Programming Language Principle and Processing

Jaruloj Chongstitvatana

เอกสารคำสอน

2301380

หลักการและการประมวลผลภาษาโปรแกรม

Programming Language Principle and Processing

จารุโลจน์ จงสถิตย์วัฒนา

ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์

จุฬาลงกรณ์มหาวิทยาลัย

สารบัญ

คำนำ

ภาษาโปรแกรมและการการประมวลภาษาโปรแกรมเป็นพื้นฐานที่สำคัญส่วนหนึ่งของวิทยาการคอมพิวเตอร์ ขั้นตอนวิธีที่ใช้ในการประมวลภาษาโปรแกรมยังใช้ในการประมวลผลข้อมูลที่เป็นพื้นฐานในงานประยุกต์หลายด้าน การประมวลข้อความ (text processing) ใช้สแกนเนอร์เพื่อแยกคำและใช้หลักการแจงส่วนเพื่อหาความสัมพันธ์ระหว่างคำ การรวบรวมข้อมูลเว็บ (web crawling) ใช้หลักการแจงส่วนเพื่อระบุความสัมพันธ์ระหว่างข้อมูลในเว็บ

ผู้ออกแบบภาษาโปรแกรมต้องการให้มนุษย์อ่านโปรแกรมได้ง่าย อย่างไรก็ตาม การประมวลภาษาโปรแกรมมีความสัมพันธ์กับการออกแบบภาษาโปรแกรมอย่างมาก ลักษณะหลายอย่างของภาษาโปรแกรมเป็นผลจากขั้นตอนวิธีที่ใช้ในการประมวลผลภาษาโปรแกรม ดังนั้นความรู้เกี่ยวกับการประมวลภาษาโปรแกรมยังช่วยให้สามารถเข้าใจความหมายของภาษาโปรแกรมได้ดีขึ้น

รายวิชานี้อธิบายขั้นตอนวิธีที่ใช้ในการประมวลผลภาษาโปรแกรมโดยเชื่อมโยงกับการออกแบบลักษณะของภาษาโปรแกรม ความเข้าใจเกี่ยวกับความสัมพันธ์นี้ช่วยให้เข้าใจแนวคิดใหม่ ๆ ในภาษาโปรแกรมได้อย่างลึกซึ้ง

หากผู้อ่านพบข้อผิดพลาดในเอกสารนี้ กรุณาแจ้งทางอีเมล jaruloj@gmail.com เพื่อการปรับปรุงเอกสารต่อไป

จารุโลจน์ จงสถิตย์วัฒนา

1. บทนำ

ในประวัติศาสตร์คอมพิวเตอร์ที่ผ่านมามีภาษาโปรแกรมที่ใช้กันเป็นจำนวนมาก ความนิยมใช้ภาษาโปรแกรมขึ้นกับปัจจัยหลายด้าน ภาษาที่ออกแบบมาเหมาะสมเป็นปัจจัยหนึ่งที่มีผลต่อความนิยม นอกจากนั้นเครื่องมือช่วยพัฒนาโปรแกรมสำหรับภาษานั้นยังมีผลต่อความนิยมด้วย หากมีตัวแปลภาษาให้ใช้ได้บนเครื่องคอมพิวเตอร์ที่ใช้ตัวประมวลผลกลาง (CPU) ต่างๆกันและใช้ได้บนระบบปฏิบัติการ (operating system) หลายระบบโปรแกรมเมอร์จะเลือกใช้ภาษาโปรแกรมนั้นมากขึ้น แหล่งข้อมูลสำหรับการเรียนรู้ภาษาก็เป็นปัจจัยอีกข้อหนึ่งที่ทำให้โปรแกรมเมอร์เลือกใช้ภาษานั้น ในบทนี้เราจะพิจารณาการออกแบบภาษาที่ดี แบบความคิด (paradigms) ของภาษาโปรแกรม และ ขั้นตอนการประมวลผลสำหรับภาษาโปรแกรม (programming language processing)

1.1 การออกแบบภาษาโปรแกรม

ในหัวข้อนี้เราจะพิจารณาการออกแบบภาษาที่ดีคือให้ใช้ง่าย ซึ่งรวมถึง อ่านง่าย (readability) เขียนง่าย (writability) และเชื่อถือได้ (reliability) ภาษาโปรแกรมที่อ่านง่ายหมายถึงภาษาที่ทำให้เขียนโปรแกรมแล้วอ่านเข้าใจได้ง่ายสำหรับทั้งผู้เขียนโปรแกรมเองและผู้ที่มาอ่านโปรแกรมภายหลัง โปรแกรมที่อ่านเข้าใจได้ง่ายมีความสำคัญในการพัฒนาโปรแกรมเนื่องจากโปรแกรมที่เขียนด้วยคนหนึ่งยังต้องมีโปรแกรมเมอร์คนอื่นมาอ่านโปรแกรมนั้นเพื่อแก้ไขปรับปรุงในภายหลังหรือผู้ที่เขียนโปรแกรมที่เกี่ยวข้องในทีมเดียวกันอาจต้องอ่านโปรแกรมนั้นเพื่อเขียนโปรแกรมส่วนอื่นให้ทำงานร่วมกันได้ถูกต้อง ดังนั้นภาษาโปรแกรมที่ดีควรทำให้โปรแกรมอ่านแล้วเข้าใจได้ง่าย

ภาษาโปรแกรมที่เขียนง่ายทำให้โปรแกรมเมอร์ใช้ความพยายามไม่มากในการเรียนรู้ภาษาและการใช้ภาษานั้นเขียนโปรแกรม ภาษาบางภาษาใช้เวลาเรียนภาษาไม่นาน แต่อาจใช้เวลาในการเขียนโปรแกรมด้วยภาษานั้นนานกว่าเมื่อใช้ภาษาอื่น ภาษาระดับต่ำ (low-level language) เช่น ภาษาแอสเซมบลี (assembly language) มีคำสั่งน้อย จึงทำให้เรียนภาษาได้เร็ว แต่เมื่อเขียนโปรแกรมด้วยภาษาแอสเซมบลีจะใช้เวลามากกว่าภาษาระดับสูง (high-level language) ดังนั้นภาษาแอสเซมบลีจึงไม่เป็นภาษาที่เขียนง่าย

ภาษาโปรแกรมบางภาษาทำให้โปรแกรมเกิดความผิดพลาดได้น้อย เริ่มตั้งแต่โปรแกรมเมอร์เขียนโปรแกรมแล้วมีโอกาสผิดพลาดน้อย ในลำดับต่อมาตัวประมวลภาษาก็ตรวจสอบแล้วแจ้งข้อผิดพลาดเพื่อให้โปรแกรมเมอร์แก้ไขได้ นอกจากนั้นแล้วโปรแกรมจะมีข้อผิดพลาดจากกรณีที่ไม่ได้คาดคิดได้น้อย ภาษาที่ออกแบบให้มีโอกาสผิดพลาดน้อยถือว่ามีความเชื่อถือได้สูง

ลักษณะเฉพาะ (characteristics) ของภาษาโปรแกรมทำให้ภาษาใช้ยาก-ง่ายต่างกัน ในที่นี้เราจะพิจารณาลักษณะเฉพาะต่อไปนี้ คือ ความเรียบง่าย (simplicity) ความไม่ขึ้นต่อกันขององค์ประกอบย่อย (orthogonality) ชนิดข้อมูล (data type) ที่มีในภาษา ไวยากรณ์ (syntax) ของภาษา การแยกระดับความหมาย (abstraction) ในภาษาการแสดงออก (expressivity) ด้วยภาษา การตรวจสอบชนิดข้อมูล (type checking) การจัดการเมื่อเกิดกรณีไม่ปกติ (exception handling) การใช้ alias

ความเรียบง่าย

ความเรียบง่ายของภาษาเกิดจากการเลือกความคิดหลักและโครงสร้างในภาษาให้เหมาะสมตามความจำเป็น หากความคิดหลักของภาษามีน้อยและความคิดหลักเหล่านี้ไม่ขัดแย้งกัน โปรแกรมเมอร์สามารถเรียนรู้ภาษาได้ง่ายเพราะมีมีจำนวนแนวคิดที่ต้องเรียนรู้ไม่มากและแนวคิดจากสถานการณ์หนึ่งสามารถนำไปใช้ในสถานการณ์อื่น ๆ ได้ ตัวอย่าง เช่น ในภาษาโปรแกรม เช่น ภาษาซี ภาษาจาวา ภาษาไพธอน มีแนวความคิดของบล็อก (block) แนวความคิดนี้สามารถนำไปใช้ได้ในโครงสร้าง if, while, for และ function โดยไม่ขัดแย้งกัน เมื่อแนวคิดหลักในภาษามีจำนวนน้อย โปรแกรมเมอร์สามารถเรียนรู้ภาษาได้เร็ว หากเปรียบเทียบภาษาซีกับภาษาจาวา ภาษาจาวามี แนวคิดเชิงวัตถุเพิ่มมาทำให้เรียนรู้ได้ช้ากว่าภาษาซี แต่การใช้วัตถุในโปรแกรมทำให้โปรแกรมเมอร์ทำงานได้ง่ายขึ้น ดังนั้นผู้ออกแบบภาษาควรเลือกแนวคิดหลักสำหรับภาษาหนึ่งให้เหมาะกับการใช้งาน

ผู้ออกแบบภาษาควรเลือกโครงสร้างที่จำเป็นในการทำงานของโปรแกรมเมอร์โดยไม่ให้เกินความจำเป็น ตัวอย่าง เช่น ภาษาซีมีตัวกระทำ ++ แต่ x++ อาจเขียนแทนด้วย x=x+1 ได้ ดังนั้นจึงไม่จำเป็นต้องมีตัวกระทำ ++ เนื่องจากถ้าภาษามีโครงสร้างที่ต้องเรียนรู้เป็นจำนวนมากโปรแกรมเมอร์จะเรียนภาษาได้ช้า นอกจากนั้นโปรแกรมเมอร์จะเลือกใช้โครงสร้างที่ตนเองถนัดและไม่เลือกใช้บางโครงสร้างได้ ดังนั้นเมื่อโปรแกรมเมอร์อ่านโปรแกรมที่ผู้อื่นเขียนไว้ อาจพบโครงสร้างที่ตนเองไม่คุ้นเคยและทำความเข้าใจได้ช้า ในทางตรงข้าม ภาษาที่มีส่วนประกอบพื้นฐานน้อยทำให้โปรแกรมเมอร์เลือกใช้โครงสร้างได้จำกัด ดังนั้นเมื่อโปรแกรมเมอร์อื่นมาอ่านโปรแกรมก็จะพบโครงสร้างที่ใช้คล้าย ๆ กันได้มาก แต่โครงสร้างที่ต่างกันแต่ละแบบอาจทำให้โปรแกรมเมอร์เลือกใช้อธิบายการทำงานได้เหมาะสมในกรณีที่ต่างกัน ตัวอย่าง เช่น โครงสร้าง while ในภาษาโปรแกรมก็เพียงพอที่จะทดแทนโครงสร้าง for แต่เมื่อต้องการอธิบายการทำงานกับค่าในตัวแปรแถวลำดับ (array) ทีละค่าเรียงตามลำดับ โครงสร้าง for ใช้สื่อความหมายได้ชัดเจนกว่า ดังนั้นผู้ออกแบบภาษาโปรแกรมต้องพิจารณาความเหมาะสมด้วย

นั่นคือภาษาที่เรียบง่ายจะทำให้โปรแกรมอ่านง่ายและเขียนง่ายแต่หากจำกัดให้ภาษาเรียบง่ายเกินไปก็จะทำให้เขียนโปรแกรมยาก สุดท้ายความเรียบง่ายของภาษายังทำให้โปรแกรมน่าเชื่อถือมากขึ้น เนื่องจากภาษาที่อ่านง่ายและเขียนง่ายทำให้เกิดความผิดพลาดน้อยและหาข้อผิดพลาดง่าย

ความไม่ขึ้นต่อกันขององค์ประกอบย่อย

เนื่องจากภาษาโปรแกรมมีองค์ประกอบหลายชิ้นซึ่งอาจไม่เกี่ยวข้องกัน การนำองค์ประกอบเหล่านี้มาประกอบกันในโปรแกรมได้โดยไม่ขึ้นกับองค์ประกอบข้างเคียงทำให้อ่านโปรแกรมได้ง่าย เช่น ตัวกระทำ + และ * สามารถใช้กับตัวแปร ค่าคงที่ หรือ นิพจน์ซึ่งเป็นจำนวนเต็มหรือจำนวนจริงก็ได้ นั่นคือความหมายของตัวกระทำ + และ * ไม่ขึ้นกับองค์ประกอบข้างเคียงว่าเป็นตัวแปรหรือค่าคงที่หรือนิพจน์และไม่ขึ้นกับว่าเป็นจำนวนเต็มหรือจำนวนจริง การที่ความหมายขององค์ประกอบหนึ่งในภาษาไม่ขึ้นกับองค์ประกอบข้างเคียงเรียกว่ามี orthogonality ซึ่งทำให้อ่านโปรแกรมได้ง่าย แต่หากนำไปใช้ในบางสถานการณ์ก็อาจทำให้สับสน เช่น ในภาษาไพธอน ตัวกระทำ + ใช้กับสตริงได้โดยหมายถึงการนำสตริงมาเชื่อมต่อกัน (concatenation) ตัว

กระทำ * ใช้กับสตริงโดยหมายถึงการนำสตริงนั้นมาเชื่อมต่อกันหลาย ๆ ครั้ง แต่ในกรณีนี้ตัวถูกกระทำตัวหนึ่งเป็นสตริงและอีกตัวหนึ่งเป็นจำนวนเต็มเพื่อระบุว่านำสตริงนั้นมาต่อกันกี่ครั้ง

หากองค์ประกอบย่อยในภาษามีความหมายเป็นอิสระต่อกันโปรแกรมเมอร์จะเรียนภาษาได้ง่ายทำให้การอ่านและเขียนโปรแกรมง่ายไปด้วย แต่การออกแบบให้องค์ประกอบย่อยไม่ขึ้นต่อกันอาจขัดกับปัจจัยอื่น เช่น การใช้ * กับจำนวน และการใช้ * กับสตริงในภาษาไพธอน ทำให้ความหมายของตัวกระทำ * ขึ้นอยู่กับตัวถูกกระทำในที่นั้นด้วย แต่การใช้สัญลักษณ์เดียวกันสำหรับการคูณจำนวนและการต่อซ้ำสตริง มีข้อดีคือโปรแกรมเมอร์คุ้นเคยกับความหมายของการคูณจำนวนทางคณิตศาสตร์อยู่แล้ว การต่อซ้ำสตริงก็ใช้สัญลักษณ์เดียวกันกับการคูณจำนวน คือ • (เช่น x•y) หรือ เขียนตัวแปรต่อกับไป (เช่น xy) ดังนั้นการเลือกใช้สัญลักษณ์เดียวกันกับตัวกระทำการคูณจำนวนก็สื่อความหมายกับโปรแกรมเมอร์ได้ดี

ชนิดข้อมูลที่มีในภาษา

หากภาษาโปรแกรมมีชนิดของข้อมูลที่เหมาะสมให้โปรแกรมเมอร์เลือกใช้จะทำให้ โปรแกรมเข้าใจได้ง่าย อย่างไรก็ตาม งานประเภทต่าง ๆ กันต้องการชนิดของข้อมูลที่ต่างกัน เช่น งานทางคณิตศาสตร์อาจต้องการใช้ข้อมูลที่เป็นจำนวนเชิงซ้อน (complex number) แต่งานทางธนาคารอาจต้องการใช้ข้อมูลที่เป็นวันเวลา ชนิดข้อมูลที่เหมาะสมทำให้ภาษาอ่านง่ายและเขียนง่าย

ไวยากรณ์ของภาษา

ปัจจัยอีกประการหนึ่งในการออกแบบที่ทำให้ภาษาโปรแกรมอ่านง่ายคือการใช้รูปแบบที่สื่อความหมายได้ชัดเจน ดังนั้นภาษาโปรแกรมส่วนใหญ่จะใช้รูปแบบที่มาจากภาษาอังกฤษหรือภาษาคณิตศาสตร์ที่คนส่วนใหญ่เข้าใจกันดีอยู่แล้ว เช่น การเลือกคำที่มีความหมายในภาษาอังกฤษ (ได้แก่ if, while, for เป็นต้น) เป็นคำสั่งในภาษาโปรแกรม การใช้สัญลักษณ์ทางคณิตศาสตร์ (ได้แก่ +, -, *, /, >, < เป็นต้น) สำหรับตัวกระทำในภาษาโปรแกรม แต่ในกรณีที่สัญลักษณ์ทางคณิตศาสตร์นั้นไม่อยู่ในชุดตัวอักขระที่ใช้กันทั่วไปในภาษาโปรแกรมผู้ออกแบบภาษาในยุคแรก ๆ จึงเลือกใช้ตัวอักขระ 2 ตัวร่วมกัน เช่น >=, <=, != ในทางตรงข้าม สัญลักษณ์ทางคณิตศาสตร์บางตัวใช้ในหลายความหมายซึ่งนักคณิตศาสตร์สามารถเลือกตีความหมายตามบริบท เช่น + อาจหมายถึงการบวกจำนวน 2 จำนวนหรือการเชื่อมสตริง 2 สตริงหรือการบวกเวคเตอร์ เป็นต้น แต่เมื่อนำมาใช้ในภาษาโปรแกรมอาจทำให้ผู้อ่านสับสนได้

การแยกระดับความหมาย (abstraction) ในภาษา

การแยกระดับความหมายในภาษาใช้สร้างส่วนของโปรแกรมในระดับความหมายที่สูงขึ้นเพื่อซ่อนรายละเอียด เช่น การสร้างฟังก์ชันเพื่อให้ผู้ใช้เรียกใช้ได้โดยไม่ต้องสนใจการทำงานภายในฟังก์ชัน แต่ระบุข้อมูลเข้าให้ถูกต้องและนำผลลัพธ์ที่ส่งจากฟังก์ชันมาใช้ได้ ภาษาเชิงวัตถุแยกระดับความหมายด้วยโครงสร้างของคลาสด้วยอีกมิติหนึ่ง การแยกระดับความหมายทำให้อ่านและเขียนโปรแกรมได้ง่ายขึ้นเพราะสามารถเขียนและทำความเข้าใจกับโปรแกรมในระดับต่าง ๆ กันได้ตามต้องการ นอกจากนั้นยังทำให้ตรวจสอบความถูกต้องของโปรแกรมทีละส่วนได้ง่ายขึ้น ดังนั้นจึงทำให้ภาษาอ่านง่าย เขียนง่ายและเชื่อถือได้

การแสดงออก (Expressivity) ด้วยภาษา

ภาษาที่แสดงออกได้ดี จะทำให้อธิบายการทำงานที่ซับซ้อนได้ง่าย คำสั่ง โครงสร้างการโปรแกรม และชนิดข้อมูลที่เหมาะสมกับงานช่วยให้อธิบายการทำงานได้ง่าย เช่น คำสั่ง for ... in ... ในภาษาไพธอนอธิบายการทำงานกับข้อมูลแต่ละตัวในโครงสร้างได้ง่าย โครงสร้าง dictionary ในภาษาไพธอนทำให้อธิบายการหาค่าในตาราง (table lookup) ได้ง่าย ดังนั้นจึงทำให้เขียนโปรแกรมได้เร็ว

การตรวจสอบชนิดข้อมูล (type checking)

ภาษาโปรแกรมมีข้อมูลชนิดต่าง ๆ มีตัวแปรที่ใช้เก็บข้อมูล และ มีตัวกระทำหรือโครงสร้างการโปรแกรมที่ต้องใช้กับข้อมูลที่กำหนดเท่านั้น ดังนั้นการตรวจสอบชนิดข้อมูลที่ใช้ให้ถูกต้องจึงเป็นการตรวจสอบความถูกต้องระดับง่ายแบบหนึ่งซึ่งทำให้ภาษาเชื่อถือได้มากขึ้น ภาษาโปรแกรมที่บังคับให้ใช้ข้อมูลตรงตามชนิดที่กำหนดโดยไม่ยอมเปลี่ยนข้อมูลให้อัตโนมัติ เรียกว่า ภาษาที่บังคับชนิดอย่างเข้ม (strongly-typed language) ภาษาโปรแกรมที่ยอมแปลงชนิดข้อมูลให้ตรงตามที่ตัวกระทำหรือโครงสร้างโปรแกรมกำหนดไว้เรียกว่าเป็นภาษาที่บังคับชนิดอย่างอ่อน (weakly-typed language) ภาษาซีเป็นภาษาที่บังคับชนิดอย่างอ่อนซึ่งยอมให้เปลี่ยนข้อมูลชนิดจำนวนเต็มให้เป็นชนิดจำนวนจริงได้ ภาษาที่บังคับชนิดอย่างเข้มจะตรวจสอบชนิดข้อมูลและพบข้อผิดพลาดของโปรแกรมเมอร์ได้มากกว่า ภาษาที่บังคับชนิดอย่างอ่อนจะยอมให้เปลี่ยนชนิดข้อมูลโดยโปรแกรมเมอร์ไม่ต้องระบุจึงอาจเกิดข้อผิดพลาดได้มากกว่า อย่างไรก็ตามการบังคับให้โปรแกรมเมอร์ระบุการเปลี่ยนชนิดก็ทำให้เขียนโปรแกรมยากขึ้น

การจัดการเมื่อเกิดกรณีไม่ปกติ (exception handling)

การจัดการเมื่อเกิดกรณีไม่ปกติเป็นส่วนประกอบของภาษาโปรแกรมที่สร้างขึ้นเพื่อจัดการกับความผิดพลาดที่เกิดระหว่างทำงาน (runtime error) ซึ่งช่วยให้ภาษาเชื่อถือได้มากขึ้น

การใช้ alias

aliasing คือการใช้ชื่อมากกว่าหนึ่งชื่ออ้างถึงหน่วยความจำที่เดียวกัน เช่น การใช้ตัวชี้ (pointer) ชี้ไปที่ตัวแปรซึ่งทำให้สามารถเปลี่ยนค่าตัวแปรโดยอ้างถึงชื่อตัวแปรหรือผ่านตัวชี้ การใช้ aliase เช่นนี้ทำให้โปรแกรมอ่านยากเนื่องจากต้องดูทั้งชื่อตัวแปรและไปตัวชี้เพื่อหาการเปลี่ยนค่าของตัวแปรนั้น นอกจากนั้นยังทำให้เกิดข้อผิดพลาดได้ง่าย คือ ความเชื่อถือได้ลดลง

1.2 แบบความคิดของภาษาโปรแกรม

ภาษาโปรแกรมที่ใช้ในปัจจุบันมีส่วนประกอบพื้นฐานที่เหมือนกัน คือ ตัวแปรซึ่งเป็นที่เก็บค่าและอาจเปลี่ยนค่าที่เก็บในตัวแปรได้ ความแตกต่างที่เป็นหลักใหญ่ระหว่างภาษาโปรแกรมเป็นผลจากแนวความคิดว่าการทำงานของโปรแกรมคืออะไร ดังนั้นส่วนประกอบพื้นฐานในภาษาต้องรองรับแนวคิดของการทำงานเช่นนั้น ในที่นี้เราจะกล่าวถึงแบบความคิดของภาษาโปรแกรม 3 แบบที่แตกต่างกันอย่างมาก คือ การโปรแกรมเชิงคำสั่ง (imperative programming) การโปรแกรมเชิงประกาศ (declarative programming) และการโปรแกรมเชิงฟังก์ชัน (functional programming)

การโปรแกรมเชิงคำสั่ง

สำหรับการโปรแกรมเชิงคำสั่ง การทำงานของโปรแกรมเป็นการจัดลำดับของคำสั่งที่มีการเขียนและอ่านค่าของตัวแปร ดังนั้นส่วนประกอบพื้นฐานของภาษาที่ใช้ในการโปรแกรมเชิงคำสั่งต้องสามารถกำหนดลำดับก่อนหลังของคำสั่งที่ต้องทำงาน ลำดับของการทำงานสามารถขึ้นกับค่าของตัวแปรด้วย ดังนั้นโปรแกรมอาจทำงานต่างกันเมื่อตัวแปรมีค่าต่างกัน สำหรับภาษาโปรแกรมเชิงคำสั่งที่ใช้กันในปัจจุบัน ส่วนประกอบที่ใช้กำหนดลำดับการทำงานของคำสั่ง ได้แก่

- การทำงานเรียงตามลำดับคำสั่งที่เขียนจากบนลงล่าง
- การทำงานแบบทางเลือกโดยใช้โครงสร้างทางเลือก เช่น คำสั่ง if และคำสั่ง switch
- การทำงานแบบวนซ้ำโดยใช้โครงสร้างวนซ้ำ เช่น โครงสร้าง while และ โครงสร้าง for
- การโดดไปทำงานย่อยให้เสร็จแล้วกลับมาทำงานที่ค้างไว้ต่อ เช่น การเรียกใช้โปรแกรมย่อย (subroutine call)

ภาษาโปรแกรมส่วนใหญ่ที่ใช้กันในปัจจุบันใช้การโปรแกรมเชิงคำสั่ง เช่น ภาษาซี (C) ภาษาปาสคาล (Pascal) ภาษาฟอร์แทรน (FORTRAN) ภาษาจาวา (Java) ภาษาไพธอน (Python) เป็นต้น

การโปรแกรมเชิงประกาศ

สำหรับการโปรแกรมเชิงประกาศ การทำงานของโปรแกรมเป็นการหาค่าของตัวแปรที่ทำให้นิยามทั้งหมดที่ประกาศในโปรแกรมเป็นจริง โปรแกรมแบบนี้ไม่ต้องระบุลำดับการทำงานของคำสั่งในโปรแกรม แต่ตัวประมวลภาษาจะหาลำดับการทำงานเพื่อให้ได้คำตอบที่ต้องการจากนิยามในโปรแกรม ภาษาโปรแกรมเชิงประกาศประกอบด้วยสิ่งที่ระบุไว้ก่อนว่าเป็นจริงและกฎที่ระบุว่า สำหรับตัวแปรชุดที่กำหนด ถ้าสมบัติบางอย่างของตัวแปรชุดนี้เป็นจริงแล้วเราสามารถสรุปสมบัติอื่นของตัวแปรบางตัวในชุดนี้ได้ด้วย เช่น สำหรับตัวแปร n ถ้าเศษของการหาร n ด้วย 2 เป็นศูนย์แล้ว เราสรุปได้ว่า n เป็นจำนวนคู่ เราจะยกตัวอย่างภาษาโปรล็อก (Prolog) ที่เป็นภาษาโปรแกรมเชิงประกาศและแสดงตัวอย่างโปรแกรมในภาษานี้

ภาษาโปรล็อกระบุกฎ (Q and R) → P ดังนี้

```
P :- Q, R.
```

และระบุว่า P เป็นจริงดังนี้

```
P.
```

โปรแกรมต่อไปนี้ใช้เพื่อทดสอบว่าจำนวนเต็มที่ระบุเป็นจำนวนคู่หรือไม่

```
even(N) :- remainder(N, 2, 0).
remainder(M, M, 0).
remainder(M, N, M) :- M>=0, M<N.
remainder(M, N, R) :- M>0, M>N, remainder(M-N, N, R).
```

โปรแกรมข้างบนนี้ even(N) เป็นประพจน์ที่เป็นจริงเมื่อ N เป็นจำนวนคู่ ถ้าเราต้องการถามว่า 8 เป็นจำนวนคู่หรือไม่ เราจะถามด้วย

:- even(8).

remainder(M, N, R) เป็นประพจน์ที่ใช้ ประกอบเพื่อตัดสินว่าจำนวนที่ระบุเป็นจำนวนคู่หรือไม่ โดย remainder(M, N, R) เป็นจริงเมื่อ R เป็นเศษของการหาร M ด้วย N ดังนั้นเราสามารถนิยามได้ว่าถ้าเศษของการหาร N ด้วย 2 เป็น 0 แล้ว N เป็นจำนวนคู่ ดังแสดงในบรรทัดที่ 1 ของโปรแกรม บรรทัดที่ 2-4 เป็นนิยามที่ใช้ในการเศษของการหาร บรรทัดที่ 2 ระบุว่าเศษของการหารจำนวนใดใดด้วยตัวมันเองเป็น 0 บรรทัดที่ 3 ระบุว่าเศษของการหาร M ด้วย N เป็น M ถ้า M ไม่เป็นจำนวนลบและ M<N บรรทัดที่ 4 ระบุว่าเศษของการหาร M ด้วย N เท่ากับเศษของการหาร M-N ด้วย N ถ้า M>0 และ M>N โปรแกรมนี้ไม่ได้ระบุลำดับการทำงานเพื่อทดสอบว่า N เป็นจำนวนคู่หรือไม่ แต่บอกว่าจำนวนจำนวนหนึ่งเป็นจำนวนคู่ ถ้า จำนวนนั้นหาร 2 ได้ลงตัว ดังนั้นตัวประมวลภาษาต้องไปหาคำตอบว่าเสร็จของการหารจำนวนที่ระบุเป็น 0 หรือไม่

จากโปรแกรมในตัวอย่างนี้ เมื่อถาม :- even(8). จากกฎในบรรทัดที่ 2 ตัวประมวลภาษาจะต้องหาคำตอบของ remainder(8, 2, 0) มาก่อน ดังนั้นจึงต้องใช้กฎ ในบรรทัดที่ 4 เพื่อหาคำตอบของ remainder(4, 2, 0) และ remainder(2, 2, 0) ตามลำดับ จากนั้นด้วยกฎในบรรทัดที่ 2 จะได้ว่า remainder(2, 2, 0) เป็นจริง ดังนั้นจึงสรุปได้ว่า remainder (4, 2, 0) และ remainder (8, 2, 0) เป็นจริงด้วยตามลำดับสุดท้ายสรุปได้ว่า even(8) เป็นจริงไปด้วย

ทำนองเดียวกัน เมื่อถาม :- even(7). จากกฎในบรรทัดที่ 2 ตัวประมวลภาษาจะต้องหาคำตอบของ remainder(7, 2, 0) มาก่อน ดังนั้นจึงต้องใช้กฎ ในบรรทัดที่ 4 เพื่อหาคำตอบของ remainder(5, 2, 0) และ remainder(3, 2, 0) ตามลำดับ จากนั้นด้วยกฎในบรรทัดที่ 2,3,4 เราสรุปไม่ได้ว่า remainder(1, 2, 0) เป็นจริง ดังนั้นจึงสรุปไม่ได้ว่า remainder (3, 2, 0), remainder (5, 2, 0) และ remainder (7, 2, 0) เป็นจริงด้วยตามลำดับ สุดท้ายจึงสรุปไม่ได้ว่า even(7) เป็นจริงไป นั่นคือ even(7) เป็นเท็จ

จากตัวอย่างนี้เห็นได้ชัดว่าโปรแกรมให้นิยามของจำนวนคู่และเศษของการหาร แต่ไม่ได้กำหนดลำดับการทำงานเพื่อทดสอบจำนวนคู่หรือหาเศษของการหาร ดังนั้นตัวประมวลภาษาจะใช้นิยามที่กำหนดไว้ในโปรแกรมและจัดลำดับการทำงานเอง

การโปรแกรมเชิงฟังก์ชัน

สำหรับการโปรแกรมเชิงฟังก์ชัน โปรแกรมคือฟังก์ชันที่สร้างจากการนำฟังก์ชันมาทำงานด้วยกัน ลำดับการทำงานในโปรแกรมเชิงฟังก์ชันขึ้นกับการประกอบฟังก์ชันเป็นโปรแกรม แนวคิดของโปรแกรมเชิงฟังก์ชันมีพื้นฐานจากฟังก์ชันทางคณิตศาสตร์และมีฟังก์ชันเป็นวัตถุหลัก ดังนั้น ฟังก์ชันเป็นการส่งค่า (map) ในโดเมน (domain) ไปยังค่าในเรนจ์ (range) โดเมนและเรนจ์อาจเป็นเซตของค่าพื้นฐาน (เช่น เซตของจำนวนเต็ม) หรือ เซตของทูเปิล n ค่า (n-tuple) หรือ อาจเป็นเซตของฟังก์ชันก็ได้ ในที่นี้ เราจะยกตัวอย่างการโปรแกรมเชิงฟังก์ชันจากภาษาไพธอน

ในภาษาโปรแกรมเชิงฟังก์ชัน การสร้างฟังก์ชันใหม่ทำได้โดยใช้การประกอบฟังก์ชัน (function composition) ฟังก์ชันเงื่อนไข (condition function) หรือฟังก์ชันให้ทำงานกับทุกตัว (Apply-all function) เป็นต้น

การประกอบฟังก์ชัน f และ g เขียนแทนด้วย f ∘ g(x) หมายถึง f(g(x)) ตัวอย่าง เช่น ถ้าให้ add(x, y) เป็นฟังก์ชันที่ส่งค่าจาก (x, y) ไปยังผลบวกของ x และ y และ mul(x, y) เป็นฟังก์ชันที่ส่งค่าจาก (x, y) ไปยังผลคูณของ x และ y การประกอบฟังก์ชัน add และ mul ดังนี้ add ∘ mul (x, y, z) = add(mul(x, y),z) ได้เป็นฟังก์ชันที่ส่งค่า x, y, z ไปยังผลบวกของผลคูณของ x และ y กับ z การประกอบฟังก์ชัน add ดังนี้ add ∘ add (x, y, z) = add(add(x, y),z) เป็นฟังก์ชันที่ส่งค่า x, y, z ไปยังผลบวกของ x, y และ z การประกอบฟังก์ชันมีใช้ในภาษาโปรแกรมระดับสูงเกือบทุกภาษา เช่น ภาษาซี ภาษาจาวา เป็นต้น

ฟังก์ชันเงื่อนไขมักใช้ชื่อ if หรือ cond โดย if(c, a, b) จะส่งค่า c, a, b ไปยัง a เมื่อ c เป็นจริง และไปยัง b เมื่อ c เป็นเท็จ จากฟังก์ชัน mul ที่กล่าวไปแล้ว if(x>=0, x, mul(x, -1)) เป็นฟังก์ชันที่ให้ค่าสัมบูรณ์ของ x ฟังก์ชันเงื่อนไขคล้ายกับคำสั่ง if ในภาษาเชิงคำสั่ง

ฟังก์ชันให้ทำงานกับทุกตัวมักใช้ชื่อ apply-all หรือ map apply-all(f, x) จะให้ค่าในโครงสร้างเกี่ยวกับ x โดยแต่ละค่าในโครงสร้างเป็น f(a) สำหรับค่า a ใน x ในภาษาไพธอนมีฟังก์ชัน map ที่ทำงานเช่นนี้ด้วย ตัวอย่าง เช่น map(abs, [1, -2, 3, -4, 5]) ให้ตัวทำซ้ำ (iterator) ที่ให้ค่าที่ได้จากการใช้ฟังก์ชัน abs กับค่าแต่ละค่าในลิสต์ คือ 1, 2, 3, 4, 5 และ map(pow, [-1, 2, 3], [2, 1, 2]) ให้ตัวทำซ้ำ (iterator) ที่ให้ค่าที่ได้จากการใช้ฟังก์ชัน pow กับคู่ของค่าจากลิสต์ 2 ลิสต์ คือ $(-1)^2$, 2^1, 3^2

นอกจากนั้น ในภาษาโปรแกรมเชิงฟังก์ชันยังให้สร้างฟังก์ชันและส่งฟังก์ชันไปให้ฟังก์ชันอื่นใช้ได้ด้วย ภาษาไพธอนยอมให้ส่งฟังก์ชันเป็นพารามิเตอร์ให้ฟังก์ชันอื่น ตัวอย่างนี้แสดงฟังก์ชัน integrate รับฟังก์ชัน f และค่า a, b แล้วคำนวณหาผลรวมของ f(x) เมื่อ x มีค่าตั้งแต่ a ถึง b ในตัวอย่างนี้ ถ้าให้ p เป็นฟังก์ชัน $3x^2 + 2x + 6$ และเมื่อเรียก integrate(p, 2, 10) จะได้ผลรวมของ $3x^2 + 2x + 6$ เมื่อ x มีค่าตั้งแต่ 2 ถึง 10

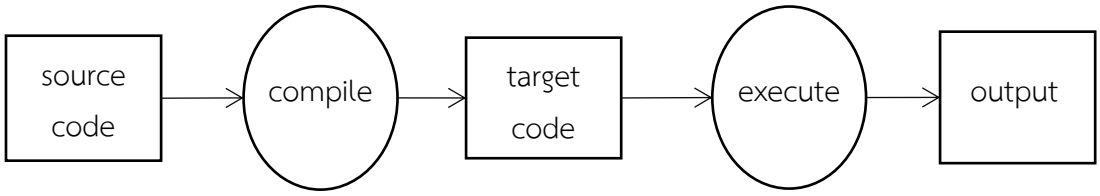
```
def integrate(f, a, b):
    return sum([f(x) for x in range(a,b, 1)])
```

ภาษาโปรแกรมเชิงฟังก์ชันแบบบริสุทธิ์ (pure functional programming language) ไม่มีการเก็บค่าในตัวแปร ค่าถูกส่งจากฟังก์ชันหนึ่งถูกส่งไปให้ฟังก์ชันที่ต้องใช้โดยไม่เก็บในตัวแปรก่อน ข้อดี คือ ทำให้โปรแกรมอ่านง่ายและเข้าใจผิดได้น้อย ข้อเสีย คือ จะไม่สามารถเก็บค่าที่ต้องใช้ในหลายฟังก์ชันในตัวแปรแล้วส่งให้แต่ละฟังก์ชันได้

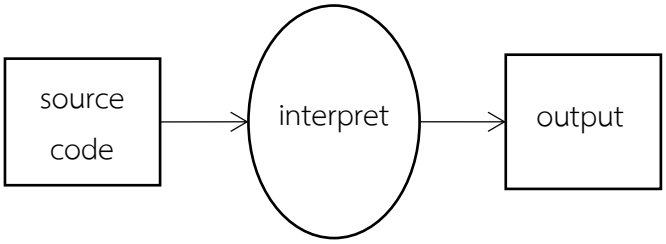
การโปรแกรมในแบบความคิดที่ต่างกันทำให้โปรแกรมเมอร์คิดวิธีแก้ปัญหาต่างกัน การเขียนโปรแกรมในแบบความคิดที่ต่างกันช่วยให้โปรแกรมเมอร์มีมุมของการแก้ปัญหากว้างขึ้นและสามารถเลือกไปประยุกต์ใช้ในสถานการณ์ต่าง ๆ ได้อย่างเหมาะสม

1.3 ขั้นตอนการประมวลผลสำหรับภาษาโปรแกรม

เมื่อโปรแกรมเมอร์เขียนโปรแกรมที่เป็นรหัสเริ่มต้น (source code) แล้ว โปรแกรมนี้เก็บเป็นไฟล์ข้อความ (text file) เมื่อต้องการให้โปรแกรมทำงานในเครื่องคอมพิวเตอร์ โปรแกรมที่เป็นรหัสเริ่มต้นนี้ต้องถูกแปลเป็นคำสั่งที่ใช้สำหรับเครื่องคอมพิวเตอร์นั้นก่อน การแปลโปรแกรมที่ใช้กันมี 2 แบบ คือ แบบคอมไพล์ (compile) และ แบบอินเทอร์พรีท (interpret) ในการคอมไพล์โปรแกรม ข้อความในไฟล์ซึ่งเป็นคำสั่งในภาษาระดับสูงถูกแปลเป็นคำสั่งในภาษาเครื่อง (machine code) ของเครื่องคอมพิวเตอร์ที่จะทำงานแล้วเก็บไว้ในไฟล์ โปรแกรมที่แปลมาแล้วนี้ เรียกว่า รหัสเป้าหมาย (target code) เมื่อต้องการให้โปรแกรมทำงาน เราจึงนำรหัสเป้าหมายไปให้เครื่องคอมพิวเตอร์ทำงาน (execute) ดังแสดงในรูปข้างล่าง ตัวแปลภาษาซี (C compiler) ที่ใช้กันทำงานแบบนี้ สำหรับการแปลโปรแกรมแบบอินเทอร์พรีท คำสั่งในภาษาระดับสูงถูกแปลเป็นคำสั่งในภาษาเครื่องและสั่งให้เครื่องคอมพิวเตอร์ทำงานทีละคำสั่งดังแสดงในรูปข้างล่าง



การแปลแบบคอมไพล์

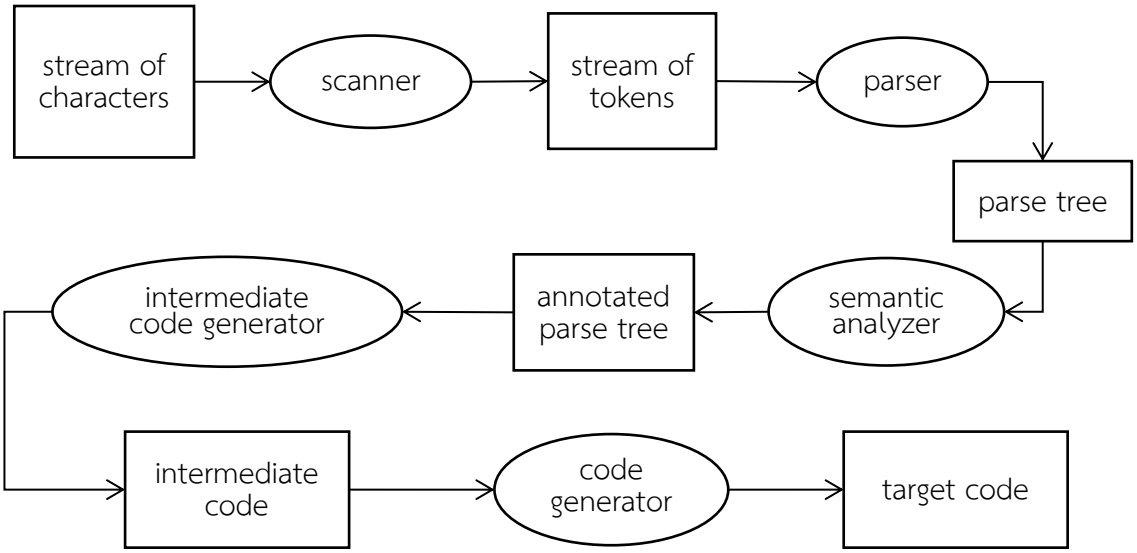


การแปลแบบอินเทอร์พรีท

ในที่นี้ เราจะเน้นการแปลแบบคอมไพล์ ขั้นตอนการคอมไพล์โปรแกรมแบ่งเป็นขั้นตอนย่อย คือ

- การสแกน (scanning)
- การแจงส่วน (parsing)
- การวิเคราะห์ความหมาย (semantic analyzing)
- การสร้างรหัสกลาง (intermediate code generating)
- การสร้างรหัส (code generator)
- การปรับรหัสที่เหมาะสม (code optimization)

ดังแสดงในรูปข้างล่าง



รหัสเริ่มต้นเป็นตัวอักขระเรียงต่อ ๆ กันเก็บในไฟล์ข้อความ ในขั้นตอนการสแกน ตัวอักขระที่เรียงต่อกันนี้ถูกแบ่งเป็นชิ้นส่วนพื้นฐานที่มีความหมายในภาษาโปรแกรม เช่น ชื่อ (identifier) จำนวนเต็ม (integer) สายอักขระ (string) สัญลักษณ์การเปรียบเทียบ >= เป็นต้น ชิ้นส่วนเหล่านี้เรียกว่าโทเคน (token) และสตริงที่ตรงกับรูปแบบที่กำหนดสำหรับแต่ละโทเคน เรียกว่า หน่วยศัพท์ (lexeme) ตัวอย่าง เช่น สำหรับรหัสเริ่มต้น

```
if (x==0) x=x+1; else x=0;
```

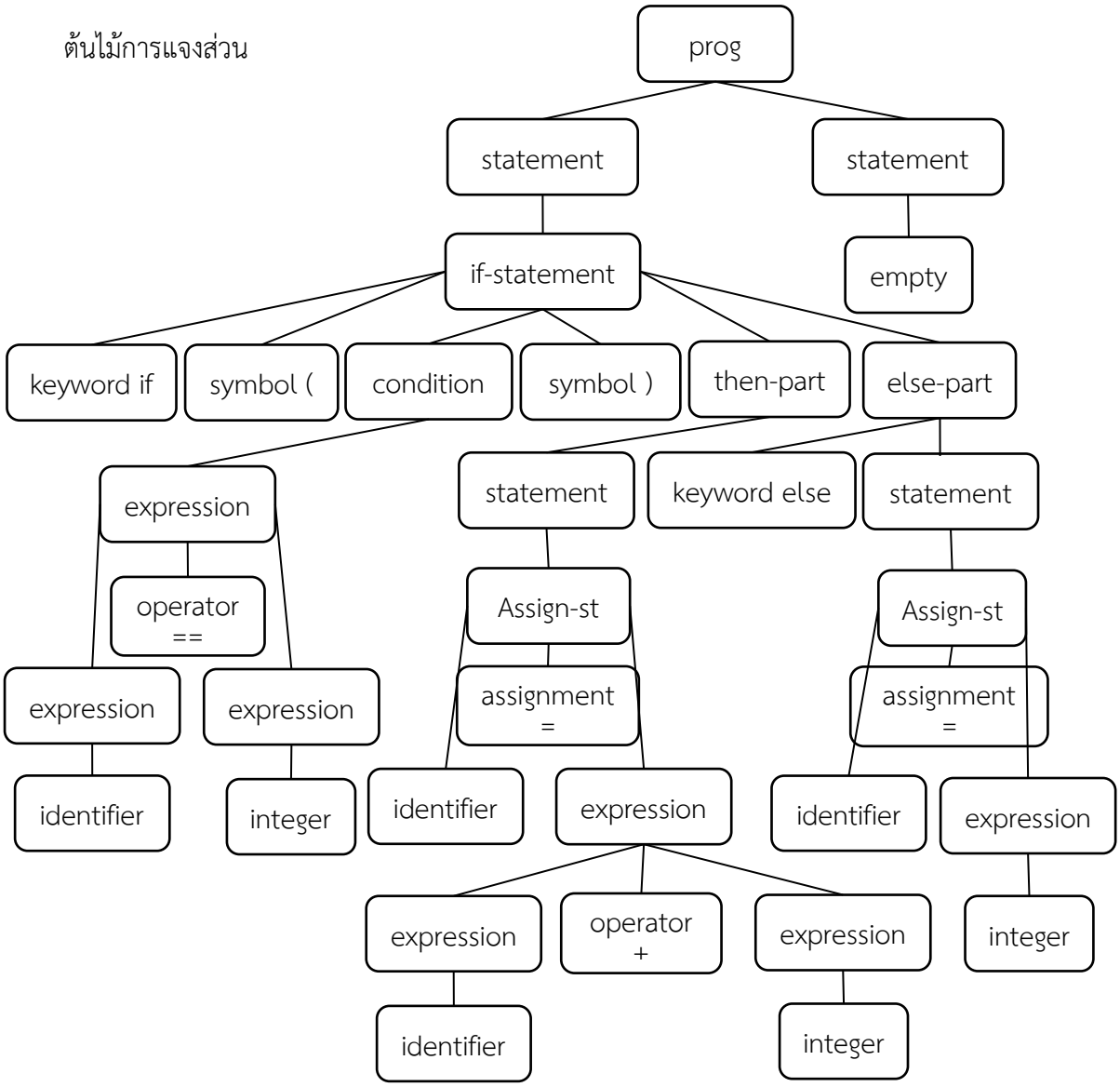
สแกนเนอร์แบ่งเป็นโทเคนดังนี้

keyword if	symbol (	identifier x	operator ==	integer 0
symbol)	identifier x	assignment =	identifier x	operator +
integer 1	symbol ;	keyword else	identifier x	assignment =
integer 0	symbol ;			

