจุดหมายที่เหมาะสมและการทำ ข้อมูลตารางที่เหมาะสม ก่อนการทำงานต่อไป ด้วย การวิเคราะห์ ของ โปรแกรมต้นฉบับ

การดำเนินการ ณ เวลาแปล หมายถึง การดำเนินการ ซึ่งจะถูกกระทำ ระหว่างการแปลภาษา เพื่อควบคุมการแปลของ โปรแกรมต้นฉบับ

(**A compile-time operation** is an operation to be performed during translation to control the translation of the source program.)

ตัวอย่าง ภาษา PL/1 จัดให้มีการดำเนินการเช่นนี้ จำนวนหนึ่ง การกำหนดค่า ณ เวลาคอมไพล์ จัดหา ความสามารถแมโครอย่างง่าย โดยยอมให้ สายอักขระใดๆ ถูกนำไปแทนที่ ให้กับ การเกิดแต่ละครั้งของ ไอเดนติไฟเออร์ การดำเนินการเวลาคอมไพล์ ซึ่งซับซ้อนมากยิ่งขึ้น ยอมให้ ส่วนต่างๆ ของโปรแกรม ถูกแปลเฉพาะ เมื่อเงื่อนไขเฉพาะซึ่งเป็นที่ต้องการ หรือ ยอมให้กลุ่มต่างๆ ของข้อความสั่ง ถูกแปลซ้ำๆ กัน เช่น ผ่าน ลูป (loop) ถูกกระทำการโดยตัวแปลภาษา, การประมวลผลใหม่ กลุ่มของข้อความสั่ง ด้วยการวนซ้ำ (iteration) แต่ละครั้ง อีกครั้งหนึ่งคือตัววิเคราะห์ความหมาย ซึ่งต้องจำแนก (identify) และกระทำการ การดำเนินการเวลาคอมไพล์ เหล่านี้ ก่อนกระบวนการของการแปล

การสังเคราะห์ โปรแกรมภาษาจุดหมาย (Synthesis of the Object Program)

ขั้นตอนสุดท้าย ของการแปลภาษา เกี่ยวกับ การสร้าง โปรแกรมกระทำการได้ จากเอ้าพุท ซึ่งได้โดยตัววิเคราะห์ความหมาย ระยะนี้เกี่ยวกับการก่อกำเนิดรหัส ที่จำเป็น และรวมทั้ง **ความเหมาะที่สุด** (optimization) **ของ** โปรแกรมก่อกำเนิด (generated program) ถ้าโปรแกรมย่อย ถูกแปลแยกต่างหากกัน หรือ ถ้าโปรแกรมย่อยเฉพาะงานจากคลัง (library subprograms) ถูกนำมาใช้ ขั้นตอนการโยง และการบรรจุ สุดท้าย มีความจำเป็น เพื่อให้โปรแกรมบริบูรณ์ พร้อมสำหรับการกระทำการ

ความเหมาะสมที่สุด (Optimization)

ตัววิเคราะห์ความหมาย ปกติให้ เอ้าพุท เป็น โปรแกรม ถูกแปลแล้วกระทำการได้ แทน **รหัสกลาง**บางอย่าง (some intermediate code) การแทนที่ภายใน เช่น สายโพลิช (Polish string) ของตัวดำเนินการ และตัวถูกดำเนินการ หรือตารางของลำดับ ตัวดำเนินการ-ตัวถูกดำเนินการ จากการแทนที่ภายในนี้ **ตัวก่อกำเนิดรหัส** (code generator) จะก่อกำเนิด (generate) รหัสจุดหมาย เอ้าพุท รูปแบบ อย่างถูกต้อง อย่างไรก็ตาม ก่อนการก่อกำเนิดรหัส ปกติ มีความเหมาะที่สุดบางอย่าง ของโปรแกรม ใน การแทนที่ภายใน โดยตัววิเคราะห์ความหมาย ก่อกำเนิด (generate) รูปแบบ โปรแกรมภายใน ทีละน้อย เป็น แต่ละเซกเมนต์ ของ โปแกรมอินพุท ซึ่งจะถูกวิเคราะห์ งานนี้กระทำได้ง่ายที่สุด ถ้าตัววิเคราะห์ความหมาย ไม่ต้องกังวลมากนัก เกี่ยวกับรหัสล้อมรอบ ซึ่ง จะก่อกำเนิดทันที ก่อนนั้น ในการทำเอ้าพุท ทีละน้อยนี้ รหัสที่แย่มาก อาจมีขึ้น (extremely poor code may be produced) เช่น เรจิสเตอร์ตัวหนึ่ง อาจถูกเก็บไว้ตอนสิ้นสุดของเซกเมนต์ก่อกำเนิดหนึ่งชุด และทันใดนั้นมีการบรรจุใหม่ (reloaded) จาก ตำแหน่งเดียวกัน ที่ตอนต้น ของ เซกเมนต์ถัดไป บ่อยครั้งคือ ยอมให้ ตัววิเคราะห์ความหมาย ก่อกำเนิดของ ลำดับรหัสแย่ จากนั้น ระหว่าง การเหมาะที่สุด (optimization) แทนลำดับเหล่านี้ ด้วย สิ่งที่ดีกว่า โดยหลีกเลี่ยง สิ่งที่ไม่มีประสิทธิภาพ ซึ่งเห็นชัดเจน

คอมไพเลอร์ จำนวนมาก ไปไกลมากแล้วจาก การเหมาะที่สุด อย่างง่ายนี้ และวิเคราะห์โปรแกรม สำหรับ การปรับให้ดีขึ้นอื่นๆ ซึ่งสามารถทำได้ ตัวอย่างเช่น การคำนวณ**นิพจน์ย่อยร่วม** (common subexpressions) เพียงครั้งเดียวเท่านั้น การตัดทิ้ง การดำเนินงานคงที่ จาก loops การเหมาะที่สุด ในการใช้เรจิสเตอร์ การเหมาะที่สุด ของการคำนวณสูตรการเข้าถึงแถวลำดับ การวิจัยส่วนใหญ่ กระทำกับ การเหมาะที่สุดของโปรแกรม และเทคนิคใหม่ๆ จำนวนมากซึ่งเป็นที่รู้จักกัน

การก่อกำเนิดรหัส (Code generation)

หลังจาก โปรแกรมที่ถูกแปลแล้ว ในการแทนที่ภายใน มีการทำให้เหมาะที่สุด มันต้องถูกประกอบให้เป็น ข้อความสั่งภาษาแอสเซมบลี รหัสเครื่อง หรือ โปรแกรมภาษาจุดหมายอื่นๆ ซึ่งเป็นเอ้าพุท ของการแปล กรรมวิธีนี้ เกี่ยวกับการ จัดรูปแบบเอ้าพุทอย่างถูกต้อง จากสาร-สนเทศ ที่อยู่ใน การแทนที่โปรแกรมภายใน รหัสเอ้าพุท อาจถูกกระทำการได้โดยตรง หรืออาจเป็น ขั้นตอนของการแปลอื่นๆ ที่ตามมา ได้แก่ assembly หรือการเชื่อมโยง และการบรรจุ

การโยงและการบรรจุ (Linking and loading)

ขั้นตอนสุดท้ายของการแปล ซึ่งอาจจะมีหรือไม่มีก็ได้ ชิ้นส่วนต่างๆ ของรหัสซึ่งเป็นผลลัพธ์จากการแปลแยกต่างหากจากกัน ของ โปรแกรมย่อย ถูกรวมเข้าด้วยกัน เป็น โปรแกรมกระทำการได้ **ขั้นสุดท้าย** (final executable program) เอ้าพุทของระยะการแปลก่อนหน้านั้น โดยปกติประกอบด้วย โปรแกรมกระทำการได้ ซึ่งเกือบจะเป็น รูปแบบสุดท้าย ยกเว้น ตำแหน่งที่โปรแกรม อ้างถึง **ข้อมูลภายนอก** หรือ โปรแกรมย่อยอื่นๆ ตำแหน่งที่ไม่บริบูรณ์เหล่านี้ ในรหัสซึ่งกำหนดไว้ให้ ผูกติดกับ **ตารางตัวบรรจุ** (loader tables) ซึ่งให้โดย ตัวแปลภาษา

โปรแกรมบรรจุการโยง หรือ เอดิเตอร์เชื่อมโยง (linker loader or link editor) บรรจุเซกเมนต์ต่างๆ ของรหัสซึ่งแปลแล้ว เข้าไปใน หน่วยความจำ และจากนั้น ใช้ ตารางตัวบรรจุผูกโยง (attached loader tables) เพื่อเชื่อมโยง สิ่งทั้งหมดเข้าด้วยกันอย่างถูกต้อง โดยใส่ในแอดเดรสข้อมูล และแอดเดรสโปรแกรมย่อย ใน รหัส ขณะที่จำเป็น ผลลัพธ์คือ โปรแกรมกระทำการได้สุดท้าย ซึ่งพร้อมสำหรับดำเนินงาน

8.4 บทนิยามทางการของวากยสัมพันธ์

(Formal definition of syntax)

ในการศึกษา ภาษาโปรแกรม แบบทางการ หัวข้อวากยสัมพันธ์และการวิเคราะห์วากย-สัมพันธ์ ได้รับการสนใจมากที่สุด

เริ่มต้น เป้าหมายของงานนี้ คือ การจัดหา บทนิยาม ถูกต้อง ของ วากยสัมพันธ์ภาษาโปรแกรม สำหรับ ผู้ใช้ (users) และการทำให้เกิดผลของภาษา อย่างไรก็ตาม มันทำให้ปรากฏขึ้นมาอย่างเร็วที่ว่า บทนิยามวากยสัมพันธ์เหล่านี้ ถูกนำมาใช้โดยตรง เป็น มูลฐาน สำหรับ การวิเคราะห์วากยสัมพันธ์ ในตัวแปลภาษา (translators) งานต่อมา มีการพัฒนาและวิเคราะห์เทคนิคหลากหลาย สำหรับ การวิเคราะห์วากยสัมพันธ์ บน หลักการ ของ บทนิยามวายสัมพันธ์ แบบทางการ

บทนิยามทางการของวากยสัมพันธ์ ของ ภาษาโปรแกรม ปกติเรียกว่า **ไวยากรณ์** (grammar) ซึ่งเหมือนกับการใช้ถ้อยคำ สำหรับภาษาธรรมชาติ

ไวยากรณ์ ประกอบด้วย เซตของบทนิยาม (เรียกว่า กฎ หรือ การผลิต) ซึ่งกำหนด ลำดับของ ตัวอักษร (หรือ ชิ้นศัพท์) ซึ่งอนุญาต ให้ประกอบเป็น โปรแกรม ในภาษาซึ่งกำลังถูกนิยาม

(A grammar consists of a set of definitions (termed **rules** or **production**) which specify the sequences of characters (or lexical items) that form allowable programs in the language being defined.)

ไวยากรณ์แบบทางการ เป็นเพียงการกำหนดไวยากรณ์ โดยใช้ เครื่องหมายการนิยาม อย่างเข้มงวด

(A **formal grammar** is just grammar specify using a strictly defined notation.)