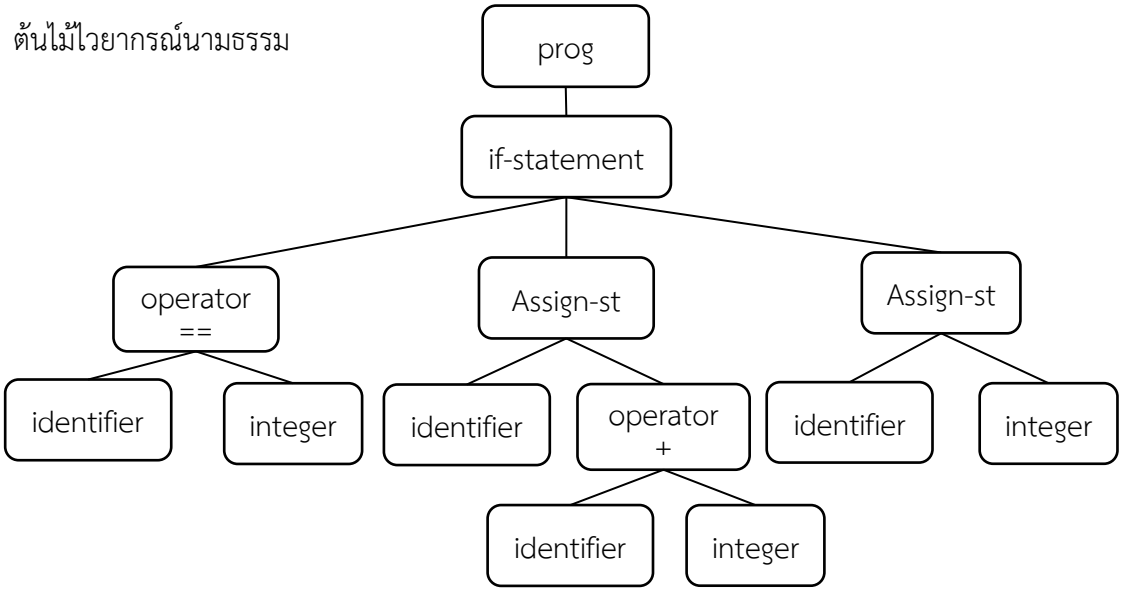
รูปแบบสำหรับแต่ละโทเคนถูกอธิบายด้วยนิพจน์ปกติซึ่งจะอธิบายต่อไปในบทที่ 2 เมื่อสแกนเนอร์ทำงานเสร็จแล้วจะได้โทเคนซึ่งบอกชนิดของชิ้นส่วนที่อยู่ในโปรแกรม

ในขั้นตอนการแจงส่วน โทเคนที่ได้จากสแกนเนอร์จะถูกจัดเป็นกลุ่มตามโครงสร้างที่ใหญ่ขึ้นของภาษาโปรแกรม เช่น นิพจน์ คำสั่งทางเลือก โครงสร้างวนซ้ำแบบ for บล็อกของโปรแกรม หรือ ฟังก์ชัน โครงสร้างของภาษาในระดับนี้อธิบายด้วยไวยากรณ์ไม่พึ่งบริบท (context-free grammar) ซึ่งจะอธิบายต่อไปในบทที่ 3 โครงสร้างนี้ถูกเก็บในรูปต้นไม้ที่เรียกว่า ต้นไม้การแจงส่วน (parse tree) หรือ ต้นไม้ไวยากรณ์นามธรรม (abstract syntax tree) ต้นไม้นี้แสดงการประกอบกันของโทเคนเป็นโครงสร้างที่ใหญ่ขึ้นตามลำดับตามไวยากรณ์ของภาษาโปรแกรมนั้น ข้างล่างนี้เป็นต้นไม้การแจงส่วนและต้นไม้ไวยากรณ์นามธรรมของโทเคนในตัวอย่างข้างต้น

ต้นไม้การแจงส่วน



ต้นไม้ไวยากรณ์นามธรรม



ขั้นตอนการวิเคราะห์ความหมายเป็นการตรวจสอบชนิดของข้อมูลที่ใช้ในโปรแกรมให้ตรงกับชนิดข้อมูลที่ระบุไว้ในภาษา เช่น ใน x = y + 2 ถ้าตัวแปร x เป็นชนิดจำนวนเต็ม แล้ว ตัวแปร y ควรเป็นชนิดจำนวนเต็มหรือจำนวนจริง หาก y เป็นจำนวนจริง ตัวแปลภาษาจะต้องใส่ตัวกระทำการแปลงจำนวนเต็มให้เป็นจำนวนจริงเพิ่มในต้นไม้ที่เรียกว่าต้นไม้แจงส่วนประกอบความหมาย (annotated parse tree) แล้วจึงเลือกว่าใช้ตัวกระทำการบวกสำหรับจำนวนจริง ขั้นตอนการวิเคราะห์ความหมายอธิบายในบทที่ 5

ต้นไม้การแจงส่วนบอกลำดับการทำงานของโปรแกรม ดังนั้นจึงสามารถสร้างโปรแกรมจากต้นไม้การแจงส่วนเพื่อให้ทำงานตามลำดับนั้นได้ โปรแกรมที่สร้างขึ้นอาจอยู่ในภาษาเครื่องหรือภาษาอื่นได้ ในการสร้างตัวแปลภาษาอาจใช้รหัสกลาง (intermediate code) ซึ่งภาษาที่อยู่ระหว่างภาษาระดับสูงและภาษาเครื่อง ในขั้นตอนการสร้างรหัสกลาง ต้นไม้การแจงส่วนจะถูกแปลงเป็นโปรแกรมในรหัสกลาง บทที่ 6 แสดงตัวอย่างของรหัสกลางรวมถึงขั้นตอนการสร้างรหัสกลาง

การสร้างรหัสกลางช่วยอำนวยความสะดวกเมื่อต้องการแปลโปรแกรมจากภาษาระดับสูงหลายภาษาไปยังภาษาเครื่องหลายภาษา ภาษาจาวาและภาษาไพธอนมีไบท์โคด (byte code) ที่ทำหน้าที่เป็นรหัสกลาง เครื่องเสมือน (virtual machine) อ่านรหัสกลางและทำงานตามคำสั่งในรหัสกลาง สำหรับภาษาอื่น เช่น ภาษาซี ตัวแปลภาษาอาจแปลโปรแกรมเป็นรหัสกลางแล้วจึงแปลเป็นภาษาเครื่องต่อไปในขั้นตอนการสร้างรหัส

การปรับรหัสให้เหมาะสม (code optimization) เป็นขั้นตอนที่ปรับปรุงโปรแกรมให้ทำงานได้ประสิทธิภาพดีขึ้น การปรับปรุงนี้อาจทำได้กับโปรแกรมในรหัสกลางและโปรแกรมในภาษาเครื่องโดยวิธีการปรับรหัสให้เหมาะสมขึ้นอยู่กับภาษาที่ใช้ ดังนั้นขั้นตอนนี้อาจทำได้ทั้งหลังขั้นตอนการสร้างรหัสกลางและหลังขั้นตอนการสร้างรหัส อย่างไรก็ตาม ตัวแปลภาษาอาจมีขั้นตอนนี้หรือไม่ก็ได้ นั่นคือตัวแปลภาษาแบบคอมไพล์อย่างน้อยต้องมีขั้นตอนการสแกน การแจงส่วน การวิเคราะห์ความหมาย และ การสร้างรหัส

2. สแกนเนอร์

สแกนเนอร์เป็นส่วนแรกของตัวแปลภาษา (compiler) ที่รับสายของอักขระซึ่งเป็นโปรแกรมที่ต้องการแปลมาแบ่งเป็นชิ้นส่วนพื้นฐานของภาษา เช่นจำนวนเต็ม จำนวนจริง สตริง ชื่อของตัวแปรหรือฟังก์ชัน เครื่องหมายต่างๆ (+ - * /) คำหลัก (keyword) เป็นต้น ชิ้นส่วนเหล่านี้เรียกว่าโทเคน (token) ซึ่งเป็นส่วนประกอบที่มีความหมายที่เล็กสุดในภาษา คล้ายกับคำในภาษาธรรมชาติ สแกนเนอร์ต้องระบุชนิดของโทเคนแต่ละตัว เพื่อส่งให้ตัวแจงส่วนตรวจโครงสร้างทางไวยากรณ์ นอกจากนั้นสแกนเนอร์จะเก็บโทเคนบางตัว เช่น ชื่อตัวแปรหรือฟังก์ชัน และ ค่าคงที่ไว้ในตารางที่เรียกว่า ตารางสัญลักษณ์ (symbol table) เพื่อให้ตัวแปลภาษาใช้ในการทำงานขั้นต่อ ๆ ไป

ในบทนี้เราจะอธิบายแนวคิดในภาษาโปรแกรมที่เกี่ยวข้องกับการสร้างสแกนเนอร์ ในหัวข้อ 2.1 แนวคิดเหล่านี้ ได้แก่ การกำหนดรูปแบบของโทเคนแต่ละชนิด การกำหนดให้คำหลัก (keyword) เป็นหรือไม่เป็น คำสงวน (reserved word) ในภาษาหรือไม่ จากนั้น หัวข้อ 2.2 อธิบายการใช้นิพจน์ปกติ (regular expression) ระบุรูปแบบของโทเคนที่ใช้ในภาษา เพื่อสื่อสารระหว่างผู้พัฒนาคอมไพเลอร์ด้วยกันและกับผู้ใช้ภาษา จากนั้นหัวข้อ 2.3 อธิบายการสร้างออโตมาตาจำกัด (finite automata) เพื่อตรวจสอบลำดับของตัวอักขระที่ตรงตามรูปแบบที่นิพจน์ปกติกำหนด เราใช้ออโตมาตาจำกัดเป็นตัวแบบสำหรับอธิบายการทำงานของสแกนเนอร์เนื่องจากเป็นตัวแบบที่เข้าใจง่ายและจำลองการทำงานด้วยโปรแกรมได้สะดวก หัวข้อ 2.4 อธิบายประเด็นต่างๆในการเขียนโปรแกรมสแกนเนอร์จากออโตมาตาจำกัด

2.1 แนวคิดในภาษาโปรแกรมที่เกี่ยวข้องกับสแกนเนอร์

การเลือกรูปแบบของโทเคนที่ใช้ในภาษาโปรแกรมให้เหมาะสมทำให้ภาษาอ่านง่าย ในหัวข้อนี้ เราจะพิจารณารูปแบบของโทเคนที่ใช้ในภาษาโปรแกรมโดยทั่วไป

2.1.1 รูปแบบของโทเคน

ภาษาโปรแกรมส่วนใหญ่มีโทเคนดังต่อไปนี้

- จำนวนเต็ม
- จำนวนจริง
- สตริง
- ชื่อ (identifier) เช่น ชื่อตัวแปร ชื่อฟังก์ชัน
- ตัวกระทำ (operator) เช่น + - * / >= <= !=
- เครื่องหมายแบ่งวรรคตอน เช่น , ; : ( ) [ ] { }

รูปแบบของสตริงสำหรับโทเคนในภาษาโปรแกรมควรใกล้เคียงกับสิ่งที่โทเคนนั้นอ้างถึงในภาษาธรรมชาติ เพื่อให้โปรแกรมเมอร์อ่านโปรแกรมเข้าใจได้ง่าย แต่เนื่องจากสแกนเนอร์ต้องแยกชนิดของโทเคนตามรูปแบบ

ของสตริงสำหรับโทเคนนั้น ภาษาโปรแกรมต้องกำหนดรูปแบบของสตริงสำหรับโทเคนให้แยกโทเคนแต่ละชนิดออกจากกันได้ด้วย ภาษาโปรแกรมที่ใช้กันในปัจจุบันกำหนดรูปแบบของโทเคนคล้าย ๆ กัน ดังนี้

จำนวนเต็ม

รูปแบบของโทเคนชนิดจำนวนเต็มเลียนแบบมาจากจำนวนเต็มที่ใช้กันในภาษาธรรมชาติ เช่น 365, 0, -123 เป็นต้น นั่นคือสตริงสำหรับโทเคนจำนวนเต็มเป็นตัวเลข 0-9 อย่างน้อยหนึ่งตัวต่อกันและอาจมีเครื่องหมาย + หรือ - อยู่ข้างหน้า เช่น -32 +7 0 546

จำนวนจริง

รูปแบบของโทเคนที่แทนจำนวนจริงมี 2 แบบคือ แบบที่ไม่มีเลขยกกำลัง ซึ่งเลียนแบบจำนวนจริงที่ใช้กันโดยทั่วไป เช่น 123, -564 และ แบบที่มีเลขยกกำลัง ซึ่งคล้ายกับจำนวนจริงที่ใช้ทางวิทยาศาสตร์ เช่น 3.1212×10^4 รูปแบบแรกประกอบด้วยตัวเลข 0-9 ต่อกันกี่ตัวก็ได้ ตามด้วยจุดทศนิยม และตัวเลข 0-9 ต่อกันอีกกี่ตัวก็ได้ โดยอาจมีแต่เลขหน้าจุดทศนิยมหรือเลขหลังจุดทศนิยมอย่างใดอย่างหนึ่งก็ได้ และมีเครื่องหมายบวกหรือลบอยู่หน้าสุดได้ เช่น -3.14159 +0.6 .543 365. รูปแบบที่ 2 เป็นการนำสตริงในรูปแบบแรกมาต่อด้วยเลขยกกำลังที่ระบุด้วย e หรือ E แล้วตามด้วยสตริงที่แทนโทเคนจำนวนเต็ม เช่น 1.3223e-5 0.9e0 -12E4

สตริง

รูปแบบของโทเคนที่แทนสตริงจะคล้ายกับการแทนคำพูดในภาษาธรรมชาติ คือ จะใช้เครื่องหมาย "... " ครอบตัวอักขระที่เรียงต่อกัน สังเกตว่าการใช้เครื่องหมาย "..." ครอบสตริงนี้ทำให้แยกระหว่างสตริงและชื่อได้

นอกจากนั้นยังทำให้มีช่องว่างและเครื่องหมายพิเศษในสตริงได้โดยไม่กำกวม แต่การมี " ในสตริง จะทำให้สแกนเนอร์ตีความผิดได้ ดังนั้นถ้าต้องการใช้เครื่องหมาย " ในสตริงจะให้ใช้ \" เพื่อระบุว่าเป็นตัวอักขระในสตริง แต่ไม่ใช่เครื่องหมายปิด

อย่างไรก็ตามบางภาษา เช่น ภาษาไพธอน หลีกเลี่ยงการใช้ \" โดยอนุญาตให้เลือกใช้ " ... " หรือ ' ... ' หรือ """ ... """ ครอบสตริง ดังนั้นถ้าต้องการใช้ " ในสตริงก็ให้เลือกครอบสตริงนั้นด้วย ' ... ' เป็นต้น

ชื่อ (identifier)

ชื่อที่ใช้ในภาษาโปรแกรมอาจเป็นชื่อตัวแปรหรือชื่อฟังก์ชัน รูปแบบของโทเคนสำหรับชื่อ เลียนแบบการตั้งชื่อตัวแปรทางคณิตศาสตร์ โดยให้ใช้ตัวอักษร A-Z a-z และตัวเลข 0-9 ในชื่อได้ แต่ในภาษาโปรแกรมไม่สามารถใช้สัญลักษณ์เลียนแบบการทำตัวห้อยในทางคณิตศาสตร์ได้ จึงอนุญาตให้ใช้สัญลักษณ์ _ ได้ด้วย แต่ชื่อต้องขึ้นต้นด้วยตัวอักษร A-Z a-z และสัญลักษณ์ _ เท่านั้น

สังเกตว่าชื่อเหล่านี้ไม่สามารถมีเครื่องหมายพิเศษ เช่น + - * / " เพื่อให้สามารถบอกได้แน่นอนว่าชื่อนั้น จบเมื่อเจอช่องว่างหรือเครื่องหมายพิเศษ เหล่านี้ เช่น เมื่อสแกนเนอร์อ่านนิพจน์ grade_380+0.5 แล้วจะสามารถบอกได้ว่ามีตัวแปรชื่อ grade_380 เครื่องหมายบวกและค่าคงที่จำนวนจริง 0.5

สัญลักษณ์พิเศษ

สัญลักษณ์พิเศษส่วนใหญ่ที่ใช้ในภาษาโปรแกรม เหมือนกับสัญลักษณ์ที่ใช้ในภาษาธรรมชาติ ยกเว้นแต่บางสัญลักษณ์ที่ไม่มีในแป้นพิมพ์คอมพิวเตอร์ เช่น เครื่องหมาย ≤ ≥ ≠ ดังนั้นภาษาโปรแกรมส่วนใหญ่จึงใช้คู่ของตัวอักขระแทนเครื่องหมาย เหล่านี้ เช่น <= >= !=

ตัวกระทำการคูณในภาษาโปรแกรมให้ใช้เครื่องหมาย * และไม่อนุญาตให้ใช้การเขียนตัวแปรติดกันหรือเขียนค่าคงที่ติดกับตัวแปรแทนการคูณ เช่น xy เนื่องจาก xy อาจหมายถึงชื่อตัวแปรหนึ่งตัวได้ด้วยดังนั้นเพื่อให้ตีความหมายได้ทางเดียวจึงใช้เครื่องหมาย * แทนการคูณในทุกกรณี

2.1.2 คำสงวนหรือคำหลัก

คำหลักเป็นคำที่มีความหมายพิเศษในภาษาโปรแกรม เช่น if else int float while ภาษาโปรแกรมบางภาษาไม่อนุญาตให้ใช้ชื่อตัวแปรซ้ำกับคำหลักเหล่านี้ จึงเรียกคำเหล่านี้ว่าเป็นคำสงวน (reserved words) แต่บางภาษาอาจยอมให้ใช้คำเหล่านี้เป็นชื่อได้และเรียกคำเหล่านี้ว่าเป็นคำหลัก (keywords) หากยอมให้ใช้คำเหล่านี้เป็นชื่อได้อาจทำให้สับสนเมื่ออ่านโปรแกรม เช่น

x = if(y=0)

ในที่นี้ if เป็นชื่อฟังก์ชัน และเราเก็บค่า 0 ในตัวแปรy และส่งให้ฟังก์ชัน if ซึ่งอาจทำให้ผู้อ่านสับสนว่าเป็นคำสั่ง if และต้องอ่านคำสั่งถัดไปก่อนจึงบอกได้ว่าเป็นคำสั่ง if หรือไม่

เนื่องจากคำสงวนมีรูปแบบเหมือนชื่อ สแกนเนอร์ต้องตรวจสอบชื่อที่อ่านมาได้ก่อนว่าเป็นคำสงวนหรือไม่ เราจะกล่าวถึงการที่สแกนเนอร์จัดการคำสงวนในหัวข้อถัดไป

2.1.3 รูปแบบของสตริง

ภาษาโปรแกรมหลายภาษา กำหนดให้ใช้เครื่องหมาย "..." ครอบสตริง เช่น "Programming Languages" แต่เมื่อต้องการให้มีสัญลักษณ์ " ในสตริงด้วย เช่น "Did I say "yes" ? " สแกนเนอร์ต้องแยกระหว่างสัญลักษณ์ " ในสตริงและสัญลักษณ์ " ที่ปิดสตริงได้ ดังนั้นภาษาโปรแกรมหลายภาษาจึงใช้สัญลักษณ์ \ นำหน้าสัญลักษณ์ " ที่เรียกว่าเป็น escape character เช่น "Did I say \"yes\" ? " ดังนั้นสแกนเนอร์ต้องตรวจสอบว่าไม่มีสัญลักษณ์ \ ก่อนหน้าสัญลักษณ์ " หรือไม่เพื่อที่จะบอกได้ว่าเป็นสัญลักษณ์ปิดท้ายสตริง

อย่างไรก็ตามภาษาไพธอนหลีกเลี่ยงการใช้สัญลักษณ์ \ นำหน้าสัญลักษณ์ " โดย ให้ใช้เครื่องหมาย '...' หรือ "..." หรือ """..""" ครอบสตริง ดังนั้น หากโปรแกรมเมอร์ต้องการใช้สัญลักษณ์ใดในสตริงก็สามารถเลือกสัญลักษณ์อีกตัวหนึ่งครอบสตริง เช่น 'I said "no".', "It's me." """I said "It's me"."""" ดังนั้นสแกนเนอร์ต้องสามารถตรวจสอบคู่สัญลักษณ์เปิด-ปิดที่ตรงกันได้ การอนุญาตให้ใช้สัญลักษณ์เปิด-ปิดสตริงได้หลายแบบทำให้ไม่จำเป็นต้องใช้ escape character แต่การใช้สัญลักษณ์ " สามตัวติดกันอาจทำให้เกิดข้อผิดพลาดได้

2.1.4 การใช้ตัวอักษรตัวใหญ่/ตัวเล็ก

ภาษาโปรแกรมส่วนใหญ่กำหนดให้ตัวอักขระตัวใหญ่และตัวเล็ก เช่น A และ a ต่างกัน ดังนั้นตัวแปร x และ ตัวแปร X ไม่ถือเป็นตัวแปรเดียวกัน ซึ่งตัวแปลภาษาสามารถตรวจสอบความแตกต่างได้ง่าย แต่โปรแกรมเมอร์อาจพิมพ์ผิดและไม่สังเกตเห็นความแตกต่าง ภาษาโปรแกรมบางภาษา เช่น FORTRAN, Cobol ถือว่าตัวอักษรตัวใหญ่และตัวเล็กไม่ต่างกัน

2.2 การอธิบายรูปแบบของโทเคนด้วยนิพจน์ปกติ

เมื่อผู้ออกแบบภาษากำหนดรูปแบบของโทเคนที่ใช้ในภาษาแล้วจะอธิบายรูปแบบของโทเคนเหล่านั้นด้วยนิพจน์ปกติ ข้อดีของนิพจน์ปกติ คือ มีความหมายชัดเจน และ ทำให้ผู้ออกแบบภาษาผู้สร้างคอมไพเลอร์และผู้ใช้ภาษาเข้าใจได้ตรงกัน นอกจากนั้นยังนำไปแปลงเป็นออโตมาตาจำกัดเพื่อใช้สร้างสแกนเนอร์ต่อไปได้ ดังจะอธิบายในหัวข้อ 2.3

รูปแบบของโทเคนที่ใช้ในภาษาโปรแกรมมักเป็นรูปแบบง่าย ๆ ที่ทำให้แยกชนิดของโทเคนได้ดังที่ได้กล่าวมาแล้ว นิพจน์ปกติเป็นนิพจน์ที่สร้างจากตัวกระทำสำหรับสตริง คือการต่อกัน (concatenation) การซ้ำ และการรวม (union) ดังนั้นเราจะนิยามนิพจน์ปกติและแสดงตัวอย่างการนำนิพจน์ปกติมากำหนดโทเคนในภาษาโปรแกรม เรามักใช้นิพจน์ปกติในการอธิบายรูปแบบของโทเคนในภาษาโปรแกรม จากนั้นจึงแปลงนิพจน์ปกตินี้เป็นออโตมาตาจำกัดซึ่งเป็นตัวแบบที่อธิบายการทำงานเพื่อตรวจสอบรูปแบบที่กำหนดด้วยนิพจน์ปกติ จากนั้นจะสร้างสแกนเนอร์จากออโตมาตาจำกัด สาเหตุที่ใช้ออโตมาตาจำกัดเป็นเพราะสามารถนำมาเขียนโปรแกรมเลียนแบบการทำงานได้ง่าย

นิพจน์ปกติเป็นนิพจน์ที่ใช้สร้างภาษาซึ่งเป็นเซตของสตริงที่สร้างจากสัญลักษณ์ที่กำหนดสำหรับภาษานั้น นิพจน์ปกติสร้างจากสัญลักษณ์เหล่านี้และตัวกระทำพื้นฐาน ได้แก่ การต่อกัน (concatenation) การซ้ำ และ การรวม (Union) นิพจน์ปกติที่ใช้ได้สำหรับภาษาทุกภาษา ได้แก่ นิพจน์ e ที่ใช้แทนเซตของสตริงว่าง และนิพจน์ ∅ ที่ใช้แทนเซตว่าง นิพจน์ปกติที่เล็กสุดสำหรับเซตของสัญลักษณ์ชุดหนึ่ง ได้แก่ สัญลักษณ์แต่ละตัวในภาษาซึ่งใช้แทนเซตของสตริงที่มีสัญลักษณ์นั้นเพียงตัวเดียว ถ้าให้เซตของสัญลักษณ์ในภาษาเป็น {0, 1} นิพจน์ปกติ 0 แทนเซตของสตริง 0 และนิพจน์ปกติ 1 แทนเซตของสตริง 1

เราสามารถนำนิพจน์ปกติใด ๆ มาสร้างเป็นนิพจน์ปกติใหม่โดยใช้ตัวกระทำสำหรับนิพจน์ปกติ ในที่นี้เราให้ตัวเข้มในนิพจน์ปกติแสดงสัญลักษณ์ในภาษา

- การต่อกันแสดงด้วยการ เขียนนิพจน์ต่อกัน หรือ ใช้สัญลักษณ์ · แทนตัวกระทำการต่อกัน เช่น **01** หรือ **0·1** หมายถึง {0, 1}
- ตัวกระทำการซ้ำใช้สัญลักษณ์ * เช่น **0*** หมายถึง {e, 0, 00, 000, ...}
- ตัวกระทำการรวมใช้สัญลักษณ์ + เช่น **0+1** หมายถึง {0, 1}

เราสามารถใช้ตัวกระทำเหล่านี้สร้างนิพจน์ปกติที่ซับซ้อนมากขึ้นได้ดังตัวอย่างต่อไปนี้

ตัวอย่าง 2.1

0·**0***=**00*** = {0, 00, 000, 0000, ...} = {w | w เป็นสตริงที่ประกอบด้วยศูนย์อย่างน้อยหนึ่งตัวและไม่มีสัญลักษณ์อื่น}

0 (**0**+**1**)* **0** = {00, 000, 010, 0000, 0010, 0100, 0110, ...} = {w| w เป็นสตริงที่มีความยาวอย่างน้อย 2 และ ขึ้นต้นและลงท้ายด้วยศูนย์}

0 + (**0**(**0**+**1**)* **0**) = {0, 00, 000, 010, 0000, 0010, 0100, 0110, ...} = {w| w เป็นสตริงที่มีความยาวอย่างน้อย 1 และขึ้นต้นและลงท้ายด้วยศูนย์}

(**0**+**1**)***111**(**0**+**1**)* = {111, 0111, 1111, 1110, 01110, 01111, 11110, 11111, ...} = {w | 111 เป็นสตริงย่อยใน w }

1*((**0**+**00**)**11***)* (**0**+**00**+e) = {e, 0, 1, 00, 01, 10, 11, 001, 010, 011, 100, 101, 110, 111, 0010, 0011, 0100, 0101, 0110, 0111, 1001, 1010, 1011, 1100, 1101, 1110, 1111, ... } = {w| ไม่มี 0 ติดกันมากกว่า 2 ตัว}

นอกจากนั้นยังมีตัวกระทำที่ใช้เขียนนิพจน์ปกติให้สั้นลง เช่น

r^n หมายถึง r ต่อกัน n ครั้งเมื่อ r เป็นนิพจน์ปกติ

r^+ หมายถึง r*r เมื่อ r เป็นนิพจน์ปกติ

r? หมายถึง r หรือสตริงว่าง เมื่อ r เป็นนิพจน์ปกติ นั่นคืออาจมีสตริงตามนิพจน์ปกติ r หนึ่งครั้งหรือไม่มีเลยก็ได้

~r หมายถึง สัญลักษณ์ใด ๆ ที่ไม่ใช่ r เมื่อ r เป็นนิพจน์ปกติ

[**a**-**z**] หมายถึง **a**+**b**+...+**z** เมื่อ a, b, ..., z เป็นสัญลักษณ์ในภาษา

[**a**-**z**A-**Z**] หมายถึง **a**+**b**+...+**z**+**A**+**B**+...+**Z** เมื่อ a, b, ..., z และ A, B, ..., Z เป็นสัญลักษณ์ในภาษา

ตัวอย่าง 2.2

[**0**-**9**]$^+$ หมายถึงเซตของสตริงที่ประกอบด้วยตัวเลข 0 ถึง 9 ต่อกันอย่างน้อยหนึ่งตัว

(**+**|**-**)?[**0**-**9**]$^+$ หมายถึงเซตของสตริงที่ประกอบด้วยตัวเลข 0 ถึง 9 ต่อกันอย่างน้อยหนึ่งตัว และอาจนำหน้าด้วยเครื่องหมาย + หรือ - หรือไม่มีเครื่องหมายนำหน้า

[**0**-**9**]^{10} หมายถึงเซตของสตริงที่ประกอบด้วยตัวเลข 0 ถึง 9 ต่อกัน 10 ตัว

59[**0**-**9**]^8 หมายถึงเซตของสตริงที่ประกอบด้วยตัวเลขต่อกัน 10 ตัว โดยที่ตัวเลขสองตัวแรกเป็น 59

โทเคนในภาษาโปรแกรมส่วนใหญ่อธิบายได้ด้วยนิพจน์ปกติดังแสดงในตัวอย่างนี้

ตัวอย่าง 2.3

จำนวนเต็มอธิบายได้ด้วยนิพจน์ปกติ (+|-)?[0-9]$^+$

จำนวนจริงอธิบายได้ด้วยนิพจน์ปกติ (+|-)? [0-9]$^+$. [0-9]* ((e|E) (+|-)?[0-9]$^+$)?

สตริงอธิบายได้ด้วยนิพจน์ปกติ " (~")* "

ชื่ออธิบายได้ด้วยนิพจน์ปกติ [A-Za-z_][A-Za-z0-9_]*

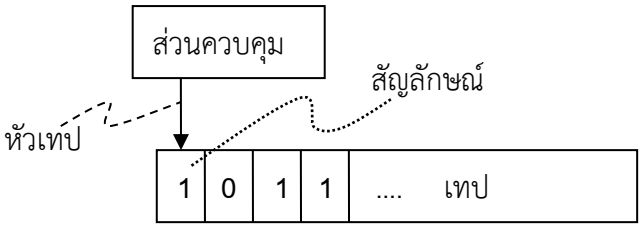
สแกนเนอร์ยังต้องตรวจหารูปแบบหนึ่งที่ไม่ใช้ในโปรแกรม คือ คอมเมนท์ (comment) คอมเมนท์ในภาษาซีอธิบายได้ด้วยนิพจน์ปกติ **/*** (~(***/**))* ***/** เมื่อสแกนเนอร์พบคอมเมนท์จะอ่านข้ามไปและไม่นำมารวมในโปรแกรม

2.3 การสร้างออโตมาตาจำกัดสำหรับสแกนเนอร์

เมื่อเราจะสร้างสแกนเนอร์เพื่อหาโทเคนที่มีรูปแบบตามกำหนดในนิพจน์ปกติดังที่กล่าวไปในหัวข้อที่แล้ว เราจะแปลงนิพจน์ปกติให้เป็นออโตมาตาจำกัดแล้วจึงนำไปสร้างสแกนเนอร์

2.3.1 ออโตมาตาจำกัด

ออโตมาตาจำกัดเป็นตัวแบบ (model) ที่ใช้ตรวจสอบรูปแบบของสตริง ออโตมาตาจำกัดประกอบด้วยส่วนควบคุม, เทปสำหรับรับข้อมูลเข้า, และ หัวเทปดังแสดงในรูปข้างล่าง



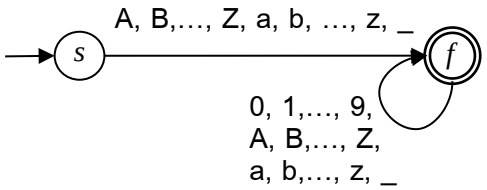
จากส่วนประกอบนี้ ออโตมาตาจำกัดอธิบายด้วย 5-สิ่งอันดับ (Q, Σ, δ, s, F) ส่วนควบคุมมีสถานะเป็นจำนวนจำกัด ซึ่ง Q เป็นเซตของสถานะ (state) ที่เป็นไปได้ของส่วนควบคุม เมื่อเริ่มทำงาน ออโตมาตาจำกัดอยู่ที่สถานะ s ที่เรียกว่า สถานะเริ่มต้น (start state) ส่วนควบคุมจะอยู่ในสถานะเดียว ณ เวลาหนึ่ง ไม่สามารถอยู่ในสองสถานะในเวลาเดียวกัน และ ไม่สามารถไม่อยู่ในสถานะใดเลย ณ เวลาหนึ่ง เทปแบ่งเป็นช่อง ๆ แต่ละช่องมีสัญลักษณ์อยู่ได้หนึ่งตัว Σ เป็นเซตของสัญลักษณ์ซึ่งอยู่บนเทป เมื่อเครื่องทำงาน ส่วนควบคุมจะเปลี่ยนสถานะตามที่ระบุในฟังก์ชันเปลี่ยนสถานะ (transition function) δ ซึ่งระบุการเปลี่ยนสถานะที่ขึ้นกับสถานะปัจจุบันและสัญลักษณ์ที่อ่านจากเทป เครื่องสามารถอ่านสัญลักษณ์ในช่องหนึ่งของเทปได้เมื่อหัวเทปอยู่ที่ช่องนั้น หัวเทปเลื่อนไปทางขวาในแต่ละครั้งที่อ่านสัญลักษณ์

เครื่องหยุดทำงานเมื่ออ่านสัญลักษณ์ตัวขวาสุดบนเทปแล้วเปลี่ยนสถานะ หากเครื่องหยุดทำงานที่สถานะใน F ที่เรียกว่าสถานะสิ้นสุด (final state) เราถือว่าเครื่องยอมรับสตริงบนเทป แต่ถ้าเครื่องจบการทำงานแล้วอยู่ในสถานะที่ไม่ใช่สถานะสิ้นสุด แล้วเครื่องจะไม่ยอมรับสตริงนั้น

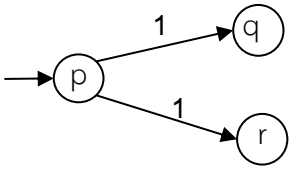
ออโตมาตาจำกัดมี 2 แบบ คือ ออโตมาตาจำกัดเชิงกำหนด (deterministic finite automata) และออโตมาตาจำกัดเชิงไม่กำหนด (non-deterministic finite automata) เราเรียกออโตมาตาจำกัดเชิงกำหนดและออโตมาตาจำกัดเชิงไม่กำหนดอย่างสั้นๆ ว่า DFA และ NFA ตามลำดับ

ออโตมาตาจำกัดเชิงกำหนดเป็นออโตมาตาที่มีการเปลี่ยนสถานะเชิงกำหนด คือ เมื่อเครื่องอยู่ในสถานะหนึ่งและอ่านได้สัญลักษณ์หนึ่ง เครื่องสามารถเปลี่ยนสถานะไปยังสถานะถัดไปได้เพียงสถานะเดียว เช่น ถ้าออโตมาตาจำกัดเชิงกำหนด M เปลี่ยนสถานะจากสถานะ p ไปยังสถานะ q เมื่ออ่านสัญลักษณ์บนเทปเป็น 1 แล้ว ทุกครั้งที่ M อยู่ในสถานะ p และอ่านได้สัญลักษณ์ 1 บนเทปแล้ว M จะเปลี่ยนไปยังสถานะ q เสมอ ดังนั้น ออโตมาตาจำกัดเชิงกำหนดมีการเปลี่ยนสถานะซึ่งอธิบายได้ด้วยฟังก์ชันของสถานะปัจจุบันและสัญลักษณ์ที่อ่านได้ ตัวอย่างต่อไปนี้แสดง DFA ที่ยอมรับชื่อในภาษาโปรแกรมที่อธิบายด้วยนิพจน์ปกติ [A-Za-z_][A-Za-z0-9_]*

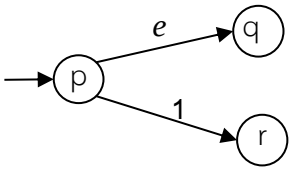
ตัวอย่าง 2.4



ออโตมาตาจำกัดเชิงไม่กำหนดเป็นออโตมาตาที่มีการเปลี่ยนสถานะเชิงไม่กำหนด คือ เมื่อเครื่องอยู่ในสถานะหนึ่งและอ่านได้สัญลักษณ์หนึ่ง เครื่องสามารถเปลี่ยนสถานะไปยังสถานะถัดไปได้มากกว่าหนึ่งสถานะ เช่น จากออโตมาตาจำกัดเชิงไม่กำหนด M ข้างล่างนี้ เมื่อ M อยู่ในสถานะ p และอ่านสัญลักษณ์บนเทปเป็น 1 แล้วอาจเปลี่ยนไปยังสถานะ q หรือสถานะ r ก็ได้

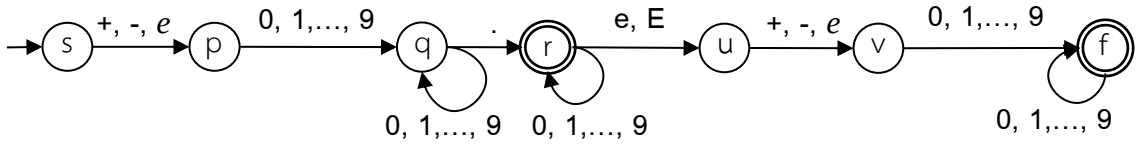


การเปลี่ยนสถานะเชิงไม่กำหนดอีกแบบ คือ เมื่อเครื่องอยู่ในสถานะหนึ่งแล้วเครื่องสามารถเปลี่ยนสถานะไปยังสถานะถัดไปโดยไม่อ่านสัญลักษณ์ใดเลยหรืออาจยังคงอยู่ในสถานะเดิมก็ได้ เช่น จากออโตมาตาจำกัดไม่เชิงกำหนด M ข้างล่างนี้ เมื่อ M อยู่ในสถานะ p แล้วอาจเปลี่ยนไปยังสถานะ q โดยไม่อ่านสัญลักษณ์บนเทป หรือ อยู่ในสถานะ p จนกว่าจะอ่านสัญลักษณ์ถัดไปบนเทป



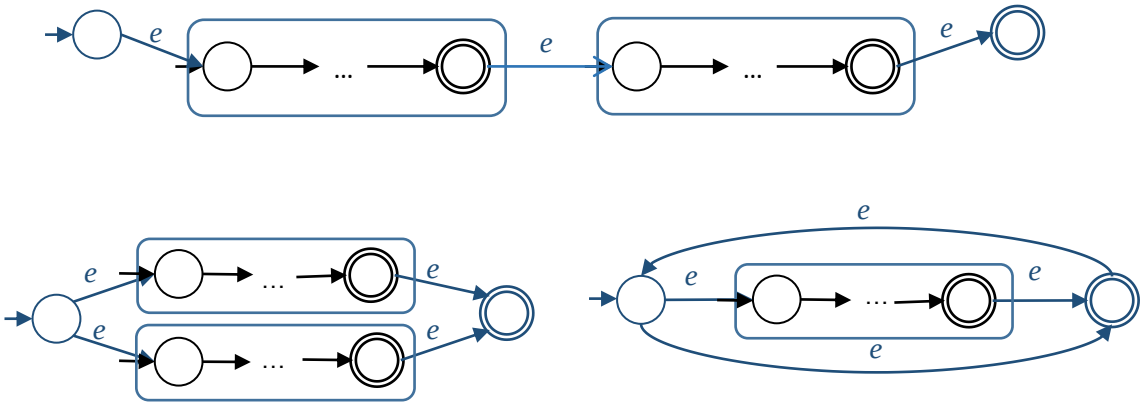
ดังนั้น ฟังก์ชันเปลี่ยนสถานะของออโตมาตาจำกัดเชิงไม่กำหนดระบุว่าหากเครื่องอยู่ที่สถานะหนึ่งและอ่านสัญลักษณ์หนึ่งแล้วสถานะถัดไปของเครื่องเป็นสถานะใดได้บ้าง ตัวอย่างต่อไปนี้แสดง NFA ที่ยอมรับจำนวนจริงในภาษาโปรแกรมที่อธิบายด้วยนิพจน์ปกติ (+|-)? [0-9]$^{+}$. [0-9]* ((e|E) (+|-)?[0-9]$^{+}$)?

ตัวอย่าง 2.5 ออโตมาตาจำกัดเชิงไม่กำหนดนี้ยอมรับจำนวนเต็มที่ไม่มีส่วนยกกำลังโดยจบการทำงานที่สถานะ r และยอมรับจำนวนเต็มที่ไม่ส่วนยกกำลังโดยจบการทำงานที่สถานะ f สัญลักษณ์ e แทนสตริงว่าง

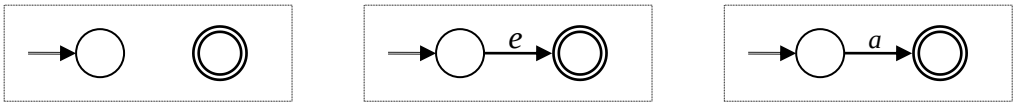


2.3.2 การสร้างออโตมาตาจำกัดจากนิพจน์ปกติ

จากตัวอย่าง 2.4 และ 2.5 จะเห็นว่าเราสร้างออโตมาตาจำกัดตามนิพจน์ปกติที่กำหนดให้ได้ อย่างไรก็ตามยังมีขั้นตอนวิธีที่ใช้แปลงนิพจน์ปกติเป็นออโตมาตาจำกัดได้ โดยสร้างออโตมาตาจำกัดสำหรับนิพจน์ปกติที่เล็กสุดก่อนแล้วค่อยนำออโตมาตาจำกัดนี้มาประกอบกันต่อไป วิธีประกอบออโตมาตาจำกัดย่อยเข้าด้วยกันนี้ขึ้นอยู่กับตัวกระทำที่ใช้สร้างนิพจน์ปกติที่ใหญ่ขึ้น รูปข้างล่างนี้แสดงการประกอบออโตมาตาจำกัดที่สร้างตามนิพจน์ปกติที่สร้างด้วยตัวกระทำ การเชื่อม การรวม และ การซ้ำ ตามลำดับ ในที่นี้ให้ M1 และ M2 เป็นออโตมาตาจำกัดที่สร้างจาก r1 และ r2 ตามลำดับ



สำหรับนิพจน์ปกติที่มีขนาดเล็กสุด เช่น นิพจน์ปกติของเซตว่าง นิพจน์ปกติของสตริงว่าง และ นิพจน์ปกติของสัญลักษณ์ตัวเดียว เราสามารถสร้างออโตมาตาจำกัดสำหรับนิพจน์ปกติเหล่านี้ได้ดังแสดงในรูปข้างล่างนี้



ตัวอย่าง 2.5 แสดงออโตมาตาจำกัดสำหรับนิพจน์ปกติ (+|-)? [0-9]$^+$. [0-9]* ((e|E) (+|-)?[0-9]$^+$)? สังเกตได้ว่าจำนวนจริงสร้างจากจำนวนเต็ม ต่อด้วย. และต่อด้วย เลขหลังจุดทศนิยมหรือไม่ก็ได้ จากนั้นตามด้วยตัวอักขระ e หรือ E แล้วตามด้วยจำนวนเต็มอีกชุดหนึ่ง ออโตมาตานี้สร้างจากการประกอบออโตมาตาสำหรับนิพจน์ปกติเล็ก ๆ ขึ้นมา

นิพจน์ปกติ (+|-)? แทนด้วยการเปลี่ยนสถานะจาก s ไปยัง p

นิพจน์ปกติ [0-9]$^+$ แทนด้วยการเปลี่ยนสถานะจาก p ไปยัง q

นิพจน์ปกติ . แทนด้วยการเปลี่ยนสถานะจาก q ไปยัง r

นิพจน์ปกติ [0-9]* แทนด้วยการเปลี่ยนสถานะจาก r ไปยัง r

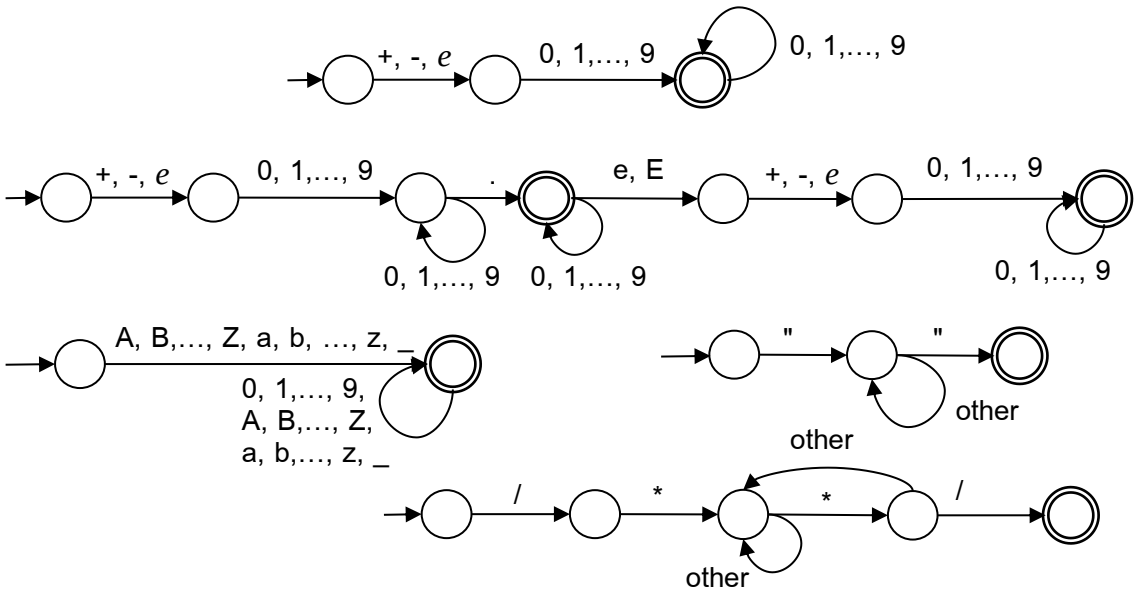
นิพจน์ปกติ ((e|E) (+|-)?[0-9]$^+$) แทนด้วยการเปลี่ยนสถานะจาก r ไปยัง f

ดังนั้นเราจะได้ออโตมาตาจำกัดเปลี่ยนสถานะจาก s ไปยัง r เมื่ออ่านได้จำนวนจริงที่ไม่มีส่วนยกกำลัง และเปลี่ยนจากสถานะ r ไปยัง f เมื่ออ่านได้ส่วนยกกำลัง

เมื่อเราสร้างออโตมาตาจำกัดสำหรับโทเคนแต่ละชนิดแล้ว จะต้องนำออโตมาตาจำกัดเหล่านี้มาประกอบกันเป็นออโตมาตาตัวเดียวด้วยตัวกระทำการรวม โดยใช้การเปลี่ยนสถานะด้วยสตริงว่างจากสถานะเริ่มต้นที่สร้างขึ้นมาใหม่ไปยังสถานะเริ่มต้นของออโตมาตาสำหรับแต่ละโทเคน ออโตมาตาที่ได้จะเป็นออโตมาตาจำกัดเชิงไม่กำหนดที่ยอมรับโทเคนแต่ละชนิด และมีสถานะสิ้นสุดสำหรับโทเคนแต่ละชนิดที่แยกกัน

ตัวอย่างต่อไปนี้แสดงออโตมาตาจำกัดที่ยอมรับโทเคนชนิด จำนวนเต็ม จำนวนจริง ชื่อ สตริง และคอมเมนท์ในภาษาซี เราจะใช้ other สำหรับการเปลี่ยนสถานะเพื่อระบุว่าการเปลี่ยนสถานะนั้นเกิดขึ้นเมื่ออ่านได้สัญลักษณ์ที่ไม่ได้ระบุไว้ในการเปลี่ยนสถานะอื่น ๆ จากสถานะนั้น

<u>ตัวอย่าง 2.6</u> ออโตมาตาจำกัดเชิงไม่กำหนดนี้ยอมรับโทเคนชนิด จำนวนเต็ม จำนวนจริง ชื่อ สตริง และ คอมเมนท์ในภาษาซี ตามลำดับ



ในหัวข้อถัดไปเราจะแสดงการนำออโตมาตาจำกัดนี้ไปสร้างเป็นโปรแกรมสแกนเนอร์

2.4 โปรแกรมสแกนเนอร์

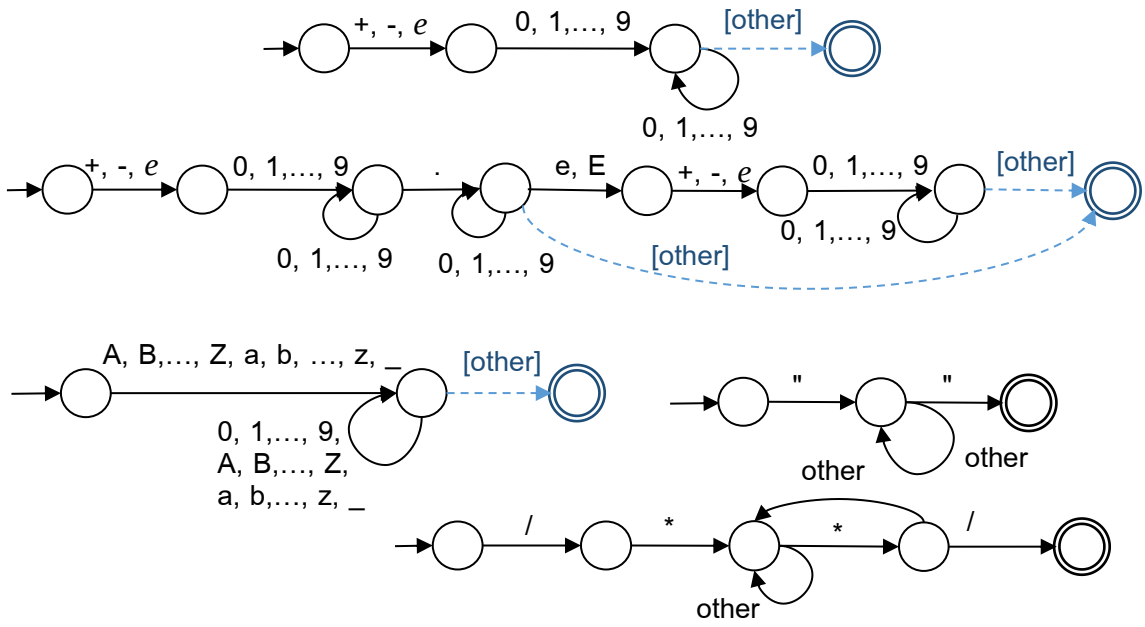
2.4.1 การปรับออโตมาตาจำกัดเพื่อสร้างสแกนเนอร์

เมื่อสร้างออโตมาตาจำกัดที่ตรวจสอบโทเคนทั้งหมดในภาษาโปรแกรมแล้ว เราสามารถสร้างสแกนเนอร์จำลองการทำงานของออโตมาตาจำกัดนี้ อย่างไรก็ตาม การทำงานของออโตมาตาจำกัดต่างจากการทำงานของสแกนเนอร์เป็นบางส่วน ออโตมาตาจำกัดหยุดทำงานและบอกว่าสายอักขระที่รับเข้ามาเป็นโทเคนที่ยอมรับหรือไม่เมื่ออ่านสายอักขระนั้นจบ แต่โปรแกรมสแกนเนอร์รับสายอักขระที่มีหลายหน่วยภาษาเรียงต่อกัน ดังนั้นสแกนเนอร์ต้องบอกได้ว่าหน่วยภาษาแต่ละหน่วยจบลงที่ใด สายอักขระ ab1 อาจแบ่งเป็นโทเคนได้หลายแบบ เช่น เป็นชื่อ a และ ชื่อ b กับ จำนวนเต็ม 1, เป็นชื่อ ab กับ จำนวนเต็ม 1, เป็นชื่อ ab1 ดังนั้นสแกนเนอร์ต้องระบุว่าจะเลือกแบบใด หลักการที่ใช้ทั่วไปคือให้เลือกสายอักขระที่ยาวที่สุดตรงกับรูปแบบของโทเคนชนิดหนึ่ง ในกรณีของสายอักขระ ab1 นี้ จะเลือกให้ได้โทเคนเดียวเป็นชื่อ ab1

ในการใช้หลักเกณฑ์นี้สแกนเนอร์อาจต้อง*มองไปข้างหน้า* (look ahead) จนพบตัวอักขระที่ไม่ทำให้หน่วยภาษานั้นตรงกับโทเคนใด ๆ ตัวอย่าง เช่น สายอักขระ ab1+ จะถูกแบ่งเป็นชื่อ ab1 เมื่ออ่านพบตัวอักขระ + ซึ่งทำให้ ab1+ ไม่ตรงกับโทเคนใดในภาษา สแกนเนอร์จึงตัดสินใจได้ว่า ab1 เป็นหน่วยภาษาที่ใหญ่สุดที่เป็นไปได้ ตัวอักขระ + จะไม่ถูกรวมอยู่ในหน่วยภาษานี้ และเรียกว่าเป็น*สัญลักษณ์มองข้างหน้า* (look-ahead symbol) จากนั้นสแกนเนอร์ต้องทำเสมือนว่ายังไม่ได้อ่านอักขระ + เข้ามาเรียกว่า*การคืน* หรือ unget สแกนเนอร์อ่านสายอักขระต่อไปโดยเริ่มจากตัวอักขระ + นี้

จากที่กล่าวมาแล้วเราจะต้องระบุการมองไปข้างหน้าและการคืนตัวอักขระโดยจะแสดงไว้ในออโตมาตาจำกัดด้วยสัญลักษณ์ที่อยู่ใน [ ] เช่น [+] ระบุว่าการเปลี่ยนสถานะนั้นเกิดเมื่ออ่านตัวอักขระ + โดย + เป็นสัญลักษณ์มองข้างหน้าและจะคืน + กลับไปเป็นตัวอักขระที่ยังไม่ได้อ่าน ตัวอย่างข้างล่างนี้ระบุการมองไปข้างหน้าที่จำเป็นสำหรับการทำงานของสแกนเนอร์ที่สร้างจากออโตมาตาจำกัดในหัวข้อที่ผ่านมา

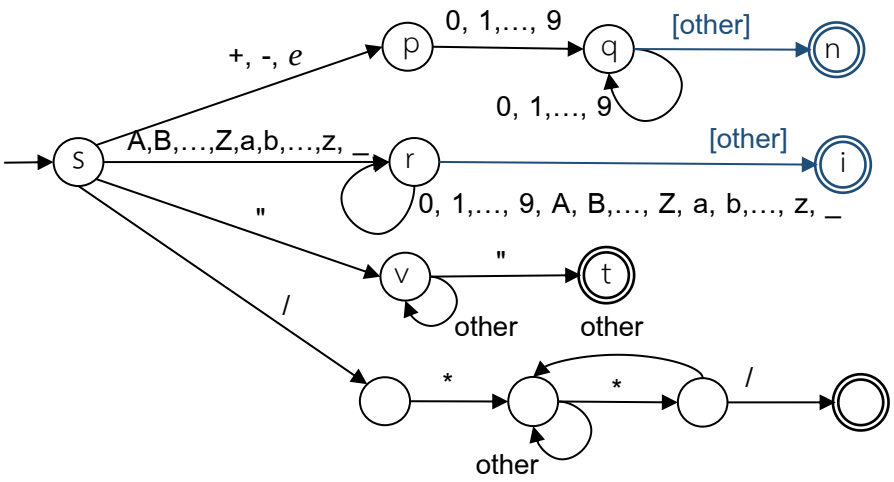
ตัวอย่าง 2.7 ออโตมาตาจำกัดนี้ระบุการมองไปข้างหน้าที่จำเป็นสำหรับการทำงานของสแกนเนอร์ที่แสดงด้วยเส้นประสีฟ้า



ข้อสังเกต: - การมองไปข้างหน้าไม่ได้จำเป็นสำหรับทุกโทเคน

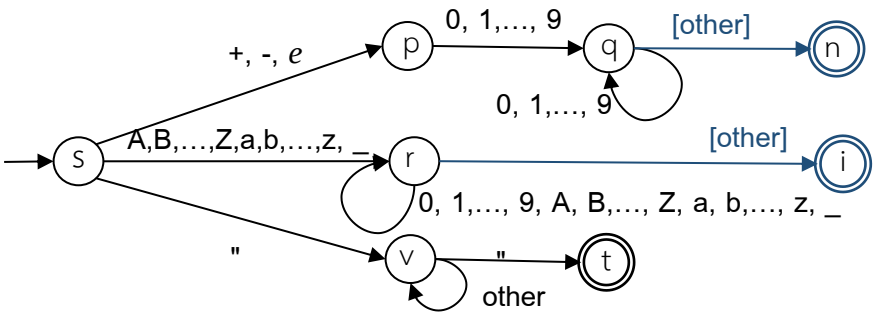
- สัญลักษณ์มองไปข้างหน้ามักเป็นสัญลักษณ์อื่นที่แสดงด้วย other แต่ other อาจใช้ในกรณีอื่นนอกจากการมองไปข้างหน้า
- สแกนเนอร์มักคืนสัญลักษณ์มองไปข้างหน้า แต่ไม่จำเป็นต้องทำเสมอ

ขั้นตอนถัดไปของ การสร้างสแกนเนอร์ คือ การนำออโตมาตาสำหรับแต่ละโทเคนมารวมให้เป็นออโตมาตาเดียวด้วยตัวกระทำการรวม ดังนั้นจะสร้างสถานะเริ่มต้นใหม่ แล้วเพิ่มการเปลี่ยนสถานะด้วยสตริงว่างจากสถานะเริ่มต้นใหม่นี้ไปยังสถานะเริ่มต้นของออโตมาตาของแต่ละโทเคน ออโตมาตานี้ไปถึงสถานะสิ้นสุดสถานะก็จะบอกได้ จากสถานะนั้นว่าโทเคนที่ได้เป็นชนิดใด ออโตมาตาในรูปข้างล่างนี้เป็นออโตมาตาที่ใช้สำหรับสแกนเนอร์ของโทเคนจำนวนเต็ม ชื่อ สตริง และ คอมเมนต์



2.4.2 โปรแกรมจำลองการทำงานของออโตมาตาจำกัดเพื่อสร้างสแกนเนอร์

เมื่อปรับออโตมาตาเพื่อสร้างสแกนเนอร์แล้วเราสร้างสแกนเนอร์โดยเขียนโปรแกรมจำลองการทำงานของออโตมาตานั้น ในที่นี้แสดงตัวอย่างโปรแกรมในภาษาไพธอนสำหรับออโตมาตาที่ยอมรับโทเคนจำนวนเต็ม ชื่อและสตริงที่แสดงข้างล่างนี้ ออโตมาตานี้สร้างจากการรวมออโตมาตาของโทเคนจำนวนเต็ม ชื่อ และ สตริง แล้วนำมาปรับลดจำนวนสถานะลง



ในโปรแกรมข้างล่างนี้ ตัวแปร states เก็บสถานะของออโตมาตาเป็นเซตของชื่อสถานะ ตัวแปร startState เก็บชื่อสถานะเริ่มต้น ตัวแปร finalStates เก็บเซตของชื่อสถานะสิ้นสุด ตัวแปร tFunc เป็น dictionary ที่เก็บฟังก์ชันการเปลี่ยนสถานะ ตัวแปร ungetTr เก็บเซตของคู่ลำดับของสถานะสิ้นสุด f และสัญลักษณ์ a เพื่อระบุว่าต้องคืนสัญลักษณ์ a หากออโตมาตาไปถึงสถานะสิ้นสุด f ด้วยสัญลักษณ์ a จะเห็นได้ว่าตัวแปรเหล่านี้เก็บข้อมูลทั้งหมดของออโตมาตาที่ใช้สร้างสแกนเนอร์

```
# Scanner for integer:(+|-|e)[0-2]+
#             string: "[a-dA-D0-2+-_ ]*"
#             identifier: [a-dA-D_][a-dA-D0-2_]*

states=set(['s', 'p', 'q', 'r', 'v', 'n', 't', 'i'])  # set of states
startState='s'                                        # start state
finalStates=set(['n', 't', 'i'])                      # set of final states

# transition function
tFunc = {('s','"'):set(['r']), ('s','_'):set(['v']), ('s','+'):set(['p']),
          ...,
         ('n','e'):set([]),    ('t','e'):set([]),    ('i','e'):set([])}

ungetTr = set([('n','+'),('n','-'),('n','_'),('n','"', ... ,
               ('i','+'),('i','-'),('i',' '),('i','"')]) # unget transition

tokenMap = {'n':'integer', 't':'string', 'i':'identifier'}

buffer='-11 a_b 12 "a1- +12"22""'                     # source code

# Find the closure of each state
setClosure={}
for state in states:
    setClosure[state]=closure(set([state]))

# main loop call scanner
bufferPtr=0
while bufferPtr<len(buffer):
    tk,bufferPtr=scan(bufferPtr)
    print(tk)
```

ฟังก์ชัน closure คำนวณหาเซตของสถานะที่อยู่ในส่วนปิดคลุม (closure) ของสถานะที่กำหนด นั่นคือการหาว่าออโตมาตาสามารถเปลี่ยนจากสถานะที่กำหนดไปอยู่ในสถานะใดด้วยสตริงว่างได้บ้าง ฟังก์ชันนี้ทำงานโดย ให้เสร็จของสถานะที่กำหนดมาอยู่ในส่วนปิดคลุม เก็บในตัวแปร clsSet ก่อน จากนั้นหาว่ามีการเปลี่ยนสถานะด้วยสตริงว่างจากเซตของสถานะในตัวแปร clsSet ไปยังสถานะใดได้บ้าง แล้วเพิ่มสถานะเหล่านั้นในตัวแปร clsSet จนไม่ได้สถานะใหม่เพิ่มขึ้นมา

```
def closure(states):    # find closure of a set of states from lambda
transiton
    clsSet=states
    for s in states:
        clsSet=clsSet.union(tFunc[(s,'e')])
        if clsSet.issubset(states):
            return states
        else:
            return states.union(closure(clsSet))
```

ฟังก์ชัน scan อ่าน ตัวอักขระหรือสัญลักษณ์ที่เก็บในตัวแปร buffer โดยเริ่มที่ตำแหน่งที่ระบุด้วยพารามิเตอร์ p ของฟังก์ชัน สถานะเริ่มต้นของออโตมาตาเก็บในตัวแปร currentState ซึ่งเมื่อเริ่มต้นทำงานกำหนดให้เป็นส่วนปิดคลุมของสถานะเริ่มต้น จากนั้นในการวนซ้ำแต่ละครั้ง จะอ่านสัญลักษณ์จากตัวแปร buffer มาเก็บในตัวแปร sym และหาว่าออโตมาตาสามารถเปลี่ยนสถานะจากแต่ละสถานะในตัวแปร currentState ไปยังสถานะใดได้บ้างโดยดูจากฟังก์ชันเปลี่ยนสถานะในตัวแปร tFunc แล้วหาส่วนปิดคลุมของสถานะเหล่านั้น เก็บในตัวแปร nextState การวนซ้ำนี้จะหยุดเมื่อออโตมาตาไปถึงสถานะสิ้นสุดสถานะหนึ่งได้ เมื่อถึงสถานะสิ้นสุดจะสามารถบอกได้ว่าสายอักขระที่อ่านมาได้เป็นหน่วยภาษาของโทเคนใด และต้องมีการคืนตัวอักขระที่อ่านมาหรือไม่ หากตรวจสอบแล้วสถานะสิ้นสุดนั้นและตัวอักขระที่อ่านมาเป็นสถานะที่ต้องมีการคืนตัวอักขระที่อ่านเข้ามาตามที่เก็บในตัวแปร ungetTr ก็จะถอยตำแหน่งของอักขระที่อ่านกลับไป 1 ตัว ฟังก์ชันนี้คืนโทเคนที่เป็นผลลัพธ์และตำแหน่งใน buffer ที่ต้องอ่านต่อไป

```
# ------------- function for scanner -----------
def scan(p):
    global states
    global startState
    global finalStates
    global ungetTr
    global buffer

    currentState=setClosure[startState]
    tk=""
    while not currentState.intersection(finalStates):
        sym=buffer[p]
        p=p+1
        nextState=set([])
        for st in currentState:
            for nxt in tFunc[(st,sym)]:
                nextState=nextState.union(setClosure[nxt])
        currentState=nextState
        tk=tk+sym
    [tState]=currentState.intersection(finalStates)
    if tState in set([st for (st, a) in ungetTr if a==sym]):
        p=p-1
```

```
        tk=tk[:-1]
    return (tokenMap[tState], p)
```

2.4.3 ตารางสัญลักษณ์

ตารางสัญลักษณ์เป็นโครงสร้างข้อมูลที่ใช้เก็บชื่อหรือค่าคงที่ที่ใช้ในโปรแกรม ดังนั้นสิ่งที่ถูกเก็บไว้ในตารางสัญลักษณ์อาจเป็นชื่อตัวแปร ชื่อฟังก์ชัน ชื่อคลาส หรือ สตริง ตารางนี้ใช้ในหลายขั้นตอนการทำงานของตัวแปลภาษา สแกนเนอร์เป็นส่วนแรกของตัวแปลภาษาที่จะนำชื่อหรือค่าคงที่ในโปรแกรมเก็บไว้ในตารางสัญลักษณ์ ดังนั้นเมื่อสแกนเนอร์อ่านได้โทเคนที่เป็นชื่อหรือค่าคงที่อื่นที่ต้องการเก็บในตารางสัญลักษณ์ก็จะมาใส่เพิ่มไว้ หากชื่อหรือค่าคงที่นั้นถูกเก็บไว้ในตารางสัญลักษณ์แล้วก็จะใส่ซ้ำอีก ดังนั้นทุกครั้งที่พบชื่อหรือค่าคงที่ในโปรแกรมสแกนหน้าต้องตรวจสอบว่ามีชื่อหรือค่านั้นในตารางสัญลักษณ์แล้วหรือยัง โครงสร้างข้อมูลที่ใช้สำหรับตารางสัญลักษณ์จึงควรเป็นโครงสร้างของข้อมูลที่เข้าถึงได้เร็ว เช่น ตารางแฮช (hash table)

ขั้นตอนการวิเคราะห์ความหมายเก็บชนิดของตัวแปรหรือข้อมูลเกี่ยวกับฟังก์ชันเพิ่มในตารางสัญลักษณ์และใช้เพื่อตรวจสอบการใช้ตัวแปรและฟังก์ชันให้ถูกชนิดและตรวจสอบขอบเขตของตัวแปรด้วย จากนั้น ในการสร้างรหัสจะกำหนดตำแหน่งในหน่วยความจำสำหรับชื่อเหล่านี้เพื่ออ้างถึงในรหัสที่สร้าง

2.4.4 การสร้างสแกนเนอร์ให้หาคำหลัก

คำหลักหรือคำสงวนเป็นคำที่มีรูปแบบตรงกับชื่อที่ใช้ในภาษาโปรแกรม โดยที่คำหลักแต่ละคำมีความหมายเฉพาะ สำหรับแต่ละคำ คำหลักแต่ละคำจึงเป็นโทเคนที่ต่างกันไป หากเราสร้างออโตมาตาจำกัดสำหรับคำหลักแต่ละคำ แล้วนำมารวมกันสร้างสแกนเนอร์ ออโตมาตาของสแกนเนอร์จะใหญ่มาก ดังนั้นสแกนเนอร์มักตรวจสอบหาชื่อและคำหลักรวมไปด้วยกัน จากนั้นจึงนำมาพิจารณาว่าเป็นชื่อหรือคำหลักโดยอาศัยตารางสัญลักษณ์ เนื่องจากเราต้องนำชื่อไปตรวจสอบว่ามีอยู่ในตารางสัญลักษณ์แล้วหรือยัง จึงสามารถใช้การตรวจสอบนี้เพื่อหาว่าชื่อนั้นเป็นคำหลักหรือไม่โดยให้เก็บคำหลักทั้งหมดไว้ในตารางสัญลักษณ์ก่อน เมื่อนำหน่วยคำที่ได้ไป หาในตารางสัญลักษณ์แล้วพบว่าหน่วยคำนั้นเป็นคำสงวนก็จะบอกได้ว่าไม่เป็นชื่อและส่งกลับ โทเคนของคำสงวนนั้นที่เก็บไว้ในตารางสัญลักษณ์ไป หากหน่วยคำนั้นอยู่ในตารางสัญลักษณ์โดยระบุว่าเป็นชื่อ ก็จะส่งค่ากลับว่าเป็นโทเคนชนิดชื่อ หากตรวจสอบแล้วไม่พบว่าอยู่ในตารางสัญลักษณ์หมายความว่าหน่วยคำนั้นเป็นชื่อที่ยังไม่เคยเก็บไว้ วิธีการ ทำงานแบบนี้ทำให้ออโตมาตาของสแกนเนอร์มีขนาดเล็กลงและสแกนเนอร์ทำงานได้ อย่างมีประสิทธิภาพ

2.4.5 การใช้ตัวอักษรตัวใหญ่/ตัวเล็ก

ในภาษาโปรแกรมส่วนใหญ่ที่ไม่ถือว่าตัวอักขระตัวใหญ่และตัวเล็กต่างกัน สแกนเนอร์สามารถเก็บชื่อทั้งหมดเป็นตัวอักขระตัวใหญ่ทั้งหมด (หรือ ตัวอักขระตัวเล็กทั้งหมด) และแปลงชื่อที่พบเป็นตัวอักขระตัวใหญ่ทั้งหมดก่อนนำไปเปรียบเทียบในตารางสัญลักษณ์

3. ตัวแจงส่วน (Parser)

ตัวแจงส่วนทำหน้าที่ตรวจสอบไวยากรณ์ของโปรแกรมที่ต้องการแปล ดังนั้นผู้ออกแบบภาษาต้องกำหนดไวยากรณ์ของภาษาก่อน หัวข้อ 3.1 กล่าวถึงประเด็นในการออกแบบภาษาที่เกี่ยวข้องกับตัวแจงส่วน ภาษาโปรแกรมเป็นภาษาที่ไม่พึ่งบริบท (context-free language) นั่นคือกฎการสร้างหน่วยของภาษา (เช่น นิพจน์ คำสั่ง if ฟังก์ชัน) ไม่ขึ้นกับสิ่งที่อยู่รอบข้างหน่วยของภาษานั้น ไวยากรณ์ของภาษาโปรแกรมอธิบายได้ด้วยไวยากรณ์ที่เรียกว่าไวยากรณ์ไม่พึ่งบริบท (context-free grammar) ซึ่งอธิบายในหัวข้อ 3.2 หัวข้อ 3.3 อธิบายออโตมาตาพูชดาว์น (pushdown automata) ที่เป็นตัวแบบที่ใช้ในการทำงานของตัวแจงส่วน หัวข้อ 3.4 แสดงขั้นตอนวิธีแจงส่วนที่ใช้ออโตมาตาพูชดาว์นที่เรียกว่า การแจงส่วนแบบ LL(1) หัวข้อ 3.5 แสดงขั้นตอนวิธีแจงส่วนแบบ LR(1) ที่เป็นการแจงส่วนอีกวิธีที่ใช้สแตค (stack) ประกอบกับออโตมาตาจำกัด

3.1 แนวคิดในภาษาโปรแกรมที่เกี่ยวข้องกับตัวแจงส่วน

ในการออกแบบภาษาโปรแกรม ผู้ออกแบบจะเลือกรูปแบบของคำสั่งหรือส่วนของโปรแกรมเพื่อให้อ่านเข้าใจได้ง่าย นอกจากนั้นยังต้องทำให้วิเคราะห์โครงสร้างของภาษาได้ง่ายด้วย นั่นคือทำให้ตัวแจงส่วนแยกส่วนประกอบในโปรแกรมได้อย่างถูกต้อง เมื่อโครงสร้างในภาษาทำให้ตีความหมายได้มากกว่าหนึ่งแบบ ผู้ออกแบบต้องเลือกระบุความหมายที่ต้องการเพียงแบบเดียว ทั้งนี้เพื่อให้ผู้ที่สร้างตัวแปลภาษาใช้ความหมายเดียวกันและตัวแปลภาษาทุกตัวสร้างโปรแกรมให้ทำงานได้เหมือนกัน

ในหัวข้อนี้เราจะพิจารณาประเด็นในการออกแบบภาษาโปรแกรมที่มีส่วนเกี่ยวข้องกับการสร้างตัวแจงส่วน

3.1.1 ลำดับการทำงานในนิพจน์

รูปแบบของนิพจน์ในภาษาโปรแกรมถูกออกแบบให้คล้ายกับนิพจน์ทางคณิตศาสตร์และตรรกศาสตร์ แต่ลำดับการทำงานของตัวกระทำในนิพจน์ที่ใช้กันในภาษาคณิตศาสตร์สามารถแสดงได้ด้วยวิธีเขียนที่ต่าง ๆ กัน เช่น สำหรับ $\frac{3+x}{y}$ เรารู้ได้ว่าต้องบวก 3 กับ x ก่อน แล้วจึงเอาผลลัพธ์ที่ได้หารด้วย y แต่เราไม่สามารถใช้รูปแบบของนิพจน์เช่นนี้ในภาษาโปรแกรม นอกจากนั้นเรายังต้องการให้ตัวแปลภาษาทุกตัวแปลให้ได้ลำดับการทำงานในนิพจน์ออกมาตรงกันเสมอ ดังนั้นภาษาโปรแกรมแต่ละภาษาจะกำหนดลำดับความสำคัญ (precedence) ของตัวกระทำเพื่อกำหนดว่าตัวกระทำใดในนิพจน์จะทำงานก่อน ลำดับความสำคัญที่กำหนดนี้จะใกล้เคียงกับที่มนุษย์ตีความนิพจน์ทางคณิตศาสตร์ ตัวอย่าง เช่น จากนิพจน์ $3x^2 - 5$ จะทำ x^2 ก่อนแล้วจึงคูณด้วย 3 แล้วลบด้วย 5 ดังนั้นการยกกำลังมีลำดับความสำคัญสูงกว่าการคูณ และ การคูณมีความลำดับสำคัญสูงกว่าการลบ เป็นต้น ตารางต่อไปนี้แสดงลำดับความสำคัญของตัวกระทำในภาษาซีและภาษาไพธอน

ตารางแสดงลำดับความสำคัญของตัวกระทำในภาษาซีและภาษาไพธอนเรียงจากต่ำไปสูง

ภาษาซี	ภาษาไพธอน
|| (logical OR) && (logical AND) | (bitwise OR) ^ (bitwise XOR) & (bitwise AND) ==, != <, <=, >, >= <<, >> (bitwise shift) +, - *, /, % +x, -x, ~x ( )	or and not in, not in, is, is not, <, <=, >, >=,!=, == | ^ & <<, >> (bitwise shift) +, - *, @, /, //, % +x, -x, ~x ** ( )

นอกจากนั้นยังต้องกำหนดลำดับการทำงานสำหรับตัวกระทำที่มีลำดับความสำคัญเท่ากัน พิจารณานิพจน์ $x-y-z$ เราจะทำ $x-y$ ก่อนแล้วจึงนำไปลบด้วย z แต่สำหรับนิพจน์ x^{2^3} เราจะทำ 2^3 ก่อนแล้วจึงนำมาเป็นกำลังของ x หลักการที่ใช้ในการกำหนดลำดับการทำงานของตัวกระทำที่มีความสำคัญเท่ากันเรียกว่า สมบัติการเชื่อมโยง (associativity) เมื่อตัวกระทำสองตัวอยู่ทางซ้ายและขวาของตัวถูกกระทำ x ดังนี้ ... $\otimes\ x\ \otimes$ ...

- หาก x ทำงานกับตัวกระทำทางซ้ายก่อน (คือ (...$\otimes\ x$) $\otimes$...) เราเรียกว่าตัวกระทำ $\otimes$ มีสมบัติการเชื่อมโยงทางซ้าย
- หาก x ทำงานกับตัวกระทำทางขวาก่อน (คือ ...$\otimes$ ($x\otimes$...)) เราเรียกว่าตัวกระทำ $\otimes$ มีสมบัติการเชื่อมโยงทางขวา

ภาษาโปรแกรมโดยทั่วไปจะให้ตัวกระทำมีสมบัติการเชื่อมโยงทางซ้าย ยกเว้นตัวกระทำยกกำลัง

3.1.2 Dangling else

โครงสร้าง if เป็นโครงสร้างการทำงานแบบทางเลือกที่มีอยู่ในเกือบทุกภาษาโปรแกรม โครงสร้าง if นี้สามารถนำมาวางซ้อนกันได้ดังแสดงในโปรแกรมภาษาซีข้างล่างนี้

```
if (x<=0)
   if (y<=0)
      y=10;
   else
      y--;
else
   x--;
```

โครงสร้าง if อาจไม่มีส่วน else ได้ เช่น `if (x==0) x=10;` เมื่อนำโครงสร้าง if สองแบบนี้มาซ้อนกัน โครงสร้างจะมี if 2 ที่และ else 1 ที่ แล้วสามารถตีความให้ else เป็นส่วนของ if ที่ตำแหน่งแรกหรือตำแหน่งที่สองก็ได้ ในภาษาโปรแกรมส่วนใหญ่ เช่น ภาษาซีจะให้ else จับคู่กับ if ก่อนหน้าที่ใกล้ที่สุดที่ไม่มี else อยู่ เช่น

```
if (x==0)          // if ตัวที่ 1
   if (y==0)       // if ตัวที่ 2
      if (z==0)    // if ตัวที่ 3
          z++;
       else        // else ตัวที่ 1
          z=0;
    else           // else ตัวที่ 2
       y=0;
```

ในโครงสร้างนี้ else ตัวที่ 1 เป็นส่วนของ if ตัวก่อนหน้าที่ใกล้สุดคือ if ตัวที่ 3 และ else ตัวที่ 2 เป็นส่วนของ if ตัวก่อนหน้าที่ใกล้สุดที่ยังไม่มี else นั่นคือ if ตัวที่ 2 ดังนั้น if ตัวที่ 1 จะเป็น if ทีไม่มี else โปรแกรมเมอร์สามารถเปลี่ยนการจับคู่ if-else ได้โดยใช้ { } กำหนดบล็อคของโปรแกรมได้ดังตัวอย่างข้างล่างนี้

```
if (x==0)          // if ตัวที่ 1
   if (y==0)       // if ตัวที่ 2
   {  if (z==0)    // if ตัวที่ 3
          z++;
   }
   else            // else ตัวที่ 1
       z=0;
else               // else ตัวที่ 2
   y=0;
```

ในภาษาไพธอน การจับคู่ if-else ระบุอย่างชัดเจนด้วยบล็อคของโปรแกรมซึ่งกำหนดด้วยการจัดย่อหน้า (indentation)

3.1.3 ตำแหน่งของคำหลักในโครงสร้าง

คำสั่งหรือโครงสร้างในภาษาจะมีรูปแบบที่แตกต่างกันอย่างชัดเจน ตัวอย่าง เช่น โครงสร้าง if โครงสร้าง while และ โครงสร้าง for เริ่มต้นด้วยคำหลัก if, while และ for เพื่อให้ตัวแจงส่วนรู้ว่าเป็นโครงสร้างชนิดใดและทำให้โปรแกรมมาอ่านเข้าใจได้ด้วย อย่างไรก็ตามการใช้คำหลักเพื่อบ่งบอกโครงสร้างอาจไม่เหมาะกับบางโครงสร้างบาง เช่น ประโยคกำหนดค่า (assignment statement) ภาษาโปรแกรมส่วนใหญ่จะใช้ชื่อตัวแปรตามด้วยเครื่องหมาย = เพื่อบ่งบอกว่าเป็นประโยคกำหนดค่า แต่ไม่ใช้คำหลักระบุโครงสร้าง เช่น assign x 10 เนื่องจากคนทั่วไปจะเข้าใจการใช้เครื่องหมาย = เพื่อกำหนดค่าได้ง่ายกว่า

3.2 การอธิบายภาษาโปรแกรมด้วยไวยากรณ์ไม่พึ่งบริบท

โครงสร้างทางไวยากรณ์ของภาษาโปรแกรมอธิบายได้ด้วยการแบ่งเป็นองค์ประกอบย่อยคล้ายกับไวยากรณ์ในภาษาธรรมชาติ ไวยากรณ์ที่ใช้อธิบายภาษาโปรแกรมไม่ซับซ้อนเท่าไวยากรณ์ที่ใช้อธิบายภาษาธรรมชาติ ในหัวข้อนี้เราจะนิยามไวยากรณ์ที่เรียกว่า ไวยากรณ์ไม่พึ่งบริบทซึ่งใช้อธิบายภาษาโปรแกรมและแสดงการเขียนไวยากรณ์สำหรับส่วนย่อยของภาษาโปรแกรม นอกจากนั้นจะแสดงการเขียนไวยากรณ์เพื่อกำหนดลำดับการทำงานของตัวกระทำในนิพจน์ตามลำดับความสำคัญและการเชื่อมโยงของตัวกระทำ

3.2.1 ไวยากรณ์ไม่พึ่งบริบท

ไวยากรณ์ไม่พึ่งบริบทระบุกฎการสร้างส่วนประกอบย่อยในภาษาจนเป็นประโยคในภาษา ในกฎเหล่านี้ใช้สัญลักษณ์แทนองค์ประกอบย่อยหรือคำในภาษา สัญลักษณ์ที่ใช้แบ่งเป็น 2 ชนิด คือ สัญลักษณ์ปลาย (terminal symbol) และสัญลักษณ์ไม่ปลาย (non-terminal symbol) สัญลักษณ์ปลาย เป็นสัญลักษณ์ที่ใช้ในประโยคในภาษา (เช่น คำในภาษาธรรมชาติ) สัญลักษณ์ไม่ปลายเป็นสัญลักษณ์แทนองค์ประกอบในภาษา (เช่น วลี ภาคแสดง ในภาษาธรรมชาติ) กฎในไวยากรณ์ระบุว่าองค์ประกอบแต่ละตัวประกอบด้วยส่วนประกอบย่อยใด กฎในไวยากรณ์ไม่พึ่งบริบทอยู่ในรูปของ X $\rightarrow$ W เมื่อ X เป็นสัญลักษณ์ไม่ปลายและ W สตริงที่สร้างจากสัญลักษณ์ใด ๆ ต่อกัน

ตัวอย่าง 3.1 ถ้าให้ S เป็นสัญลักษณ์ไม่ปลาย (และ) เป็นสัญลักษณ์ปลาย

S $\rightarrow$ (S) S

S $\rightarrow$ e

เป็นกฎที่อธิบายการสร้างวงเล็บที่จับคู่เปิดปิดกันอย่างถูกต้อง เช่น ( ), ( )( ), (( )), (( ) ( )) (( )) เป็นต้น

สังเกตว่าทางซ้ายมือของกฎในไวยากรณ์ไม่พึ่งบริบทมีสัญลักษณ์ไม่จบเพียงตัวเดียว นั่นหมายความว่าการระบุว่าองค์ประกอบที่แทนด้วยสัญลักษณ์นั้นสร้างจากองค์ประกอบใดบ้างจะไม่ขึ้นกับบริบท (context) หรือสิ่งที่อยู่รอบข้างของสัญลักษณ์นั้น นอกจากนั้นในไวยากรณ์จะระบุสัญลักษณ์ที่เรียกว่าสัญลักษณ์เริ่มต้น (start symbol) ไว้ และสิ่งที่สร้างจากสัญลักษณ์เริ่มต้นตามกฎในไวยากรณ์นั้นเป็นสิ่งที่อยู่ในภาษานั้น

เราอาจนิยามไวยากรณ์ไม่พึ่งบริบทได้ดังนี้

> G = (V, T, P, S) อธิบายไวยากรณ์ไม่พึ่งบริบทเมื่อ V เป็นเซตของสัญลักษณ์ไม่ปลาย, T เป็นเซตของสัญลักษณ์ปลาย, S เป็นสัญลักษณ์เริ่มต้นที่อยู่ใน V และ P เป็นเซตของกฎที่ใช้สร้างสตริงในภาษาโดยที่กฎ X $\rightarrow$ W ใน P X ต้องมาเป็นสัญลักษณ์ไม่จบและ W สร้างจากสัญลักษณ์ใด ๆ ในภาษาต่อกัน

ตัวอย่าง 3.2 ไวยากรณ์ที่อธิบายในตัวอย่าง 3.1 ก่อนหน้านี้สามารถเขียนได้เป็น ({S}, { (,)}, {S $\rightarrow$ (S) S, S $\rightarrow$ e}, S}

การแปลง (derivation) เป็นขั้นตอนการใช้กฎในไวยากรณ์เพื่อสร้างประโยค และใช้เครื่องหมาย $\Rightarrow$ เพื่อบอกการแปลงสัญลักษณ์ไม่ปลายหนึ่งตัวในประโยคตามกฎในไวยากรณ์ ตัวอย่างต่อไปนี้แสดงการแปลงที่ทำให้ได้ ( ), ( )( ), (( )), (( ) ( )) (( ))

ตัวอย่าง 3.3 ลำดับการแปลงข้างล่างนี้แสดงการแปลงที่ให้ ( ), ( )( ), (( )), (( ) ( )) (( )) ตามลำดับ

$S \Rightarrow (S) S \Rightarrow (S) \Rightarrow ( )$

$S \Rightarrow (S) S \Rightarrow (S) (S) S \Rightarrow ( ) (S) S \Rightarrow ( ) ( ) S \Rightarrow ( ) ( )$

$S \Rightarrow (S) S \Rightarrow ((S) S) S \Rightarrow ((S) S) \Rightarrow (( ) S) \Rightarrow (( ))$

$S \Rightarrow (S) S \Rightarrow ((S) S) S \Rightarrow (( ) S) S \Rightarrow (( ) (S) S) S \Rightarrow (( ) ( ) S) S$
$\Rightarrow (( ) ( )) S \Rightarrow (( ) ( )) (S) S \Rightarrow (( ) ( )) ((S) S) S \Rightarrow (( ) ( )) (( ) S) S$
$\Rightarrow (( ) ( )) (( )) S \Rightarrow (( ) ( )) (( ))$

สตริงที่ได้จากการแปลงแต่ละขั้นเรียกว่าแบบประโยค (sentential form) สัญลักษณ์ไม่ปลายในแบบประโยคถูกเปลี่ยนตามกฎในแต่ละขั้นของการแปลง หากการแปลงแต่ละขั้นเลือกแปลงสัญลักษณ์ไม่ปลายตัวซ้ายสุด การแปลงนั้นเรียกว่า การแปลงซ้ายสุด (leftmost derivation) หากการแปลงแต่ละขั้นเลือกแปลงสัญลักษณ์ไม่ปลายตัวขวาสุด การแปลงนั้นเรียกว่า การแปลงขวาสุด (rightmost derivation)

บางครั้งการแปลงประกอบด้วยหลายขั้นตอน เราใช้สัญลักษณ์ $\Rightarrow^*$ แสดงการแปลงหลายขั้นที่ต่อกัน ดังแสดงในตัวอย่างข้างล่างนี้

ตัวอย่าง 3.4 ลำดับการแปลงข้างล่างนี้แสดงการแปลงที่ให้ (( ) ( )) (( )) โดยใช้ $\Rightarrow^*$

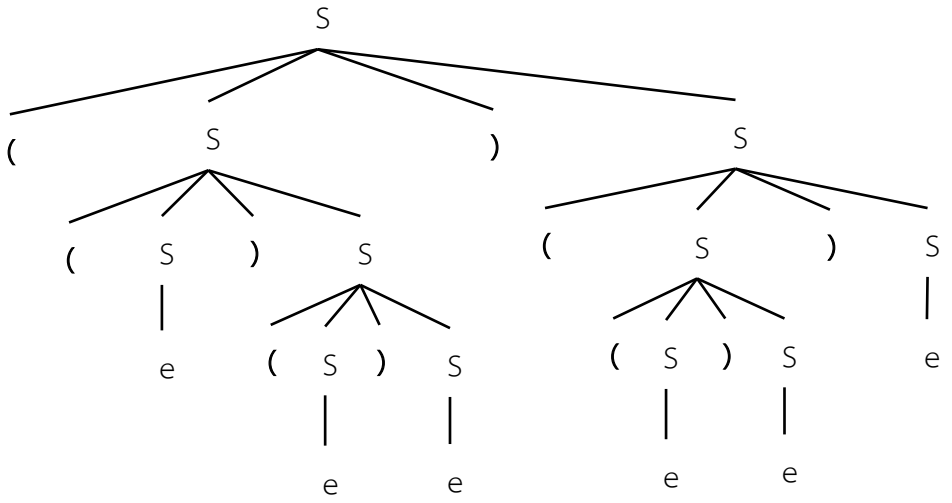
$S \Rightarrow (S) S \Rightarrow ((S) S) S \Rightarrow (( ) S) S$
$\Rightarrow (( ) (S) S) S \Rightarrow^* (( ) ( )) S \Rightarrow (( ) ( )) (S) S$
$\Rightarrow (( ) ( )) ((S) S) S \Rightarrow^* (( ) ( )) (( ))$

ขั้นตอนการแปลงสามารถแสดงได้เป็นต้นไม้ที่มีรากเป็นสัญลักษณ์เริ่มต้น การแตกกิ่งของโหนดภายในแต่ละโหนดเป็นไปตามการแปลงด้วยกฎในไวยากรณ์หนึ่งครั้งซึ่งโหนดพ่อเป็นสัญลักษณ์ทางซ้ายมือของกฎและโหนดลูกเป็นสัญลักษณ์ที่อยู่ทางขวามือของกฎ ต้นไม้นี้เรียกว่า ต้นไม้แจงส่วน (parse tree)

กำหนดไวยากรณ์ G = (V, T, P, S) และสตริง w ที่สร้างจากไวยากรณ์ G **ต้นไม้แจงส่วน**ของ w มีสมบัติดังนี้

- รากของต้นไม้นี้เป็นโหนดที่แทนสัญลักษณ์เริ่มต้น S,
- โหนดภายในแทนสัญลักษณ์ปลาย,
- โหนดใบแทนสัญลักษณ์ปลายซึ่งเมื่อเรียงสัญลักษณ์ของโหนดใบจากซ้ายไปขวาจะได้ w, และ
- สำหรับโหนดภายในที่แทนสัญลักษณ์ X ลูกของเป็นโหนดนั้นแทนสัญลักษณ์ K L ... M โดย $X \rightarrow K L \ldots M$ อยู่ใน P

ตัวอย่าง 3.5 ต้นไม้ข้างล่างนี้เป็นต้นไม้การแจงส่วนของสตริง (( ) ( )) (( )) ที่แสดงการแปลงในตัวอย่าง 3.4



ภาษาที่สร้างจากไวยากรณ์คือเซตของสตริงที่ได้จากการแปลงสัญลักษณ์เริ่มต้นตามกฎในไวยากรณ์ นั่นคือ

ภาษาที่สร้างจากไวยากรณ์ G = (V, T, P, S) เขียนแทนด้วย L(G) คือ {w| S $\Rightarrow^*$ w}

ตัวอย่าง 3.6 กำหนดไวยากรณ์ G1 = ({S}, { (,)}, {S $\rightarrow$ (S) S, S $\rightarrow$ e}, S)

L(G1) = {w| w เป็นสตริงที่มีวงเล็บเปิดและปิดที่จับคู่กันได้อย่างถูกต้อง}

จาก S $\rightarrow$ e จะเห็นว่าสตริงเล็กสุดที่สร้างจาก S เป็นสตริงว่าง นั่นคือหากจับคู่วงเล็บเปิดและปิดที่จะจับคู่กันได้ถูกต้อง จาก S $\rightarrow$ (S) S ถ้า S ทางขวาทั้ง 2 ตัวเป็นสตริงที่มีวงเล็บเปิดและปิดที่จับคู่กันได้อย่างถูกต้อง จะเห็นว่าสตริงที่สร้างตามกฎนี้จะต้องมีวงเล็บเปิดและปิดที่จับคู่กันได้อย่างถูกต้องด้วย

ตัวอย่าง 3.7 กำหนดไวยากรณ์ G2 = ({ E, A }, { id, num, +, -, *, / }, P, E) โดยที่ P = { E $\rightarrow$ E A E, E $\rightarrow$ id, E $\rightarrow$ num, A $\rightarrow$ +, A $\rightarrow$ -, A $\rightarrow$ *, A $\rightarrow$ / }

L(G2) เป็นเซตของสตริงที่เป็นนิพจน์ทางคณิตศาสตร์ที่มีตัวกระทำ +, -, *, / และตัวถูกกระทำเป็นโทเคน id และ num

จาก E $\rightarrow$ id, E $\rightarrow$ num จะเห็นว่านิพจน์เล็กสุดคือ id หรือ num จาก E $\rightarrow$ E A E นิพจน์สร้างจาก 2 นิพจน์เชื่อมด้วยตัวกระทำ +, -, *, /

ตัวอย่าง 3.8 กำหนดไวยากรณ์ G3 = ({ K, R, E, A }, { id, num, +, -, *, /, >, <, == }, P, K) โดยที่ P = { K $\rightarrow$ E R E, E $\rightarrow$ E A E, E $\rightarrow$ id, E $\rightarrow$ num, A $\rightarrow$ +, A $\rightarrow$ -, A $\rightarrow$ *, A $\rightarrow$ /, R $\rightarrow$ >, R $\rightarrow$ <, R $\rightarrow$ == }

L(G3) เป็นเซตของสตริงที่เป็นนิพจน์การเปรียบเทียบนิพจน์ทางคณิตศาสตร์ด้วยตัวกระทำ >, <, ==

ในไวยากรณ์นี้ E เป็นชื่อ (id) หรือจำนวน (num) จาก K → E R E นิพจน์การเปรียบเทียบสร้างจากนิพจน์ทางคณิตศาสตร์เชื่อมด้วยตัวกระทำเปรียบเทียบ >, <, ==

<u>ตัวอย่าง 3.9</u> กำหนดไวยากรณ์ G4 = ({ S, F, As, K, R, E, A }, { if, else, id, num, =, >, <, == }, P, S) โดยที่ P = { S → F, S → As, F → if K S, F → if K S else S, As → id = E, K → E R E, E → id, E → num, R → >, R → <, R → == }

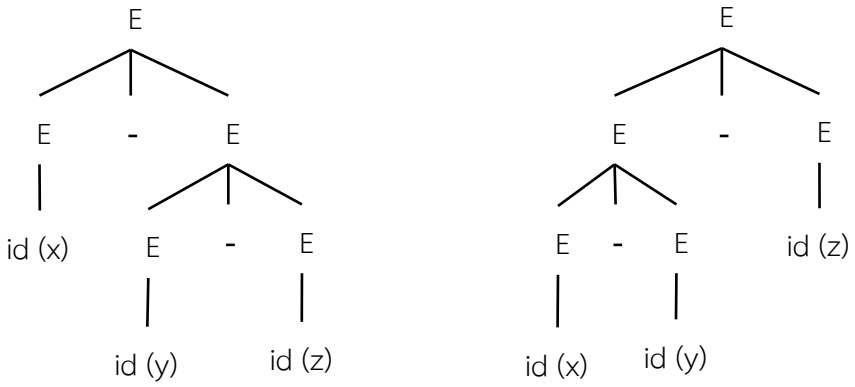
L(G4) เป็นเซตของสตริงที่เป็นคำสั่ง if หรือคำสั่งกำหนดค่า (assignment)

จาก F → if K S, F → if K S else S F เป็นคำสั่ง if ที่ประกอบด้วยโทเคน if, นิพจน์เปรียบเทียบ, คำสั่งใดๆ ที่เป็น then part และอาจมี else part หรือไม่ก็ได้ else part ประกอบด้วยโทเคน else และคำสั่งใดๆ จาก As → id = E As เป็นคำสั่งกำหนดค่าที่ประกอบด้วยโทเคน id, โทเคน = และนิพจน์ทางคณิตศาสตร์ สุดท้าย S เป็นคำสั่ง if หรือคำสั่งกำหนดค่าจาก S → F, S → As