ไวยากรณ์แบบทางการ ชนิดที่ เป็นที่รู้จักกัน มากที่สุด คือ **ไวยากรณ์ BNF** (หรือ ไวยากรณ์ไม่พึ่งบริบท)

(The best-known type of formal grammar is the **BNF grammar** (or **context-free grammar**))

ซึ่งพบว่า นำไปประยุกต์ใช้กว้างขวาง ทั้ง ในบทนิยามภาษา และในการวิจัยภาษาธรรมชาติ ตัวหลากหลายของ รูปแบบ BNF ที่มีความสำคัญ คือ ใช้ใน บทนิยาม ของ ภาษา COBOL ซึ่งเรียกว่า ไวยากรณ์ CBL (COBOL-like grammar)

ไวยากรณ์ BNF (BNF Grammars)

ไวยากรณ์ BNF (Backus-Naur form) พัฒนาขึ้นมาเพื่อใช้เป็น บทนิยาม วากยสัมพันธ์ ของภาษา ALGOL ผู้คิดค้นคือ John Backus ในปี ค.ศ.1960 ในเวลาใกล้ๆ กันนั้น รูปแบบไวยากรณ์ที่คล้ายกัน เรียกว่า ไวยากรณ์ไม่พึ่งบริบท พัฒนาโดย นักภาษาศาสตร์ ชื่อ Noam Chomsky ในปี ค.ศ. 1959 เพื่อใช้สำหรับ บทนิยาม ของ วากยสัมพันธ์ภาษาธรรมชาติ รูปแบบ BNF และ ไวยากรณ์ไม่พึ่งบริบท ในทางกำลัง (power) แล้วเท่าเทียมกัน ความแตกต่าง ที่สำคัญ มีเพียงเครื่องหมายเท่านั้น ด้วยเหตุผลนี้ คำว่า ไวยากรณ์ BNF และไวยากรณ์ไม่พึ่งบริบท ปกติ ในการอภิปรายเรื่อง วากยสัมพันธ์ จึงสลับที่กันได้

ไวยากรณ์ BNF ประกอบด้วย เซตจำกัดของ กฎไวยากรณ์ BNF ซึ่งรวมเข้าด้วยกัน เพื่อ นิยามภาษา ในที่นี้ คือ ภาษาโปรแกรม ก่อนที่เราจะดูการประกอบกัน ของ กฎไวยากรณ์เหล่านี้ คำว่า **ภาษา** (language) ในที่นี้ สมการมีการอธิบายที่ขยาย บางอย่าง เพราะว่า วากยสัมพันธ์ เกี่ยวข้องเฉพาะ รูปแบบ ไม่ใช่ ความหมาย ภาษาโปรแกรม พิจารณาเฉพาะวากยสัมพันธ์ ประกอบด้วย เซตของ โปรแกรมถูกต้องเชิงวากยสัมพันธ์ แต่ละชุดคือสายอักขระ

(A **programming language** , considered syntactically, consists of a set of syntactically correct programs, each of which is simply a character string.)

โปรแกรมถูกต้องเชิงวากยสัมพันธ์ ไม่จำเป็นต้องมี ความหมายใดๆ นั่นคือ ถ้ามันถูกกระทำการ ไม่จำเป็นต้องคำนวณ สิ่งใดที่เป็นประโยชน์ หรือ สิ่งใดทั้งหมดเพื่อเหตุผลนั้น มันอาจเป็นเพียง ลูป (loop)

โดยทั่วไป ไวยากรณ์แบบทางการ สำหรับภาษาโปรแกรม ยอมให้โปรแกรมถูกต้องเชิงวากยสัมพันธ์เช่นนั้น จำนวนมาก แต่ โปรแกรม ไม่มีความหมายเชิงความหมาย โปรดระลึกว่า ไวยากรณ์นิยามเฉพาะ เซตของ สายอักขระ แต่ไม่กำหนดความหมายใดๆ ให้กับสายอักขระเหล่านั้น

เรานำ ความเกี่ยวข้อง ที่ขาดหายไปนี้ กับความหมาย หนึ่งขั้นตอนต่อไป และยอมรับบทนิยาม ดังนี้

ภาษา หมายถึง เซตใดๆ ของ สายอักขระ (ความยาวจำกัด) (ด้วย ตัวอักขระ เลือกจากตัวอักษรจำกัดคงที่ ของสัญลักษณ์ บางอย่าง)

(A **language** is any set of (finite-length) character strings (with characters chosen from some fixed finite alphabet of symbols.)

ภายใต้ บทนิยามนี้ FORTRAN หมายถึง ภาษา (ประกอบด้วย สายอักขระทั้งหมด แทนที่โปรแกรม FORTRAN ซึ่งถูกต้องเชิงวากยสัมพันธ์) แต่เซตของ ข้อความสั่งกำหนดค่า FORTRAN ทั้งหมด หรือแม้แต่ เซต ประกอบด้วย ลำดับของ a's และ b's ซึ่ง a's ทั้งหมด อยู่ข้างหน้า b's ทั้งหมด (ตัวอย่างเช่น ab, aab, abb, ...) ภาษา อาจประกอบด้วย เฉพาะเซตจำกัดของสายอักขระ เท่านั้น ตัวอย่างเช่น ภาษาประกอบด้วย ตัวคั่น Pascal ทั้วหมด ได้แก่ **begin**, **end**, **if**, **then** และอื่นๆ ข้อจำกัดเฉพาะภาษา คือ สายอักขระแต่ละชุดของมัน ต้องมีความยาวจำกัด และต้องประกอบด้วย ตัวอักขระ เลือกแล้วจาก ตัวอักษรจำกัดคงที่ ของสัญลักษณ์ ภาษาโดยตัวมันเอง อาจประกอบด้วย จำนวนไม่จำกัด ของสายอักขระ

ไวยากรณ์ BNF นิยามภาษา ในลักษณะตรงไปตรงมา ในกรณีง่ายที่สุด กฎไวยากรณ์อาจเป็นรายการสมาชิก ของภาษาจำกัด ตัวอย่างเช่น

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

กฎ BNF ข้อนี้ นิยามว่า ภาษาประกอบด้วย สายอักขระ สิบตัว (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) โดยแสดงรายการ เซตของทางเลือก กฎไวยากรณ์ข้างต้น อ่านว่า

"A digit is either a '0' or a '1' or a '2' or a . . ."