3.2.2 ไวยากรณ์กำกวม

เราสามารถอธิบายไวยากรณ์ของภาษาโปรแกรมได้ดังแสดงในหัวข้อ 3.2.1 ไวยากรณ์ G3 และ G4 เป็นไวยากรณ์ที่เรียกว่าไวยากรณ์กำกวม (ambiguous grammar) เนื่องจากเราสามารถสร้างต้นไม้แจงส่วนมากกว่าหนึ่งแบบสำหรับสายของโทเคนหนึ่ง ไวยากรณ์กำกวมทำให้เกิดปัญหาในการสร้างตัวแปลภาษา เนื่องจากโปรแกรมเมอร์ที่สร้างตัวแปลภาษา 2 ตัวอาจเลือกสร้างต้นไม้ที่ต่างกันและทำให้โปรแกรมที่ได้จากตัวแปลภาษา 2 ตัวทำงานต่างกัน ตัวอย่างต่อไปนี้แสดงต้นไม้แจงส่วน 2 แบบของนิพจน์ทางคณิตศาสตร์หนึ่ง

<u>ตัวอย่าง 3.10</u> สำหรับนิพจน์ `x-y-z` ซึ่ง `x`, `y`, `z` เป็นโทเคน id สายของโทเคนที่สแกนเนอร์ส่งให้ตัวแจงส่วนคือ id – id – id เราสามารถสร้างต้นไม้แจงส่วนของนิพจน์นี้ตามไวยากรณ์ G3 ได้ 2 แบบดังนี้



หาก x, y, z มีค่า 8, 5, 2 ตามลำดับ ต้นไม้ทางซ้ายทำให้นิพจน์นี้ให้ค่า (8-(5-2)) = 5 ส่วนต้นไม้ทางขวาทำให้นิพจน์นี้ให้ค่า ((8-5)-2) = 1

ตัวอย่างต่อไปแสดงต้นไม้แจงส่วน 2 แบบของคำสั่ง if ที่ซ้อนกัน

ตัวอย่าง 3.11 สำหรับคำสั่ง if ที่ซ้อนกันดังนี้

```
if x==y if x==0 y=0 else x=0
```

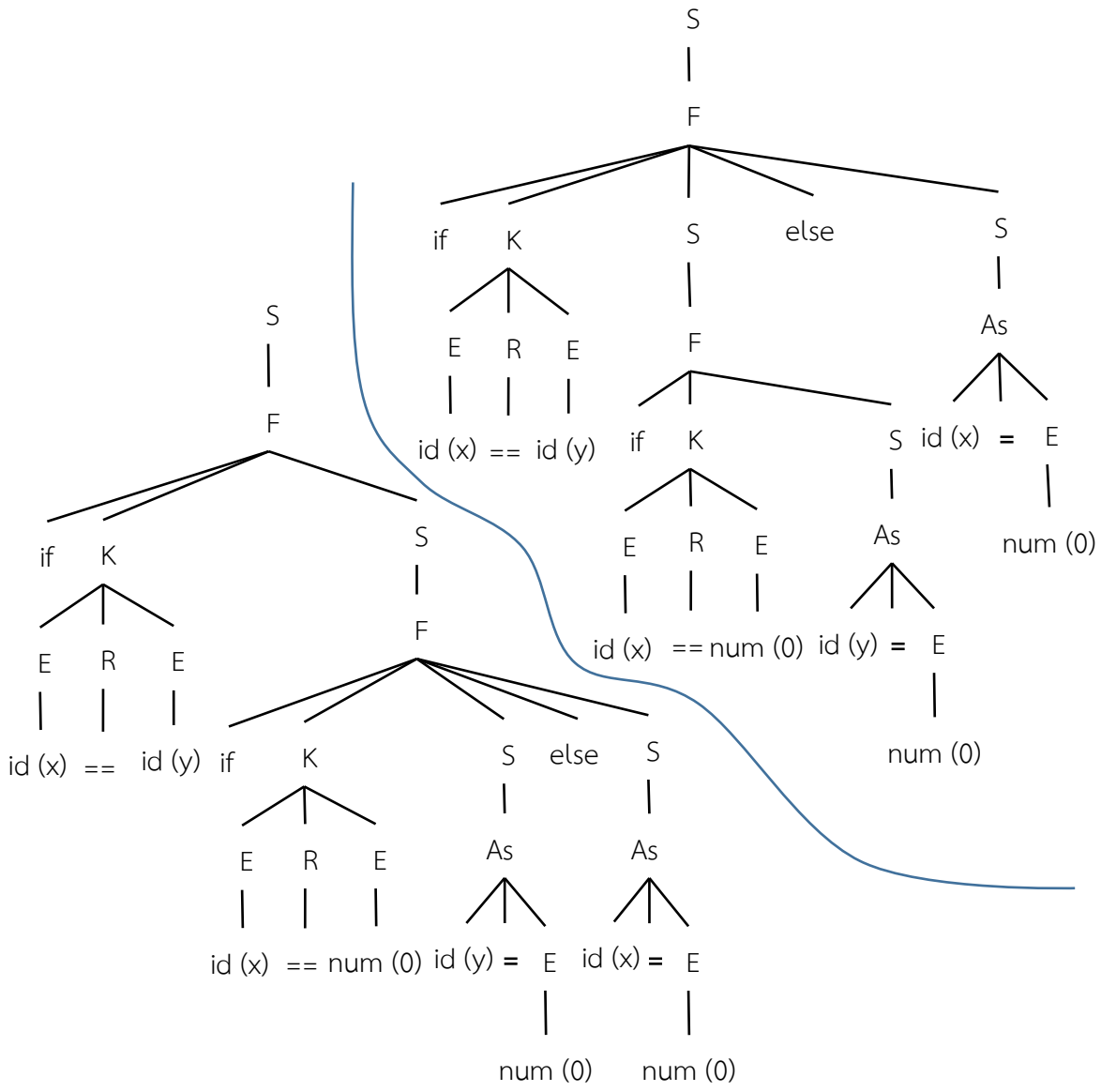
ซึ่ง x, y, z เป็นโทเคน id เราสามารถสร้างต้นไม้แจงส่วนของนิพจน์นี้ตามไวยากรณ์ G4 ได้ 2 แบบ ต้นไม้แบบแรกจับคู่ else กับ if ตัวแรกดังนี้

```
if x==y   {if x==0 y=0}   else x=0
```

ต้นไม้แบบที่สองจับคู่ else กับ if ตัวที่สองดังนี้

```
if x==y   {if x==0 y=0 else x=0}
```

ต้นไม้แจงส่วนทั้งสองแบบแสดงในรูปต่อไปนี้



เนื่องจากเราต้องการให้ตัวแปลภาษาทุกตัวแปลโปรแกรมแล้วทำงานได้ผลลัพธ์เหมือนกัน ไวยากรณ์ของภาษาต้องไม่กำกวม เราจะอธิบายการเขียนไวยากรณ์ของภาษาให้ไม่กำกวมในหัวข้อต่อไป

3.2.3 การเขียนไวยากรณ์เพื่อบังคับลำดับการทำงานของตัวกระทำในนิพจน์

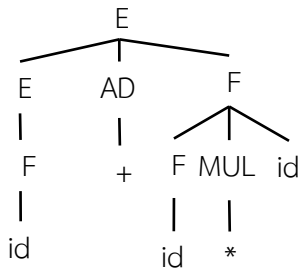
หัวข้อ 3.1.1 อธิบายการกำหนดลำดับการทำงานของตัวกระทำในภาษาโปรแกรมด้วยลำดับความสำคัญและสมบัติการเชื่อมโยง ในหัวข้อนี้ เราจะใช้ลำดับการทำงานของตัวกระทำที่กำหนดในหัวข้อ 3.1.1 แก้ความกำกวมของไวยากรณ์ G3 โดยเขียนไวยากรณ์ให้บังคับลำดับการทำงานของตัวกระทำตามลำดับความสำคัญและสมบัติการเชื่อมโยง การบังคับลำดับทำได้โดยสร้างสัญลักษณ์ไม่ปลายสำหรับนิพจน์ที่มีตัวกระทำที่มีลำดับความสำคัญต่าง ๆ กันดังตัวอย่างต่อไปนี้

<u>ตัวอย่าง 3.12</u> เราสามารถแก้ไวยากรณ์ G3 ในตัวอย่าง 3.8 ให้กำหนดลำดับความสำคัญของการคูณและหารสูงกว่าการบวกและลบได้ดังแสดงในไวยากรณ์ G5 = ({E, AD, F, MUL }, { id, num, +, -, *, / }, P, E) โดยที่ P = { E → E AD E, E → F, AD → +, AD → -, F → F MUL F, F → id, F → num, MUL → *, MUL → / }

ในภาษานี้ ตัวกระทำ + และ – มีลำดับความสำคัญเท่ากันแต่น้อยกว่า * และ / เราให้ E เป็นสัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำ +, –, * และ / ส่วน F เป็นสัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำ * และ / เท่านั้น

จากกฎ F → F MUL F และ F → id, F → num จะเห็นว่า F เป็นสัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำ * และ / แต่ไม่มีตัวกระทำ + และ – ส่วนกฎ E → E AD E ระบุว่า E เป็นสัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำ + และ – แต่จาก E → F ทำให้ E เป็นสัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำ * และ / ได้ด้วย ดังนั้น E เป็นสัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำ +, -, * และ / เนื่องจาก E ⇒ F ได้ แต่ F ⇒* E ไม่ได้ โหนด F จะมีโหนด E เป็นลูกหลานไม่ได้ ดังนั้นตัวกระทำ * และ / ต้องอยู่ใกล้โหนดใบมากกว่าตัวกระทำ + และ –

นิพจน์ `x+y*z` ที่ทำให้ได้สายของโทเคน id + id * id และต้นไม้แจงส่วนของ id + id * id แสดงในต้นไม้ข้างล่างนี้



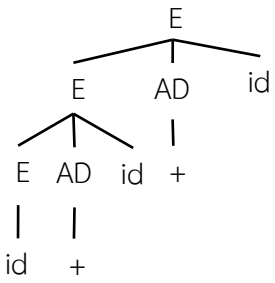
ถ้าต้องการให้ทำ + ก่อน * เราต้องสร้างจาก E ⇒ F ⇒ F MUL F แต่เนื่องจากเราไม่สามารถสร้าง id + id จาก F ได้ ดังนั้นนิพจน์จะทำ + ก่อน * ไม่ได้

นั่นคือ เราสามารถกำหนดลำดับความสำคัญของตัวกระทำได้โดยแยกใช้สัญลักษณ์ที่สร้างนิพจน์ที่มีตัวกระทำที่มีลำดับความสำคัญต่าง ๆ กัน แล้วให้สัญลักษณ์ที่แทนนิพจน์ที่สร้างด้วยตัวกระทำที่มีลำดับความสำคัญน้อยกว่าแปลงเป็นสัญลักษณ์ที่แทนนิพจน์ที่สร้างด้วยตัวกระทำที่มีลำดับความสำคัญมากกว่าได้

นอกจากลำดับความสำคัญแล้ว เรายังใช้สมบัติการเชื่อมโยงเพื่อบังคับลำดับการทำงานในนิพจน์ เราสามารถเขียนไวยากรณ์ที่กำหนดสมบัติการเชื่อมโยงของตัวกระทำโดยเลือกให้สัญลักษณ์ที่อยู่ทางซ้ายหรือขวาของตัวกระทำนั้นเป็นสัญลักษณ์ที่แทนนิพจน์ชนิดเดิมดังแสดงในตัวอย่างนี้

ตัวอย่าง 3.13 เราสามารถเขียนไวยากรณ์ให้กำหนดสมบัติความเชื่อมโยงของตัวกระทำ + และ – เป็นแบบเชื่อมโยงทางซ้ายได้ดังแสดงในไวยากรณ์ G6 = ({ E, AD }, { id, num, +, - }, P, E) โดยที่ P = { E → E AD id, E → E AD num, E → id, E → num, AD → +, AD → - }

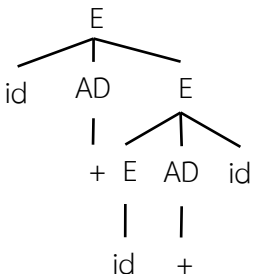
จาก E → E AD id จะเห็นว่าเราสามารถสร้างต้นไม้แจงส่วนของ id + id + id ดังแสดงในรูปข้างล่าง ในต้นไม้นี้ จะบวกด้วยตัวกระทำ + ที่อยู่ทางซ้ายของนิพจน์ก่อน ดังนั้นตัวกระทำนี้มีสมบัติการเชื่อมโยงทางซ้าย



ไวยากรณ์ในตัวอย่างต่อไปนี้กำหนดให้ตัวกระทำ + และ - มีสมบัติการเชื่อมโยงทางขวา

ตัวอย่าง 3.14 กำหนดไวยากรณ์ G7 = ({ E, AD }, { id, num, +, - }, P, E) โดยที่ P = { E → id AD E, E → num AD E, E → id, E → num, AD → +, AD → - }

จาก E → id AD E จะเห็นว่าเราสามารถสร้างต้นไม้แจงส่วนของ id + id + id ดังแสดงในรูปข้างล่าง ในต้นไม้นี้ จะบวกด้วยตัวกระทำ + ที่อยู่ทางขวาของนิพจน์ก่อน ดังนั้นตัวกระทำนี้มีสมบัติการเชื่อมโยงทางขวา



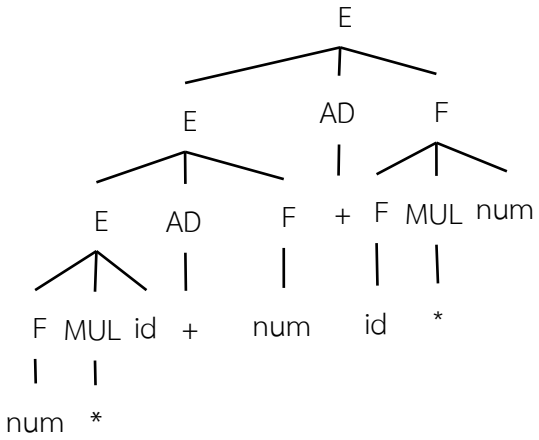
เราสามารถเขียนไวยากรณ์ที่กำหนดทั้งลำดับความสำคัญและสมบัติการเชื่อมโยงได้ดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 3.15 กำหนดไวยากรณ์ G8 = ({E, AD, F, MUL }, { id, num, +, -, *, / }, P, E) โดยที่ P = { E → E AD F, E → F, AD → +, AD → -, F → F MUL id, F → F MUL num, F → id, F → num, MUL → *, MUL → / }

การแปลงที่ให้สายของโทเคน num * id + num + id * num แสดงได้ดังนี้

E ⇒ E AD F ⇒ E AD F AD F ⇒ F AD F AD F
⇒ F MUL id AD F AD F ⇒ num MUL id AD F AD F ⇒ num * id AD F AD F
⇒ num * id AD F AD F ⇒ num * id + F AD F ⇒ num * id + num AD F
⇒ num * id + num + F ⇒ num * id + num + F MUL num
⇒ num * id + num + id MUL num
⇒ num * id + num + id * num

ต้นไม้แจงส่วนข้างล่างนี้แสดงการแปลงของ num * id + num + id * num



3.2.4 การเขียนไวยากรณ์เพื่อกำหนดการจับคู่ของ else กับ if

จากหัวข้อ 3.1.2 ภาษาโปรแกรมกำหนดให้ else จับคู่กับ if ก่อนหน้าที่ใกล้สุดที่ยังไม่มี else คู่ด้วย เราสามารถแก้ปัญหาไวยากรณ์กำกวมสำหรับคำสั่ง if ได้โดยเขียนไวยากรณ์ให้บังคับการจับคู่ของ else กับ if ดังที่กำหนดนี้ เราสร้างสัญลักษณ์ Matched สำหรับ คำสั่ง if ที่มี else จบคู่กันได้ครบและสัญลักษณ์ Unmatched สำหรับคำสั่ง if ที่มี else จับคู่กันได้ไม่ครบ ดังตัวอย่างต่อไปนี้

<u>ตัวอย่าง 3.16</u> เราสามารถแก้ไวยากรณ์ G4 ในตัวอย่าง 3.9 ที่เป็นไวยากรณ์กำกวมให้กำหนดการจับคู่ else กับ if โดยแยกคำสั่งเป็น 2 ประเภท คือ คำสั่งถ้ามี if แล้วจะมี else จับคู่กับ if ได้ครบ และ คำสั่งที่มี else จับคู่กับ if ไม่ครบ คำสั่งแบบแรกแทนด้วยสัญลักษณ์ Matched และแบบหลังแทนด้วยสัญลักษณ์ Unmatched คำสั่งที่แทนด้วย Matched ได้แก่

```
x=0,
if x==0 x=x+1 else x=0,
if x==0 x=x+1 else if x>0 x=x-1 else x=x+1
```

คำสั่งเหล่านี้อธิบายได้ด้วยกฎ Matched → As และ Matched → if K Matched else Matched

คำสั่งที่แทนด้วย Unmatched ได้แก่

```
if x==0 x=x+1,
if x>=0 if x>0 x=x-1 else x=x+1
```

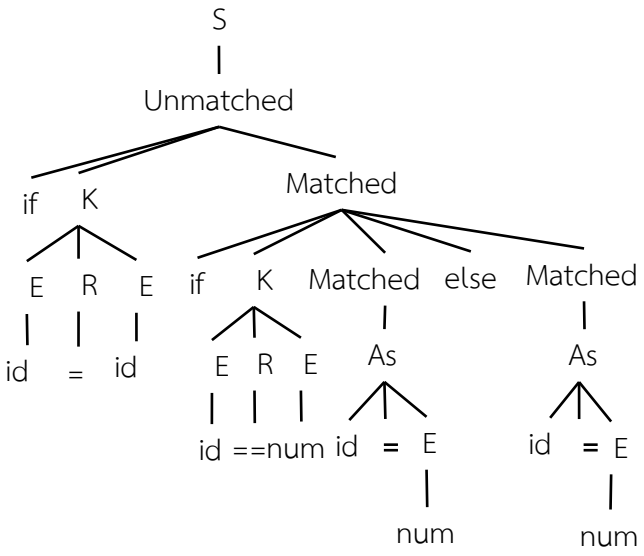
คำสั่งเหล่านี้อธิบายได้ด้วยกฎ Unmatched → if K Matched และ Unmatched → if K Matched else Unmatched กฎทั้งสองนี้บังคับให้จับคู่ else กับ if ที่ได้จาก Matched จากข้างในก่อนและปล่อยให้ if ที่เหลือท้ายสุดไม่มี else

ไวยากรณ์ G9 ต่อไปนี้ไม่กำกวมเพราะบังคับให้ else จับคู่กับ if ก่อนหน้าที่ใกล้สุดที่ยังไม่จับคู่กับ else ใด G9 = ({ S, Matched, Unmatched, As, K, R, E }, { if, else, id, num, =, >, <, == }, P, S) โดยที่ P = { S → Matched, S → Unmatched, Matched → As, Matched → if K Matched else Matched, Unmatched → if K Matched, Unmatched → if K Matched else Unmatched, As → id = E, K → E R E, E → id, E → num, R → >, R → <, R → == }

คำสั่ง `if x==y if x==0 y=0 else x=0` ที่เป็นสายของโทเคน if id == id if id == num id = num else id = num สายของโทเคนนี้สร้างด้วยไวยากรณ์ G9 ได้ดังนี้

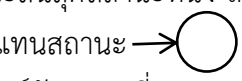
S ⇒ Unmatched (หากใช้กฎ S → Matched จะแปลงจนจบไม่ได้)
⇒ if K Matched (ใช้ Unmatched → if K Matched else Unmatched แล้วมี else เกิน)
⇒ if K if K Matched else Matched
⇒ if E R E if K Matched else Matched
⇒* if id == id if K Matched else Matched
⇒ if id == id if E R E Matched else Matched
⇒* if id == id if id == num Matched else Matched
⇒* if id == id if id == num As else Matched
⇒* if id == id if id == num id = num else Matched
⇒ if id == id if id == num id = num else As
⇒* if id == id if id == num id = num else id = num

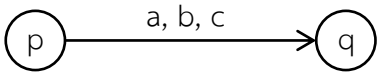
การแปลงนี้แสดงด้วยต้นไม้แจงส่วนข้างล่างที่ให้ else จับคู่กับ if ตัวก่อนหน้าที่ยังไม่มี else



3.3 ออโตมาตาพูชดาว์น

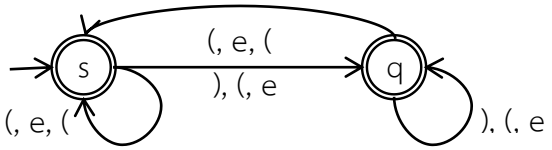
จากการศึกษาด้านภาษาและออโตมาตา ภาษาที่อธิบายด้วยไวยากรณ์ไม่พึ่งบริบทสามารถตรวจสอบด้วยออโตมาตาพชูดาว์นได้ (ในทำนองเดียวกับที่ภาษาที่อธิบายด้วยนิพจน์ปกติสามารถตรวจสอบด้วยออโตมาตาจำกัดได้) นอกจากนั้นการทำงานของตัวแจงส่วนบางประเภทก็เป็นการจำลองการทำงานของออโตมาตาพูชดาว์น ออโตมาตาพูชดาว์นคล้ายกับออโตมาตาจำกัดคือมีส่วนควบคุมที่มีสถานะเป็นจำนวนจำกัด มีเทปซึ่งเก็บสตริงที่ต้องการตรวจสอบโดยอ่านจากซ้ายไปขวาเท่านั้น แต่ส่วนที่ออโตมาตาพูชดาว์นมีเพิ่มขึ้นมา คือ สแตค (stack) ที่สามารถเก็บสัญลักษณ์ใดใดที่ต้องการได้โดยอ่านและเขียนข้อมูลตัวบนสุดของสแตคเท่านั้น ออโตมาตาพูชดาว์น มีสองแบบที่ต่างกันตามชนิดของการเปลี่ยนสถานะ ออโตมาตาพูชดาว์นแบบกำหนดมีการเปลี่ยนสถานะแบบกำหนด คือ เมื่อเครื่องอยู่ในสถานะหนึ่ง อ่านได้สัญลักษณ์หนึ่งบนเทปและอ่านได้สัญลักษณ์หนึ่งจากสแตคแล้ว เครื่องมีสถานะถัดไปที่เป็นไปได้เพียงสถานะเดียว ส่วนออโตมาตาพูชดาว์นแบบไม่กำหนดมีการเปลี่ยนสถานะแบบไม่กำหนด คือ เมื่อเครื่องอยู่ในสถานะหนึ่ง อ่านได้สัญลักษณ์หนึ่งบนเทปและอ่านได้สัญลักษณ์หนึ่งจากสแตคแล้ว เครื่องอาจมีสถานะถัดไปที่เป็นไปได้มากกว่าหนึ่งสถานะ หรือเลือกเปลี่ยนสถานะโดยไม่อ่านสัญลักษณ์บนเทปและสแตคเลย เครื่องจะเลือกเปลี่ยนสถานะไปยังสถานะที่ทำให้ออโตมาตายอมรับสตริงรับเข้าหากเป็นไปได้ แต่ในการสร้างตัวแจงส่วนใช้ออโตมาตาพูชดาว์นแบบกำหนดเท่านั้น

ออโตมาตาพูชดาว์นนิยามได้ด้วย (Q, S, T, d, s, F) ซึ่ง Q เป็นเซตจำกัดของสถานะที่เป็นไปได้ของออโตมาตาโดยมี s เป็นสถานะเริ่มต้นและ F เป็นเซตของสถานะสิ้นสุด S และ T เป็นเซตจำกัดของสัญลักษณ์ที่ใช้บนเทปและสแตคตามลำดับ d เป็นฟังก์ชั่นเปลี่ยนสถานะโดย d(p, a, b) = (q, c) หมายความว่าเมื่อออโตมาตาอยู่ในสถานะ p และอ่านได้สัญลักษณ์ a จากเทปและสัญลักษณ์ b จากสแตค แล้วออโตมาตาจะเปลี่ยนสถานะไปยังสถานะ q และ เขียน c ลงบนสแตคแทน b ดังนั้นหาก c เป็นสตริงว่าง หมายความว่า b จะถูก pop ออกจากสแตค และหาก c เป็น db หมายความว่า d จะถูกใส่เพิ่มบนสแตค ออโตมาตาพูชดาว์นจะหยุดทำงานเมื่ออ่านสตริงบนเทปหมดและสแตคว่าง หากเครื่องจบการทำงานที่สถานะสิ้นสุดสถานะหนึ่ง เครื่องจะยอมรับสตริงรับเข้า ออโตมาตาพูชดาว์น แสดงเป็นแผนภาพได้โดยให้  แทนสถานะ  แทนสถานะเริ่มต้น  แทนสถานะสิ้นสุด และเส้นเชื่อมระหว่างสถานะอธิบายฟังก์ชันการเปลี่ยนสถานะดังรูปข้างล่างที่แสดง d(p, a, b) = (q, c)



ตัวอย่างต่อไปนี้เป็นออโตมาตาพูชดาว์นที่ยอมรับสตริงที่มีสัญลักษณ์เปิดวงเล็บและปิดวงเล็บที่จับคู่กันได้อย่างถูกต้อง

ตัวอย่าง 3.17 กำหนด M = ({s, q}, { (,) }, { (,) }, d, s, {s, q}) โดยที่ d เป็นฟังก์ชันเปลี่ยนสถานะที่แสดงในรูปข้างล่าง



เราใช้ (q, x, y) เป็น**โครงแบบ (configuration)** ของออโตมาตาพูชดาว์นที่บอกว่า q เป็นสถานะของเครื่องในขณะนั้นโดย x เป็นสตริงบนเทปที่ยังไม่ได้อ่านและ y เป็นสตริงในสแตค นอกจากนั้น (p, v, x) |- (q, w, y) แสดง**การเปลี่ยนจากโครงแบบ** (p, v, x) **ไปยัง**โครงแบบ (q, w, y) ในหนึ่งขั้น เมื่อ w=av และมี d(p, a, b) = (q, c) โดย x = bz, y = cz สำหรับบางสตริง z ตัวอย่างต่อไปนี้แสดงการเปลี่ยนโครงแบบของออโตมาตาในตัวอย่าง 3.17

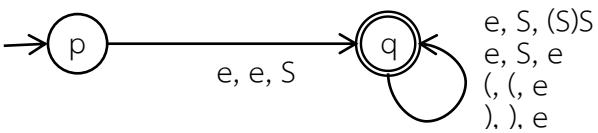
ตัวอย่าง 3.18 สมมติให้สตริงรับเข้าบนเทปของออโตมาตาในตัวอย่าง 3.17 เป็น ()(()(())) ออโตมาตาทำงานโดยมีการเปลี่ยนโครงแบบดังต่อไปนี้

(s, ()(()(())), e) |- (q,)(()(())), ( ) |- (s , (()(())), e) |- (q, ()(())), ( ) |- (q,)(())), (( ) |- (s, (())), ( )
|- (s, ())), (( ) |- (s,))), ((( ) |- (q,)), (( ) |- (q,), ( ) |- (q, e, e)

ออโตมาตาพูชดาว์น M ยอมรับสตริง w ก็ต่อเมื่อให้ M เริ่มทำงานด้วยสแตคว่างและมีสตริง w บนเทปแล้วเมื่อ M อ่านสตริง w หมดจะจบการทำงานที่สถานะสิ้นสุดและสแตคว่าง จากตัวอย่าง 3.18 เมื่อให้ ()(()(())) เป็นสตริงที่อยู่บนเทปเมื่อ M เริ่มต้นทำงาน แล้วเมื่อ M อ่านสตริงบนเทปหมด M อยู่ในโครงแบบ (q, e, e) ที่ q เป็นสถานะสิ้นสุด ดังนั้น M ยอมรับ ()(()(())) สุดท้ายเรานิยามให้ภาษาที่ยอมรับด้วยออโตมาตาหนึ่งเป็นเซตของสตริงที่ออโตมาตานั้นยอมรับ ดังนั้นภาษาที่ยอมรับด้วยออโตมาตาในตัวอย่าง 3.17 เป็นเซตของสตริงที่มีวงเล็บเปิดและวงเล็บปิดจับคู่กันได้อย่างถูกต้อง

ในการสร้างตัวแจงส่วน เราใช้ออโตมาตาพูชดาว์นที่ใช้ในการพิสูจน์ว่าออโตมาตาพูชดาว์นสามารถยอมรับภาษาที่สร้างด้วยไวยากรณ์ไม่พึ่งบริบทได้ ออโตมาตาพูชดาว์นนี้สร้างแบบประโยคที่ได้จากการแปลงเก็บไว้ในสแตคและตรวจว่าแบบประโยคที่อยู่ในสแตคมีส่วนหน้าที่ตรงกับส่วนหน้าของสตริงบนเทปหรือไม่ และ pop สัญลักษณ์ที่ตรวจสอบได้แล้วออกจากสแตค หากสัญลักษณ์ที่อยู่บนสุดของสแตคเป็นสัญลักษณ์ไม่ปลายในไวยากรณ์ ออโตมาตาจะแปลงสัญลักษณ์นั้นตามกฎในไวยากรณ์ เช่น ถ้า X อยู่บนสุดของสแตคและมีกฎ X → ab ... c ออโตมาตาจะแทน X ด้วย ab ... c ตัวอย่างต่อไปนี้แสดงการสร้างออโตมาตาพูชดาว์นจากไวยากรณ์ในหัวข้อ 3.2.1

ตัวอย่าง 3.19 จากไวยากรณ์ G1 = ({S}, { (,)}, {S → (S) S, S → e}, S) เราสามารถสร้างออโตมาตาพูชดาว์นที่ยอมรับ L(G1) ได้ดังแสดงในรูปข้างล่างนี้



เริ่มต้น ออโตมาตาต้องใส่สัญลักษณ์เริ่มต้นในสแตคดังแสดงด้วย d(p, e, e) = (q, S) จากนั้นที่สถานะ q เราจะตรวจสอบว่าสัญลักษณ์ที่สร้างตามไวยากรณ์ตรงกับสัญลักษณ์บนเทปด้วย d(q, (, ( ) = (q, e) และ d(q,),)) = (q, e) เราสร้างฟังก์ชันเปลี่ยนสถานะสำหรับกฎ S → (S) S และ S → e ใน G1 คือ d(q, e, S) = (q, (S)S) และ d(q, e, S) = (q, e) ตามลำดับ

รูปข้างล่างนี้แสดงการเปลี่ยนสถานะของออโตมาตาพูชดาว์นเทียบกับการแปลงด้วยไวยากรณ์ สัญลักษณ์ที่แสดงด้วยตัวหนาในการแปลงเป็นสัญลักษณ์ที่ถูก pop ออกจากสแตคแล้ว

การเปลี่ยนสถานะครั้งที่	**การเปลี่ยนสถานะของออโตมาตาพูชดาว์น**	**การแปลงด้วยไวยากรณ์**
	(p, ()((())()), e)	
1	|- (q, ()((())()), S)	S
2	|- (q, ()((())()), (S)S)	⇒ (S)S
3	|- (q,)((())()), S)S)	
4	|- (q,)((())()),)S)	⇒ **()**S
5	|- (q, (((())()), S)	
6	|- (q, (((())()), (S)S)	⇒ **()**(S)S
7	|- (q, ((())()), S)S)	
8	|- (q, ((())()), (S)S)S)	⇒ **()(**(S)S)S
9	|- (q, ())()), S)S)S)	
10	|- (q, ())()), (S)S)S)S)	⇒ **()(((**(S)S)S)S
11	|- (q,))()), S)S)S)S)	
12	|- (q,))()),)S)S)S)	⇒ **()(((**S)S)S
13	|- (q,)()), S)S)S)	
14	|- (q,)()),)S)S)	⇒ **()((()**S)S
15	|- (q, ()), S)S)	
16	|- (q, ()), (S)S)S)	⇒ **()((()**(S)S)S
17	|- (q,)), S)S)S)	
18	|- (q,)),)S)S)	⇒ **()((())()**S)S
19	|- (q,), S)S)	
20	|- (q,),)S)	⇒ **()((())()**)S
21	|- (q, e, S)	
22	|- (q, e, e)	⇒ **()((())())**

ในหัวข้อต่อไป เราจะอธิบายขั้นตอนวิธีแจงส่วนที่ใช้การทำงานของออโตมาตาพูชดาว์นที่กล่าวมานี้

3.4 ขั้นตอนวิธีแจงส่วนแบบ LL(1)

การแจงส่วนแบบ LL(1) อ่านสายของโทเคนจากซ้ายไปขวา (แสดงในชื่อLL(1) ด้วย L ตัวแรก) และเลียนแบบการทำงานของพูชดาว์นออโตมาตาซึ่งจำลองการแปลงซ้ายสุด (แสดงในชื่อLL(1) ด้วย L ตัวหลัง) วิธีนี้สร้างต้นไม้แจงส่วนจากบนลงล่าง (topdown) นอกจากนั้นวิธีนี้ดูโทเคนถัดไป 1 ตัวเพื่อเลือกกฎที่ใช้ใน

การแปลง (แสดงในชื่อLL(1) ด้วย 1) หัวข้อนี้อธิบายขั้นตอนวิธีแจงส่วนแบบ LL(1) พร้อมทั้งอธิบายหลักการที่ใช้ในขั้นตอนวิธีนี้

3.4.1 ขั้นตอนวิธีจำลองการทำงานของออโตมาตาพูชดาว์น

ตัวแจงส่วนทำหน้าที่ตรวจสอบไวยากรณ์ของโปรแกรมที่ต้องการแปลและสร้างต้นไม้แจงส่วนที่แยกองค์ประกอบของโปรแกรมเพื่อนำไปแปลเป็นรหัสเป้าหมาย เนื่องจากออโตมาตาพูชดาว์นสามารถเลียนแบบการแปลงสตริงตามไวยากรณ์ที่กำหนดได้ เราสามารถสร้างตัวแจงส่วนโดยเขียนโปรแกรมจำลองการทำงานของออโตมาตาพูชดาว์นดังที่กล่าวในหัวข้อก่อนหน้านี้ ขั้นตอนวิธีที่จำลองการทำงานมีดังนี้

```
1. push(d, S)                                  // S เป็นสัญลักษณ์เริ่มต้นและ d เป็นสแตค
2. t = scan_next_token()
3. WHILE d ไม่เป็นสแตคว่าง และ t ไม่เป็น null
       top = pop(d)
       IF top เป็นสัญลักษณ์ไม่ปลาย
           เลือกกฎ head → body ที่ head=top    // ทำการแปลง
           push(d, reverse(body))
       ELSE                                    // top เป็น สัญลักษณ์ปลาย
           IF t==top                           // token t ตรงกับสัญลักษณ์บนสุดในสแตค
               t = scan_next_token()           // ทิ้ง token เดิม
           ELSE                                // token t ไม่ตรงกับสัญลักษณ์บนสุดในสแตค
               BREAK                           // มีข้อผิดพลาด
4. IF d เป็นสแตคว่าง และ t เป็น null           // ตรวจสอบ token ตรงกับที่แปลงได้
       RETURN TRUE
   ELSE                                        // ตรวจสอบ token ไม่ตรงกับที่แปลงได้
       RETURN FALSE
```

scan_next_token() เป็นฟังก์ชันที่ทำหน้าที่เป็นสแกนเนอร์ที่อ่านโทเคน push และ pop เป็นฟังก์ชันที่ใส่สัญลักษณ์ลงบนสแตคและอ่านสัญลักษณ์บนสุดจากสแตค reverse เป็นฟังก์ชันที่เรียงสัญลักษณ์ในสตริงย้อนจากหลังไปหน้า

ขั้นตอนวิธีนี้จำลองการทำงานของออโตมาตาพูชดาว์นได้ดังแสดงในตัวอย่างนี้

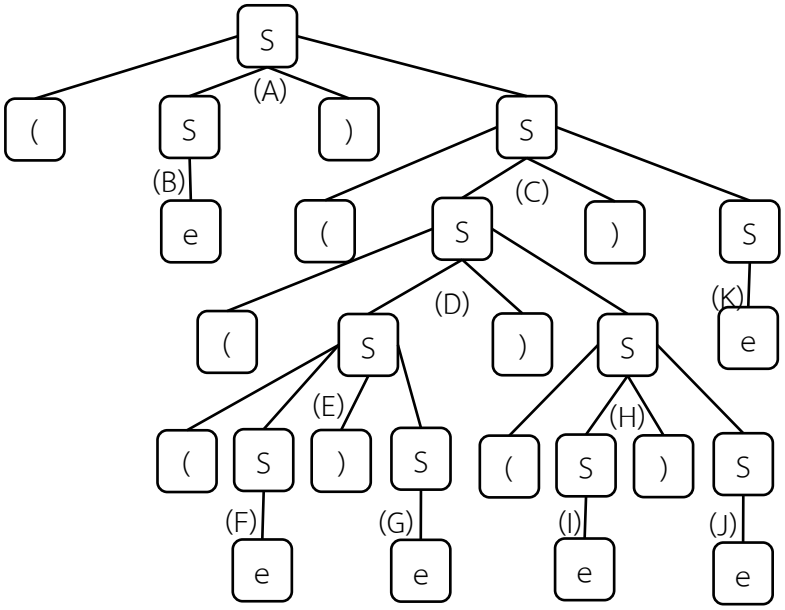
ตัวอย่าง 3.20 กำหนด S → (S) S, S → e เป็นกฎในไวยากรณ์ และให้สายของโทเคนเป็น ( ) ((( )) ( ))

ขั้นตอนวิธีจำลองการทำงานของออโตมาตาพูชดาว์นข้างบนเลียนแบบการทำงานของออโตมาตาในตัวอย่าง 3.19 ดังแสดงต่อไปนี้ ขั้นตอนวิธีนี้อ่านสายของโทเคนจากซ้ายไปขวาและสัญลักษณ์ที่เป็นตัวหนาเป็นโทเคนที่อ่านมาเก็บในตัวแปร t คอลัมน์ที่ 1 และ 2 ในตารางนี้แสดงค่าในสแตคและโทเคนที่อ่านเมื่อทำการ

วนซ้ำหนึ่งครั้ง โทเคนที่อ่านในขณะนั้นแสดงด้วยตัวอักขระเข้ม คอลัมน์ที่ 3 อธิบายการกระทำที่ทำให้ค่าในสแตคเปลี่ยน คอลัมน์ที่ 4 เทียบการทำงานแต่ละขั้นกับการทำงานของออโตมาตาพูชดาว์น

สแตค d (top ⇨ bottom)	**สายของโทเคน**	**การกระทำ**	**การเปลี่ยนสถานะ**
S	**(**)(()())()		
		(A) แทน S ด้วย (S) S และ pop (	ครั้งที่ 2-3
S) S	(**)**(()())()		
		(B) แทน S ด้วย e และ pop (	ครั้งที่ 4-5
S	()**(**()())()		
		(C) แทน S ด้วย (S) S และ pop (	ครั้งที่ 6-7
S) S	()(**(**)())		
		(D) แทน S ด้วย (S) S และ pop (	ครั้งที่ 8-9
S) S) S	(()((**(**))())		
		(E) แทน S ด้วย (S) S และ pop (	ครั้งที่ 10-11
S) S) S) S	(()(()**)**())		
		(F) แทน S ด้วย e และ pop)	ครั้งที่ 12-13
S) S) S	(()(())**)**()		
		(G) แทน S ด้วย e และ pop)	ครั้งที่ 14-15
S) S	(()(()**(**))		
		(H) แทน S ด้วย (S) S และ pop (	ครั้งที่ 16-17
S) S) S	(()(()**)**)		
		(I) แทน S ด้วย e และ pop)	ครั้งที่ 18-19
S) S	(()(())**)**		
		(J) แทน S ด้วย e และ pop)	ครั้งที่ 20-21
S	(()(())**)**		
		(K) แทน S ด้วย e	ครั้งที่ 22
empty	empty		

เราสามารถสร้างต้นไม้แจงส่วนจากบนลงล่างระหว่างการทำงานนี้ได้ดังแสดงในรูปข้างล่าง สัญลักษณ์แต่ละตัวในสแตคแทนโหนดในต้นไม้ เริ่มต้นเมื่อ push สัญลักษณ์เริ่มต้นตัวแรกมา เราให้สัญลักษณ์เริ่มต้นนั้นเป็นรากของต้นไม้ เมื่อเลือกกฎที่แปลงสัญลักษณ์บนสุดของสแตคและ push สตริงที่ใช้แทนในสแตค เราจะเพิ่มโหนดสำหรับสัญลักษณ์ในสตริงในต้นไม้ด้วย ดังนั้นเมื่อสแตคว่างละใช้โทเคนหมด ต้นไม้ที่ได้เป็นต้นไม้แจงส่วนของสายของโทเคนที่อ่านมา



จากตัวอย่างนี้จะเห็นว่าการแทน (A) ใช้กฎ S → (S)S แต่การแทน (B) ใช้กฎ S → e หากไม่เลือกใช้กฎตามนี้แล้วจะสร้างต้นไม้แจงส่วนไม่ได้ แต่ขั้นตอนที่แสดงนี้ ไม่ได้ระบุว่าเลือกใช้กฎใด ขั้นตอนวิธีแจงส่วนแบบ LL(1) กำหนดให้เลือกกฎโดยพิจารณาว่าสามารถสร้างสัญลักษณ์ถัดไปตรงกับโทเคนที่ต้องการได้

3.4.2 ขั้นตอนวิธีแจงส่วนแบบ LL(1)

ขั้นตอนวิธีแจงส่วนแบบ LL(1) ใช้กฎที่ทำการแปลงแล้วได้โทเคนที่ต้องการเป็นลำดับถัดไปซึ่งอธิบายด้วยแนวคิดของ first set กับ follow set ดังต่อไปนี้

First set

ถ้า N เป็นสัญลักษณ์ไม่ปลายแล้ว first set ของ N (เขียนแทนด้วย first(N)) เป็นเซตของสัญลักษณ์ปลายที่เป็นสัญลักษณ์แรกของสตริงที่ได้จากการแปลง N

ตัวอย่าง 3.21 กำหนดไวยากรณ์ S → (S) S, S → e

สตริงที่ได้จากการแปลง S ได้แก่ e, (), ()(), (()), (()()), ... ดังนั้นสัญลักษณ์ (ต้องอยู่ใน first(S) นอกจากนั้น e อยู่ใน first(S) เพราะ S ⇒* e ด้วย

เราสามารถนิยาม first(N) เมื่อ N เป็นสัญลักษณ์ไม่ปลายได้ดังนี้

ให้ N เป็นสัญลักษณ์ไม่ปลาย
first(N) = {a| N ⇒* aB a เป็นสัญลักษณ์ปลายและ B เป็นสตริงใด ๆ} ถ้า N ไม่สามารถแปลงเป็น e ได้
first(N) = {a| N ⇒* aB a เป็นสัญลักษณ์ปลายและ B เป็นสตริงใด ๆ} U {e} ถ้า N แปลงเป็น e ได้

เราสามารถขยายนิยามของ first ให้ใช้กับสัญลักษณ์ปลายได้ดังนี้

first(N) = { N } เมื่อ N เป็นสัญลักษณ์ปลาย

นอกจากนั้นเรายังสามารถนิยาม first set ของสตริงได้ดังนี้

ถ้า N เป็นสตริงว่าง แล้ว first(N) = {e}

ถ้า N = xY เป็นสตริง และ x เป็นสัญลักษณ์ใด ๆ แล้ว

- first(x)-{e} $\subseteq$ first(N)
- first(Y) $\subseteq$ first(N) ถ้า e $\in$ first(x)
- e $\in$ first(N) ถ้า e $\in$ first(x) และ e $\in$ first(Y)
- สัญลักษณ์อื่นที่ไม่ได้ระบุในเงื่อนไข 3 ข้อนี้ ไม่อยู่ใน first(N)

นอกจากนั้น เราสามารถนิยาม first set จากกฎในไวยกรณ์ได้ดังนี้

ให้ N เป็นสัญลักษณ์ไม่ปลาย และ N $\rightarrow$ X_1 X_2 ... X_n โดยที่ X_1 X_2 ... และ X_n เป็นสัญลักษณ์ใด ๆ

- first(X_1) – {e} $\subseteq$ first(N)
- สำหรับ 1<i<n first(X_i) – {e} $\subseteq$ first(N) ถ้า e $\in$ first(X_1 X_2 ... X_{i-1})
- e $\in$ first(N) ถ้า e $\in$ first(X_1 X_2 ... X_n)

จากนิยามของ first set นี้ เราสามารถสร้างขั้นตอนวิธีการหา first set ได้ดังนี้

<u>ขั้นตอนวิธีการหา first set</u>

1. FOR EACH non-terminal X:
   first(X) = {}
2. FOR EACH production L $\rightarrow$ e :
   first(L) = {e}
3. DO
   FOR EACH production L $\rightarrow$ R_1 R_2 ... R_n :
   i = 1
   emptyPrefix = TRUE
   WHILE emptyPrefix AND i $\leq$ n :
   first(L) = first(L) $\cup$ first(R_i) – {e}
   emptyPrefix = emptyPrefix AND (e $\in$ first(R_i))
   i = i+1
   IF emptyPrefix :
   first(L) = first(L) $\cup$ {e}

   WHILE there is a change in first(X), for some nonterminal X

ตัวอย่างต่อไปนี้แสดงการคำนวณหา first set ด้วยขั้นตอนวิธีข้างบน

ตัวอย่าง 3.22 กำหนดไวยากรณ์ G10 ที่มีกฎต่อไปนี้ E → F T, T → AD F T, T → e, AD → +, AD → -, F → L K, K → MUL L K, K → e, MUL → *, MUL → /, L → (E), L → id เราคำนวณหา first set ของสัญลักษณ์ไม่ปลายตามขั้นตอนวิธีข้างบนนี้ได้ดังนี้

ขั้นที่ 1 กำหนดให้ first set ของสัญลักษณ์ไม่ปลายทุกตัวเป็นเซตว่าง

ขั้นที่ 2 เพิ่ม e ใน first(T) และ first(K) set เพราะมีกฎ T → e และ K → e

ขั้นที่ 3 เราบอกจากกฎในไวยากรณ์ได้ว่า

first(F)-{e} ⊆ first(E),

first(T)-{e} ⊆ first(E) ถ้า e ∈ first(F),

first(AD)-{e} ⊆ first(T),

{+, -} ⊆ first(AD),

first(L)-{e} ⊆ first(F),

first(K)-{e} ⊆ first(F) ถ้า e ∈ first(L),

{(, id} ⊆ first(L),

first(MUL)-{e} ⊆ first(K) และ

{*, /} ⊆ first(MUL)

ดังนั้น

ในรอบที่ 1 จะได้ + และ - เพิ่มใน first(AD), (และ id เพิ่มใน first(L), * และ / เพิ่มใน first(MUL)

ในรอบที่ 2 จะได้ + และ - เพิ่มใน first(T), (และ id เพิ่มใน first(F), * และ / เพิ่มใน first(K)

ในรอบที่ 3 จะได้ (และ id เพิ่มใน first(E)

ในรอบที่ 4 ไม่มีการเปลี่ยนค่า จึงหยุดทำงานได้

ตารางต่อไปนี้แสดง first set ที่คำนวณได้ในการทำงานแต่ละขั้นของขั้นตอนวิธีข้างบน

Nonterminals	ขั้นที่ 1	ขั้นที่ 2	ขั้นที่ 3 รอบที่ 1	ขั้นที่ 3 รอบที่ 2	ขั้นที่ 3 รอบที่ 3	ขั้นที่ 3 รอบที่ 4
E	{}	{}	{}	{}	{(, id}	{(, id}
T	{}	{e}	{e}	{e, +, -}	{e, +, -}	{e, +, -}
AD	{}	{}	{+, -}	{+, -}	{+, -}	{+, -}
F	{}	{}	{}	{(, id}	{(, id}	{(, id}
K	{}	{e}	{e}	{e, *, /}	{e, *, /}	{e, *, /}
MUL	{}	{}	{*, /}	{*, /}	{*, /}	{*, /}
L	{}	{}	{(, id}	{(, id}	{(, id}	{(, id}

Follow Set

ถ้า N เป็นสัญลักษณ์ไม่ปลายแล้ว follow set ของ N (เขียนแทนด้วย follow(N)) เป็นเซตของสัญลักษณ์ปลายที่อาจปรากฏอยู่หลัง N ในการแปลงที่เริ่มต้นจากสัญลักษณ์เริ่มต้น

ตัวอย่าง 3.23 กำหนดไวยากรณ์ $S \rightarrow (S)S$, $S \rightarrow e$

สัญลักษณ์) ต้องอยู่ใน follow(S) เนื่องจาก S ที่สร้างจากกฎ $S \rightarrow (S)S$ ต้องตามด้วย) เสมอ เนื่องจากเราต้องตรวจสอบว่าสายของโทเคนจบที่ใด เราใช้ $ แทนสัญลักษณ์สิ้นสุดสายของโทเคน ดังนั้น $ อยู่ใน follow(S) ด้วยเพราะ S เป็นสัญลักษณ์เริ่มต้นที่แปลงไปจนได้สายของโทเคนที่ถูกต้องตามไวยากรณ์ซึ่งจบท้ายด้วย $ เสมอ

เราสามารถนิยาม follow set ของสัญลักษณ์ไม่ปลาย N ได้ดังนี้

- $a \in follow(N)$ ถ้า $S \Rightarrow^* XNaY$ โดยที่ a เป็นสัญลักษณ์ปลาย, S เป็นสัญลักษณ์เริ่มต้น, X และ Y เป็นสตริงใด ๆ
- $\$ \in follow(N)$ ถ้า $S \Rightarrow^* XN$ โดยที่ S เป็นสัญลักษณ์เริ่มต้น และ X เป็นสตริงใด ๆ

นอกจากนั้น เราสามารถนิยาม follow set จากกฎในไวยกรณ์ได้ดังนี้

- $\$ \in follow(S)$ ถ้า S เป็นสัญลักษณ์เริ่มต้น
- $first(Y) - \{e\} \subseteq follow(N)$ ถ้า $L \rightarrow XNY$ โดยที่ X และ Y เป็นสตริงใด ๆ
- $follow(L) \subseteq follow(N)$ ถ้า $L \rightarrow XN$
- $follow(L) \subseteq follow(N)$ ถ้า $L \rightarrow XNY$ และ $e \in first(Y)$ โดยที่ X และ Y เป็นสตริงใด ๆ

จากนิยามของ follow set นี้ เราสามารถสร้างขั้นตอนวิธีการหา follow set ได้ดังนี้

ขั้นตอนวิธีการหา follow set

1. FOR EACH non-terminal X:
   IF X is the start symbol:
   follow(X) = {$}
   ELSE follow(X) = {}
2. DO
   FOR EACH production $L \rightarrow R_1 R_2 \ldots R_n$:
   FOR EACH nonterminal R_i in $R_1 R_2 \ldots R_n$:
   $follow(R_i) = follow(R_i) \cup (first(R_{i+1} \ldots R_n) - \{e\})$
   IF $e \in first(R_{i+1} \ldots R_n)$:
   $follow(R_i) = follow(R_i) \cup follow(L)$
   WHILE there is a change in follow (X), for some nonterminal X

ตัวอย่างต่อไปนี้แสดงการคำนวณหา follow set ด้วยขั้นตอนวิธีข้างบน

ตัวอย่าง 3.24 จากไวยากรณ์ใน G10 ตัวอย่าง 3.22 ที่มีกฎต่อไปนี้ E → F T, T → AD F T, T → e, AD → +, AD → -, F → L K, K → MUL L K, K → e, MUL → *, MUL → /, L → (E), L → id และ first set ต่อไปนี้

first(E) = {(, id}, first(T) = {e, +, -}, first(AD) = {+, -},
first(F) = {(, id }, first(K) = {e, *, / }, first(MUL) = {*, /}, first(L) = {(, id}

เราคำนวณหา follow set ของสัญลักษณ์ไม่ปลายตามขั้นตอนวิธีข้างบนได้ดังนี้

ขั้นที่ 1 กำหนดให้ follow set ของสัญลักษณ์ไม่ปลายทุกตัวเป็นเซตว่าง เว้นแต่สัญลักษณ์เริ่มต้น E ที่มีfollow set เป็น {$}

ขั้นที่ 2 จากกฎทั้งหมด เราบอกได้ว่า

กฎ	ความสัมพันธ์ที่ได้จากกฎ
E → F T	first(T) - {e} = {+, -} ⊆ follow(F)
	follow(E) ⊆ follow(T)
	follow(E) ⊆ follow(F) since ∈ first(T)
T → AD F T	first(F) - {e} = {(, id} ⊆ follow(AD)
	first(T) - {e} = {+, -} ⊆ follow(F)
	follow(T) ⊆ follow(F)
F → L K	first(K) – {e} = {+, -} ⊆ follow(L)
	follow(F) ⊆ follow(K)
	follow(F) ⊆ follow(L) since e ∈ first(K)
K → MUL L K	first(K) – {e} = {*, /} ⊆ follow(L)
	first(L) – {e} = {(, id} ⊆ follow(MUL)
	follow(K) ⊆ follow(L) since e ∈ first(K)
L → (E)	{) } ⊆ follow(E)

นั่นคือ {+, -,)} ∪ follow(T) ∪ follow(E) ⊆ follow(F),
follow(E) ⊆ follow(T),
{(, id} ⊆ follow(AD),
{*, /} ∪ follow(F) ∪ follow(K) ⊆ follow(L),
follow(F) ⊆ follow(K),
{(, id} ⊆ follow(MUL) และ
{) } ⊆ follow(E)

ดังนั้น ขั้นตอนวิธีคำนวณหา follow set ทำงานได้ดังแสดงในตารางข้างล่างนี้ โดยขั้นที่ 2 หยุดทำงานในรอบที่ 3 ซึ่งไม่มีการเปลี่ยน follow set ของสัญลักษณ์ใด ๆ

ตารางต่อไปนี้แสดง follow set ที่คำนวณได้ในการทำงานแต่ละขั้นของขั้นตอนวิธีข้างบน

Nonterminals	ขั้นที่ 1	ขั้นที่ 2 รอบที่ 1	ขั้นที่ 2 รอบที่ 2	ขั้นที่ 2 รอบที่ 3
E	{$}	{$,)}	{$,)}	{$,)}
T	{}	{$}	{$,)}	{$,)}
AD	{}	{(, id}	{(, id}	{(, id}
F	{}	{+, -, $}	{+, -, $)}	{+, -, $)}
K	{}	{+, -, $}	{+, -, $,)}	{+, -, $,)}
MUL	{}	{(, id }	{(, id }	{(, id }
L	{}	{+, -, *, /, $}	{*, /, +, -, $,)}	{*, /, +, -, $,)}

เมื่อออโตมาตาพูชดาว์นจำลองการแปลงมีการเปลี่ยนสถานะเชิงไม่กำหนดเมื่อมีกฎในการแปลงสัญลักษณ์ไม่ปลาย N ที่อยู่บนสุดของสแตคให้เลือกมากกว่าหนึ่งกฎ ขั้นตอนวิธีแจงส่วนแบบ LL(1) ใช้โทเคนที่อ่านได้ถัดไปมาช่วยตัดสินใจ หากโทเคนถัดไปที่อ่านได้เป็นโทเคน X เราจะเลือกกฎที่แปลง N เป็นสตริงที่ขึ้นต้นด้วย X ได้ กฎที่แปลงสัญลักษณ์ N ไปเป็นสตริงว่างจะใช้เมื่อ การทำให้สัญลักษณ์ N หายไปแล้วสัญลักษณ์ ที่ตามมาเป็นสัญลักษณ์ที่แปลงเป็นสตริงที่ขึ้นต้นด้วยโทเคน X ได้ จากหลักการที่กล่าวมาทั้งสองข้อนี้เราสามารถอธิบายวิธีการเลือกกฎของการแปลงจาก first set และ follow set ได้ดังนี้

ให้ N เป็นสัญลักษณ์ไม่ปลายที่อยู่บนสุดของสแตค และ T เป็นสัญลักษณ์ปลายหรือโทเคนที่อ่านได้ถัดไป

กฎ $N \rightarrow R$ จะใช้ได้ ถ้า

T อยู่ใน first(R) หรือ
$e \in$ first(R) (นั่นคือ $R \Rightarrow^* e$) และ T อยู่ใน follow(N)

ดังนั้น ขั้นตอนวิธีแจงส่วนแบบ LL(1) ใช้เกณฑ์การเลือกกฎการแปลงที่กล่าวมานี้รวมกับขั้นตอนวิธีจำลองการแปลงด้วยออโตมาตาพูชดาว์น โดยใช้ตารางที่เรียกว่าตารางกฎการแจงส่วน (parsing table) เป็นที่เก็บกฎที่ใช้ในกรณีต่าง ๆ แยกตามสัญลักษณ์ไม่ปลายที่อยู่บนสุดของสแตคและโทเคนที่อ่านได้ ตัวอย่างต่อไปนี้แสดงการสร้างตารางกฎการแจงส่วน

<u>ตัวอย่าง 3.25</u> จากไวยากรณ์ในตัวอย่าง 3.22 ที่มีกฎ $E \rightarrow F\ T$, $T \rightarrow AD\ F\ T$, $T \rightarrow e$, $AD \rightarrow +$, $AD \rightarrow -$, $F \rightarrow L\ K$, $K \rightarrow MUL\ L\ K$, $K \rightarrow e$, $MUL \rightarrow *$, $MUL \rightarrow /$, $L \rightarrow (\ E\ )$, $L \rightarrow id$ และ first set กับ follow set ที่คำนวณในตัวอย่าง 3.23 และ 3.24 ดังนี้

Nonterminals	E	T	AD	F	K	MUL	L
first	{(, id}	{e, +, -}	{+, -}	{(, id}	{e, *, /}	{*, /}	{(, id}
follow	{$,)}	{$,)}	{(, id}	{+, -, $)}	{+, -, $,)}	{(, id }	{*, /, +, -, $}

เราสร้างตารางกฎการแจงส่วนได้ดังแสดงในรูปข้างล่าง

Nonterminals	id	(	)	+	-	*	/	$
E	E → F T	E → F T						
T			T → e	T → AD F T	T → AD F T			T → e
AD				AD → +	AD → -			
F	F → L K	F → L K						
K			K → e	K → e	K → e	K → MUL L K	K → MUL L K	K → e
MUL						MUL → *	MUL → /	
L	L → id	L → (E)						

เมื่อสร้างตารางกฎการแจงส่วนแล้วเราสามารถใช้ขั้นตอนวิธีจำลองการแปลงด้วยออโตมาตาพูชดาว์นร่วมกับการเลือกกฎด้วยตารางกฎการแจงส่วนดังแสดงข้างล่างนี้

```
1. push(d, S)                          // S เป็นสัญลักษณ์เริ่มต้นและ d เป็นสแตค
2. t = scan_next_token()
3. WHILE d ไม่เป็นสแตคว่าง และ t ไม่เป็น null
       top = pop(d)
       IF top เป็นสัญลักษณ์ไม่ปลาย
           (top → body) = parseT(top ,t)   // ทำการแปลง
           push(d, reverse(body))
       ELSE                                // top เป็น สัญลักษณ์ปลาย
           IF t==top                       // token t ตรงกับสัญลักษณ์บนสุดในสแตค
               t = scan_next_token()       // ทิ้ง token เดิม
           ELSE                            // token t ไม่ตรงกับสัญลักษณ์บนสุดในสแตค
               BREAK                       // มีข้อผิดพลาด
4. IF d เป็นสแตคว่าง และ t เป็น null       // ตรวจสอบ token ตรงกับที่แปลงได้
       RETURN TRUE
   ELSE                                    // ตรวจสอบ token ไม่ตรงกับที่แปลงได้
       RETURN FALSE
```

ตัวอย่างต่อไปนี้แสดงการแจงส่วนแบบ LL(1)

<u>ตัวอย่าง 3.26</u> การแจงส่วนแบบ LL(1) ของสายของโทเคน id + id * (id + id) โดยใช้ตารางกฎการแจงส่วนในตัวอย่าง 3.25 แสดงตามลำดับขั้นได้ดังนี้

stack	tokens	action
$ E	id + id * (id + id) $	
		ใช้ E → F T
$ T F	id + id * (id + id) $	
		ใช้ F → L K
$ T K L	id + id * (id + id) $	
		ใช้ L → id
$ T K id	id + id * (id + id) $	
		ทิ้ง id ในสแตคและสายโทเคน
$ T K	+ id * (id + id) $	
		ใช้ K → e
$ T	+ id * (id + id) $	
		ใช้ T → AD F T
$ T F AD	+ id * (id + id) $	
		ใช้ AD → +
$ T F +	+ id * (id + id) $	
		ทิ้ง + ในสแตคและสายโทเคน
$ T F	id * (id + id) $	
		ใช้ F → L K
$ T K L	id * (id + id) $	
		ใช้ L → id
$ T K id	id * (id + id) $	
		ทิ้ง id ในสแตคและสายโทเคน
$ T K	* (id + id) $	
		ใช้ K → MUL id K
$ T K L MUL	* (id + id) $	
		ใช้ MUL → *
$ T K L *	* (id + id) $	
		ทิ้ง * ในสแตคและสายโทเคน
$ T K L	(id + id) $	
		ใช้ L → (E)
$ T K) E (	(id + id) $	
		ทิ้ง (ในสแตคและสายโทเคน
$ T K) E	id + id) $	
		ใช้ E → F T
$ T K) T F	id + id) $	
		ใช้ F → L K
$ T K) T K L	id + id) $	
		ใช้ L → id
$ T K) T K id	id + id) $	

		ทิ้ง id ในสแตคและสายโทเคน
$ T K) T K	+ id) $	
		ใช้ K → e
$ T K) T	+ id) $	
		ใช้ T → AD F T
$ T K) T F AD	+ id) $	
		ใช้ AD → +
$ T K) T F +	+ id) $	
		ทิ้ง + ในสแตคและสายโทเคน
$ T K) T F	Id) $	
		ใช้ F → L K
$ T K) T K L	Id) $	
		ใช้ L → id
$ T K) T K id	Id) $	
		ทิ้ง id ในสแตคและสายโทเคน
$ T K) T K	) $	
		ใช้ K → e
$ T K) T	) $	
		ใช้ T → e
$ T K)	) $	
		ทิ้ง) ในสแตคและสายโทเคน
$ T K	$	
		ใช้ K → e
$ T	$	
		ใช้ T → e
$	$	
		ยอมรับ

จากขั้นตอนวิธีการแจงส่วนแบบ LL(1) ที่อธิบายไป จะเห็นได้ว่าขั้นตอนวิธีนี้ไม่สามารถเลือกกฎเพื่อใช้ในการแปลงได้หากมีกฎมากกว่าหนึ่งข้อสำหรับบางช่องในตารางกฎการแจงส่วน หากไวยากรณ์ใดทำให้ตารางกฎการแจงส่วนมีกฎมากกว่าหนึ่งข้อในช่องใดช่องหนึ่ง เราถือว่าไวยากรณ์นั้นไม่เป็นไวยากรณ์แบบ LL(1) ตัวอย่างต่อไปนี้แสดงการตรวจสอบว่าไวยากรณ์เป็นไวยากรณ์แบบ LL(1) หรือไม่จากตารางกฎการแจงส่วนแบบ LL(1)

ตัวอย่าง 3.27 กำหนดไวยากรณ์ G11 สำหรับนิพจน์ที่มีตัวกระทำ +, -, *, / และตัวถูกกระทำ id โดยมี ( ) ที่มีลำดับความสำคัญสูงสุด ตัวกระทำ * และ / มีลำดับความสำคัญรองลงมา ตัวกระทำ + และ – มีลำดับความสำคัญต่ำสุด นอกจากนั้นตัวกระทำทั้งหมดเป็นชนิดที่มีการเชื่อมโยงทางซ้ายดังแสดงด้วยกฎดังนี้

E → E AD F, E → F, AD → +, AD → -,
F → F MUL L , F → L, MUL → *, MUL → /,
L → (E), L → id

จากกฎชุดนี้เราได้ first(E)=first(F)=first(L) = { (, id }, first(AD) = {+, -} และ first(MUL) = {*, /} ซึ่งทำให้สร้างตารางกฎแจงส่วนดังนี้

Nonterminals	id	(	)	+	-	*	/	$
E	E → E AD F E → F	E → E AD F E → F						
AD				AD → +	AD → -			
F	F → F MUL L F → L	F → F MUL L F → L						
MUL						MUL → *	MUL → /	
L	L → id	L → (E)						

เนื่องจาก first(E) = first(F) = {id, (} ดังนั้นกฎ E → E AD F และ E → F ใช้ได้เมื่อสัญลักษณ์บนสแตคเป็น E และโทเคนถัดไปเป็น id หรือ (ทำนองเดียวกัน เนื่องจากกฎ F → F MUL L และ F → L ใช้ได้เมื่อสัญลักษณ์บนสแตคเป็น F และโทเคนถัดไปเป็น id หรือ (นั่นคือไวยากรณ์นี้ไม่เป็นไวยากรณ์แบบ LL(1)

การเวียนซ้ำทางซ้าย (left recursion)

ปัญหาหนึ่งของขั้นตอนวิธีแจงส่วนแบบ LL(1) เกิดจากกฎที่อยู่ในรูปแบบของ N → N X เราเรียกกฎแบบนี้ว่ามี การเวียนซ้ำทางซ้าย (left recursion) เมื่อ N อยู่บนสุดของสแตค และโทเคนที่อ่านได้อยู่ใน first(N) กฎ N → N X จะถูกใช้ในการแปลง การแปลงด้วยกฎนี้แทน N ในสแตคด้วย N X หลังจากการแปลงครั้งที่ 1 สัญลักษณ์ที่อยู่บนสุดของสแตคยังเป็น N และโทเคนที่อ่านได้ยังอยู่ใน first(N) ดังนั้นกฎ N → N X จะถูกใช้ในการแปลงซ้ำไปเรื่อย ๆ และ ขั้นตอนวิธีแจงส่วนแบบ LL(1) ไม่สามารถทำงานได้

เนื่องจากกฎที่มีการเวียนซ้ำทางซ้ายทำให้ขั้นตอนวิธีแจงส่วนแบบ LL(1) เราต้องเขียนกฎให้สามารถอธิบายไวยากรณ์ได้เหมือนเดิมแต่ไม่มีการเวียนซ้ำทางซ้าย จากกฎ N → N X จะเห็นว่าต้องมีกฎ N → Y สำหรับสตริง Y ที่ไม่ขึ้นต้นด้วย N ด้วยจึงจะใช้อธิบายภาษาได้ จากกฎ N → N X และ N → Y จะเห็นได้ว่า N ⇒* Y X* ดังนั้นเราสามารถใช้กฎต่อไปนี้อธิบาย N

N → Y K
K → X K
K → e

ตัวอย่าง 3.28 ไวยากรณ์ G11 ในตัวอย่าง 3.27 มีการเวียนซ้ำทางซ้าย เราสามารถกำจัดการเวียนซ้ำทางซ้ายได้โดยแปลง

E → E AD F, E → F เป็น
E → F T, T → AD F T, T → e

และแปลง

F → F MUL L, F → L เป็น

F → L K, K → MUL L K, K → e

ไวยากรณ์ที่ได้จากการกำจัดการเวียนซ้ำทางซ้ายนี้ คือไวยากรณ์ G10 ในตัวอย่าง 3.22 ที่เป็นไวยากรณ์แบบ LL(1) ดังแสดงในตารางกฎการแจงส่วนแบบ LL(1) ในตัวอย่าง 3.25

หากมีกฎที่มีการเวียนซ้ำทางซ้ายมากกว่าหนึ่งกฎดังนี้

N → N X1

N → N X2

...

N → N Xn

และมีกฎที่ไม่มีการเวียนซ้ำทางซ้ายสำหรับ N ดังนี้

N → Y1

N → Y2

...

N → Ym

เราสามารถใช้กฎต่อไปนี้อธิบาย N

N → Y1 K

N → Y2 K

...

N → Ym K

K → X1 K

K → X2 K

...

K → Xn K

K → e

นอกจากนั้นยังมีการเวียนซ้ำทางซ้าย ที่ผ่านสัญลักษณ์ไม่ปลายมากกว่าหนึ่งตัว นั่นคือ N ⇒* N X ในกรณีการกำจัดการเวียนซ้ำทางซ้ายจะซับซ้อนมากขึ้นและจะไม่ขออธิบายไว้ในที่นี้

การดึงตัวร่วมทางซ้าย (Left factoring)

ขั้นตอนวิธีการแจงส่วนแบบ LL(1) ทำงานได้ถูกต้องเมื่อมีกฎที่เลือกใช้ในแต่ละขั้นตอนเพียงกฎเดียว สาเหตุหนึ่งที่ทำให้เลือกใช้กฎได้มากกว่าหนึ่งกฎในการแปลงคือ มีกฎมากกว่าหนึ่งกฎที่ทางซ้ายเป็นสัญลักษณ์เดียวกันและ ด้านขวาของกฎเริ่มต้นด้วยสัญลักษณ์เดียวกันด้วย เช่น กฎ N → X Y และ N → X Z เมื่อ

สัญลักษณ์ที่อยู่บนสุดของสแตคเป็น N และโทเคนที่อ่านได้อยู่ใน first(X) เราอาจเลือกกฎได้กฎหนึ่งใช้ได้เช่นกัน ปัญหานี้แก้ได้โดยเขียนกฎใหม่ดังนี้

N → X R, R → Y, R → Z

วิธีนี้เรียกว่าการดึงตัวร่วมทางซ้าย (left factoring) ตัวอย่างต่อไปนี้แสดงการดึงตัวร่วมทางซ้าย

ตัวอย่าง 3.29 ไวยากรณ์ G9 ในตัวอย่าง 3.16 เป็นไวยากรณ์ไม่กำกวม แต่ไวยากรณ์นี้ยังมีตัวร่วมทางซ้ายในกฎ Unmatched → if K Matched และ Unmatched → if K Matched else Unmatched เมื่อสร้างตารางการแจงส่วนแบบ LL(1) จะได้ว่ากฎที่ใช้สำหรับสัญลักษณ์ไม่ปลาย Unmatched และโทเคน if มี 2 กฎ คือ Unmatched → if K Matched และ Unmatched → if K Matched else Unmatched ดังแสดงในตารางข้างล่างนี้

Nonterminals \ Tokens	if	else	id	...
Unmatched	Unmatched → if K Matched Unmatched → if K Matched else Unmatched			
Matched	Matched → if K Matched else Matched		Matched → As	
...				

เราจะดึงตัวร่วมทางซ้ายและแทนด้วย Unmatched → if K Matched Elsepart, Elsepart → e และ Elsepart → else Unmatched เมื่อสร้างตารางการแจงส่วนแบบ LL(1) จะได้ว่ากฎที่ใช้สำหรับสัญลักษณ์ไม่ปลาย Unmatched และโทเคน if มีเพียงกฎเดียว คือ Unmatched → if K Matched elsepart ส่วน Elsepart → else Unmatched ใช้สำหรับสัญลักษณ์ไม่ปลาย Elsepart และโทเคน else และ Elsepart → e ใช้สำหรับสัญลักษณ์ไม่ปลาย Elsepart และโทเคนที่เป็นสัญลักษณ์แรกของคำสั่ง เช่น if, id ดังแสดงในตารางข้างล่างนี้

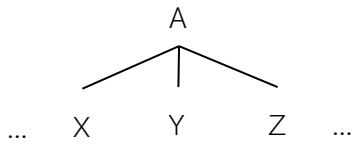
Nonterminals \ Tokens	if	else	id	...
Unmatched	Unmatched → if K Matched Elsepart			
Elsepart	Elsepart → e	Elsepart → else Unmatched	Elsepart → e	
Matched	Matched → if K Matched else Matched		Matched → As	
...				

ดังนั้นเมื่อต้องการสร้างตัวแจงส่วนแบบ LL(1) จะต้องเริ่มจากไวยากรณ์ที่ไม่กำกวม (ไวยากรณ์กำกวมจะไม่เป็นไวยากรณ์แบบ LL(1)) จากนั้นจึงตรวจสอบว่า ไวยากรณ์นั้นมีการเวียนซ้ำทางซ้ายหรือไม่ หากมีก็ให้นำไวยากรณ์นั้นมากำจัดการเวียนซ้ำทางซ้าย แล้วจึงนำไปดึงตัวร่วมทางซ้าย ถ้ามี สำหรับสองขั้นตอนนี้เราจะต้องกำจัดการเวียนซ้ำทางสายก่อนดึงตัวร่วมทางซ้าย เพราะเมื่อกำจัดการเวียนซ้ำทางซ้ายแล้วอาจได้กฎที่มีตัวร่วมทางซ้าย จากนั้นเราจึงสร้างตารางกฎการแจงส่วนแบบ LL(1) แล้วตรวจสอบว่า เป็นไวยากรณ์แบบ

LL(1) หรือไม่ โดยตรวจสอบว่าตารางกฎการแจงส่วนแบบ LL(1) มีช่องที่มีกฎมากกว่าหนึ่งกฎหรือไม่ หากเป็นไวยากรณ์แบบ LL(1) ก็ใช้ตารางนี้สำหรับขั้นตอนการแจงส่วนแบบ LL(1) ได้

3.5 ขั้นตอนวิธีแจงส่วนแบบ SLR(1)

การแจงส่วนแบบ LR เป็นการแจงส่วนจากล่างขึ้นบน (bottom-up parsing) นั่นคือต้นไม้แจงส่วนถูกสร้างจากใบขึ้นไปสู่ราก โทเคนถูกอ่านจากซ้ายไปขวาและสร้างเป็นใบของต้นไม้แจงส่วน ระหว่างการทำงาน เราจะได้ต้นไม้ย่อยหลายต้นที่นำมาประกอบกันเป็นต้นไม้ที่ใหญ่ขึ้นเรื่อย ๆ การเลือกกฎที่ใช้สร้างต้นไม้พิจารณาจากด้านขวาของกฎว่าตรงกับโหนดที่มีอยู่ในขณะนั้นหรือไม่ หากโหนดในระดับสูงสุดของต้นไม้ย่อย ๆ เรียงกันได้เป็น X Y Z ... และมีกฎ A → X Y Z เราจะเลือกกฎนี้เพื่อสร้างต้นไม้ที่มีโหนด A เป็นรากและโหนด X, Y, Z เป็นลูกของโหนด A ดังแสดงในรูปข้างล่าง การสร้างต้นไม้แจงส่วนนี้เสร็จสมบูรณ์เมื่อโหนดใบทั้งหมดรวมอยู่ในต้นไม้ต้นเดียว



3.5.1 หลักการทำงานของการแจงส่วนแบบ LR

ขั้นตอนวิธีแจงส่วนแบบ LR ใช้สแตคเพื่อเก็บโหนดรากของต้นไม้ย่อยที่สร้างมาและใช้ออโตมาตาจำกัดเพื่อจำว่าโหนดรากที่สร้างมาเรียงต่อกันเป็นส่วนของด้านขวาของกฎได้บ้าง

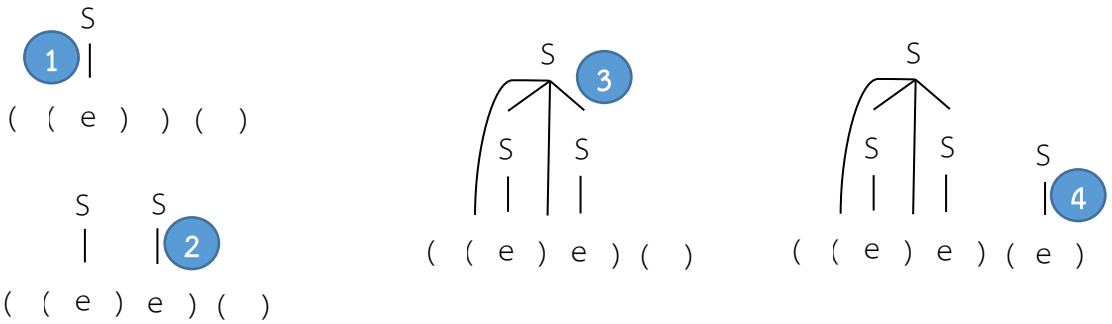
การแจงส่วนแบบ LR ทำงานเลียนแบบการแปลงขวาสุด (rightmost derivation) ย้อนจากขั้นสุดท้ายไปถึงขั้นแรก ในแต่ละขั้นเราเลือกระหว่างการทำงานสองแบบ คือ การเลื่อน (shift) และ การลด (reduce) การเลื่อนเป็นการเอาโทเคนที่อ่านไว้ใส่ลงในสแตค การลดเป็นการแทนสตริงที่อยู่บนสุดของสแตคด้วยสัญลักษณ์ไม่ปลาย การลดนี้ต้องเป็นไปตามกฎในไวยากรณ์ เช่น การลดตามกฎ A → X Y Z ทำได้เมื่อ Z Y X (อ่านจากบนลงล่าง) อยู่บนสุดของสแตคและเราลด Z Y X ไปเป็น A การลดในที่นี้เทียบได้กับการแปลงที่ใช้กฎ A → X Y Z

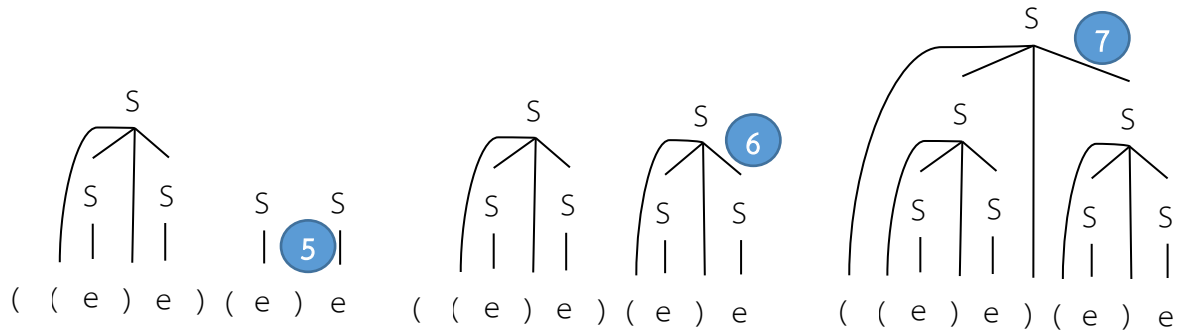
ตัวอย่างต่อไปนี้แสดงการแจงส่วนแบบ LR ด้วยการเลื่อนและการลด พร้อมทั้งเปรียบเทียบกับการแปลงขวาสุด

ตัวอย่าง 3.30 กำหนดไวยากรณ์ G ที่มีกฎ S → (S) S, S → e และสายของโทเคน (( )) ( ) การแจงส่วนด้วยการเลื่อนและการลดแสดงเทียบกับการแปลงขวาสุดได้ดังนี้

สแตค (ล่าง -> บน)	สายของโทเคน	การกระทำ	การแปลง
$	(( )) ( ) $		
		เลื่อน	
$ (	( )) ( ) $		
		เลื่อน	
$ ((	)) ( ) $		(( )) ( )
		ลด S → e	1 ⇒
$ ((S	)) ( ) $		
		เลื่อน	
$ ((S)	) ( ) $		((S)) ( )
		ลด S → e	2 ⇒
$ ((S) S	) ( ) $		((S) S) ( )
		ลด S → (S) S	3 ⇒
$ (S	) ( ) $		
		เลื่อน	
$ (S)	( ) $		
		เลื่อน	
$ (S) (	) $		(S) ( )
		ลด S → e	4 ⇒
$ (S) (S	) $		
		เลื่อน	
$ (S) (S)	$		(S) (S)
		ลด S → e	5 ⇒
$ (S) (S) S	$		(S) (S) S
		ลด S → (S) S	6 ⇒
$ (S) S	$		(S) S
		ลด S → (S) S	7 ⇒
$ S			S

สังเกตว่า การแปลงที่แสดงข้างบนสร้างเป็นต้นไม้แจงส่วนได้ดังนี้





ข้อสังเกต:

- การลดใช้กับสตริงในสแตค ดังนั้นเราต้องใส่โทเคนในสแตคด้วยการเลื่อนก่อน
- สตริงในสแตค (อ่านจากล่างขึ้นบน) ต่อกับโทเคนที่ยังไม่ได้อ่าน รวมกันเป็นแบบประโยค
- เราเรียกการแจงส่วนแบบ LR ว่า การแจงส่วนแบบเลื่อนลด (shift-reduce parsing) เพราะทำงานด้วยการเลื่อนและการลด

3.5.2 ออโตมาตาจำกัดของรายการ

ในการแจงส่วนแบบ LR เรายังต้องตัดสินใจในแต่ละขั้นว่าจะเลื่อนหรือลด ถ้าจะลดเราต้องใช้กฎใดในไวยากรณ์ หัวข้อนี้อธิบายออโตมาตาจำกัดที่ใช้ในการตัดสินใจนี้

รายการ LR(0) (LR(0) item)

การแจงส่วนแบบ LR อาจเลือกทำการลดได้เมื่อสิ่งที่อยู่บนสุดของสแตคตรงกับด้านขวาของกฎบางกฎในไวยากรณ์ อย่างไรก็ตาม ตัวแจงส่วนอาจยังไม่ทำการลดแต่เลื่อนโทเคนลงมาในสแตคเพื่อใช้กฎอื่นต่อไปก็ได้ เราจะสร้างออโตมาตาจำกัดที่ใช้ช่วยตัดสินใจเลือกระหว่างการเลื่อนและการลดด้วยกฎแต่ละกฎในไวยากรณ์ ออโตมาตานี้ใช้จำว่าสตริงที่อยู่บนสุดของสแตคตรงกับด้านขวาของกฎใดบ้าง **รายการ LR(0)** บอกว่าสตริงที่อยู่บนสุดของสแตคตรงกับส่วนหน้า (prefix) ใดในด้านขวาของกฎโดยสัญลักษณ์ · แสดง**ตำแหน่งแยก (distinguished position)** ว่าส่วนหน้าที่อยู่ในสแตคกับส่วนที่เหลือ

- รายการ $L \rightarrow X\ Y \cdot Z$ เป็นรายการ LR(0) ที่บอกว่าสตริงที่อยู่หน้า · (คือ X Y) เป็นสตริงที่อยู่บนสุดของสแตค (นั่นคือได้สร้างส่วนของต้นไม้แจงส่วนที่มีโหนดรากเป็น X และ Y แล้ว) และจะใช้กฎ $L \rightarrow X\ Y\ Z$ ทำการลดได้เมื่อสร้างสตริงที่อยู่หลัง · (คือ Z) ได้
- รายการ $L \rightarrow X\ Y\ Z \cdot$ เป็นรายการ LR(0) ที่บอกว่าสร้างสตริง X Y Z ไว้ในสแตคแล้ว (นั่นคือได้สร้างส่วนของต้นไม้แจงส่วนที่มีโหนดรากเป็น X, Y และ Z แล้ว) ดังนั้นจะใช้กฎ $L \rightarrow X\ Y\ Z$ ทำการลดได้

ดังนั้น รายการ LR(0) ของไวยากรณ์ใด ๆ ประกอบด้วยกฎในไวยากรณ์ที่ระบุตำแหน่งแยกในทุกตำแหน่งที่เป็นไปได้ของกฎ เช่น รายการ LR(0) ที่สร้างจากกฎ $L \rightarrow X\ Y\ Z$ คือ $L \rightarrow \cdot X\ Y\ Z$, $L \rightarrow X \cdot Y\ Z$, $L \rightarrow X\ Y \cdot Z$ และ $L \rightarrow X\ Y\ Z \cdot$ ส่วนรายการ LR(0) ที่สร้างจากกฎ $L \rightarrow e$ คือ $L \rightarrow \cdot$ เพราะสตริงว่างมีความยาว 0 จึงมีตำแหน่งแยกได้ตำแหน่งเดียว

ตัวอย่าง 3.31 ไวยากรณ์ที่มีกฎ $S \rightarrow (S) S, S \rightarrow e$ มีรายการ LR(0) ต่อไปนี้

$S \rightarrow \cdot (S) S, S \rightarrow (\cdot S) S, S \rightarrow (S \cdot) S, S \rightarrow (S) \cdot S, S \rightarrow (S) S \cdot$ และ $S \rightarrow \cdot$

ไวยากรณ์ที่มีกฎ $E \rightarrow E + n, E \rightarrow E * n, E \rightarrow n$ มีรายการ LR(0) ต่อไปนี้

$E \rightarrow \cdot E + n, E \rightarrow E \cdot + n, E \rightarrow E + \cdot n, E \rightarrow E + n \cdot ,$

$E \rightarrow \cdot E * n, E \rightarrow E \cdot * n, E \rightarrow E * \cdot n, E \rightarrow E * n \cdot ,$

$E \rightarrow \cdot n$ และ $E \rightarrow n \cdot$

รายการ LR(0) ที่มีตำแหน่งแยกอยู่หน้าสัญลักษณ์แรกในกฎ เรียกว่า **รายการเริ่ม (initial item)** เป็นรายการที่แสดงว่าจะเริ่มพิจารณาสร้างต้นไม้ย่อยของสัญลักษณ์ทางซ้ายของกฎได้หรือไม่

รายการ LR(0) ที่มีตำแหน่งแยกอยู่หลังสัญลักษณ์สุดท้ายในกฎ เรียกว่า **รายการสมบูรณ์ (complete item)** เป็นรายการที่แสดงว่าในสแตคเก็บสตริงทางขวาของกฎนั้นได้สมบูรณ์แล้ว ดังนั้นอาจเลือกใช้กฎนั้นในการลด

ตัวอย่าง 3.32 จากตัวอย่าง 3.31

ไวยากรณ์ที่มีกฎ $S \rightarrow (S) S, S \rightarrow e$ มีรายการเริ่มต้น

$S \rightarrow \cdot (S) S$ และ $S \rightarrow \cdot$

มีรายการสมบูรณ์

$S \rightarrow (S) S \cdot$ และ $S \rightarrow \cdot$

ไวยากรณ์ที่มีกฎ $E \rightarrow E + n, E \rightarrow E * n, E \rightarrow n$ มีรายการเริ่มต้น

$E \rightarrow \cdot E + n, E \rightarrow \cdot E * n$ และ $E \rightarrow \cdot n$

มีรายการสมบูรณ์

$E \rightarrow E + n \cdot , E \rightarrow E * n \cdot$ และ $E \rightarrow n \cdot$

ออโตมาตาจำกัดของรายการ LR(0)

เราสร้างออโตมาตาจำกัดเพื่อจำว่าตัวแจงส่วนสร้างต้นไม้ย่อยใดได้แล้วบ้าง ดังนั้นแต่ละสถานะของออโตมาตาจำกัดจำรายการ LR(0) แต่ละรายการ ออโตมาตาจะเปลี่ยนสถานะจากสถานะของ $L \rightarrow X \cdot Y Z$ ไปยังสถานะของ $L \rightarrow X Y \cdot Z$ เมื่อได้สัญลักษณ์ Y ซึ่งอาจเป็นสัญลักษณ์ปลาย (คือโทเคน) หรือสัญลักษณ์ไม่ปลาย

- ถ้า Y เป็นโทเคน การเปลี่ยนจากสถานะของ $L \rightarrow X \cdot Y Z$ ไปยังสถานะของ $L \rightarrow X Y \cdot Z$ เกิดเมื่อตัวแจงส่วนเลื่อนโทเคน Y ลงในสแตค
- ถ้า Y เป็นสัญลักษณ์ไม่ปลาย การเปลี่ยนจากสถานะของ $L \rightarrow X \cdot Y Z$ ไปยังสถานะของ $L \rightarrow X Y \cdot Z$ เกิดเมื่อตัวแจงส่วนใส่ Y ลงในสแตคโดยการลดสตริงบนสแตคเป็น Y

นอกจากนั้น เมื่อ Y เป็นสัญลักษณ์ไม่ปลาย ออโตมาตาอาจเปลี่ยนจากสถานะ $L \rightarrow X \cdot Y Z$ ไปยังสถานะของ $Y \rightarrow \cdot P Q$ ด้วยสตริงว่าง การเปลี่ยนสถานะนี้เป็นการเปลี่ยนสถานะเชิงไม่กำหนด คือ เลือก

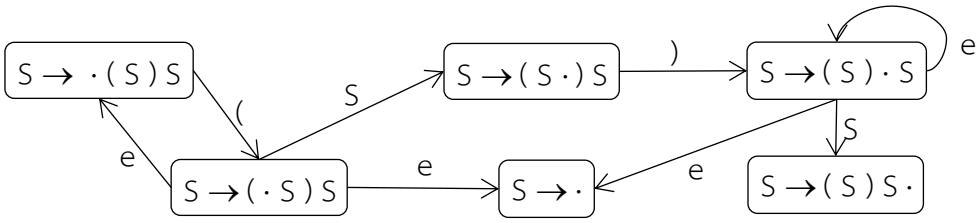
เปลี่ยนสถานะเมื่อรู้ล่วงหน้า (ด้วยความสามารถเชิงไม่กำหนด) ว่าต้องสร้างสัญลักษณ์ Y ด้วยการลดในอนาคต เพื่อรายการที่มี Y หลังตำแหน่งแยก เช่น L → X Y · Z

ตัวอย่างต่อไปนี้แสดงการสร้างออโตมาตาจำกัดของรายการ LR(0)

<u>ตัวอย่าง 3.33</u> ไวยากรณ์ที่มีกฎ S → (S) S, S → e มีรายการ LR(0) ต่อไปนี้

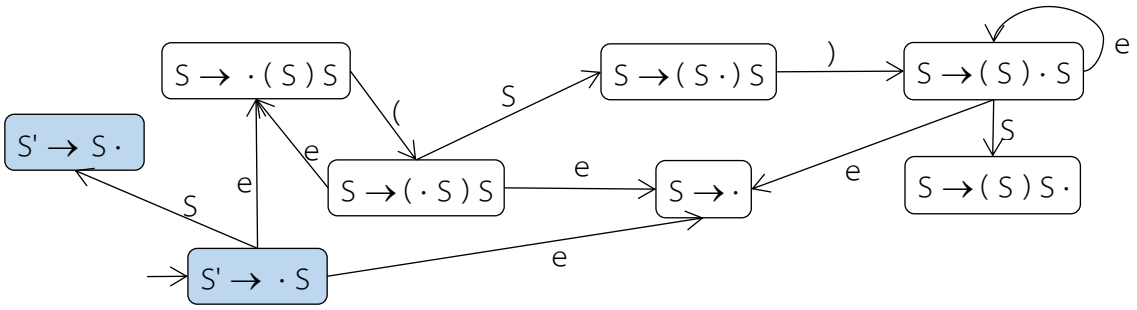
S → · (S) S, S → (· S) S, S → (S ·) S, S → (S) · S, S → (S) S · และ S → ·

และมีการเปลี่ยนสถานะในออโตมาตาจำกัดดังแสดงในรูปข้างล่างนี้



จากตัวอย่าง 3.33 ออโตมาตาที่สร้างมานี้ยังไม่มีสถานะเริ่มต้น สถานะเริ่มต้นควรมีรายการที่ใช้ในการแปลงครั้งแรกที่อยู่ที่รากของต้นไม้แจงส่วน ดังนั้นจึงอาจเป็นรายการเริ่มต้นที่สร้างจากกฎที่มีสัญลักษณ์เริ่มต้นอยู่ทางซ้ายซึ่งอาจมีได้มากกว่าหนึ่งกฎ ดังนั้นเราจะเพิ่มสัญลักษณ์ S' เป็นสัญลักษณ์เริ่มต้นใหม่และกฎ S' → S เมื่อ S เป็นสัญลักษณ์เริ่มต้นเก่า การเพิ่มกฎนี้ในไวยากรณ์ไม่ทำให้ภาษาเปลี่ยนไป แต่การแปลงด้วยไวยากรณ์นี้เริ่มต้นด้วยการใช้กฎ S' → S เสมอเพราะมีเพียงกฎเดียวที่ใช้ S' ไวยากรณ์ที่เพิ่มกฎนี้ไปเรียกว่า **ไวยากรณ์เสริม (augmented grammar)**

กฎที่เสริมมาทำให้มีรายการ S' → · S และ S' → S · เพิ่มมา เราสร้างออโตมาตาจากไวยากรณ์เสริมตามที่อธิบายก่อนนี้และให้สถานะของ S' → · S เป็นสถานะเริ่มต้นได้ดังแสดงในรูปข้างล่างนี้ สถานะที่เพิ่มขึ้นมาแสดงด้วยการแรเงา

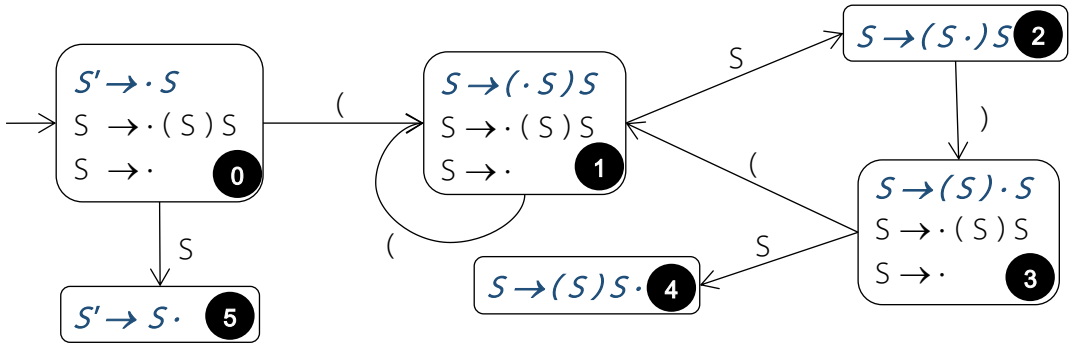


ออโตมาตานี้เป็นออโตมาตาจำกัดเชิงไม่กำหนดซึ่งทำให้การสร้างโปรแกรมจากออโตมาตานี้ซับซ้อน ดังนั้นเราจะแปลงออโตมาตานี้เป็นออโตมาตาจำกัดเชิงกำหนดโดยหาส่วนปิดคลุม (closure) ของแต่ละสถานะ (ดูจากตำราเกี่ยวกับออโตมาตาจำกัดทั่วไป) และให้สถานะของออโตมาตาจำกัดเชิงกำหนดแทนส่วนปิดคลุมของสถานะของออโตมาตาจำกัดเชิงไม่กำหนดดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 3.34 จากออโตมาตาจำกัดเชิงไม่กำหนดในตัวอย่าง 3.33 เราหาส่วนปิดคลุมของแต่ละสถานะได้ดังแสดงในตารางข้างล่างนี้

สถานะในออโตมาตาจำกัดเชิงไม่กำหนด	ส่วนปิดคลุม	สถานะในออโตมาตาจำกัดเชิงกำหนด
S' → · S	S' → · S, S → · (S) S, S → ·	0
S' → S ·	S' → S ·	5
S → · (S) S	S → · (S) S	ไม่เกิดในการทำงาน
S → (· S) S	S → (· S) S, S → · (S) S, S → ·	1
S → (S ·) S	S → (S ·) S	2
S → (S) · S	S → (S) · S, S → · (S) S, S → ·	3
S → (S) S ·	S → (S) S ·	4
S → ·	S → ·	ไม่เกิดในการทำงาน

เราสร้างออโตมาตาจำกัดเชิงกำหนดโดยให้แต่ละสถานะแทนส่วนปิดคลุมของสถานะของออโตมาตาจำกัดเชิงไม่กำหนดจากตารางข้างบน การเปลี่ยนสถานะของออโตมาตาจำกัดเชิงกำหนดแสดงในรูปข้างล่าง



จากออโตมาตาข้างบน สถานะ 6 และ 7 ไม่อยู่ในออโตมาตาจำกัดเชิงกำหนดนี้เนื่องจากรายการ S → · (S) S และ S → · ไม่เกิดแยกกับรายการอื่น จึงไม่แสดงไว้ในรูป