คำว่า digit เรียกว่า **ประเภทวากยสัมพันธ์** (syntactic category) หรือ **nonterminal** โดยหลักการ คือ ชื่อหนึ่ง สำหรับภาษา นิยามโดย กฎไวยากณ์ สัญลักษณ์ ::= หมายถึง " i s defined as" หรือ เพียงแค่ "is" และเครื่องหมาย | อ่านว่า "or" และแยกทางเลือกต่างๆ ในบทนิยาม

เราเคยนิยาม เซตพื้นฐาน ของ ประเภทวากยสัมพันธ์ (จริงๆ คือ ภาษาย่อย) เราอาจใช้สิ่งเหล่านี้ ในการสร้าง ภาษาซับซ้อนมากขึ้น

ตัวอย่าง กฎ

<conditional statement> ::=

if <Boolean expression> **then** <statement> **else** <statement> |

if <Boolean expression> **then** <statement>

นิยามภาษา ประกอบด้วย <conditional statement> โดยใช้ประเภทวากยสัมพันธ์ <Boolean epression> และ <statement> ซึ่งต้องถูกนิยามต่อไป โดยใช้กฎไวยากรณ์อื่น โปรดสังเกตว่า กฎข้างต้นแสดงว่าข้อความสั่งมีเงื่อนไข มี รูปแบบสองทางเลือก (แยกกันด้วยสัญลักษณ์ |) แต่ละทางเลือก ถูกสร้าง จาก การต่อกัน ของสมาชิก หลายตัว ซึ่งอาจเป็น literal strings (เช่น **if** หรือ **else** หรือ ประเภทวากยสัมพันธ์ เมื่อ ประเภทวากยสัมพันธ์ ถูกกำหนดขึ้น หมายถึง any string ในภาษาย่อย นิยามโดย ประเภทวากยสัมพันธ์ <Boolean expression> ประกอบด้วย เซตของ strings แทน นิพจน์แบบบลู ที่ถูกต้อง กฎข้างต้น อนุญาตให้ string หนึ่งชุด ใส่เข้าไประหว่าง **if** และ **then** ของ ข้อความสั่งมีเงื่อนไข

อีกหนึ่งรูปแบบของกฎไวยากรณ์ มีประโยชน์ คือ ใช้ ประเภทวากยสัมพันธ์ นิยาม การเรียกซ้ำ ใน บทนิยาม ดังนั้น เป็นเทคนิค ซึ่งใช้ในกฎ BNF เพื่อระบุการทำซ้ำ ตัวอย่างเช่น กฎ

<unsigned integer> ::= <digit> | <unsigned integer><digit>

นิยาม จำนวนเต็มไม่มีเครื่องหมาย เป็นลำดับของ <digit>s โดยใช้ ประเภทวากยสัมพันธ์ <unsigned integer> เรียกซ้ำ ทางเลือกที่หนึ่ง ของกฎ ยอมให้ <digit> หนึ่งตัว เป็น <unsigned integer>

ดังนั้น <digit> สองตัว ในลำดับ ยังคงเป็น รูปแบบ <unsigned integer> และเนื่องจาก สิ่งนี้เป็นจริง สำหรับ two <digit>s และเนื่องจาก สิ่งนี้เป็นจริง สำหรับ <digit>s , และยังคงเป็นจริง สำหรับ three <digit>s ในลำดับ เช่นนี้เรื่อยไป

ไวยากรณ์ BNF บริบูรณ์ หมายถึง เซตของกฎไวยากรณ์ ซึ่งรวมกัน เพื่อนิยาม ลำดับชั้นของ ภาษาย่อย นำไปสู่ ประเภทวากยสัมพันธ์ ระดับบนสุด ซึ่งสำหรับภาษาโปรแกรม ปกติหมายถึง ประเภท <program> รูป 9-3 แสดงให้เห็น ไวยากรณ์ซับซ้อน มากยิ่งขึ้น ใช้สำหรับนิยามวากยสัมพันธ์ ชนิดของ ข้อความสั่งกำหนดค่าอย่างง่าย โดยใช้ ประเภทวากยสัมพันธ์พื้นฐาน <identifier> และ <number> ดังนี้

<identifier statement> ::= <variable> := <arithmetic expresion>

<arithmetic statement> ::= <term> I <arithmetic expression> + term |
<arithmetic expression> - <term>

<term> ::= <factor> I <term> * <factor> <term> / <factor>

<factor> ::= <primary> I<factor> ↑ <primary)

<primary> ::= <variable> I <number> (<arithmetic expression>)

<variable> ::= <identifier> I <identifier> [<subscript list>]

<subscript list> ::= <arithmetic expression> I <subscript list>,
<arithmetic expression>

รูป 8-3 ไวยากรณ์ BNF สำหรับ ข้อความสั่งกำหนดค่า อย่างง่าย

หน้าที่ของ ไวยากรณ์ BNF (Functions of a BNF grammar)

การใช้ ไวยากรณ์แบบทางการ เพื่อนิยาม วากยสัมพันธ์ของภาษาโปรแกรม มีความสำคัญทั้ง ผู้ใช้ภาษา (language user) และ ผู้ทำภาษาให้เกิดผล (implementor) ผู้ใช้ ดู ไวยากรณ์ แบบทางการ เพื่อ ตอบคำถามปลีกย่อยเกี่ยวกับ รูปแบบโปรแกรม, เครื่องหมายกำกับวรรคตอน, และโครงสร้าง

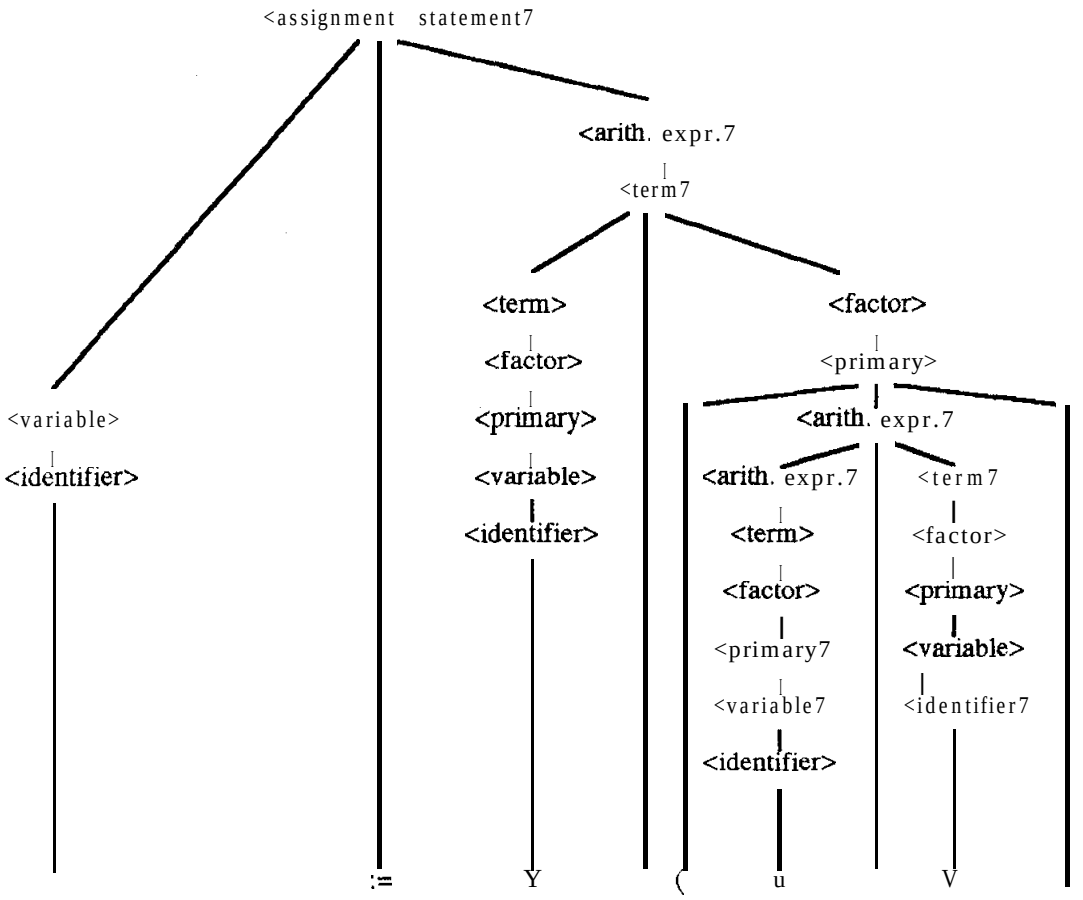
ผู้ทำภาษาให้เกิดผล อาจใช้ ไวยากรณ์แบบทางการ เพื่อหากรณีที่เป็นไปได้ทั้งหมด ของ

โครงสร้าง โปรแกรมอินพุท ซึ่งยอมให้ใช้ได้ และสิ่งเหล่านี้ ซึ่ง ตัวแปลภาษาของมัน จะต้องเกี่ยวข้องด้วย โปรแกรมเมอร์ และ implementor ทั้งคู่ มีความเห็นตรงกันบน บทนิยาม ซึ่งอาจถูกนำมาใช้ เพื่อแก้ปัญหา การโต้แย้ง เกี่ยวกับ โครงสร้างวากยสัมพันธ์ ที่ใช้ได้ บทนิยามวากย-สัมพันธ์ แบบทางการ จะช่วยขจัด ความแตกต่างเชิงวากยสัมพันธ์ส่วนย่อย ระหว่าง การทำให้เกิดผลของภาษา

หน้าที่หลักของไวยากรณ์ คือ การแยกความแตกต่างระหว่าง string ซึ่งถูกต้องเชิงวากย-สัมพันธ์ กับ string ซึ่งไม่ถูกต้องเชิงวากยสัมพันธ์

(The basic function of any grammar is that of distinguishing between syntactically correct and syntactically incorrect strings.)