รายการในส่วนปิดคลุมที่มาจากออโตมาตาจำกัดเชิงไม่กำหนดเรียกว่า **รายการแกน (kernel item)** ซึ่งแสดงด้วยตัวเอียงในรูปข้างบน สังเกตว่าถ้าสถานะ 2 สถานะมีรายการแกนเหมือนกันแล้วจะต้องมีส่วนปิดคลุมเหมือนกันและเป็นสถานะเดียวกัน

3.5.3 การสร้างตารางการแจงส่วนแบบ SLR(1)

ตัวแจงส่วนสามารถใช้ออโตมาตาของรายการเพื่อจำว่าส่วนของแบบประโยคที่สร้างไว้ในสแตคตรงกับส่วนหน้าของสตริงทางขวาในกฎใด ดังนั้นตัวแจงส่วนจะดูว่าออโตมาตาอยู่ในสถานะใดโดยไม่ต้องเข้าไปในสแตค ตัวแจงส่วนสามารถตัดสินใจว่าจะทำการเลื่อนหรือลดโดยดูจากรายการในสถานะของออโตมาตาดังนี้

- สถานะที่มีรายการสมบูรณ์ $X \rightarrow y \cdot$ แสดงว่า สแตคมีสตริงที่ y อยู่ ดังนั้นตัวแจงส่วนสามารถใช้กฎของรายการดังกล่าวทำการลดเมื่ออยู่ในสถานะนั้นได้ หากพิจารณาโทเคนถัดไปประกอบด้วย จะเห็นว่าการลดนั้นทำให้สร้างต้นไม้แจงส่วนต่อไปได้หากโทเคนถัดไปอยู่ใน follow(X)
- เมื่อตัวแจงส่วนอยู่ในสถานะที่มีรายการ $X \rightarrow w \cdot Y z$ แสดงว่ามีสตริง w ในสแตคแล้ว ถ้า Y เป็นสัญลักษณ์ปลายที่ตรงกับโทเคนถัดไป ตัวแจงส่วนอาจเลื่อนโทเคน Y ใส่ลงในสแตค เพื่อให้มี w Y ในสแตค หากโทเคนถัดไปไม่ใช่ Y ตัวแจงส่วนไม่สามารถเลื่อนโทเคนนั้นลงในสแตคได้เพราะจะไม่สามารถทำให้สแตคตรงกับขวามือของกฎนั้นได้
- เมื่อตัวแจงส่วนอยู่ในสถานะที่มีรายการ $X \rightarrow w \cdot Y z$ แสดงว่ามีสตริง w ในสแตคแล้ว ถ้า Y เป็นสัญลักษณ์ไม่ปลายและได้สัญลักษณ์นั้นใส่เพิ่มมาในสแตค *จากการลด* ตัวแจงส่วนเปลี่ยนสถานะหลังการลดที่ทำให้ได้สัญลักษณ์ไม่ปลายนั้นไปยังสถานะที่มีรายการ $X \rightarrow w Y \cdot z$

ตัวแจงส่วนแบบ simple LR(1) หรือ SLR(1) ใช้การแจงส่วนแบบ LR อย่างง่ายที่ดูโทเคนล่วงหน้า 1 ตัวร่วมกับออโตมาตาของรายการดังที่กว่ามา จากออโตมาตาของรายการ เราสร้างตารางการแจงส่วนแบบ SLR(1) ที่ระบุการทำงาน (การเลื่อนหรือการลด) ของตัวแจงส่วนเมื่อได้สัญลักษณ์หนึ่งและออโตมาตาอยู่ในแต่ละสถานะดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 3.35 จากออโตมาตาจำกัดเชิงกำหนดในตัวอย่าง 3.34 เราสร้างตารางการแจงส่วนแบบ SLR(1) ได้ดังแสดงในตารางข้างล่างนี้

Symbol / State	(	)	$	S
0	shift & goto 1	reduce: $S \rightarrow \varepsilon$	reduce: $S \rightarrow \varepsilon$	goto 5
1	shift & goto 1	reduce: $S \rightarrow \varepsilon$	reduce: $S \rightarrow \varepsilon$	goto 2
2		shift & goto 3		
3	shift & goto 1	reduce: $S \rightarrow \varepsilon$	reduce: $S \rightarrow \varepsilon$	goto 4
4		reduce: $S \rightarrow (S) S$	reduce: $S \rightarrow (S) S$	
5			accept	

จากออโตมาตาจำกัดเชิงกำหนดในตัวอย่าง 3.34 ตารางการแจงส่วนแบบ SLR(1) สร้างได้ดังนี้

ที่สถานะ 0

- จากรายการ $S \rightarrow \cdot (S) S$ ถ้าโทเคนถัดไปเป็นสัญลักษณ์ (แล้วออโตมาตาจะเปลี่ยนไปยังสถานะ 1 และตัวแจงส่วนจะเลื่อน (พร้อมกับสถานะ 1 ของออโตมาตาลงในสแตค (shift & goto 1)
- จากรายการ $S' \rightarrow \cdot S$ เมื่อตัวแจงส่วนได้สัญลักษณ์ S จากการลด ออโตมาตาจะเปลี่ยนไปยังสถานะ 5 และตัวแจงส่วนจะเลื่อน S พร้อมกับสถานะ 5 ของออโตมาตาลงในสแตค (goto 5)
- จากรายการ $S \rightarrow \cdot$ ถ้าโทเคนถัดไปเป็นสัญลักษณ์) หรือ $ ที่อยู่ใน follow(S) ตัวแจงส่วนจะทำการลดด้วยกฎ $S \rightarrow \varepsilon$ โดย pop สตริงว่างออกจากสแตค (reduce: $S \rightarrow \varepsilon$) นั่นคือไม่เปลี่ยนสแตค ดังนั้น

สถานะของออโตมาตาจะถอยไปตามที่ pop สตริงออกจากสแตค แล้วจึงเปลี่ยนสถานะตามสัญลักษณ์ S ที่ลงในสแตค

ที่สถานะ 1

- จากรายการ S →(· S) S เมื่อตัวแจงส่วนได้สัญลักษณ์ S จากการลด ออโตมาตาจะเปลี่ยนไปยังสถานะ 2 และตัวแจงส่วนจะเลื่อน S พร้อมกับสถานะ 2 ของออโตมาตาลงในสแตค (goto 2)
- จากรายการ S →· (S) S ถ้าโทเคนถัดไปเป็นสัญลักษณ์ (แล้วออโตมาตาจะเปลี่ยนไปยังสถานะ 1 และตัวแจงส่วนจะเลื่อน (พร้อมกับสถานะ 1 ของออโตมาตาลงในสแตค (shift & goto 1) ทำนองเดียวกับที่สถานะ 0
- จากรายการ S →· ถ้าโทเคนถัดไปเป็นสัญลักษณ์) หรือ $ ที่อยู่ใน follow(S) ตัวแจงส่วนจะทำการลดด้วยกฎ S → ε (reduce: S → ε) ทำนองเดียวกับที่สถานะ 0

ที่สถานะ 2

- จากรายการ S →(S ·) S ถ้าโทเคนถัดไปเป็นสัญลักษณ์) แล้วออโตมาตาจะเปลี่ยนไปยังสถานะ 3 และตัวแจงส่วนจะเลื่อน) พร้อมกับสถานะ 3 ของออโตมาตาลงในสแตค (shift & goto 3)

ที่สถานะ 3

- จากรายการ S →(S) · S เมื่อตัวแจงส่วนได้สัญลักษณ์ S จากการลด ออโตมาตาจะเปลี่ยนไปยังสถานะ 4 และตัวแจงส่วนจะเลื่อน S พร้อมกับสถานะ 4 ของออโตมาตาลงในสแตค (goto 4)
- จากรายการ S →· (S) S ถ้าโทเคนถัดไปเป็นสัญลักษณ์ (แล้วออโตมาตาจะเปลี่ยนไปยังสถานะ 1 และตัวแจงส่วนจะเลื่อน (พร้อมกับสถานะ 1 ของออโตมาตาลงในสแตค (shift & goto 1) ทำนองเดียวกับที่สถานะ 0
- จากรายการ S →· ถ้าโทเคนถัดไปเป็นสัญลักษณ์) หรือ $ ที่อยู่ใน follow(S) ตัวแจงส่วนจะทำการลดด้วยกฎ S → ε (reduce: S → ε) ทำนองเดียวกับที่สถานะ 0

ที่สถานะ 4

- จากรายการ S →(S) S · ถ้าโทเคนถัดไปเป็นสัญลักษณ์) หรือ $ ที่อยู่ใน follow(S) ตัวแจงส่วนจะทำการลดด้วยกฎ S →(S) S โดย pop สตริง (S) S ออกจากสแตค (reduce: S →(S) S) ดังนั้นสถานะของออโตมาตาจะถอยไปตามที่ pop สตริงออกจากสแตค แล้วจึงเปลี่ยนสถานะตามสัญลักษณ์ S ที่ลงในสแตค

ที่สถานะ 5

- จากรายการ S' →S · เมื่อตัวแจงส่วนได้สัญลักษณ์ $ ที่บอกว่าอ่านสายของโทเคนหมดแล้ว ตัวแจงส่วนจะยอมรับสายของโทเคนที่อ่านมา และ ส่งต้นไม้แจงส่วนกลับไป

ตัวอย่างต่อไปนี้แสดงการแจงส่วนแบบ SLR(1) โดยใช้ตารางการแจงส่วนในตัวอย่าง 3.35 สแตคเก็บสัญลักษณ์ซึ่งเป็นโหนดในต้นไม้แจงส่วนคู่กับสถานะ เช่น S **2** หมายถึง โหนดที่เก็บสัญลักษณ์ S กับสถานะ 2

ตัวอย่าง 3.36 จากตารางการแจงส่วนในตัวอย่าง 3.35 หากให้สายของโทเคนที่นำไปสร้างต้นไม้แจงส่วนเป็น (( )) ( ) จะมีการทำงานดังแสดงต่อไปนี้

สแตค (ล่าง -> บน)	สายของโทเคน	การกระทำ	การแปลง
$ 0	(( )) ( ) $		
		เลื่อน	
$ 0 (1	( )) ( ) $		
		เลื่อน	
$ 0 (1 (1	)) ( ) $		(( )) ( )
		ลด S → ε	1 ⇒
$ 0 (1 (1 S 2	)) ( ) $		
		เลื่อน	
$ 0 (1 (1 S 2) 3	) ( ) $		((S)) ( )
		ลด S → ε	2 ⇒
$ 0 (1 (1 S 2) 3 S 4	) ( ) $		((S) S) ( )
		ลด S → (S) S	3 ⇒
$ 0 (1 S 2	) ( ) $		
		เลื่อน	
$ 0 (1 S 2) 3	( ) $		
		เลื่อน	
$ 0 (1 S 2) 3 (1	) $		(S) ( )
		ลด S → ε	4 ⇒
$ 0 (1 S 2) 3 (1 S 2	) $		
		เลื่อน	
$ 0 (1 S 2) 3 (1 S 2) 3	$		(S) (S)
		ลด S → ε	5 ⇒
$ 0 (1 S 2) 3 (1 S 2) 3 S 4	$		(S) (S) S
		ลด S → (S) S	6 ⇒
$ 0 (1 S 2) 3 S 4	$		(S) S
		ลด S → (S) S	7 ⇒
$ 0 S 5	$		S
		ยอมรับ	

สังเกตการลด S → ε ครั้งแรก สแตคเปลี่ยนจาก $ 0 (1 (1 เป็น $ 0 (1 (1 S 2 เพราะ pop ε ออกจากสแตคคือไม่มีการเปลี่ยนสแตค ดังนั้นออโตมาตาอยู่ที่สถานะ 1 และเมื่อได้ S จึงเปลี่ยนไปอยู่ในสถานะ 2 ตามที่ระบุในตารางการแจงส่วน

สังเกตการลด S → (S) S ครั้งแรก สแตคเปลี่ยนจาก $ 0 (1 (1 S 2) 3 S 4 เป็น $ 0 (1 S 2 เพราะ pop (S) S ออกจากสแตคพร้อมทั้งสถานะที่อยู่คู่กัน เหลือ $ 0 (1 ในสแตค จากตารางการแจงส่วนเมื่ออยู่ที่สถานะ 1 และได้ S จึงเปลี่ยนไปอยู่ในสถานะ 2 ดังนั้นจึงใส่ S 2 ลงในสแตค

ตารางการแจงส่วนแบบ SLR(1) ระบุว่าตัวแจงส่วนจะเลือกทำการเลื่อนหรือลดด้วยกฎใด ช่องที่ไม่ระบุว่าตัวแจงส่วนสามารถเลือกทำงานอย่างไร หมายถึง สถานการณ์ที่มีข้อผิดพลาดทางไวยากรณ์ (syntax error) ในสายของโทเคนดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 3.37 จากตารางการแจงส่วนในตัวอย่าง 3.35 หากให้สายของโทเคนที่นำไปสร้างต้นไม้แจงส่วนเป็น ( )) ที่ไม่ถูกต้องตามไวยากรณ์ที่กำหนด การแจงส่วนจะเป็นดังนี้

สแตค (ล่าง -> บน)	สายของโทเคน	การกระทำ	การแปลง
$ 0	( )) $		
		เลื่อน	
$ 0 (1	)) $		
		ลด S → ε	
$ 0 (1 S 2	)) $		(( )) ( )
		เลื่อน	⇒
$ 0 (1 S 2) 3	) $		
		ลด S → ε	
$ 0 (1 S 2) 3 S 4	) $		((S)) ( )
		ลด S → (S) S	⇒
$ 0 S 5	) $		((S) S) ( )
		ผิดไวยากรณ์	

เมื่ออยู่ที่สถานะ 5 และได้โทเคน) ตารางแจงส่วนไม่มีการทำงานที่ใช้ได้ นั่นคือมีข้อผิดพลาดทางไวยากรณ์

การแจงส่วนแบบ SLR(1) มีความสามารถมากกว่าการแจงส่วนแบบ LL(1) เพราะสามารถใช้กับไวยากรณ์ที่มีตัวร่วมทางซ้ายได้ดังแสดงในตัวอย่างต่อไปนี้

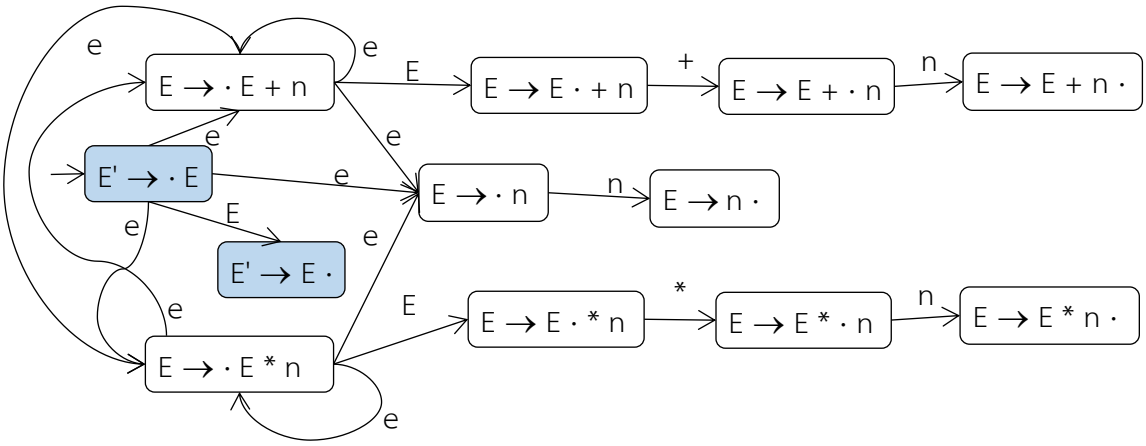
ตัวอย่าง 3.38 กำหนดไวยากรณ์ที่มีตัวร่วมทางซ้ายดังนี้

$E \rightarrow E + n,$
$E \rightarrow E * n,$
$E \rightarrow n$

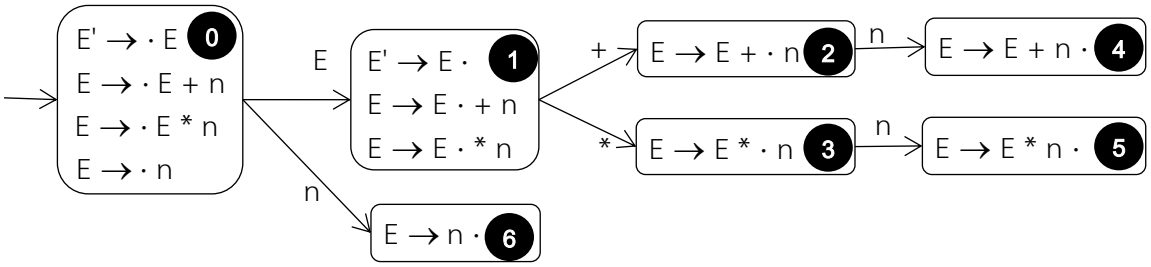
และมีรายการ LR(0) ต่อไปนี้

$E \rightarrow \cdot E + n,$ $E \rightarrow E \cdot + n,$ $E \rightarrow E + \cdot n,$
$E \rightarrow E + n \cdot ,$ $E \rightarrow \cdot E * n,$ $E \rightarrow E \cdot * n,$
$E \rightarrow E * \cdot n,$ $E \rightarrow E * n \cdot,$ $E \rightarrow \cdot n$ และ
$E \rightarrow n \cdot$

เราสามารถสร้างออโตมาตาจำกัดเชิงไม่กำหนดของรายการ LR(0) ได้ดังแสดงในรูปข้างล่างนี้



ออโตมาตาจำกัดเชิงไม่กำหนดนี้แปลงเป็นออโตมาตาจำกัดเชิงกำหนดดังแสดงข้างล่าง



เราสร้างตารางการแจงส่วนแบบ SLR(1) จากออโตมาตาจำกัดเชิงกำหนดนี้และการหา follow(E) = {+, *, $} ได้ดังนี้

Symbol / State	n	+	*	$	E
0	shift & goto 6				goto 1
1		shift & goto 2	shift & goto 3	accept	
2	shift & goto 4				
3	shift & goto 5				
4		reduce:E → E + n	reduce: E → E + n	reduce: E → E + n	
5		reduce:E → E * n	reduce: E → E * n	reduce: E → E * n	
6		reduce:E → n	reduce: E → n	reduce: E → n	

3.5.4 ไวยากรณ์แบบ non-SLR(1)

การแจงส่วนแบบ SLR(1) อาจมีสถานการณ์ที่มีการทำงานที่เป็นไปได้มากกว่าหนึ่งแบบ เราเรียกว่าเกิด**ข้อขัดแย้ง (conflict)** นั่นคือในตารางการแจงส่วนแบบ SLR(1) มีการลดและการเลื่อนในช่องเดียวกันที่เรียกว่า**ข้อขัดแย้งแบบเลื่อน-ลด (shift-reduce conflict)** หรือ มีการลดด้วยกฎหลายกฎในช่องเดียวกันที่เรียกว่า**ข้อขัดแย้งแบบลด-ลด (reduce-reduce conflict)** ไวยากรณ์ที่ทำให้เกิดข้อขัดแย้งในตารางการแจงส่วนแบบ SLR(1) เรียกว่า **ไวยากรณ์แบบ non-SLR(1)** ตัวอย่างต่อไปนี้แสดงในไวยากรณ์แบบ non-SLR(1)

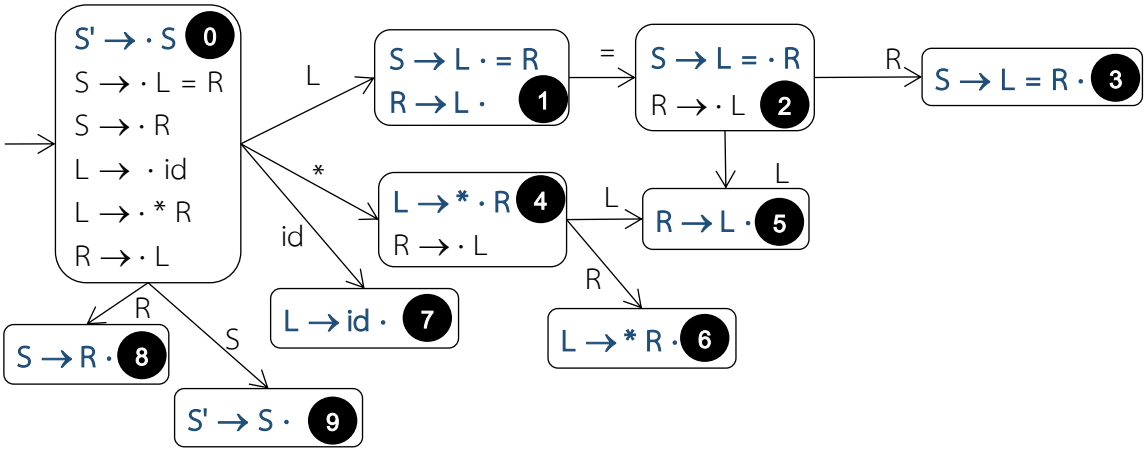
ตัวอย่าง 3.39 กำหนดไวยากรณ์ต่อไปนี้

$S' \rightarrow S$, $S \rightarrow L = R$, $S \rightarrow R$, $L \rightarrow id$, $L \rightarrow * R$, $R \rightarrow L$

และมีรายการ LR(0) ต่อไปนี้

$S' \rightarrow \cdot S$, $S' \rightarrow S \cdot$,
$S \rightarrow \cdot L = R$, $S \rightarrow L \cdot = R$, $S \rightarrow L = \cdot R$, $S \rightarrow L = R \cdot$,
$S \rightarrow \cdot R$, $S \rightarrow R \cdot$,
$L \rightarrow \cdot id$, $L \rightarrow id \cdot$,
$L \rightarrow \cdot * R$, $L \rightarrow * \cdot R$, $L \rightarrow * R \cdot$,
$R \rightarrow \cdot L$, และ $R \rightarrow L \cdot$

เราสามารถสร้างออโตมาตาจำกัดเชิงกำหนดของรายการ LR(0) ได้ดังแสดงในรูปข้างล่างนี้



เราสร้างตารางการแจงส่วนแบบ SLR(1) จากออโตมาตาจำกัดเชิงกำหนดนี้และการหา follow(S) = {$}, follow(L) = {=, $} follow(R) = {=, $} ได้ดังนี้

Symbol / State	=	*	id	$	S	L	R
0		shift & goto 4	shift & goto 7		goto 9	goto 1	goto 8
1	reduce: R → L shift & goto 2			reduce: R → L			
2						goto 5	goto 3
3				reduce: S → L = R			
4						goto 5	goto 6
5	reduce: R → L			reduce: R → L			
6	reduce: L → * R			reduce: L → * R			
7	reduce: L → id			reduce: L → id			
8				reduce: S → R			
9				accept			

จากตารางแจงส่วนข้างบนนี้ เมื่ออยู่ในสถานะ 1 และโทเคนถัดไปเป็น = แล้วตัวแจงส่วนอาจเลือกลดด้วยกฎ R → L หรือเลื่อนและเปลี่ยนสถานะไปยังสถานะ 2 ซึ่งเป็นข้อขัดแย้งแบบเลื่อนลด ดังนั้น ไวยากรณ์นี้เป็นแบบ non-SLR(1)

การแก้ข้อขัดแย้งในไวยกรณ์แบบ non-SLR(1)

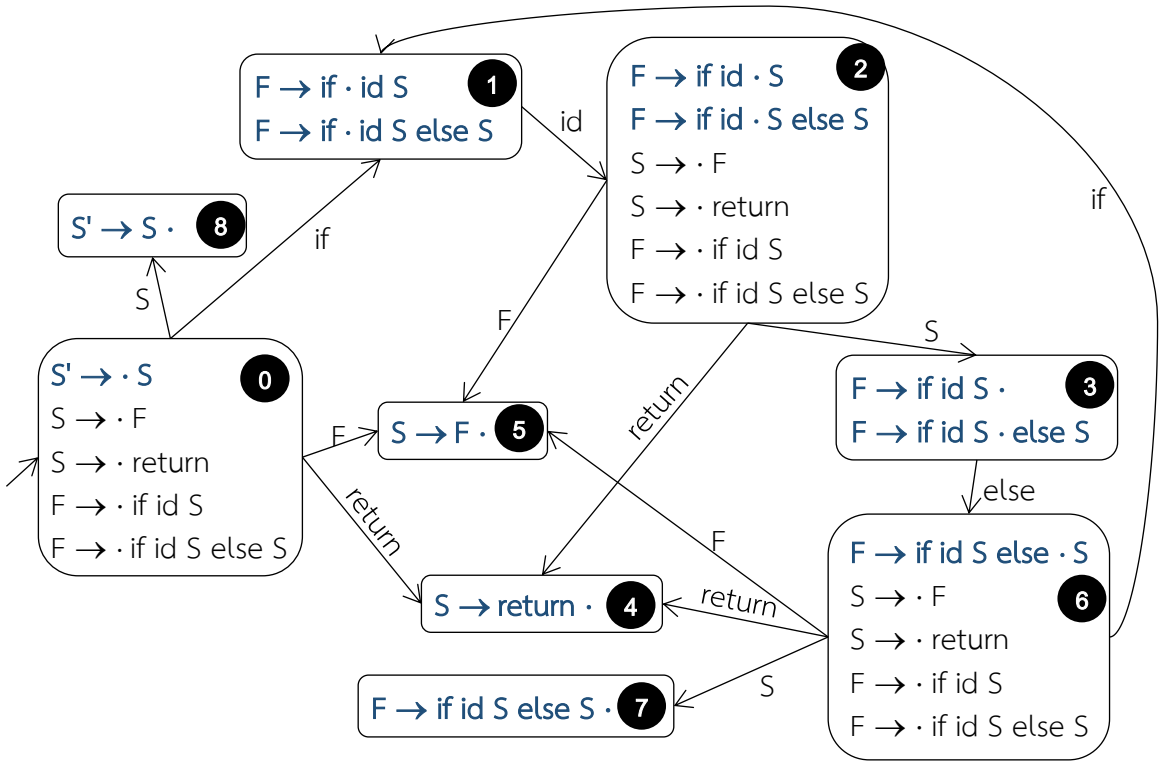
เมื่อมีข้อขัดแย้งในตารางการแจงส่วนแบบ SLR(1) ผู้ออกแบบภาษาอาจเลือกการทำงานหนึ่งแบบจากการทำงานทั้งหมดที่เป็นไปได้ ในไวยากรณ์ของภาษาโปรแกรมส่วนใหญ่ เมื่อมีข้อขัดแย้งแบบเลื่อน-ลด จะเลือกการเลื่อนแทนการลด ซึ่งเป็นการเลือกสร้างโครงสร้างใหญ่แทนที่จะสร้างโครงสร้างเล็ก จากตัวอย่าง 3.39 เมื่อตัวแจงส่วนอยู่ในสถานะ 1 และโทเคนถัดไปเป็น = หากเลือกทำการเลื่อนหมายความว่าตัวแจงส่วนจะสร้างโครงสร้างประโยคกำหนดค่า (assignment statement) จากกฎ S → L = R หากเลือกทำการลดหมายความว่าตัวแจงส่วนจะสร้างโครงสร้างสำหรับนิพจน์ทางขวาของประโยคกำหนดค่าจากกฎ R → L

ไวยากรณ์ของประโยคเงื่อนไข (if statement) มีความกำกวมและทำให้เกิดข้อขัดแย้งได้ดังแสดงในตัวอย่างต่อไปนี้

ตัวอย่าง 3.41 กำหนดไวยากรณ์ของประโยคเงื่อนไขต่อไปนี้ซึ่งเป็นไวยากรณ์กำกวม

S' → S, S → F, S → return, F → if id S, F → if id S else S

เราสามารถสร้างออโตมาตาจำกัดเชิงกำหนดของรายการ LR(0) ได้ดังแสดงในรูปข้างล่างนี้



เราสร้างตารางการแจงส่วนแบบ SLR(1) จากออโตมาตาจำกัดเชิงกำหนดนี้และการหา follow(S) = follow(F) = {$, else} ได้ดังนี้

Symbol / State	id	if	else	return	$	S	F
0		shift & goto 1		shift & goto 4			goto 5
1	shift & goto 2						
2		shift & goto 1		shift & goto 4		goto 3	goto 5
3			shift & goto 6 reduce: F → if id S		reduce: F → if id S		
4			reduce: S → F		reduce: S → F		
5			reduce: S → return		reduce: S → return		
6		shift & goto 1		shift & goto 4		goto 7	goto 5
7			reduce: F → if id S else S				
8					accept		

เมื่อตัวแจงส่วนอยู่ในสถานะ 3 และโทเคนถัดไปเป็น else ตัวแจงส่วนอาจเลือกทำการเลื่อนเพื่อให้ได้โครงสร้าง if id S else S เพื่อให้ใช้กฎ F → if id S else S ได้ในเวลาต่อมา ดังนั้นจะทำให้สร้างประโยคเงื่อนไขแบบไม่มี else ก่อน วิธีนี้ทำให้จับคู่ else กับ if ใกล้สุดก่อนหน้าที่ยังไม่มีคู่ก่อน หากตัวแจงส่วนเลือกทำการลดโดยใช้กฎ F → if id S ทำให้เลือกสร้างประโยคเงื่อนไขแบบไม่มี else ก่อน ในกรณีนี้จะไม่จับคู่ else กับ if ใดเลย ดังนั้นสำหรับภาษาโปรแกรมที่มีคำสั่งเงื่อนไขเช่นนี้ เราอาจสร้างตัวแจงส่วนแบบ SLR(1) และแก้ข้อขัดแย้งนี้โดยเลือกการเลื่อนแทนการลด

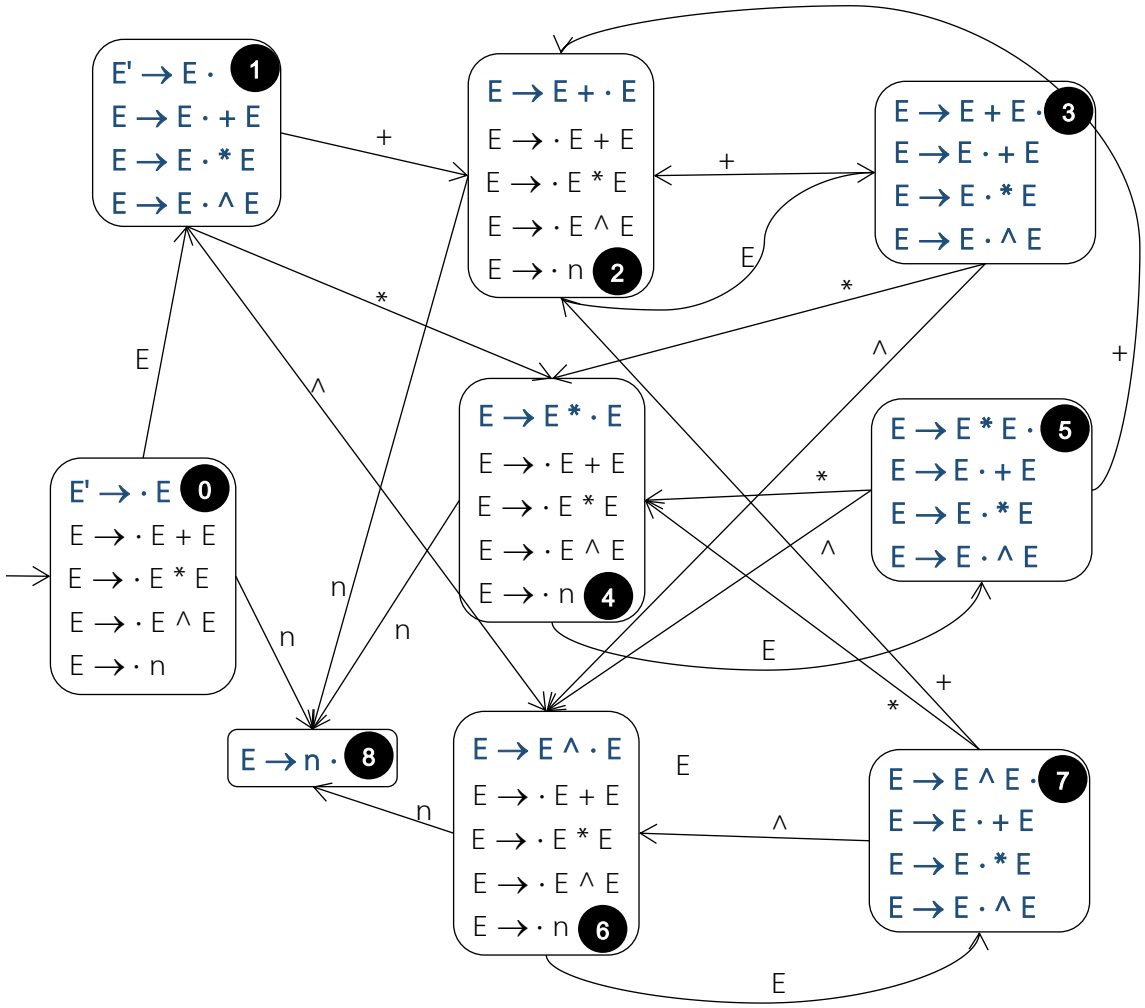
ตัวอย่างต่อไปนี้เป็นการสร้างตารางการแจงส่วนแบบ SLR(1) ของไวยากรณ์ที่อธิบายนิพจน์ที่ใช้ตัวกระทำ + * และ ^

ตัวอย่าง 3.41 กำหนดไวยากรณ์ต่อไปนี้

E → E + E, E → E * E, E → E ^ E, E → n

ไวยากรณ์นี้เป็นไวยากรณ์กำกวมที่ไม่ระบุว่าตัวกระทำใดในนิพจน์จะถูกทำก่อน ดังนั้นเราอาจใช้ลำดับความสำคัญและสมบัติการเชื่อมโยงของตัวกระทำมาใช้แก้ข้อขัดแย้ง ในที่นี้ให้ตัวกระทำ ^ * + มีลำดับความสำคัญเรียงจากมากไปน้อย ตัวกระทำ + และ * มีสมบัติการเชื่อมโยงทางซ้าย และตัวกระทำ ^ มีสมบัติการเชื่อมโยงทางขวา

เราสามารถสร้างออโตมาตาจำกัดเชิงกำหนดของรายการ LR(0) ได้ดังแสดงในรูปข้างล่างนี้



เราสร้างตารางการแจงส่วนแบบ SLR(1) จากออโตมาตาจำกัดเชิงกำหนดนี้และการหา follow(E) = {+, *, ^, $} ได้ดังนี้

Symbol / State	n	+	*	^	$	E
0	shift & goto 6					goto 1
1		shift & goto 2	shift & goto 4	shift & goto 6	accept	
2	shift & goto 8					goto 3
3		shift & goto 2 reduce: E → E+E	shift & goto 4 reduce: E → E+E	shift & goto 6 reduce: E → E+E	reduce: E → E+E	
4	shift & goto 8					goto 5
5		shift & goto 2 reduce: E → E*E	shift & goto 4 reduce: E → E*E	shift & goto 6 reduce: E → E*E	reduce: E → E*E	
6	shift & goto 8					goto 7
7		shift & goto 2 reduce: E → E^E	shift & goto 4 reduce: E → E^E	shift & goto 6 reduce: E → E^E	reduce: E → E^E	
8		reduce: E → n	reduce: E → n	reduce: E → n	reduce: E → n	

ในที่นี้เราจะแก้ข้อขัดแย้งโดยใช้ลำดับความสำคัญและสมบัติการเชื่อมโยงของตัวกระทำ สำหรับข้อขัดแย้ง (การลดด้วยกฎ E → E + E และการเลื่อน) เมื่อได้โทเคน + * หรือ ^ และตัวแจงส่วนอยู่ที่สถานะ 3 ซึ่งมี E + E อยู่ในสแตค เราสามารถแก้ข้อขัดแย้งได้ดังนี้

- เมื่อได้โทเคน + ตัวแจงส่วนจะเลือกทำการลดด้วยกฎ E → E + E เพื่อให้สร้างนิพจน์จากตัวกระทำ + ตัวทางซ้ายก่อนตามสมบัติการเชื่อมโยงทางซ้ายของตัวกระทำ +
- เมื่อได้โทเคน * หรือ ^ ตัวแจงส่วนจะไม่เลือกทำการลด แต่จะเลื่อนโทเคน * หรือ ^ ลงในสแตค เพื่อให้สร้างนิพจน์จากตัวกระทำ * หรือ ^ ก่อนเนื่องจากตัวกระทำ + มีลำดับความสำคัญต่ำกว่าตัวกระทำ * และ ^

สำหรับข้อขัดแย้ง (การลดด้วยกฎ E → E * E และการเลื่อน) เมื่อได้โทเคน + * หรือ ^ และตัวแจงส่วนอยู่ที่สถานะ 5 ซึ่งมี E * E อยู่ในสแตค เราสามารถแก้ข้อขัดแย้งได้ดังนี้

- เมื่อได้โทเคน + ตัวแจงส่วนจะเลือกการลดด้วยกฎ E → E * E เพื่อให้สร้างนิพจน์จากตัวกระทำ * ทางซ้ายก่อนเนื่องจากตัวกระทำ + มีลำดับความสำคัญต่ำกว่าตัวกระทำ *
- เมื่อได้โทเคน * ตัวแจงส่วนจะลดด้วยกฎ E → E * E เพื่อให้สร้างนิพจน์จากตัวกระทำ * ตัวทางซ้ายก่อนตามสมบัติการเชื่อมโยงทางซ้ายของตัวกระทำ *
- เมื่อได้โทเคน ^ ตัวแจงส่วนจะไม่เลือกทำการลด แต่จะเลื่อนโทเคน ^ ลงในสแตคเพื่อให้สร้างนิพจน์จากตัวกระทำ ^ ก่อนเนื่องจากตัวกระทำ ^ มีลำดับความสำคัญสูงกว่าตัวกระทำ *

สำหรับข้อขัดแย้ง (การลดด้วยกฎ E → E ^ E และการเลื่อน) เมื่อได้โทเคน + * หรือ ^ และตัวแจงส่วนอยู่ที่สถานะ 7 ซึ่งมี E ^ E อยู่ในสแตค เราสามารถแก้ข้อขัดแย้งได้ดังนี้

- เมื่อได้โทเคน + หรือ * ตัวแจงส่วนจะเลือกการลดด้วยกฎ E → E ^ E เพื่อให้สร้างนิพจน์จากตัวกระทำ ^ ก่อนเนื่องจากตัวกระทำ + และ * มีลำดับความสำคัญต่ำกว่าตัวกระทำ ^
- เมื่อได้โทเคน ^ ตัวแจงส่วนจะไม่เลือกทำการลด แต่จะเลื่อนโทเคน ^ ลงในสแตคเพื่อให้สร้างนิพจน์จากตัวกระทำ ^ ทางขวาก่อนตามสมบัติการเชื่อมโยงทางขวาของตัวกระทำ ^

4. ตัววิเคราะห์ความหมาย

ในภาษาโปรแกรมเราสนใจความหมายขององค์ประกอบย่อยในโปรแกรมซึ่งรวมเป็นตัวกำหนดความหมายของโปรแกรมโดยรวม ตัวแปลภาษาที่ใช้ในปัจจุบันตรวจสอบเพียงแค่ความหมายระดับพื้นฐานของโครงสร้าง (construct) ในโปรแกรม ได้แก่ ชนิดและขอบเขต (scope) ของตัวแปร ความหมายเหล่านี้ไม่สามารถตรวจสอบจากไวยากรณ์ได้ ตัววิเคราะห์ความหมาย (semantic analyzer) ทำงานกับต้นไม้แจงส่วนที่สร้างโดยตัวแจงส่วน การทำงานร่วมกันระหว่างตัวแจงส่วนและตัววิเคราะห์ความหมายเป็นได้สองแบบ สำหรับแบบที่หนึ่ง ตัววิเคราะห์ความหมายทำงานไปพร้อมกับตัวแจงส่วนในขณะที่กำลังสร้างต้นไม้แจงส่วน สำหรับแบบที่สอง ตัวแจงส่วนสร้างต้นไม้แจงส่วนให้เสร็จก่อนแล้วตัววิเคราะห์ความหมายจึงตรวจสอบความหมายจากต้นไม้ที่ครบสมบูรณ์แล้ว

หัวข้อ 4.1 กล่าวถึงความหมายที่สนใจในภาษาโปรแกรมโดยเน้นในส่วนที่เกี่ยวข้องกับตัววิเคราะห์ความหมาย หัวข้อ 4.2 อธิบายการกำหนดความสัมพันธ์ของความหมายในภาษาโปรแกรมด้วยไวยากรณ์ของลักษณะประจำ (attribute grammar) หัวข้อ 4.3 อธิบายการเขียนโปรแกรมเพื่อตรวจสอบความหมายตามไวยากรณ์ของลักษณะประจำสำหรับภาษาที่กำหนด

4.1 แนวคิดเกี่ยวกับความหมายในภาษาโปรแกรมที่เกี่ยวกับตัววิเคราะห์ความหมาย

ตัวระบุ (identifier) ใช้อ้างถึงสิ่งที่โปรแกรมเมอร์ต้องการสร้างเพื่อใช้ในโปรแกรม เช่น ตัวแปร ฟังก์ชัน คลาส (class) ชนิดข้อมูลที่ผู้ใช้กำหนด (user-defined data type) ตารางสัญลักษณ์เก็บตัวระบุและข้อมูลที่เกี่ยวข้อง ส่วนต่าง ๆ ของตัวแปลภาษาดึงข้อมูลเหล่านี้จากโปรแกรมมาเก็บในตารางสัญลักษณ์

- สแกนเนอร์หาชื่อของตัวระบุในโปรแกรม
- ตัววิเคราะห์ความหมายวิเคราะห์หาประเภทและชนิดของตัวระบุ เช่น เป็นตัวแปรชนิดจำนวนเต็ม เป็นฟังก์ชันที่คืนค่าชนิดจำนวนจริง
- ตัวสร้างรหัสกำหนดที่เก็บของตัวระบุในหน่วยความจำ

ข้อมูลที่เกี่ยวกับตัวระบุที่ได้มานี้จะถูกนำมาเก็บในตารางสัญลักษณ์ ข้อมูลที่เกี่ยวกับความหมายของตัวแปรในแต่ละภาษาอาจต่างกันไป ในที่นี้เราจะพิจารณาชนิดและขอบเขตของตัวแปรที่เป็นข้อมูลเกี่ยวกับความหมายที่ตรวจสอบด้วยตัววิเคราะห์ความหมาย

4.1.1 ชนิดของตัวแปร

ชนิดของตัวแปรในภาษาโปรแกรมแต่ละภาษาอาจต่างกัน ชนิดข้อมูลพื้นฐาน (primitive data type) ที่มีในภาษาโปรแกรมโดยทั่วไป ได้แก่ จำนวนเต็ม จำนวนจริง ตัวอักขระ สายอักขระ เป็นต้น นอกจากนั้นยังมีชนิดข้อมูลที่สร้างจากชนิดข้อมูลพื้นฐาน เรียกว่า ชนิดข้อมูลกลุ่ม (aggregate data type) ข้อมูลกลุ่มที่ประกอบด้วยข้อมูลชนิดเดียวกันทั้งหมด เรียกว่า แถวลำดับ (array) ข้อมูลกลุ่มที่ประกอบด้วยข้อมูลชนิดต่างกัน ได้แก่ struct ในภาษาซี tuple ในภาษาไพธอน

ภาษาโปรแกรมบางภาษา เช่น ภาษาซี ภาษาจาวา ไม่ให้ตัวแปรเปลี่ยนชนิดระหว่างการทำงาน ตัวแปรแบบนี้เป็นตัวแปรแบบ static type ในภาษาที่ใช้ตัวแปรแบบนี้ ตัวแปรมีชนิดตามที่ประกาศไว้ในคำสั่งประกาศตัวแปร (declaration statement) และไม่อาจเปลี่ยนจากชนิดหนึ่งไปเป็นอีกชนิดหนึ่งในระหว่างการทำงานได้ ภาษาโปรแกรมบางภาษา เช่น ภาษาไพธอน ยอมให้ตัวแปรเปลี่ยนชนิดระหว่างการทำงาน ตัวแปรแบบนี้เป็นตัวแปรแบบ dynamic type

ในโปรแกรมภาษาไพธอนข้างล่างนี้ เมื่อทำงานในบรรทัดที่ 1 `x` เป็นตัวแปรชนิดจำนวนเต็ม จากนั้นเมื่อรับข้อมูลเข้าในบรรทัดที่ 2 และข้อมูลเข้าเป็น `'f'` แล้วค่าที่นำไปใส่ในตัวแปร `x` ในบรรทัดที่ 3 มีชนิดเป็นจำนวนจริง ดังนั้นตัวแปร `x` เปลี่ยนชนิดเป็นจำนวนจริง หากข้อมูลเข้าไม่เป็น `'f'` ค่าที่นำไปใส่ในตัวแปร `x` ในบรรทัดที่ 6 มีชนิดสายอักขระ ดังนั้นตัวแปร `x` เปลี่ยนชนิดสายอักขระ

```
1: x = 0
2: t = input('choose type :')
3: if t=='f':      x=3.14159
4: else:           x='error'
```

การยอมให้ตัวแปรเปลี่ยนชนิดได้ระหว่างที่โปรแกรมทำงานทำให้เขียนโปรแกรมง่ายขึ้น แต่การจัดการตัวแปรในภาษาประเภทนี้มีขั้นตอนมากกว่าภาษาที่ไม่ให้เปลี่ยนชนิดของตัวแปร สำหรับภาษาไม่ให้เปลี่ยนชนิดของตัวแปร ตัวสร้างรหัส จะเก็บตำแหน่งในหน่วยความจำ (memory address) ของตัวแปรไว้ในตารางสัญลักษณ์และตำแหน่งนี้ไม่เปลี่ยนตลอดอายุของตัวแปร ตัวสร้างรหัสสร้างโปรแกรมในภาษาระดับที่ใกล้เคียงกับภาษาเครื่องโดยใช้ตารางสัญลักษณ์ดังแสดงในรูปข้างล่าง (ภาษาที่ใช้ในตัวอย่างนี้มีความหมายเหมือนภาษาซี)

name	type	…	address
…	…	…	…
x	int	…	0318
…	…	…	…

คำสั่ง: `x = 3`

Intermediate code:

```
t = 0318 //addr of x, stored in the symbol table
*t = 3
```

หากยอมให้ตัวแปรเปลี่ยนชนิดได้ ตำแหน่งในหน่วยความจำที่เก็บค่าของตัวแปรหนึ่งอาจมีการเปลี่ยนแปลงเมื่อตัวแปรเปลี่ยนชนิดและต้องใช้หน่วยความจำมากขึ้น (เช่น เมื่อตัวแปรเปลี่ยนจากชนิดตัวอักขระที่ใช้หน่วยความจำ 2 ไบท์ เป็นชนิดจำนวนเต็มที่ใช้หน่วยความจำ 4 ไบท์) เนื่องจากหน่วยความจำที่ตำแหน่งเดิมอาจไม่มีที่ว่างพอจึงต้องหาหน่วยความจำที่ตำแหน่งอื่น อย่างไรก็ตามการเปลี่ยนชนิดของตัวแปรและย้ายตำแหน่งที่เก็บค่าจะเกิดขึ้นในระหว่างที่โปรแกรมทำงาน แต่ตัวสร้างรหัสจะต้องกำหนดตำแหน่งในหน่วยความจำที่ใช้ในการเข้าถึงตัวแปรในขั้นตอนของการแปลภาษาแล้ว ดังนั้นจึงให้ตารางสัญลักษณ์เก็บตำแหน่งในหน่วยความจำที่ใช้เก็บตำแหน่งที่เก็บค่าจริงดังแสดงในรูปข้างล่าง

name	type	…	address
…	…	…	…
x	int	…	0318
…	…	…	…

Memory address	ค่า
…	…
0318	0577
…	…
0577	3

คำสั่ง: `x = 3`

Intermediate code:

```
t = 0318 //from symbol table
a = *t   //a = 0577
*a= 3    //store 3 at mem# 0577
```

เมื่อมีการเปลี่ยนชนิดของตัวแปรและต้องการเพิ่มขนาดของตัวแปรในหน่วยความจำ ระบบจะหาที่ในหน่วยความจำให้และนำตำแหน่งใหม่นี้มาเก็บแทนและสามารถใช้งานรหัสกลางเดิมดังแสดงในรูปข้างล่าง

name	type	…	address
… x …	… int …	… … …	… 0318 …

Memory address	ค่า
… 0318 … 0577 … 0610	… 0610 … 3 … 3.5

คำสั่ง: `x = 3.5`

Intermediate code:

```
t = 0318 //from symbol table
a = *t   //a = 0610
*a= 3.5 //store 3.5 at mem# 0610
```

วิธีการนี้ทำให้สามารถเปลี่ยนชนิดของตัวแปรขณะที่โปรแกรมทำงานได้แต่การเข้าถึงค่าในตัวแปรมีขั้นตอนมากขึ้นกว่าภาษาที่ไม่ยอมให้เปลี่ยนตัวแปรขณะทำงาน

4.1.2 ขอบเขต (scope) ของตัวแปร

ขอบเขตของตัวแปรหมายถึงตำแหน่งในโปรแกรมที่สามารถอ้างถึงตัวแปรนั้นได้ ในภาษาโปรแกรมที่ใช้กันในปัจจุบันตัวแปรมีขอบเขตแบบสถิต (static scope) นั่นคือขอบเขตของตัวแปรระบุโดยตำแหน่งของการประกาศตัวแปรโดยไม่ขึ้นกับลำดับการทำงานของคำสั่งในโปรแกรม ภาษาซีและภาษาจาวาใช้ขอบเขตแบบสถิต นั่นคือตัวแปรตัวหนึ่งจะมองเห็นได้ภายในบล็อคที่ประกาศตัวแปรไว้ ในตัวอย่างโปรแกรมภาษาซีข้างล่างนี้ ในบล็อคของฟังก์ชัน `blk0` จะมองเห็นตัวแปร `x` ที่ประกาศในบล็อกของตัวเองและมองไม่เห็นตัวแปร `x` ที่ประกาศในบล็อก `blk1` ทำนองเดียวกันในบล็อคของฟังก์ชัน `blk1` มองเห็นตัวแปร `x` ในบล็อคของฟังก์ชัน `blk1` และ ไม่เห็นตัวแปร `x` ในบล็อคของฟังก์ชัน ฟังก์ชัน `blk0` ขอบเขตของตัวแปรกำหนดด้วยขอบเขตของบล็อคที่ประกาศตัวแปรนั้นไว้ ขอบเขตนี้ขึ้นกับการกำหนดบล็อกในโปรแกรมซึ่งไม่มีการเปลี่ยนแปลงในขณะทำงานดังนั้นจึงเรียกขอบเขตแบบนี้ว่าขอบเขตแบบสถิต

```
void blk0(void) {
   int x = 1;
   printf("x=%d", x) // x=1 here.
   y = blk1();
   printf("x=%d", x) // x=1 still.
}
```

```
void blk1(void) {
   int x = 0;

   printf("%d", x) // x=0 here.
}
```

สำหรับภาษาที่ใช้ขอบเขตแบบพลวัต (dynamic scope) ขอบเขตของตัวแปรขึ้นกับลำดับการเรียกฟังก์ชัน ในตัวอย่างโปรแกรมข้างล่างนี้ ฟังก์ชัน `blk0` เรียกฟังก์ชัน `blk1` ดังนั้นตัวแปรในฟังก์ชัน `blk0` จะมองเห็นจากฟังก์ชัน `blk1` ด้วย ดังนั้นตัวแปร `x` ในฟังก์ชัน `blk1` เป็นตัวแปรเดียวกันกับในฟังก์ชัน `blk0`

```
void blk0(void) {
   int x = 1;

   printf("x=%d", x) // x is 1 here.
   y = blk1();
   printf("x=%d", x) // x is 0 here.
}
```

```
void blk1(void) {
   x = 0;  // x is changed here.
   printf("%d", x) // x is 0 here.
}
```

เนื่องจากตัวแปรที่อ้างถึงในฟังก์ชันมีขอบเขตเปลี่ยนไปตามการเรียกใช้ฟังก์ชันทำให้อ่านโปรแกรมเข้าใจได้ยาก ภาษาโปรแกรมที่ใช้ในปัจจุบันจึงใช้ขอบเขตแบบสถิต ตัววิเคราะห์ความหมายมีหน้าที่ตรวจสอบเกี่ยวกับขอบเขตของตัวแปรด้วย นั่นคือเมื่อมีการอ้างถึงตัวแปรแต่ละครั้งในโปรแกรม ตัววิเคราะห์ความหมายต้องตรวจสอบว่าอยู่ในขอบเขตของตัวแปรนั้นหรือไม่หากมีการอ้างถึงตัวแปรนอกขอบเขตของตัวแปรนั้น ตัววิเคราะห์ความหมายต้องรายงานข้อผิดพลาดด้วย

4.2 การกำหนดความสัมพันธ์ของความหมายด้วยไวยากรณ์ของลักษณะประจำ

ในการออกแบบภาษาโปรแกรมผู้ออกแบบต้องกำหนดความหมายในภาษาที่ต้องการตรวจสอบ ในภาษาโปรแกรมทั่วไปความหมายที่ตรวจสอบเกี่ยวกับชนิดข้อมูลของส่วนประกอบในโปรแกรม ผู้ออกแบบภาษาต้องกำหนดชนิดข้อมูลที่ต้องใช้ในโครงสร้างต่างๆของโปรแกรมและชนิดข้อมูลที่เป็นผลลัพธ์ของโครงสร้างใหม่โปรแกรม เครื่องมือที่ใช้อธิบายข้อกำหนดเหล่านี้ ให้ชัดเจนเพื่อนำไปสร้างตัววิเคราะห์ความหมายได้ถูกต้องคือไวยากรณ์ลักษณะประจำ

4.2.1 ไวยากรณ์ของลักษณะประจำ

ไวยากรณ์ของลักษณะประจำใช้อธิบายความหมายขององค์ประกอบย่อยในโปรแกรม ไวยากรณ์ของลักษณะประจำประกอบด้วย (1) ไวยากรณ์แบบไม่พึ่งบริบท (2) ลักษณะประจำ (attribute) และ (3) กฎของความหมาย (semantic rule)

- ไวยากรณ์แบบไม่พึ่งบริบทเป็นชุดของกฎที่อธิบายโครงสร้างของภาษาดังกล่าวไปแล้วในบทที่ 3
- ลักษณะประจำของโครงสร้างทางไวยากรณ์เป็นสิ่งที่เราใช้เก็บความหมายที่ต้องการตรวจสอบในขั้นตอนนี้ ลักษณะประจำที่ใช้ในตัววิเคราะห์ความหมายอาจเป็นชนิดหรือขอบเขตของตัวแปร
- กฎของความหมายระบุความสัมพันธ์ระหว่างลักษณะประจำของโครงสร้างย่อยในโครงสร้างใหญ่

ตัวอย่างข้างล่างนี้เป็นการกำหนดชนิดของตัวแปรในประโยคประกาศตัวแปร (declaration statement) ด้วยไวยากรณ์ของลักษณะประจำ ลักษณะประจำ `type` แทนชนิดข้อมูลซึ่งเป็นความหมายที่ต้องการ `varlist.type` แทนชนิดข้อมูลของโครงสร้าง `varlist` ไวยากรณ์ลักษณะประจำนี้หาชนิดข้อมูลในประโยคประกาศตัวแปรแล้วเก็บชนิดของตัวแปรในตารางสัญลักษณ์

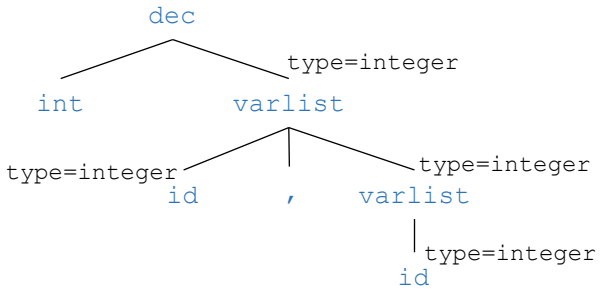
	Grammar rule	Semantic rule
1	`dec` → **`int`** `varlist`	`varlist.type = integer`
2	`dec` → **`float`** `varlist`	`varlist.type = real`
3	`varlist` → **`id`**	`id.type = varlist.type` `addType(id, id.type)`

4	$varlist_1$ → **`id`** , $varlist_2$	`id.type = varlist`$_1$`.type`
		`addType(id, id.type)`
		$varlist_2$`.type = varlist`$_1$`.type`

กฎของความหมายสองกฎแรกในไวยากรณ์ลักษณะประจำนี้กำหนดว่า ในประโยคประกาศตัวแปรที่แทนด้วยสัญลักษณ์ `dec` รายการของตัวแปรที่ตามมา (`varlist`) มีชนิดข้อมูลเป็น `integer` เมื่อประโยค

ประกาศตัวแปรขึ้นต้นด้วยโทเคน **int** แต่รายการของตัวแปรที่ตามมามีชนิดข้อมูลเป็น real เมื่อประโยคประกาศตัวแปรขึ้นต้นด้วยโทเคน **float** กฎที่ 3 กำหนดว่า เมื่อรายการของตัวแปร (varlist) ประกอบด้วยตัวแปร (id) เดียว ชนิดของตัวแปรนั้น (id.type) เป็นชนิดเดียวกับชนิดข้อมูลของรายการของตัวแปร (varlist.type) และให้เพิ่มชนิดของตัวแปรนั้นในตารางสัญลักษณ์ด้วยฟังก์ชัน addType กฎที่ 4 กำหนดว่า เมื่อรายการของตัวแปร ($varlist_1$) ประกอบด้วยตัวแปร (id) ตัวหนึ่งกับรายการของตัวแปรอีกชุด (เช่น x, y, z) ชนิดของตัวแปรนั้น (id.type) และชนิดข้อมูลของรายการตัวแปรที่เหลือ ($varlist_2$.type) เป็นชนิดเดียวกับชนิดข้อมูลของรายการของตัวแปร ($varlist_1$.type) ทั้งหมดและให้เพิ่มชนิดของตัวแปรในตารางสัญลักษณ์ หลังจากนี้ตัวแปลภาษาสามารถดึงชนิดของตัวแปรที่เก็บไว้ในตารางสัญลักษณ์มาใช้ในการตรวจสอบชนิดข้อมูล

จากไวยากรณ์ลักษณะประจำที่กำหนดนี้ ต้นไม้แจงส่วนของโปรแกรมที่มีประโยคประกาศตัวแปร int x, y แสดงด้วยรูปข้างล่างโดยแสดงลักษณะประจำ type กำกับที่โหนด varlist และ id กฎของความหมายกฎแรกทำให้ได้ลักษณะประจำ type ของโหนด varlist ที่ความสูงเป็น 1 มีค่าเป็น integer จากค่านี้และกฎข้อที่ 4 ทำให้ได้ลักษณะประจำ type ของโหนด id และโหนด varlist ที่ความสูงเป็น 2 มีค่าเป็น integer จากกฎข้อที่ 3 และค่าที่โหนด varlist ที่ได้นี้ทำให้ได้ลักษณะประจำ type ของโหนด id ที่ความสูงเป็น 3 มีค่าเป็น integer ต้นไม้แจงส่วนรวมกับลักษณะประจำของแต่ละโหนดที่แสดงข้างบนนี้เรียกว่า ต้นไม้แจงส่วนประกอบความหมาย (annotated parse tree)



4.2.2 การส่งค่าของลักษณะประจำในต้นไม้แจงส่วนประกอบความหมาย

การส่งค่าของลักษณะประจำในต้นไม้แจงส่วนประกอบความหมายเป็นปัจจัยที่ต้องพิจารณาในการเขียนโปรแกรมเพื่อคำนวณค่าของลักษณะประจำ เมื่อพิจารณากฎของความหมาย $L \rightarrow R_1\ R_2\ \ldots\ R_n$ ในไวยากรณ์ลักษณะประจำที่มีลักษณะประจำ $a, a_1, a_2, \ldots, a_n$ การส่งค่าของลักษณะประจำในต้นไม้แจงส่วนประกอบความหมายมีได้ 4 แบบดังนี้

- การส่งค่าจากบนลงล่างโดยส่งค่าจากโหนด L ลงไปยังโหนด R_1, R_2 … หรือ R_n ด้วยกฎของความหมาย $R_1.a=f(L.a_1)$, $R_2=g(L.a_2), \ldots$, $R_n=h(L.a_n)$ เมื่อ f, g,…, h เป็นฟังก์ชัน
- การส่งค่าจากล่างขึ้นบนโดยส่งค่าจากโหนด R_1, R_2 … หรือ R_n ขึ้นไปยังโหนด L ด้วยกฎของความหมาย $L.a=f(R_1.a_1, R_2.a_2, \ldots, R_n.a_n)$ เมื่อ f เป็นฟังก์ชัน

- การส่งค่าจากซ้ายไปขวาโดยส่งค่าจากโหนดลูก R_i ไปยังโหนดลูกที่อยู่ทางขวา R_j (เช่น จากโหนด R_2 ไปยังโหนด R_4) ด้วยกฎของความหมาย $R_j.a=f(R_i.a_i)$ เมื่อ f เป็นฟังก์ชัน
- การส่งค่าจากขวาไปซ้ายโดยส่งค่าจากโหนดลูก R_i ไปยังโหนดลูกที่อยู่ทางซ้าย R_j (เช่น จากโหนด R_4 ไปยังโหนด R_2) ด้วยกฎของความหมาย $R_j.a=f(R_i.a_i)$ เมื่อ f เป็นฟังก์ชัน

ตัวอย่างต่อไปนี้แสดงไวยากรณ์ลักษณะประจำที่ส่งค่าของลักษณะประจำจากบนลงล่าง เมื่อพิจารณากฎไวยากรณ์ varlist → id ที่มีกฎของความหมาย id.type=varlist.type จะเห็นว่า type ของโหนด id ได้ค่ามาจาก type ของโหนด varlist ซึ่งเป็นโหนดแม่และเพิ่มชนิดข้อมูลของตัวแปร id ลงในตารางสัญลักษณ์ ทำนองเดียวกัน สำหรับกฎไวยากรณ์ $varlist_1$ → id , $varlist_2$ ที่มีกฎของความหมาย id.type = $varlist_1$.type และ $varlist_2$.type = $varlist_1$.type จะเห็นว่า type ของโหนด id และ $varlist_2$ ได้ค่ามาจาก type ของโหนด $varlist_1$ ซึ่งเป็นโหนดแม่

	Grammar rule	Semantic rule
1	dec → **int** varlist	varlist.type = integer
2	dec → **float** varlist	varlist.type = real
3	varlist → **id**	id.type = varlist.type addType(id, id.type)
4	$varlist_1$ → **id** , $varlist_2$	id.type = $varlist_1$.type addType(id, id.type) $varlist_2$.type = $varlist_1$.type

ตัวอย่างต่อไปนี้แสดงไวยากรณ์ลักษณะประจำที่ส่งค่าของลักษณะประจำจากล่างขึ้นบน ไวยากรณ์ลักษณะประจำนี้ใช้ลักษณะประจำ type แทนชนิดข้อมูลที่ได้จากนิพจน์ซึ่งสร้างจากตัวแปร นิพจน์ย่อย และตัวกระทำ ฟังก์ชัน getType(id) ใช้หาชนิดของตัวแปร id จากตารางสัญลักษณ์ ในที่นี้สมมติว่าลักษณะประจำ type มีค่าเป็น integer หรือ real เท่านั้น

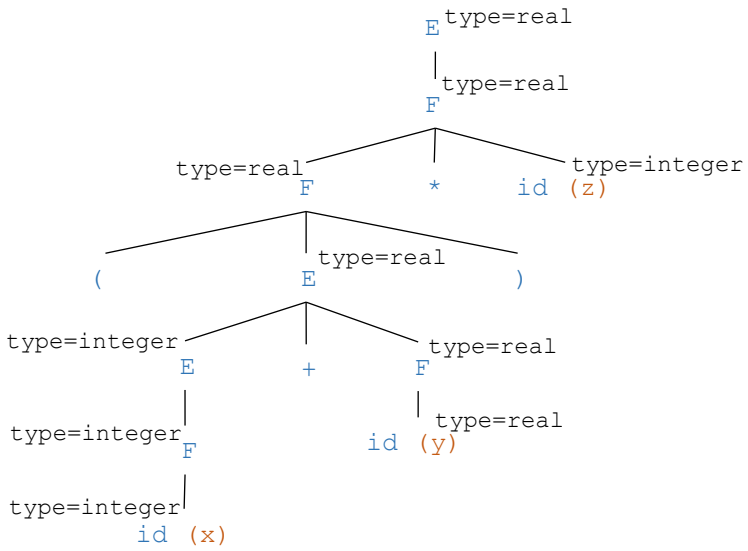
	Grammar rule	Semantic rule
1	E_1 → E_2 **+** F	E_1.type = (E_2.type==F.type)?F.type:real
2	E → F	E.type = F.type
3	F_1 → F_2 ***** **id**	F_1.type = (F_2.type==getType(id))?F_2.type:real
4	F → **id**	F.type = getType(id)
5	F → **(** E **)**	F.type=E.type

กฎของความหมายในแถวแรกใช้เมื่อนิพจน์สร้างจากนิพจน์ย่อย 2 นิพจน์และตัวกระทำ + กฎนี้ระบุว่า ถ้านิพจน์ย่อยมีชนิดเดียวกัน (integer ทั้งคู่ หรือ real ทั้งคู่) ชนิดของนิพจน์ใหญ่จะเป็นชนิดเดียวกับนิพจน์ย่อย ถ้านิพจน์ย่อยมีชนิดต่างกัน (integer กับ real) นิพจน์ใหญ่เป็นชนิด real ดังนั้นค่าของ E_1.type ส่งขึ้นมาจากค่าของ E_2.type และ F.type ในโหนดลูก ในทำนองเดียวกัน กฎของความหมายในแถว 3 ใช้เมื่อนิพจน์สร้างจากตัวกระทำ * นิพจน์ย่อย และตัวแปร กฎนี้ระบุว่า ถ้านิพจน์ย่อยและตัวแปรมีชนิดเดียวกัน ชนิดของนิพจน์ใหญ่จะเป็นชนิดเดียวกับนิพจน์ย่อย ถ้านิพจน์ย่อยมีชนิดต่างกับตัวแปร นิพจน์ใหญ่เป็นชนิด real ดังนั้นค่าของ F_1.type ส่งขึ้นมาจากค่าของ F_2.type และ getType(id) ในโหนดลูก

กฎของความหมายในแถว 2 และ 5 ใช้เมื่อนิพจน์สร้างจากนิพจน์ย่อยนิพจน์เดียว กฎนี้ระบุว่านิพจน์ใหญ่มีชนิดเดียวกันนิพจน์ย่อย ดังนั้น จากกฎของความหมายในแถว 2 ค่าของ `E.type` ส่งขึ้นมาจากค่าของ `F.type` ในโหนดลูก และจากกฎของความหมายในแถว 5 ค่าของ `F.type` ส่งขึ้นมาจากค่าของ `E.type` ในโหนดลูก

กฎของความหมายในแถว 4 ใช้เมื่อนิพจน์สร้างจากตัวแปรตัวเดียว กฎนี้ระบุว่านิพจน์มีชนิดเดียวกับตัวแปร และ ชนิดของตัวแปรหาได้จากตารางสัญลักษณ์ด้วยฟังก์ชัน `getType`

ต้นไม้แจงส่วนประกอบความหมายของนิพจน์ `(x+y)*z` ที่สร้างจากไวยากรณ์ที่กำหนดนี้แสดงในรูปข้างล่างเมื่อ `x` และ `z` เป็นตัวแปรชนิด `integer` ส่วน `y` เป็นตัวแปรชนิด `real` ค่าของลักษณะประจำ `type` ในต้นไม้นี้ส่งจากล่างขึ้นบน กฎของความหมายในไวยากรณ์นี้กฎของความหมายที่เกี่ยวข้องกับ `id` ใช้ฟังก์ชัน `getType(id)` ดึงชนิดข้อมูลของตัวแปร `id` เพื่อส่งให้โหนดแม่ ดังนั้นนิพจน์ `x+y` ที่แทนด้วยโหนด `E` ในต้นไม้มี `type` เป็น `real` และนิพจน์ `(x+y)*z` ที่แทนด้วยโหนด `F` ในต้นไม้มี `type` เป็น `real` ด้วย



ตัวอย่างต่อไปนี้แสดงไวยากรณ์ลักษณะประจำที่ส่งค่าของลักษณะประจำจากซ้ายไปขวา ไวยากรณ์ลักษณะประจำนี้ใช้ลักษณะประจำ `type` แทนชนิดข้อมูลของตัวแปรที่ได้จากประโยคประกาศตัวแปร กฎของความหมายในแถวแรกระบุว่าค่าของลักษณะประจำ `type` ของรายการของตัวแปร `varlist` ได้มาจากค่าของ `type` ของ `typeD` ซึ่งเป็นคำสงวน `int` หรือ `float` ดังนั้นค่าของลักษณะประจำ `type` ส่งจากซ้ายไปขวากฎของความหมายในแถวที่ 4 และ 5 ส่งค่าของลักษณะประจำ `type` จากบนลงล่าง

	Grammar rule	Semantic rule
1	`dec → typeD varlist`	`varlist.type = typeD.type`
2	`typeD →` **`int`**	`typeD.type = int`
3	`typeD →` **`float`**	`typeD.type = real`
4	`varlist →` **`id`**	`id.type = varlist.type` `addType(id, id.type)`

5	$varlist_1$ `→` **`id`** `,` $varlist_2$	`id.type =` $varlist_1$`.type`
		`addType(id, id.type)`
		$varlist_2$`.type =` $varlist_1$`.type`

การส่งค่าของลักษณะประจำจากขวาไปซ้ายคล้ายกับการส่งค่าจากซ้ายไปขวา การส่งค่าแบบนี้ใช้ไม่บ่อยในภาษาโปรแกรมเนื่องจากทำให้ไม่สามารถหาค่าของลักษณะประจำพร้อมกับการสร้างต้นไม้แจงส่วน

4.2.3 การอธิบายชนิดข้อมูลในโปรแกรมด้วยไวยากรณ์ของลักษณะประจำ

การสร้างตัววิเคราะห์ความหมายเพื่อตรวจสอบชนิดข้อมูลในโปรแกรมทำได้โดยเขียนกฎของความหมายที่ระบุว่าโครงสร้างย่อยในโปรแกรมเช่นตัวแปรนิพจน์คำสั่งให้ค่าเป็นข้อมูลชนิดใด และกำหนดว่าโครงสร้างแต่ละชนิดต้องใช้ข้อมูลชนิดใด หัวข้อนี้แสดงตัวอย่างกฎของความหมายที่ระบุชนิดข้อมูลของตัวแปรและค่าที่ได้จากนิพจน์รวมทั้งตรวจสอบชนิดข้อมูลที่ใช้ในโครงสร้างย่อย

	Grammar rule	Semantic rule
1	dec → typeD varlist	varlist.type = typeD.type
2	typeD → **int**	typeD.type = int
3	typeD → **float**	typeD.type = real
4	varlist → **id**	id.type = varlist.type addType(id, id.type)
5	$varlist_1$ → **id** , $varlist_2$	id.type = $varlist_1$.type addType(id, id.type) $varlist_2$.type = $varlist_1$.type
6	E_1 → E_2 **+ id**	t1= E_2.type, t2= getType(id) E_1.type = (t1==bool or t2==bool)? error: (t1==int and t2==int)? int:real
7	E → **id**	E.type = getType(id)
8	F → E_1 **<** E_2	t1= E_1.type, t2= E_2.type F.type = ((t1==int or t1==real)and (t2==int or t2==real))? bool:error
9	B → F_1 **or** F_2	t1= F_1.type, t2= F_2.type B.type = (t1==bool and t2 ==bool)?bool:error
10	B → F	B.type = F.type
11	st_1 → **if** B st_2 **else** st_3	t1=st_2.type, t2=st_3.type st_1.type = (B.type==bool and t1==t2)?t1:error
12	st → **id =** E	st.type = (E.type==getType(id))?E.type:error

กฎของความหมายในแถวที่ 1-5 กำหนดชนิดข้อมูลของตัวแปรจากการประกาศตัวแปรเช่น จากคำสั่งประกาศตัวแปร `int x,y` ตัวแปร `x` และ `y` เป็นชนิด `int` กฎของความหมายในแถวที่ 6 และ 7 กำหนดชนิดข้อมูลของนิพจน์ที่สร้างจากตัวกระทำ `+` เช่น นิพจน์ `x+y` ให้ผลลัพธ์เป็นข้อมูลชนิด `int` เมื่อ `x` และ `y` เป็นตัวแปรชนิด `int` กฎของความหมายในแถวที่ 8 กำหนดชนิดข้อมูลของนิพจน์ที่สร้างจากตัวกระทำ `<` เช่น นิพจน์ `x<y` ให้ผลลัพธ์เป็นข้อมูลชนิด `bool` ถ้า `x` และ `y` เป็นตัวแปรชนิด `int` หรือ `real` แต่ถ้า `x` หรือ `y` มีชนิด `bool` แล้ว `x<y` ให้ผลลัพธ์เป็น `error` กฎของความหมายในแถวที่ 9 และ 10 กำหนดชนิดข้อมูลของนิพจน์ทางตรรกศาสตร์ที่อาจสร้างจากตัวกระทำ `or` เช่น `a or b` โดยกำหนดว่าถ้านิพจน์ย่อยทั้งหมดเป็นชนิด `bool` นิพจน์ใหญ่จะมีชนิดเป็น `bool` ด้วย มิฉะนั้นนิพจน์ใหญ่มีชนิดเป็น error คือเกิดข้อผิดพลาด

ไวยากรณ์ลักษณะประจำหนึ่งอาจมีลักษณะประจำมากกว่าหนึ่งตัวที่ใช้ประกอบกันเพื่อกำหนดความหมายที่ต้องการ ผู้ออกแบบภาษาสามารถสร้างลักษณะประจำตามที่ต้องการใช้

4.3 โปรแกรมตัววิเคราะห์ความหมาย

การเขียนโปรแกรมตัววิเคราะห์ความหมายทำได้โดยเพิ่มการคำนวณค่าของลักษณะประจำให้กับโหนดในต้นไม้แจงส่วน เนื่องจากกฎของความหมายสร้างไว้คู่กับกฎของไวยากรณ์ค่าของลักษณะประจำของโหนดในต้นไม้แจงส่วนขึ้นกับค่าในโหนดแม่ โหนดลูก หรือ โหนดพี่น้องเท่านั้น ดังนั้นการคำนวณค่าของลักษณะประจำทำได้โดยท่อง (traverse) ต้นไม้แจงส่วนในลำดับที่เหมาะสมกับการส่งค่า เช่นการท่องต้นไม้แบบก่อนลำดับ (preorder traversal) ใช้กับการส่งค่าจากบนลงล่าง เป็นต้น อย่างไรก็ตามค่าของลักษณะประจำอาจส่งจากบนลงล่าง ส่งจากล่างขึ้นบน หรือส่งจากซ้ายไปขวาก็ได้ หัวข้อนี้อธิบายการเขียนโปรแกรมสำหรับการส่งค่าแบบต่างๆ โดยใช้ไวยากรณ์ลักษณะประจำในหัวข้อ 4.2

สำหรับโปรแกรมในหัวข้อนี้ใช้ฟังก์ชัน `nodeType` ที่บอกว่าโหนดนั้นเป็นโหนดของสัญลักษณ์ไม่ปลายตัวใด ฟังก์ชัน `leftChild` และ `rightChild` ที่ให้ลูกสายสุดและรูปขวาสุดของโหนดนั้น ฟังก์ชัน `child2` และ `child3` ที่ให้ลูกโหนดที่ 2 และ 3 ของโหนดนั้น ฟังก์ชันที่เกี่ยวกับตารางสัญลักษณ์ ได้แก่ ฟังก์ชัน `addType` ที่เก็บชนิดของตัวแปรลงในตารางสัญลักษณ์ และ ฟังก์ชัน `getType` ที่อ่านชนิดของตัวแปรจากตารางสัญลักษณ์

4.3.1 โปรแกรมสำหรับลักษณะประจำที่ส่งค่าจากบนลงล่าง

การเขียนโปรแกรมสำหรับส่งค่าของสัญลักษณ์ประจำจากบนลงล่างใช้การท่องต้นไม้แบบก่อนลำดับ (preorder traversal) ซึ่งจะหาค่าของลักษณะประจำที่โหนดนั้นก่อนแล้วจึงลงไปทำงานกับโหนดลูกดังแสดงข้างล่าง

```
procedure Eval (T:node)
{     compute all attributes of T
      for each child C of T
            Eval(C);
}
```

พิจารณาไวยากรณ์ลักษณะประจำที่ส่งค่าจากบนลงล่างจากหัวข้อ4. 2.2 ที่แสดงข้างล่างนี้

	Grammar rule	Semantic rule
1	dec → **int** varlist	varlist.type = integer
2	dec → **float** varlist	varlist.type = real
3	varlist → **id**	id.type = varlist.type addType(id, id.type)
4	$varlist_1$ → **id** , $varlist_2$	id.type = $varlist_1$.type addType(id, id.type) $varlist_2$.type = $varlist_1$.type

โปรแกรมข้างล่างนี้เป็นฟังก์ชัน `Eval` ที่ทำงานแบบเรียกซ้ำ (recursive call) ฟังก์ชันนี้คำนวณค่าของลักษณะประจำ `type` ในโหนดนั้นก่อนที่จะเรียกฟังก์ชัน `Eval` ซ้ำสำหรับโหนดลูกของมัน การทำงานแต่ละขั้นในโปรแกรมแปลงมาจากไวยากรณ์ลักษณะประจำดังแสดงด้วยคำอธิบาย (comment) ในแต่ละบรรทัดของโปรแกรม

```
procedure Eval (T:node)
{  switch nodeType(T):
   {  case dec:                        // dec -> int varlist | float varlist
         NodeVarlist=rightChild(T)
         if (nodeType(leftChild(T))==int)       // dec -> int varlist
            varlist.type=integer                // varlist.type = integer
         if (nodeType(leftChild(T))==float)     // dec -> float varlist
            varlist.type=real                   // varlist.type = real
         Eval(NodeVarlist)           // sending the attribute down
         break
      case varlist:                  // varlist1 -> id | id , varlist2
         Node_id=leftChild(T)
         NodeVarlist=rightChild(T)   // id.type = varlist1.type
         addType(Node_id, T.type)    // addType(id, id.type)
         if (NodeVarlist!=NULL)      // varlist1 -> id , varlist2
         {  NodeVarlist.type=T.type // varlist2.type = varlist1.type
            Eval(NodeVarlist)        // sending the attribute down
         }
         break
   }
}
```

4.3.2 โปรแกรมสำหรับลักษณะประจำที่ส่งค่าจากล่างขึ้นบน

การเขียนโปรแกรมสำหรับส่งค่าของสัญลักษณ์ประจำจากล่างขึ้นบนใช้การท่องต้นไม้แบบหลังลำดับ (postorder traversal) ซึ่งจะเรียกฟังก์ชัน Eval ซ้ำ เพื่อหาค่าของลักษณะประจำที่โหนดลูกก่อนแล้วจึงคำนวณค่าของลักษณะประจำในโหนดนั้นดังแสดงข้างล่าง

```
procedure Eval (T:node)
{      for each child C of T
             Eval(C);
       compute all attributes of T
}
```

พิจารณาไวยากรณ์ลักษณะประจำที่ส่งค่าจากล่างขึ้นบนจากหัวข้อ 4.2.2 ที่แสดงข้างล่างนี้

	Grammar rule	Semantic rule
1	$E_1 \rightarrow E_2$ **+** F	E_1.type = (E_2.type==F.type)?F.type:real
2	E → F	E.type = F.type
3	$F_1 \rightarrow F_2$ *** id**	F_1.type = (F_2.type==getType(id))?F_2.type:real
4	F → **id**	F.type = getType(id)
5	F → **(** E **)**	F.type=E.type

โปรแกรมข้างล่างนี้เป็นฟังก์ชัน `Eval` ที่ทำงานแบบเรียกซ้ำ (recursive call) ฟังก์ชันนี้เรียกตัวเองซ้ำเพื่อหาคำนวณค่าของลักษณะประจำ `type` ในโหนดลูกก่อนที่จะคำนวณค่าในโหนดนั้น การทำงานแต่ละขั้นในโปรแกรมแปลงมาจากไวยากรณ์ลักษณะประจำดังแสดงด้วยคำอธิบายในแต่ละบรรทัดของโปรแกรม

```
procedure Eval (T:node)
{  switch nodeType(T):
   {  case E:                                  // E1 -> F | E2 + F
         LNode=leftChild(T)
         RNode=rightChild(T)
         Eval(LNode)                           // sending the attribute up
         if (RNode!=NULL)                      // E1 -> E2 + F
         {  Eval(RNode)                        // sending the attribute up
            if (LNode.type==RNode.type)        // (E2.type==F.type)?
               T.type=LNode.type               // E1.type=F.type
            else
               T.type=real                     // E1.type=real
         }
         else                                  // E1 -> F
            T.type=LNode.type
         break
      case F:                                  //F1 -> id | F2 * id | ( E )
         LNode=leftChild(T)
         MNode=Child2(T)
         RNode=rightChild(T)
         if RNode==NULL                        //  F1 -> id
            T.type=getType(LNode)              // F.type = getType(id)
         else if (nodeType(MNode)==*)          //F1 -> F2 * id
         {  Eval(Lnode)                        // sending the attribute up
            if (LNode.type==getType(RNode))// F2.type==getType(id))?
               T.type=LNode.type               // F1.type = F2.type
            else
               T.type=real                     // F1.type = real
         }
         else                                  //  F1 -> ( E )
         {  Eval(MNode)                        // sending the attribute up
            T.type=MNode.type                  // F.type=E.type
         }
         break
   }
}
```

4.3.3 โปรแกรมสำหรับลักษณะประจำที่ส่งค่าจากซ้ายไปขวา

การเขียนโปรแกรมสำหรับส่งค่าของสัญลักษณ์ประจำจากซ้ายไปขวาต้องทำงานกับโหนดลูกเรียงลำดับจากซ้ายไปขวาด้วย พิจารณาไวยากรณ์ลักษณะประจำที่ส่งค่าจากล่างขึ้นบนจากหัวข้อ 4.2.2 ที่แสดงข้างล่างนี้

	Grammar rule	Semantic rule
1	dec → typeD varlist	varlist.type = typeD.type
2	typeD → **int**	typeD.type = int
3	typeD → **float**	typeD.type = real
4	varlist → **id**	id.type = varlist.type addType(id) id.type)
5	$varlist_1$ → **id** , $varlist_2$	id.type = $varlist_1$.type addType(id) id.type) $varlist_2$.type = $varlist_1$.type

โปรแกรมข้างล่างนี้เป็นฟังก์ชัน `Eval` ที่ทำงานแบบเรียกซ้ำ (recursive call) ฟังก์ชันนี้เรียกตัวเองซ้ำเพื่อหาคำนวณค่าของลักษณะประจำ `type` โดยทำงานกับโหนดลูกเรียงจากซ้ายไปขวา การทำงานแต่ละขั้นในโปรแกรมแปลงมาจากไวยากรณ์ลักษณะประจำดังแสดงด้วยคำอธิบายในแต่ละบรรทัดของโปรแกรม

```
procedure Eval (T:node)
{  switch nodeType(T):
   {  case dec:                              // dec -> typeD varlist
         NodeTypeD=leftChild(T)
         NodeVarlist=rightChild(T)
         Eval(NodeTypeD)                     // send the attribute up
         NodeVarlist.type=NodeTypeD.type     // varlist.type=typeD.type
                                             // send the attribute L to R
         Eval(NodeVarlist)                   // send the attribute down
         break
      case typeD:                            // typeD -> int | float
         child=leftchild(T)
         if (nodeType(child)==int)    T.type=integer   // typeD.type=int
         if (nodeType(child)==float)  T.type=real      // typeD.type=real
         break
      case varlist:                          // varlist1 -> id | id , varlist2
         Node_id=leftChild(T)
         NodeVarlist=rightChild(T)
         Node_id.type=T.type                 // id.type = varlist.type
         addType(Node_id,Node_id.type)       //addType(id,id.type)
         if (NodeVarlist!=NULL)              // varlist1 -> id , varlist2
         {  NodeVarlist.type= T.type         // varlist2.type = varlist1.type
            Eval(NodeVarlist)                // send the attribute down
         }
         break
   }
}
```

4.3.4 โปรแกรมสำหรับลักษณะประจำที่ส่งค่าแบบผสม

ไวยากรณ์ลักษณะประจำที่ใช้ในภาษาโปรแกรมทั่วไปมีการส่งค่าทั้งแบบบนลงล่าง แบบล่างขึ้นบน และแบบซ้ายไปขวาอยู่ด้วยกันดังที่แสดงในหัวข้อ 4.2.3 ในหัวข้อนี้ เราจะแสดงโปรแกรมสำหรับไวยากรณ์ลักษณะประจำที่ส่งค่าหลายแบบโดยใช้ไวยากรณ์ลักษณะประจำในหัวข้อ 4.2.3 ที่แสดงข้างล่างนี้

	Grammar rule	Semantic rule
1	dec → typeD varlist	varlist.type = typeD.type
2	typeD → **int**	typeD.type = int
3	typeD → **float**	typeD.type = real
4	varlist → **id**	id.type = varlist.type addType(id, id.type)
5	$varlist_1$ → **id** , $varlist_2$	id.type = $varlist_1$.type addType(id, id.type) $varlist_2$.type = $varlist_1$.type
6	E_1 → E_2 **+ id**	t1= E_2.type, t2= getType(id) E_1.type = (t1==bool or t2==bool)? error: (t1==int and t2==int)? int:real
7	E → **id**	E.type = getType(id)

8	$F \rightarrow E_1$ **<** E_2	t1= E_1.type, t2= E_2.type F.type = ((t1==int or t1==real)and (t2==int or t2==real))? bool:error
9	$B \rightarrow F_1$ **or** F_2	t1= F_1.type, t2= F_2.type B.type = (t1==bool and t2 ==bool)?bool:error
10	$B \rightarrow F$	B.type = F.type
11	$st_1 \rightarrow$ **if** B st_2 **else** st_3	t1=st_2.type, t2=st_3.type st_1.type = (B.type==bool and t1==t2)?t1:error
12	st → **id =** E	st.type = (E.type==getType(id))?E.type:error

โปรแกรมข้างล่างนี้เป็นฟังก์ชัน Eval ที่ทำงานแบบเรียกซ้ำ (recursive call) ฟังก์ชันนี้เรียกตัวเองซ้ำเพื่อหาคำนวณค่าของลักษณะประจำ type ตามไวยากรณ์ลักษณะประจำข้างบนนี้ การทำงานแต่ละขั้นในโปรแกรมแปลงมาจากไวยากรณ์ลักษณะประจำดังแสดงด้วยคำอธิบายในแต่ละบรรทัดของโปรแกรม

```
procedure Eval (T:node)
{  switch nodeType(T):
   {  case dec:                              // dec -> typeD varlist
         NodeTypeD=leftChild(T)
         NodeVarlist=rightChild(T)
         Eval(NodeTypeD)                     // send the attribute up
         NodeVarlist.type=NodeTypeD.type     //varlist.type=typeD.type
                                             // send the attribute L to R
         Eval(NodeVarlist)                   // send the attribute down
         break

      case typeD:                            // typeD -> int | float
         child=leftchild(T)
         if (nodeType(child)==int)    T.type=integer    // typeD.type=int
         if (nodeType(child)==float)  T.type=real       // typeD.type=real
         break

      case varlist:                          // varlist1 -> id | id , varlist2
         Node_id=leftChild(T)
         NodeVarlist=rightChild(T)
         Node_id.type=T.type                 // id.type = varlist.type
         addType(Node_id,Node_id.type)       //addType(id, id.type)
         if (NodeVarlist!=NULL)              // varlist1 -> id , varlist2
         {  NodeVarlist.type= T.type         // varlist2.type = varlist1.type
            Eval(NodeVarlist)                // send the attribute down
         }
         break

      case E:                                // E1 -> id | E2 + id
         LNode=leftChild(T)
         RNode=rightChild(T)
         if RNode!=NULL                      // E1 -> E2 + id
         {  Eval(LNode)                      // sending the attribute up
            t1=LNode.type                    // t1=E2.type
            t2=getType(RNode)                // t2=getType(id)
            if (t1==bool or t2==bool)        // (t1==bool or t2==bool)?
               T.type=error                  // E1.type = error
            else if (t1==int and t2==int)    // (t1==int and t2==int)?
```

```
            T.type=int                  // E1.type = int
         else T.type=real               // E1.type = real
      }
      else                              // E1 -> id
      {  Eval(LNode)                    // sending the attribute up
         T.type=LNode.type              // E1.type = getType(id)
      }
      break
   case F:                              //F -> E1 < E2
      LNode=leftChild(T)   RNode=rightChild(T)
      Eval(LNode)                       // sending the attribute up
      Eval(Rnode)                       // sending the attribute up
      t1=LNode.type                     // t1=E1.type
      t2=RNode.type                     // t2=E2.type
      if ((t1==int or t1==real) and     // ((t1==int or t1==real)and
          (t2==int or t2==real))        //  (t2==int or t2==real))?
         T.type=bool                    // F.type=bool
      else
         T.type=real                    // F.type=error
      break

   case B:                              //B -> F | F1 or F2
      LNode=leftChild(T)
      RNode=rightChild(T)
      if RNode!=NULL                    //B -> F1 or F2
      {  Eval(LNode)                    // sending the attribute up
         Eval(Rnode)                    // sending the attribute up
         t1=LNode.type                  // t1=F1.type
         t2=RNode.type                  // t2=F2.type
         if (t1==bool and t2==bool)     // (t1==bool and t2 ==bool)?
            T.type=bool                 // B.type = bool
         else
            T.type=error                // B.type = error
      }
      else                              //B -> F
      {  Eval(LNode)                    // sending the attribute up
         T.type=LNode.type              // B.type = F.type
      }
      break

   case st:                             //st1 -> if B st2 else st3|id = E
      LNode=leftChild(T)
      if (nodetype(LNode)==if)          // st1 -> if B st2 else st3
      {  NodeCond=child2(T)
         NodeThen=child3(T)
         NodeElse=right(T)
         Eval(NodeCond)                 // sending attribute up
         Eval(NodeThen)                 // sending attribute up
         Eval(NodeElse)                 // sending attribute up
         t1=NodeThen.type               // t1=st2.type
         t2=NodeElse.type               // t2=st3.type
         if (NodeCond.type==bool and t1==t2)//(B.type==bool and t1==t2)?
            T.type=t1                   // st1.type = t1
         else
            T.type=error                // st1.type = error
      }
      else if (nodetype(LNode)==id)     // st1 -> id = E
```

```
        {  RNode=rightChild(T)
           Eval(RNode)                    // sending attribute up
           if (RNode.type==gettype(LNode)) // E.type==getType(id))?
              T.type=RNode.type           // st.type = E.type
           else
              T.type=error                // st.type = error
   }
}
```

5. ตัวก่อกำเนิดรหัส

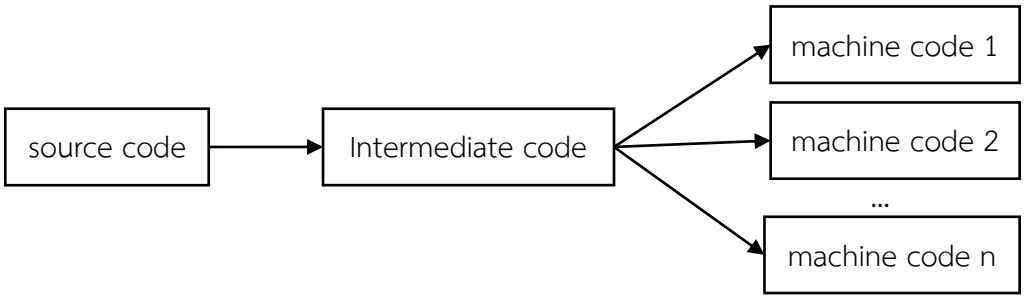
ตัวก่อกำเนิดรหัสเป็นขั้นตอนหนึ่งในการแปลภาษา ขั้นตอนนี้รับโครงสร้างที่ได้จากรหัสต้นฉบับ (source code) เช่นต้นไม้การแจงส่วน แล้วแปลงเป็นรหัสที่ต้องการ ในการทำงานของตัวแปลภาษารหัสที่ต้องการอาจเป็นรหัสเครื่อง (machine code) หรือรหัสกลาง (intermediate code)

ในการสร้างตัวก่อกำเนิดรหัส เราใช้ไวยากรณ์ลักษณะประจำอธิบายวิธีการสร้างรหัสโดยมีลักษณะประจำแทนรหัสที่สร้างและกฎของความหมายกำหนดวิธีสร้างรหัส หลังจากนั้นจึงสร้างโปรแกรมให้ทำงานตามที่กำหนดในไวยากรณ์ลักษณะประจำดังที่อธิบายไปในบทที่ 4 ดังนั้นผู้ที่สร้างตัวก่อกำเนิดรหัสต้องระบุว่าโครงสร้างต่าง ๆ ในภาษาระดับสูงมีการทำงานตามรหัสเครื่องหรือรหัสกลางอย่างไร ในที่นี้เราจะอธิบายโดยใช้รหัสกลางเป็นตัวอย่าง

หัวข้อ 5.1 แสดงตัวอย่างรหัสกลางสองแบบคือ P-code ที่ใช้กับเครื่องจำลองที่ทำงานกับสแตคและ 3-address code ที่มีรูปแบบคำสั่งอย่างง่าย เช่น `x = y + z` ซึ่งหมายความว่าให้เอาค่าในตัวแปร `y` และ `z` ไปกระทำด้วยตัวกระทำ `+` แล้วนำผลลัพธ์ที่ได้ไปเก็บไว้ในตัวแปร `x` หัวข้อ 5.2 แสดงการใช้ไวยากรณ์ลักษณะประจำกำหนดการทำงานของตัวก่อกำเนิดรหัสสำหรับโครงสร้างลำดับการทำงานในภาษาโปรแกรม หัวข้อ 5.3 อธิบายชนิดของข้อมูลที่นิยมใช้ในภาษาโปรแกรมและการสร้างรหัสกลางเพื่อเข้าถึงข้อมูลชนิดต่าง ๆ นี้