รูปแบบไวยากรณ์ BNF ทำสิ่งนี้ผ่าน เซตของ กฎ BNF ซึ่งนิยามเซตของ strings ถูกต้องเชิงวากยสัมพันธ์ ทั้งหมด ในการตัดสินใจว่า string ที่กำหนดให้ แทน โปรแกรมถูกต้องเชิงวากยสัมพันธ์ ในภาษา ซึ่งนิยาม โดย ไวยากรณ์ BNF หรือไม่ เราต้องใช้ กฎไวยากรณ์เพื่อสร้างการวิเคราะห์วากยสัมพันธ์ หรือ **วิเคราะห์กระจาย** (parse) string ถ้าการวิเคราะห์กระจายประสบความสำเร็จ แสดงว่า มันเป็น string ถูกต้องเชิงวากยสัมพันธ์ ในภาษานั้น ถ้าไม่มีวิธีใดที่จะวิเคราะห์กระจาย string ด้วย กฎ ไวยากรณ์ที่กำหนดให้ แสดงว่า string นั้น ไม่อยู่ในภาษานั้น รูป 9-4 แสดงให้เห็น **ต้นไม้วิเคราะห์กระจาย** (parse tree) ซึ่งเป็นผลลัพธ์ จากการวิเคราะห์กระจาย ของ ข้อความสั่งกำหนดค่า โดยใช้ไวยากรณ์ BNF ในรูป 9-3



รูป 8-4 ต้นไม้วิเคราะห์กระจายสำหรับข้อความสั่งกำหนดค่า

ขณะที่ หน้าที่หลัก ของ ไวยากรณ์ คือ การแยกความแตกต่างระหว่าง string ถูกต้อง กับ string ไม่ถูกต้อง, กฎไวยากรณ์ ยังมีหน้าที่ที่สอง ซึ่ง ในทางปฏิบัติ มีความสำคัญ เกือบเท่ากัน ไวยากรณ์ BNF กำหนด **โครงสร้าง** (structure) ให้ string แต่ละชุดในภาษา ซึ่งนิยาม โดย ไวยากรณ์ จะเห็นได้จากรูป 8-4

โปรดสังเกตว่า โครงสร้างซึ่งถูกกำหนด ให้เป็นต้นไม้ เนื่องจากข้อจำกัด บน กฎไวยากรณ์ BNF ทุกใบ (each leaf) ของต้นไม้วิเคราะห์กระจาย เป็น อักขระหนึ่งตัว หรือ **ชิ้นศัพท์** (lexical item) ใน input string แต่ละจุดกิ่ง ระดับกลาง ในต้นไม้ ถูกทำเครื่องหมายด้วย ประเภทวากยสัมพันธ์

ซึ่งกำหนดชนิด ให้กับ ต้นไม้ส่วนย่อย ระดับต่ำของมัน โหนดรากของต้นไม้ ถูกทำเครื่องหมาย ด้วย ประเภทวากยสัมพันธ์ กำหนดให้กับ ภาษาโดยรวม ในกรณีนี้ คือ ประเภท <assignment statement>

ต้นไม้วิเคราะห์กระจายกำหนด ให้กับ string ถูกต้องเชิงวากยสัมพันธ์ แต่ละชุด ในภาษา มีความสำคัญ เพราะว่า มันสามารถนำไปใช้ จัดชนิด ของ โครงสร้างความหมาย คาดการณ์ได้ อย่างมากของโปรแกรม ตัวอย่างเช่น ไวยากรณ์ BNF ของภาษา Pascal กำหนดว่า โครงสร้าง ของโปรแกรม คือ ลำดับของ การประกาศ และ ข้อความสั่ง ด้วย บล็อกซ้อนกัน

ข้อความสั่ง หมายถึง โครงสร้าง โดยใช้ นิพจน์ชนิดต่างๆ และ นิพจน์ ประกอบขึ้นจาก ตัวแปรอย่างง่ายและตัวแปรมีดรรชนีล่าง ตัวดำเนินการดังเดิม การเรียกฟังก์ชัน และอื่นๆ ณ ระดับต่ำสุด จะเป็น ไอเดนติไฟเออร์ และ เลข ถูกแยกจากกัน ให้เป็นส่วนประกอบต่างๆ ของมัน จากการศึกษาไวยากรณ์, โปรแกรมเมอร์ อาจได้เห็นโดยตรง ของ โครงสร้างหลากหลาย ซึ่งรวมกัน เพื่อประกอบเป็น โปรแกรมถูกต้อง

ข้อสังเกต สิ่งสำคัญ คือ ไม่มีไวยากรณ์ใดๆ ซึ่งจำเป็น ต้องกำหนดให้กับโครงสร้าง ซึ่งเราอาจคาดคิด ที่จะเป็น สมาชิก โปรแกรมที่กำหนดให้

(It is important to note that no grammar must **necessarily** assign the structure one would expect to a given program element.)

ภาษาเดียวกัน อาจถูกนิยาม โดย ไวยากรณ์ แตกต่างกัน มากมาย อาจทำให้เห็นอย่างง่าย โดย เปลี่ยนแปลง ไวยากรณ์ รูป 8-3 เล็กน้อย ให้เป็นรูป 8-5 ตัวอย่างเช่น ให้ไวยากรณ์ เพื่อ นิยาม ภาษาเดียวกัน เช่น ไวยากรณ์ ในรูป 8-3 แต่โปรดสังเกตว่า โครงสร้าง กำหนดโดย ไวยากรณ์ใหม่นี้ ไม่เหมือนกับ โครงสร้าง ซึ่งเราคาดคิดไว้

<assignment statement> ::= <variable> := <arithmetic expression>

<arithmetic expression> ::= <term> I <arithmetic expression> ↑ <term>

I <arithmetic expression> * <term>

I <arithmetic expression> + <term>

<term> ::= <primary> I <term> - <primary> I <term> / <primay>

<primary> ::= <variable> I <number> I (<arithmeticexpression>)

<variable> ::= <identifie> I <identifier> [<subscript list>]

<subscript list> ::= <arithmetic expression> I <arithmetic expression>,

<subscript list>

รูป 8-5 ไวยากรณ์ BNF เพื่อใช้นิยามภาษาเดียวกัน เช่น ไวยากรณ์ในรูป 8-3

ไวยากรณ์ BNF ทั้งๆ ที่มี โครงสร้างอย่างง่ายมากมายของมัน ถูกนำมาใช้ เพื่อทำงานที่ดีอย่างน่าประหลาดใจ ของ การนิยามวากยสัมพันธ์ ของ ภาษาโปรแกรมส่วนใหญ่ ตัวอย่างเช่น ไวยากรณ์ BNF ของภาษา ALGOL (Naur, 1963) เพิ่มเติมด้วย คำกล่าวภาษาอังกฤษถึงข้อจำกัดวากยสัมพันธ์ เพิ่มเติม จำนวน ไม่มาก อธิบายเซตของ โปรแกรมภาษา ALGOL ที่ถูกต้องอย่างใกล้ชิดมาก

เนื้อหาวากยสัมพันธ์ ซึ่ง ไม่สามารถถูกนิยาม ได้โดย ไวยากรณ์ BNF สิ่งเหล่านี้ จะเกี่ยวข้องกับ ความขึ้นอยู่กับ เนื้อความ (contextual dependence)

"ไอเดนติไฟเออร์ตัวเดียวกัน จะประกาศสองครั้งในบล็อกเดียวกันไม่ได้"

"The same identifier may not be declared twice in the same block"

"ไอเดนติไฟเออร์ทุกตัว ต้องถูกประกาศ ในบล็อก ปิดล้อม จุด การใช้ของมัน"

"every identifier must be declared in some block enclosing the point of its use"

และ

"แถวลำดับ ซึ่งประกาศให้มี สองมิติ ไม่สามารถอ้างถึงด้วย ดรรชนีล่าง สามตัว"

"au array declared to have hvo dimensions can not be referenced with three subscripts"

ทุกข้อ ไม่สามารถ นิยาม โดยใช้เฉพาะ ไวยากรณ์ BNF ข้อจำกัด ชนิดนี้ ต้องนิยามโดย การเพิ่มเรื่องราว ให้กับ ไวยากรณ์ BNF แบบทางการ

ไวยากรณ์ COBOL (เหมือน COBOL) : ความหลากหลายของ BNF เครื่องหมายที่มีประโยชน์ (CBL (COBOL-like) Grammars : A Useful Notation Variant of BNF)

นอกจาก กำลัง (power), ความสวยงาม (elegance), และความง่าย (simpliity) ของ ไวยากรณ์ BNF แล้ว มันยังเป็น เครื่องหมายที่ดี สำหรับการสื่อสาร กฎต่างๆ ของ วากยสัมพันธ์ ภาษาโปรแกรม กับ โปรแกรมเมอร์ฝึกหัด

เหตุผลข้อแรก คือ ความง่ายของ กฎ BNF ซึ่งบังคับการแทนที่ ซึ่งไม่ค่อยเป็นธรรมชาติ สำหรับ โครงสร้างวากยสัมพันธ์ ร่วมของสมาชิกละเว้นได้ (optional elements), สมาชิกเลือก (alternative elements) และ สมาชิกซ้ำ (repeated elements) ภายในกฎไวยากรณ์ ตัวอย่างเช่น เพื่อแสดง ความคิดวากยสัมพันธ์ อย่างง่าย

"a **signed integer** is a sequence of digits preceded by an optional plus or minus"

เราต้องเขียน เซตของกฎเรียกซ้ำ ซับซ้อน ด้วย BNF ดังนี้

<signed integer> ::= + <integer> | - <integer> | <integer>

<integer> ::= <digit> | <integer><digit>

เครื่องหมายเหล่านี้ เป็นข้อบกพร่อง ของ BNF ซึ่งส่วนใหญ่ไม่สำคัญต่อการพัฒนาเชิงทฤษฎีของ คุณสมบัติของไวยากรณ์ แต่มีหลายคนซึ่งใช้ BNF สำหรับอธิบายวากยสัมพันธ์ ของ ภาษาจริง (actual languages) ซึ่งมีการ ขยายเครื่องหมาย ในหลายวิธี เพื่อจัดหากฎการเขียน ที่เป็นวิธี ธรรมชาติมากกว่า การใช้เครื่องหมายทางเลือกเหล่านี้ อย่างกว้างขวาง มากที่สุด คือที่นำมาใช้ใน บทนิยาม ของ ภาษา COBOL ซึ่งดูเหมือนว่า ถูกพัฒนาอย่างเป็นอิสระ และ เป็นเวลาเดียวกับ การพัฒนา BNF

เครื่องหมายนี้ นำมาใช้ค่อนข้าง ไม่เป็นทางการ และได้รับ การวิเคราะห์อย่างเข้ม ไม่มากนัก ซึ่งประยุกต์กับ ไวยากรณ์ BNF

ข้อดีในการพิจารณา กฎ CBL คือ เป็นการขยายของ BNF ซึ่งใส่เครื่องหมายใหม่ ต่อไปนี้

1. ภายในกฎไวยากรณ์, สมาชิกละเว้นได้ แสดงให้เห็นโดยการปิดล้อมสมาชิกนั้นในวงเล็บใหญ่ [...]

2. สมาชิกเลือก แสดงให้เห็น โดยการเขียนรายการ ทางเลือกต่างๆ ปิดล้อมด้วย วงเล็บปีกกา {...}*
3. ทางเลือก ละเว้นได้ แสดงด้วย การเขียนรายการ ทางเลือก แล้วปิดล้อมด้วย วงเล็บใหญ่ [...]
4. สมาชิกซ้ำ แสดงด้วยการเขียนรายการ สมาชิกหนึ่งตัว (ปิดล้อมในวงเล็บปีกกา หรือวงเล็บใหญ่ ถ้าจำเป็น) ตามด้วยสัญกรณ์ ...
5. คำหลักที่ต้องเขียน ให้ขีดเส้นใต้ ส่วนคำรบกวน ไม่ต้องขีดเส้นใต้

ตัวอย่าง

1. <signed integer> ::= [±] <digit> . . .
2. <identifier> ::= <letter> <letter> . . . <digit>
3. <ALGOL conditional statement> ::= **if** <Boolean expression> **then** <unconditional statement> [**else** <statement>]
4. <COBOL ADD statement> ::= **ADD** <identifier> , <identifier> . . . **TO** <numbe> , <number> <identifier>[**ROUNDED**][,<identifier>[**ROUNDED**]] ... [; ON **SIZE ERROR** <statement>]

รูปแบบไวยากรณ์ CBL นี้ มีความเป็นธรรมชาติ และความรวบรัด เมื่อเปรียบเทียบกับ BNF จะเห็นได้ชัด โดยดูความแตกต่าง ในบทนิยามของ <signed integer> ในรูปแบบทั้งสอง หรือ โดยการเขียน บทนิยาม BNF ที่ครบถ้วน ของ <COBOL ADD statement> (แบบฝึกหัดข้อ 6)
อย่างไรก็ตาม รูปแบบไวยากรณ์ BNF และ CBL มีความเท่าเทียมกัน ในทาง power

ภาษาใดๆ ซึ่งถูกนิยามด้วย ไวยากรณ์ ใน หนึ่งรูปแบบ สามารถนิยามได้ ด้วย ไวยากรณ์ ในรูปแบบอื่นได้ด้วย

(Any language that can be defined by a grammar in one form can also be defined by a grammar in the other.)

---

* แทน vertical bar ของ BNF

แบบฝึกหัด

1. จงพิจารณา กฎไวยากรณ์ BNF ต่อไปนี้

<pop> : : = [<bop>, <pop>] I <bop>

<bop> ::= <boop> I (<pop>)

<boop> : : = a I b I c

สำหรับ strings แต่ละชุด ข้างล่างนี้ จงบอกว่า มันเป็นสมาชิก ของ วากยสัมพันธ์ประเภท อะไรบ้าง

a) c

b) (a)

c) [b]

d) ([a, b])

e) [(a), b]

f) [(a), [b, all

2. จงเขียนไวยากรณ์ BNF สำหรับภาษา ประกอบด้วย เลขฐานสองทั้งหมด ซึ่งประกอบด้วย อย่างน้อยที่สุด 1's ติดกันสามตัว (ภาษาประกอบด้วย string ข้างล่างนี้)

valid string 000011111110100

valid string 1111110

invalid string 0011000101011

3. วากยสัมพันธ์ของภาษา ลิง ง่ายมาก ลิงสามารถพูดโดยไม่ผิด ตัวอักษรของภาษา ได้แก่ {a, b, c, Λ} เมื่อ A หมายถึงที่ว่าง จงพิจารณาไวยากรณ์

<stop> ::= b I d

<plosive> ::= <stop> a

<syllable> ::= <plosive> I <plosive><stop> I a <plosive> I a<stop>

<word> ::= <syllable> I <syllable><word><syllable>

<sentence> ::= <word> I <sentence> A <word>

ผู้พูดคนไหน คือ ตัวแทนลับ ใน บรรดาการปลอมตัวของลิง (Which of the following speakers is the secret agent in monkey disguise?

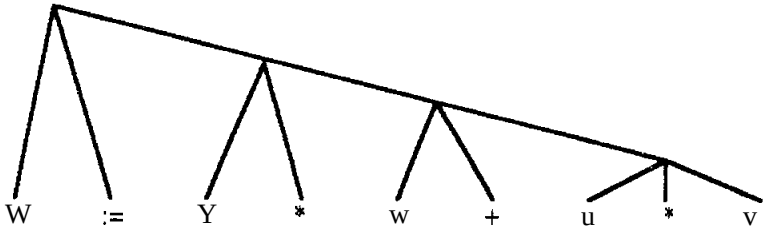
Ape : baΛ ababdadaΛ badΛ dabbada

Chimp : abdabaadabΛ ada

Baboon : dadΛ adΛ abaadad Λ badadbaad

4. จงเขียนไวยากรณ์ CBL สำหรับภาษา นิยามโดย ไวยากรณ์ BNF ของรูป 9-3

5. จงเขียนไวยากรณ์ ของรูป 8-3 ใหม่แสดง โครงสร้างถูกต้อง สำหรับ นิพจน์ สมมติเป็น กฎ APL สำหรับ การทำก่อน และ การสลับที่ ตัวคำเนินการ : ตัวคำเนินการทั้งหมด รวมทั้ง การกำหนดคือ ของ การทำก่อนเท่ากัน และการสลับที่ จาก ขวาไปซ้าย ตัวอย่างเช่น ข้อความสั่งกำหนดค่า W := Y * W + U * V ควรกำหนดโครงสร้าง ดังนี้



6. จงเขียน ไวยากรณ์ BNF สำหรับ ข้อความสั่ง ADD ของภาษา COBOL นิยามโดยใช้ ไวยากรณ์ CBL ในหัวข้อ 8-4

7. จงสร้างต้นไม้วิเคราะห์กระจาย สำหรับ ข้อความสั่งกำหนดค่า โดยใช้ ไวยากรณ์ BNF ของรูป 8-3

a) A[2] := B + 1

b) AII, JI := A[J, I]

c) X := u - v $\uparrow$ W + X / Y

d) P := U / (V / (W / X))