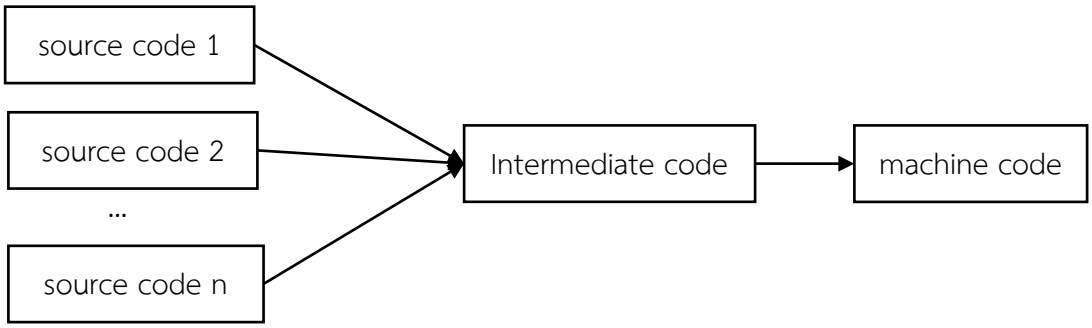
5.1 รหัสกลาง

ในการสร้างตัวแปลภาษา เราอาจแปลรหัสเริ่มต้นเป็นภาษากลางก่อนแล้วจึงแปลเป็นรหัสเครื่องอีกครั้ง วิธีนี้ทำให้สามารถแปลรหัสเริ่มต้นหนึ่งไปเป็นรหัสเครื่องหลายภาษาได้โดยแปลรหัสเริ่มต้นเป็นรหัสกลางแล้วแปลรหัสกลางเป็นรหัสเครื่องสำหรับหลายเครื่องดังแสดงในรูปข้างล่าง จาวาไบท์โคด (Java byte code) เป็นรหัสกลางแบบหนึ่งที่ทำให้โปรแกรมภาษาจาวาสามารถใช้งานในเครื่องใด ๆ ได้ จาวาไบท์โคดเป็นภาษาที่ใช้สำหรับเครื่องเสมือนจาวา (Java virtual machine) ที่เรียกย่อว่า JVM โปรแกรมภาษา Java จะถูกแปลเป็นจาวาไบท์โคด ซึ่งทำงานบน JVM JVM ถูกสร้างตามภาษาเครื่องของคอมพิวเตอร์ที่ใช้ทำงาน (เช่น JVM สำหรับหน่วยประมวลผล ARM, JVM สำหรับหน่วยประมวลผล Xeon) เพื่อจำลองการทำงานของโปรแกรมบนเครื่องนั้น



ในทำนองเดียวกันเราสามารถใช้รหัสกลางอำนวยความสะดวกในการแปลรหัสเริ่มต้นจากภาษาระดับสูงหลายภาษาไปเป็นภาษาเครื่องภาษาหนึ่งได้ โดยแปลรหัสเริ่มต้นแต่ละภาษาเป็นรหัสกลาง แล้วแปลรหัสกลางนั้นเป็นภาษาเครื่องที่ต้องการดังแสดงในรูปข้างล่าง ภาษาระดับสูงหลายภาษา เช่น JScala, Jython

มีตัวประมวลภาษาที่แปลจากภาษาเหล่านั้นเป็นจาวาไบท์โคดเพื่อทำงานบน JVM



5.1.1 P-code

P-code เป็นภาษาที่ใช้กับเครื่องเสมือน (virtual machine) ที่ทำงานกับสแตคเป็นหลัก เครื่องเสมือนนี้ใช้อ้างถึงตัวแปรได้เหมือนภาษาระดับสูง แต่เมื่อต้องการทำงาน (เช่น การกระทำทางคณิตศาสตร์ การเปรียบเทียบ) กับตัวแปรจะต้องนำค่าในตัวแปรมาเก็บลงสแตคก่อนเนื่องจากคำสั่งของ P-code ใช้ค่าบนสุดของสแตคและเก็บผลลัพธ์ที่ได้ในสแตคด้วย คำสั่งของ P-code ที่จะกล่าวถึงในส่วนนี้แบ่งเป็นคำสั่งที่นำค่าเข้าหรือออกจากสแตค การกระทำทางคณิตศาสตร์ เปรียบเทียบและตรรกศาสตร์ คำสั่งกระโดดให้ไปทำงานที่คำสั่งอื่น และคำสั่งเกี่ยวกับเลขที่อยู่ (address)

คำสั่งนำค่าเข้าหรือออกจากสแตค

`loadc k` นำค่าคงที่ `k` ใส่ไว้บนแสตค

`loadv x` นำค่าของตัวแปร `x` ใส่ไว้บนแสตค

`loada x` นำตำแหน่ง (address) ของตัวแปร `x` ใส่บนแสตค

`store` อ่านค่าบนสุด (*top) และค่าถัดลงไป (*(top-1)) ในสแตค แล้วนำค่าบนสุดของสแตคไปเก็บที่หน่วยความจำที่ตำแหน่ง (address) ที่ระบุด้วยค่าถัดลงไปในสแตค

`nstore` อ่านค่าบนสุด (*top) และค่าถัดลงไป (*(top-1)) ในสแตค แล้วนำค่าบนสุดของสแตคไปเก็บที่หน่วยความจำที่ตำแหน่ง (address) ที่ระบุด้วยค่าถัดลงไปในสแตค แล้วนำค่าบนสุดของสแตคที่อ่านมากลับไปเก็บในสแตค

การกระทำทางคณิตศาสตร์ เปรียบเทียบและตรรกศาสตร์

`add, sub, lessthan, grtthan, equal, and, or`
นำค่าบนสุดสองค่า ออกจากสแตคมาทำงานตามแต่ตัวกระทำ (เช่น บวก ลบ เทียบว่าน้อยกว่า มากกว่า) แล้วนำผลลัพธ์ที่ได้เก็บบนแสตค

`not` นำค่าบนสุดออกจากสแตคมาทำนิเสธแล้วเก็บผลลัพธ์ที่ได้เก็บบนแสตค

คำสั่งกระโดดไปทำงานที่อื่น

`label A` กำหนดตำแหน่งนั้นในโปรแกรมให้ชื่อ `A` เพื่อให้ใช้อ้างถึงในการกระโดดมาทำงาน

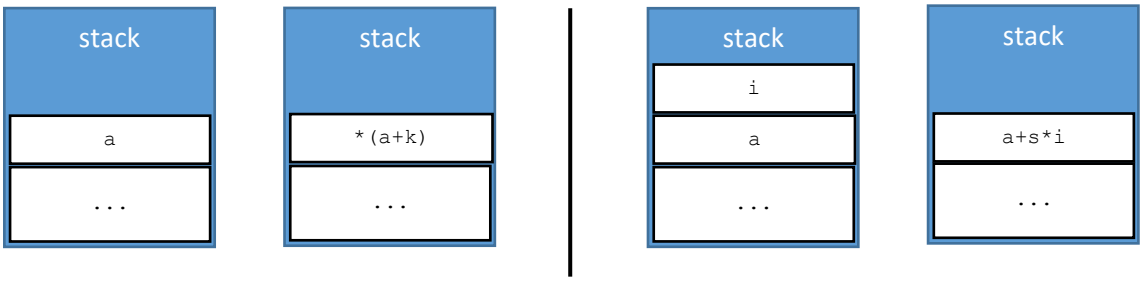
`jump A` กระโดดไปทำงานที่คำสั่งหลัง `label A`

`jumpF A` ดึงค่าบนสุดจากสแตคออกมาตรวจสอบ ถ้าค่านั้นเป็นจริงให้กระโดดไปทำงานที่คำสั่งหลัง `label A` มิฉะนั้นจะทำงานคำสั่ง ต่อไป

คำสั่งเกี่ยวกับเลขที่อยู่

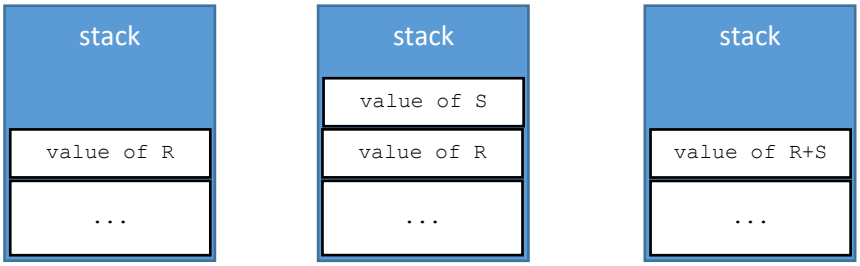
`ind k` นำค่าบนสุดของสแตค คือ `a` ออกมาและดึงข้อมูลจากหน่วยความจำที่เลขที่อยู่ `a+k` มาเก็บในสแตค แทนดังแสดงในรูปข้างล่างทางซ้าย

`ixa s` นำค่าบนสุดของ stack คือ `i` และค่าถัดลงไปคือ `a` ออกจากสแตค และคำนวณค่า `a+s*i` เก็บในสแตคดังแสดงในรูปข้างล่างทางขวา



การคำนวณนิพจน์ด้วย P-code

การหาค่าของนิพจน์ด้วย P-code เป็นการทำงานแบบ postfix นั่นคือถ้ามีนิพจน์ย่อย `R` กับ `S` แล้วต้องการหาค่าของ `R+S` ก็ต้องเขียน P-code สำหรับคำนวณค่าของ `R` ซึ่งทำให้ได้ค่าของนิพจน์ `R` อยู่บนสุดของสแตคดังแสดงในรูปด้านล่าง รูปแรก แล้วจึงต่อด้วย P-code ที่คำนวณค่าของ `S` ซึ่งทำให้ได้ค่าของนิพจน์ `S` บนแสตคดังแสดงในรูปด้านล่างรูปที่สอง จากนั้นจึงต่อด้วยคำสั่ง `add` ซึ่งทำให้ได้ผลลัพธ์ของนิพจน์ `R+S` อยู่ในสแตคดังแสดงในรูปข้างล่างรูปซ้าย



ส่วนของ P-code ต่อไปนี้แสดงตัวอย่างการคำนวณนิพจน์ในรูปแบบต่างๆ

P-code ที่คำนวณนิพจน์ `x+y`

```
loadv x
loadv y
add
```

P-code ที่คำนวณนิพจน์ `x+1`

```
loadv x
loadc 1
add
```

P-code ที่คำนวณนิพจน์ `x-(y-2)`

```
loadv x
loadv y
loadc 2
sub      // get y-2
sub      // get x-(y-2)
```

P-code ที่คำนวณนิพจน์ `x+y<2-y`

```
loadv x
loadv y
add      // get x-y
loadc 2
loadv y
sub      // get 2-y
lessthan
```

P-code ที่คำนวณนิพจน์ `x<2 and y==5`

```
loadv x
loadc 2
lessthan // get x<2
loadv y
loadc 5
equal    // get y==5
and
```

การกำหนดค่าตัวแปรด้วย P-code

การกำหนดค่าตัวแปรใน P-code ทำได้ด้วยคำสั่ง `store` ซึ่งนำค่าบนสุดของสแตคมาเก็บในตัวแปรที่กำหนด ดังนั้นจึงต้องคำนวณหาค่า ตามที่ระบุทางด้านขวาของเครื่องหมาย = ในคำสั่งกำหนดค่า (assignment statement) แล้วใส่ไว้บนแสตค จากนั้นจึงใช้คำสั่ง `store` เพื่อเก็บค่าลงในตัวแปร ค่าที่นำมาเก็บอาจเป็นค่าคงที่ (เช่น `x=0`) ค่าของตัวแปรอื่น (เช่น `x=y`) หรือค่าของนิพจน์ (เช่น `x=x+y`) ตัวอย่างต่อไปนี้แสดงการกำหนดค่าตัวแปรด้วยค่าคงที่ ค่าของตัวแปร และค่าของนิพจน์

ส่วนของ P-code ต่อไปนี้แสดงตัวอย่างการกำหนดค่าตัวแปรในรูปแบบต่างๆ

P-code สำหรับ `x=0`

```
loada x
loadc 0
store
```

P-code สำหรับ `x=y`

```
loada x
loadv y
store
```

P-code สำหรับ `x=x+y`

```
loada x
loadv x
loadv y
add
store
```

การทำงานแบบทางเลือกด้วย P-code

การทำงานแบบทางเลือกในโปรแกรมโดยทั่วไปมีสองแบบ คือ if-then และ if-then-else การทำงานทั้งสองแบบเริ่มต้นด้วยการตรวจสอบเงื่อนไข ดังนั้นต้องมี P-code ที่คำนวณค่าของเงื่อนไขก่อน และใช้คำสั่ง `jumpF` เพื่อเลือกส่วนของโปรแกรมที่จะทำงานถัดไป ในกรณีของ if-then หากเงื่อนไขเป็นจริงจะไปทำงานในส่วนของ then หากเงื่อนไขไม่เป็นจริงจะข้ามส่วนของ then และไปทำงานในคำสั่งถัดไปเลย การทำงานใน

ส่วนนี้จะใช้คำสั่ง `jumpF Q` ซึ่งตรวจสอบว่าหากค่าบนสุดของสแตคเป็นเท็จจะกระโดดไปทำงานที่ตำแหน่ง `Q` ดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
if (cond)
{  then-part
}
```

โปรแกรม P-code

```
<code for cond>
jumpF Q
<code for then-part>
label Q
```

ส่วนของ P-code ต่อไปนี้แสดงตัวอย่างการทำงานแบบ if-then

โปรแกรมในภาษาระดับสูง

```
if (x==0)
{  y=0
   x=x+2
}
```

โปรแกรม P-code

```
loadv x
loadc 0
equal        // x==0
jumpF OUT
loada y
loadc 0
store        // y=0
loada x
loadv x
loadc 2
add          // get x+2
store // x=x+2
label OUT
```

ในกรณีของ if-then-else หากเงื่อนไขเป็นจริงจะไปทำงานในส่วนของ then แล้วจึงไปทำงานตามคำสั่งถัดไป ในส่วนนี้จะคล้ายกับการทำงานแบบ if-then แต่ต้องใส่คำสั่ง `jump` เพื่อให้กระโดดข้ามโปรแกรมส่วนของ else ไปทำงานที่คำสั่งถัดจากโครงสร้าง if-then-else หากเงื่อนไขไม่เป็นจริงจะข้ามส่วนของ then ไปทำงานในส่วนของ else แล้วจึงไปทำงานตามคำสั่งถัดไปดังแสดงดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
if (cond)
{  then-part
}
else
{  else-part
}
```

โปรแกรม P-code

```
<code for cond>
jumpF ELSE
<code for then-part>
jump OUT
label ELSE
<code for else-part>
label OUT
```

ส่วนของ P-code ต่อไปนี้แสดงตัวอย่างการทำงานแบบ if-then-else

โปรแกรมในภาษาระดับสูง

```
if (x==0)
{  y=0
   x=x+2
}
else
{   y=1
    x=x-1
}
```

โปรแกรม P-code

```
loadv x
loadc 0
equal        // x==0
jumpF ELSE
loada y
loadc 0
store        // y=0
loada x
loadv x
loadc 2
add          // get x+2
store        // x=x+2
jump OUT
label ELSE
loada x
loadc 1
store y      // y=1
loadv x
loadc 1
sub          // get x-1
store        // x=x-1
label OUT
```

การทำงานวนซ้ำแบบ while

การทำงานวนซ้ำแบบ while เริ่มต้นด้วยการตรวจสอบเงื่อนไข ดังนั้นต้องมี P-code ที่คำนวณค่าของเงื่อนไขก่อน และใช้คำสั่ง `jumpF` เพื่อกระโดดออกจากโปรแกรมส่วนทำซ้ำ ส่วนของโปรแกรมที่ต้องทำซ้ำตามด้วยคำสั่ง `jump` เพื่อกระโดดกลับไปตรวจสอบเงื่อนไขดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
while (cond)
{  body
}
```

โปรแกรม P-code

```
label START
<code for cond>
jumpF OUT
<code for body>
jump START
label OUT
```

ส่วนของ P-code ต่อไปนี้แสดงตัวอย่างการทำงานวนซ้ำแบบ while

โปรแกรมในภาษาระดับสูง

```
while (x>0)
{   n=n+x
    x=x-1
}
```

โปรแกรม P-code

```
label WHSTART
loadv x
loadc 0
grtthan      // x>0
jumpF OUT
loada n
loadv n
loadv x
add
store        // n=n+x
loada x
loadv x
loadc 1
sub          // get x-1
store        // x=x-1
jump WHSTART
label OUT
```

การทำงานวนซ้ำแบบ for

การทำงานวนซ้ำแบบ for เริ่มต้นด้วยการทำงานเริ่มต้น `init` แล้วตรวจสอบเงื่อนไข `cond` ส่วนการวนซ้ำจะคล้ายกับการวนซ้ำแบบ while แต่เพิ่มการทำงาน `incr` หลังส่วนทำซ้ำซึ่งตามด้วยคำสั่ง `jump` เพื่อกระโดดกลับไปตรวจสอบเงื่อนไขดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
for (init; cond ; incr)
{   body
}
```

โปรแกรม P-code

```
<code for init>
label START
<code for cond>
jumpF OUT
<code for body>
<code for incr>
jump START
label OUT
```

ส่วนของ P-code ต่อไปนี้แสดงตัวอย่างการทำงานวนซ้ำแบบ for

โปรแกรมในภาษาระดับสูง

```
for (i=0; i<10; i=i+1)
{   n=n+i
}
```

โปรแกรม P-code

```
loada i
loadc 0
store        // i=0
label WHSTART
loadv i
loadc 10
lessthan     // i<10
jumpF OUT
loada n
loadv n
loadv i
add
store        // n=n+i
loada i
loadv i
loadc 1
add
store        // i=i+1
jump WHSTART
label OUT
```

5.1.2 3-address Code

3-address code เป็นภาษาที่มีคำสั่งระดับพื้นฐานที่ไม่ซับซ้อน แต่ละคำสั่งใน 3-address code มีค่าที่เกี่ยวข้องกับตัวแปรได้ไม่เกิน 3 ค่า 3-address code ประกอบด้วยคำสั่งคำนวณและคำสั่งกระโดดไปทำงานที่อื่น

คำสั่งคำนวณ

คำสั่งคำนวณใน 3-address code อยู่ในรูปของ `x = y op z` โดย `x`, `y` และ `z` เป็นตัวแปรที่ไม่ซ้ำกัน ส่วน `op` เป็นตัวกระทำทางคณิตศาสตร์ การเปรียบเทียบ และทางตรรกศาสตร์ดังนี้

ตัวกระทำทางคณิตศาสตร์: `+`, `-`, `*`
ตัวกระทำการเปรียบเทียบ: `>`, `>=`, `<`, `<=`, `==`
ตัวกระทำทางตรรกศาสตร์: `not` (ซึ่งใช้ 2-address เท่านั้น), `and`, `or`
ตัวกระทำเกี่ยวกับตำแหน่งของตัวแปร: `*`, `&`

คำสั่งเหล่านี้นำค่าในตัวแปร `y` และ `z` มากระทำตามที่กำหนดแล้วเก็บผลลัพธ์ในตัวแปร `x` ตัวกระทำเกี่ยวกับตำแหน่งของตัวแปร `*x` หมายถึง ค่าที่เก็บในหน่วยความจำที่มีตำแหน่ง (address) เป็นค่าในตัวแปร `x`

ถ้าให้ตัวแปร `x` เก็บค่า 3200
`*x=y` เป็นคำสั่งที่เก็บค่าของตัวแปร `y` ในหน่วยความจำที่ตำแหน่ง 3200 และ
`z=*x` เป็นคำสั่งที่เก็บค่าที่อยู่ในหน่วยความจำที่ตำแหน่ง 3200 ในตัวแปร `z`

ตัวกระทำ `&x` หมายถึงตำแหน่ง (address) ของหน่วยความจำที่เก็บตัวแปร `x`

ถ้าตัวแปร `x` เก็บที่ตำแหน่ง 4600 ในหน่วยความจำ

`y=&x` เป็นคำสั่งที่นำค่าของตำแหน่งของตัวแปร x (คือ 4600) ไปเก็บในตัวแปร `y`

สังเกตว่า `&x=y` เป็นคำสั่งที่ไม่สามารถทำได้เพราะ `&x` อ้างถึงค่าตำแหน่งในหน่วยความจำ ซึ่งไม่อนุญาตให้โปรแกรมเมอร์เปลี่ยนเองได้

คำสั่งกระโดดไปทำงานที่อื่น

`label A`	กำหนดตำแหน่งนั้นในโปรแกรมให้ชื่อ `A` เพื่อใช้อ้างถึงในการกระโดดมาทำงาน
`goto A`	กระโดดไปทำงานที่คำสั่งหลัง `label A`
`ifFalse x goto A`	ตรวจสอบค่าของตัวแปร `x` ถ้ามีค่าเป็นจริงจะทำงานที่คำสั่งถัดไป หากมีค่าเป็นเท็จจะกระโดดไปทำงานที่คำสั่งหลัง `label A`

การคำนวณนิพจน์ด้วย 3-address code

การหาค่าของนิพจน์ด้วย 3-address code ทำงานโดยหาค่าของนิพจน์ย่อยเก็บไว้ในตัวแปรชั่วคราว จากนั้นจึงนำค่าในตัวแปรชั่วคราวนี้มาคำนวณต่อเพื่อหาค่าของนิพจน์ใหญ่ นั่นคือถ้ามีนิพจน์ย่อย `R` กับ `S` แล้วต้องการหาค่าของ `R+S` จะต้องเขียน 3-address code สำหรับคำนวณค่าของนิพจน์ `R` แล้วเก็บไว้ในตัวแปรชั่วคราว `t1` ทำนองเดียวกันต้องมี 3-address code สำหรับคำนวณค่าของนิพจน์ `S` เก็บไว้ในตัวแปรชั่วคราว `t2` จากนั้นจึงใช้คำสั่ง `t=t1+t2`

ส่วนของ 3-address code ต่อไปนี้แสดงตัวอย่างการคำนวณนิพจน์ในรูปแบบต่าง ๆ

3-address code ที่คำนวณนิพจน์ `x+y`

```
t=x+y
```

3-address code ที่คำนวณนิพจน์ `x-(y-2)`

```
t = y-2
tt = x-t
```

3-address code สำหรับ `x=x+1`

```
t=x+1
x=t
```

3-address code สำหรับ `z=x-(y-2)`

```
t1 = y-2
z = x-t1
```

3-address code ที่คำนวณนิพจน์ `x+y<2-y`

```
t1=x+y
t2=2-y
t3=t1<t2
```

3-address code ที่คำนวณนิพจน์ `x<2 and y==5`

```
t1 = x<2
t2 = y==5
t3 = t1 and t2
```

การทำงานแบบทางเลือกด้วย 3-address code

การทำงานแบบทางเลือกในโปรแกรมโดยทั่วไปมีสองแบบ คือ if-then และ if-then-else การทำงานทั้งสองแบบเริ่มต้นด้วยการตรวจสอบเงื่อนไข ดังนั้นต้องมี 3-address code ที่คำนวณค่าของเงื่อนไขก่อน และใช้คำสั่ง `ifFalse ... goto ...` เพื่อเลือกส่วนของโปรแกรมที่จะทำงานถัดไป ในกรณีของ if-then หากเงื่อนไขเป็นจริงจะไปทำงานในส่วนของ then หากเงื่อนไขไม่เป็นจริงจะข้ามส่วนของ then และไปทำงาน

ในคำสั่งถัดไปเลย การทำงานในส่วนนี้จะใช้คำสั่ง `ifFalse … goto Q` ซึ่งตรวจสอบว่าหากค่าของนิพจน์เป็นเท็จจะกระโดดไปทำงานที่ตำแหน่ง `Q` ดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
if (cond)
{  then-part
}
```

โปรแกรม 3-address code

```
<code for cond, the result's in c>
ifFalse c goto Q
<code for then-part>
label Q
```

ส่วนของ 3-address code ต่อไปนี้แสดงตัวอย่างการทำงานแบบ if-then

โปรแกรมในภาษาระดับสูง

```
if (x==0)
{  y=0
   x=x+2
}
```

โปรแกรม 3-address code

```
c = x==0
ifFalse c goto OUT
y = 0
t = x+2
x = t
label OUT
```

ในกรณีของ if-then-else หากเงื่อนไขเป็นจริงจะไปทำงานในส่วนของ then แล้วจึงไปทำงานตามคำสั่งถัดไป ในส่วนนี้จะคล้ายกับการทำงานแบบ if-then แต่ต้องใส่คำสั่ง `goto` เพื่อให้กระโดดข้ามโปรแกรมส่วนของ else ไปทำงานที่คำสั่งถัดจากโครงสร้าง if-then-else หากเงื่อนไขไม่เป็นจริงจะข้ามส่วนของ then ไปทำงานในส่วนของ else แล้วจึงไปทำงานตามคำสั่งถัดไปดังแสดงดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
if (cond)
{  then-part
}
else
{  else-part
}
```

โปรแกรม 3-address code

```
<code for cond,the result's in c >
ifFalse c goto ELSE
<code for then-part>
goto OUT
label ELSE
<code for else-part>
label OUT
```

ส่วนของ 3-address code ต่อไปนี้แสดงตัวอย่างการทำงานแบบ if-then-else

โปรแกรมในภาษาระดับสูง

```
if (x==0)
{  y=0
   x=x+2
}
else
{   y=1
    x=x-1
}
```

โปรแกรม 3-address code

```
c = x==0
ifFalse c goto ELSE
y = 0
t = x+2
x = t
goto OUT
label ELSE
y = 1
t = x-1
x = t
label OUT
```

การทำงานวนซ้ำแบบ while

การทำงานวนซ้ำแบบ while เริ่มต้นด้วยการตรวจสอบเงื่อนไข ดังนั้นต้องมี 3-address code ที่คำนวณค่าของเงื่อนไขก่อน และใช้คำสั่ง `ifFalse ... goto ...` เพื่อกระโดดออกจากโปรแกรมส่วนทำซ้ำ ส่วนของโปรแกรมที่ต้องทำซ้ำตามด้วยคำสั่ง `jump` เพื่อกระโดดกลับไปตรวจสอบเงื่อนไขดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
while (cond)
{  body
}
```

โปรแกรม 3-address code

```
label START
<code for cond,the result is in c>
ifFalse c goto OUT
<code for body>
goto START
label OUT
```

ส่วนของ 3-address code ต่อไปนี้แสดงตัวอย่างการทำงานวนซ้ำแบบ while

โปรแกรมในภาษาระดับสูง

```
while (x>0)
{  n=n+x
   x=x-1
}
```

โปรแกรม 3-address code

```
label WHSTART
c = x>0
ifFalse c goto OUT
t = n+x
n = t
s = x-1
x = s
jump WHSTART
label OUT
```

การทำงานวนซ้ำแบบ for

การทำงานวนซ้ำแบบ for เริ่มต้นด้วยการทำงานเริ่มต้น `init` แล้วตรวจสอบเงื่อนไข `cond` ส่วนการวนซ้ำจะคล้ายกับการวนซ้ำแบบ while แต่เพิ่มการทำงาน incr หลังส่วนทำซ้ำซึ่งตามด้วยคำสั่ง `goto` เพื่อกระโดดกลับไปตรวจสอบเงื่อนไขดังแสดงต่อไปนี้

โปรแกรมในภาษาระดับสูง

```
for (init; cond ; incr)
{  body
}
```

โปรแกรม 3-address code

```
<code for init>
label START
<code for cond,the result is in c>
ifFalse c goto OUT
<code for body>
<code for incr>
goto START
label OUT
```

ส่วนของ 3-address code ต่อไปนี้แสดงตัวอย่างการทำงานวนซ้ำแบบ for

โปรแกรมในภาษาระดับสูง

```
for (i=0; i<10; i=i+1)
{   n=n+i
}
```

โปรแกรม 3-address code

```
i = 0
label WHSTART
c = i<10
ifFalse c goto OUT
t = n+i
n = t
t = i+1
i = t
goto WHSTART
label OUT
```

5.2 ไวยากรณ์ลักษณะประจำกำหนดการลำดับการทำงาน

ไวยากรณ์ลักษณะประจำที่ใช้สร้างรหัสกลางหรือรหัสเครื่องจะใช้ลักษณะประจำเพื่อเก็บรหัสที่สร้างสำหรับโครงสร้างย่อยและนำมาสร้างรหัสสำหรับโครงสร้างใหญ่ นอกจากนั้นยังอาจมีลักษณะประจำอื่นที่ใช้ส่งค่าที่จำเป็น เช่น ตัวแปรที่เก็บค่าชั่วคราว หัวข้อนี้อธิบายการสร้างรหัสสำหรับโครงสร้างกำหนดลำดับการทำงานในโปรแกรม คือ การคำนวณนิพจน์และกำหนดค่าให้ตัวแปร การทำงานแบบทางเลือก การทำงานวนซ้ำแบบ while และ for โดยแบ่งเป็นการสร้าง p-code ในหัวข้อ 5.2.1 และการสร้าง 3-address code ในหัวข้อ 5.2.2 การสร้าง P-code ใช้ลักษณะประจำตัวเดียวเพื่อเก็บ P-code แต่การสร้าง 3-address code ต้องใช้ลักษณะประจำเพิ่มอีกตัวหนึ่งสำหรับตัวแปรที่ใช้เก็บค่าชั่วคราวสำหรับนิพจน์ย่อย แต่การสร้าง p-code ไม่จำเป็นต้องใช้ตัวแปรนี้เนื่องจากสามารถเก็บค่าชั่วคราวของนิพจน์ย่อยในสแตค

เนื่องจากรหัสกลางหรือรหัสเครื่องที่สร้างขึ้นเป็นสตริง ในหัวข้อนี้เราใช้ฟังก์ชันเชื่อมสตริง (concatenation) ซึ่งแทนด้วยสัญลักษณ์ `+` , ฟังก์ชันดึงชื่อตัวแปรจากโหนดในต้นไม้แจงส่วนคือ `getName(node)`, ฟังก์ชันสร้างตัวแปรใหม่คือ `newVar()` ซึ่งส่งชื่อตัวแปรที่สร้างขึ้นให้ใหม่โดยชื่อนี้ไม่ซ้ำกับตัวแปรอื่นในโปรแกรม และ ฟังก์ชัน `newLabel()` ที่ส่งชื่อของตำแหน่งในโปรแกรมที่สร้างขึ้นให้ใหม่โดยชื่อนี้ไม่ซ้ำกับตำแหน่งที่ใช้ไปแล้ว

5.2.1 ไวยากรณ์ลักษณะประจำที่สร้าง P-code

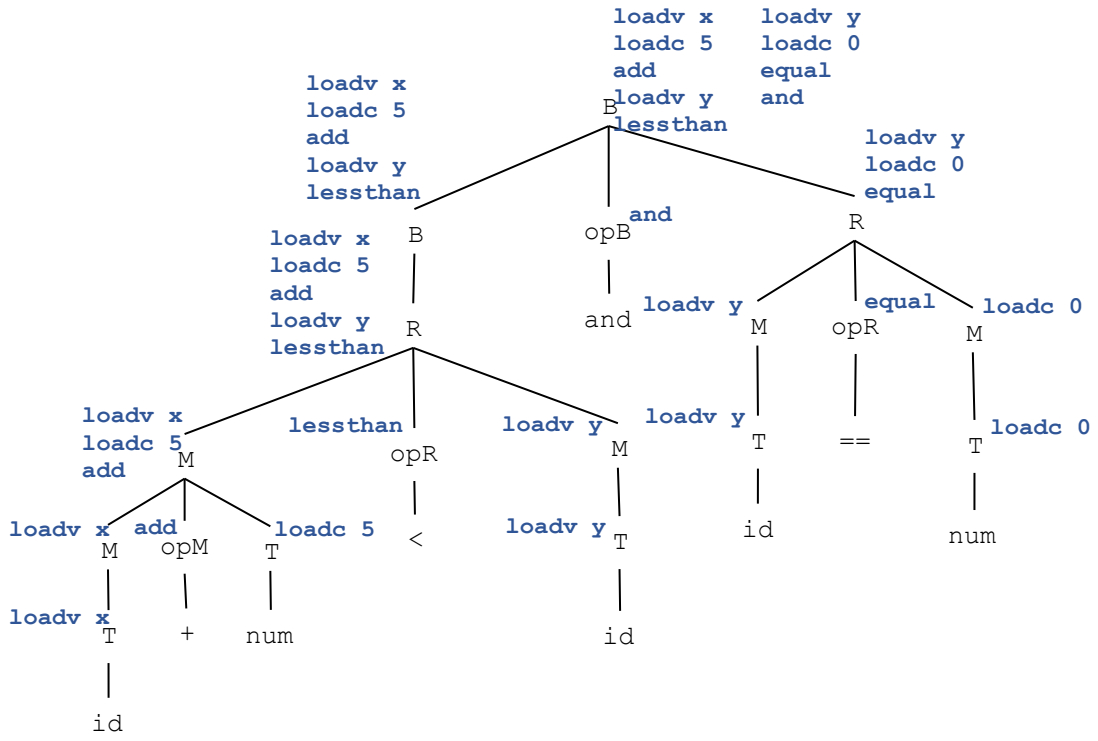
ในหัวข้อนี้เราใช้ลักษณะประจำ `code` สำหรับเก็บ p-code ที่สร้างสำหรับโครงสร้างต่าง ๆ ในภาษา เช่น `ifSt.code` เก็บ P-code สำหรับโครงสร้างที่แทนด้วยสัญลักษณ์ `ifSt` ในไวยากรณ์ของภาษา และลักษณะประจำนี้แสดงในต้นไม้แจงส่วนด้วยตัวอักษรสีฟ้า

ไวยากรณ์ลักษณะประจำที่สร้าง P-code สำหรับนิพจน์

กำหนดไวยากรณ์สำหรับนิพจน์อย่างง่ายในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ กฎของความหมายที่แสดงนี้สร้าง p-code ที่คำนวณนิพจน์โดยสร้าง p-code ของนิพจน์ย่อยแล้วนำมาประกอบกันเป็น p-code สำหรับนิพจน์รวมดังนี้

	Grammar rule	Semantic rule
1	$B_1 \rightarrow B_2$ opB R	B_1.code = B_2.code+R.code+opB.code
2	$B_1 \rightarrow$ **(** B_2 **)**	B_1.code = B_2.code
3	B → R	B.code = R.code
4	opB → **and**	opB.code = "and"
5	opB → **or**	opB.code = "or"
6	R → M_1 opR M_2	R.code = M_1.code+M_2.code+opR.code
7	$R_1 \rightarrow$ **(** R_2 **)**	R_1.code = R_2.code
8	opR → **<**	opR.code = "lessthan"
9	opR → **==**	opR.code = "equal"
10	$M_1 \rightarrow M_2$ opM T	M_1.code = M_2.code+T.code+opM.code
11	M → T	M.code = T.code
12	opM → **+**	opM.code = "add"
13	opM → **-**	opM.code = "sub"
14	T → **(** M **)**	T.code = M.code
15	T → **id**	T.code = "loadv "+getName(**id**)
16	T → **num**	T.code = "loadc "+getNum(**num**)

ไวยากรณ์ลักษณะประจำนี้ใช้สร้าง p-code โดยคำนวณลักษณะประจำ code สำหรับโหนดในต้นไม้แจงส่วน รูปข้างล่างนี้เป็นต้นไม้แจงส่วนของนิพจน์ x+5 < y and y==0 การคำนวณลักษณะประจำ code เป็นไปตามกฎของความหมาย เช่น ต้นไม้ย่อยที่มี R เป็นโหนดรากสร้างจากกฎข้อ 6 R → M opR M ลักษณะประจำ code ของโหนดนี้กำหนดด้วยกฎของความหมายที่คู่กัน คือ R.code = M_1.code + M_2.code + opR.code

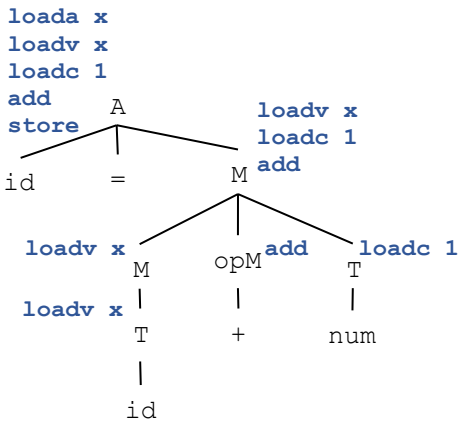


ไวยากรณ์ลักษณะประจำที่สร้าง P-code สำหรับคำสั่งกำหนดค่าให้ตัวแปร

กำหนดไวยากรณ์สำหรับคำสั่งกำหนดค่าให้ตัวแปรในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ กฎของความหมาย 2 ข้อนี้ระบุว่า p-code ของคำสั่งนี้ประกอบด้วยคำสั่ง `loada v` เมื่อ `v` เป็นตัวแปรทางซ้ายของเครื่องหมาย = ตามด้วย p-code ที่คำนวณนิพจน์ทางขวาของเครื่องหมาย = (คือ `M.code` หรือ `B.code`) ซึ่งทำให้ได้ค่าของนิพจน์อยู่บนแสตค ดังนั้นเมื่อ p-code ทำงานส่วนนี้จบ ค่าที่คำนวณจากนิพจน์จะอยู่บนสุดของสแตคและเลขที่อยู่ของตัวแปร `v` อยู่ถัดลงไป จากนั้นจึงใช้คำสั่ง `store` เพื่อเก็บค่านั้นในตัวแปร `v`

	Grammar rule	Semantic rule
17	`A → id = M`	`A.code = "loada "+getName(id)+M.code+"store"`
18	`A → id = B`	`A.code = "loada "+getName(id)+B.code+"store"`

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง `x=x+1` ลักษณะประจำ `code` ที่แสดงในต้นไม้นี้สร้างจากไวยากรณ์ลักษณะประจำข้างบน

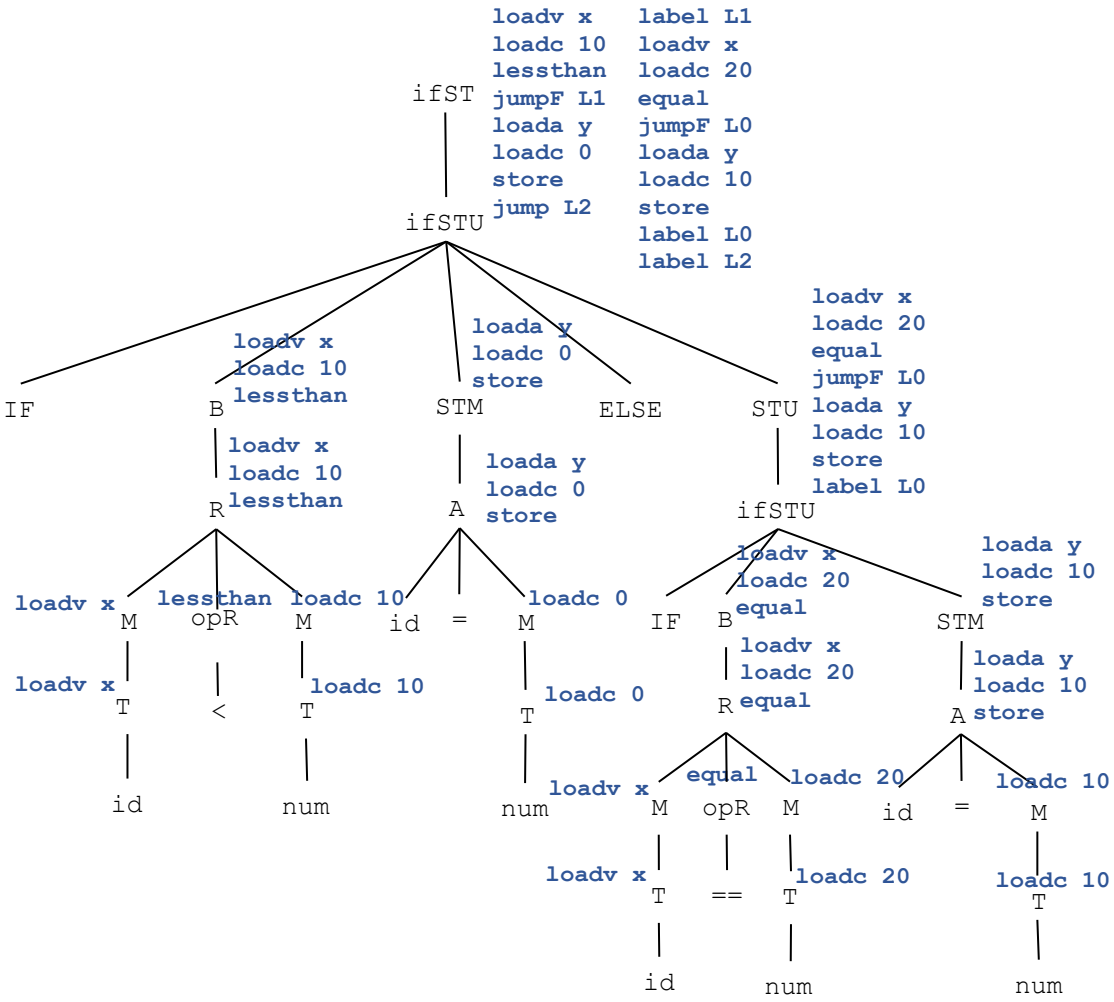


ไวยากรณ์ลักษณะประจำที่สร้าง P-code สำหรับคำสั่ง if

กำหนดไวยากรณ์สำหรับคำสั่ง if ในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ กฎของความหมายในข้อ 24-26 ระบุว่า p-code สำหรับคำสั่ง if สร้างจาก p-code ที่คำนวณเงื่อนไขซึ่งทำให้ได้ค่าของเงื่อนไขอยู่บนสุดของสแตค จากนั้นใช้คำสั่ง `jumpF` เพื่อกระโดดไปทำงานในส่วน else เมื่อเงื่อนไขเป็นเท็จ ตำแหน่งที่จะกระโดดไปทำงานนี้สร้างด้วยฟังก์ชัน `newLabel()` p-code ของส่วน then จะอยู่ต่อจากคำสั่ง `jumpF` และต้องตามด้วยคำสั่ง `jump` ที่จะทำให้กระโดดข้ามส่วน else ไป

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง `IF x<10 y=0 ELSE IF x==20 y=10` ลักษณะประจำ `code` ที่แสดงในต้นไม้นี้สร้างจากไวยากรณ์ลักษณะประจำข้างบน ในการสร้างลักษณะประจำ `code` ของโหนด `ifSTU` ในต้นไม้ย่อยด้านขวาล่าง เราต้องสร้างชื่อที่ระบุตำแหน่งในโปรแกรมก่อนด้วยฟังก์ชัน `newLabel()` ในตัวอย่างนี้สมมติว่า `OT` เป็นตำแหน่งที่ระบุด้วย `"L0"` จากนั้น `code` ของโหนดนี้สร้างจาก `code` ของโหนดลูก `B` ที่เป็นเงื่อนไขของคำสั่ง if ต่อด้วยคำสั่ง `jumpF` ไปยังตำแหน่ง `L0` ในโปรแกรม หลังคำสั่งนี้เป็น `code` ของโหนดลูก `STM` ที่เป็นส่วน `then` คำสั่งท้ายสุดเป็นการระบุตำแหน่ง `L0` ด้วยคำสั่ง `label`

	Grammar rule	Semantic rule
19	STM → ifSTM	STM.code = ifSTM.code
20	STM → A	STM.code = A.code
21	STU → ifSTU	STU.code = ifSTU.code
22	ifST → ifSTM	ifST.code = ifSTM.code
23	ifST → ifSTU	ifST.code = ifSTU.code
24	ifSTM → **IF** B STM_1 **ELSE** STM_2	EL=newLabel() OT=newLabel() ifSTM.code = B.code+"jumpF "+EL+STM_1.code+"jump "+OT+ "label "+EL+STM_2.code+"label "+OT
25	ifSTU → **IF** B STM	OT=newLabel() ifSTM.code = B.code+"jumpF "+OT+STM.code+"label "+OT
26	ifSTU → **IF** B STM_1 **ELSE** STU_2	EL=newLabel() OT=newLabel() ifSTM.code = B.code+"jumpF "+EL+STM_1.code+"jump "+OT+ "label "+EL+STU_2.code+"label "+OT

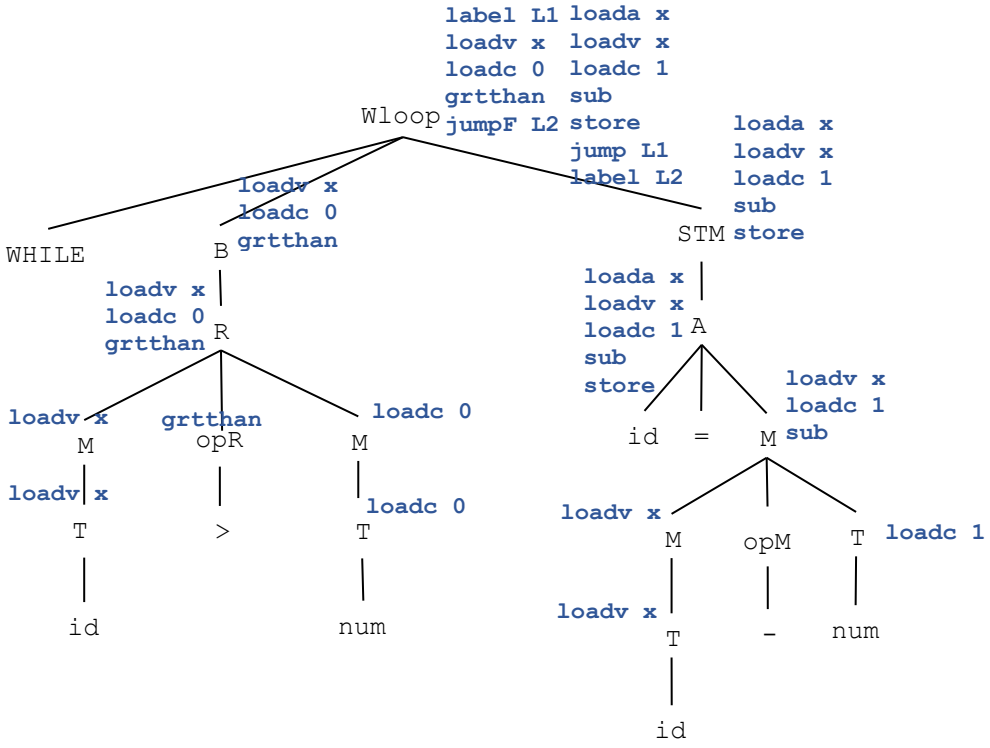


ไวยากรณ์ลักษณะประจำที่สร้าง P-code สำหรับคำสั่ง while

กำหนดไวยากรณ์สำหรับคำสั่ง while ในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ p-code สำหรับคำสั่ง while ต้องระบุตำแหน่งของคำสั่งแรกที่ตรวจสอบเงื่อนไข (ST ในกฎของความหมาย) เพื่อให้วนกลับมาทำงานซ้ำ และตำแหน่งของคำสั่งแรกที่อยู่ต่อจาก while loop (OT ในกฎของความหมาย) เพื่อให้กระโดดออกจากการวน กฎของความหมายนี้ระบุว่า p-code สำหรับคำสั่ง while สร้างจาก p-code ที่คำนวณเงื่อนไขตามด้วยคำสั่ง `jumpF OT` เพื่อกระโดดออกจากการวนซ้ำหากเงื่อนไขเป็นเท็จ แล้วตามด้วย p-code ของส่วนที่ทำซ้ำ ท้ายสุดเป็นคำสั่ง `jump` ที่จะทำให้กระโดดวนไปทำซ้ำที่ ST อีก

	Grammar rule	Semantic rule
27	Wloop → **WHILE** B STM	ST=newLabel() OT=newLabel() Wloop.code = "label "+ST+B.code+"jumpF "+OT +STM.code+"jump "+ST+"label "+OT

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง `WHILE x>0 x=x-1` ลักษณะประจำ code ที่แสดงในต้นไม้นี้สร้างจากไวยากรณ์ลักษณะประจำข้างบน ในการสร้างลักษณะประจำ code ของโหนด Wloop เราต้องสร้างชื่อที่ระบุตำแหน่งในโปรแกรมก่อนด้วยฟังก์ชัน `newLabel()` ในตัวอย่างนี้สมมติว่า ST เป็นตำแหน่งที่ระบุด้วย "L1" และ OT เป็นตำแหน่งที่ระบุด้วย "L2" จากนั้น code ของโหนดนี้เริ่มต้นด้วยคำสั่ง `label L1` ที่ระบุตำแหน่งเริ่มต้นการวนซ้ำ จากนั้นจึงต่อด้วย code ของโหนดลูก B ที่เป็นเงื่อนไขของคำสั่ง while ที่ต่อด้วยคำสั่ง `jumpF` ไปยังตำแหน่งในโปรแกรมสร้างเก็บไว้ในตัวแปร L2 ตามด้วย code ของโหนด STM คำสั่งท้ายสุดเป็นการระบุตำแหน่งคำสั่งแรกนอกโครงสร้าง while ที่สร้างไว้แล้วด้วยคำสั่ง `label`

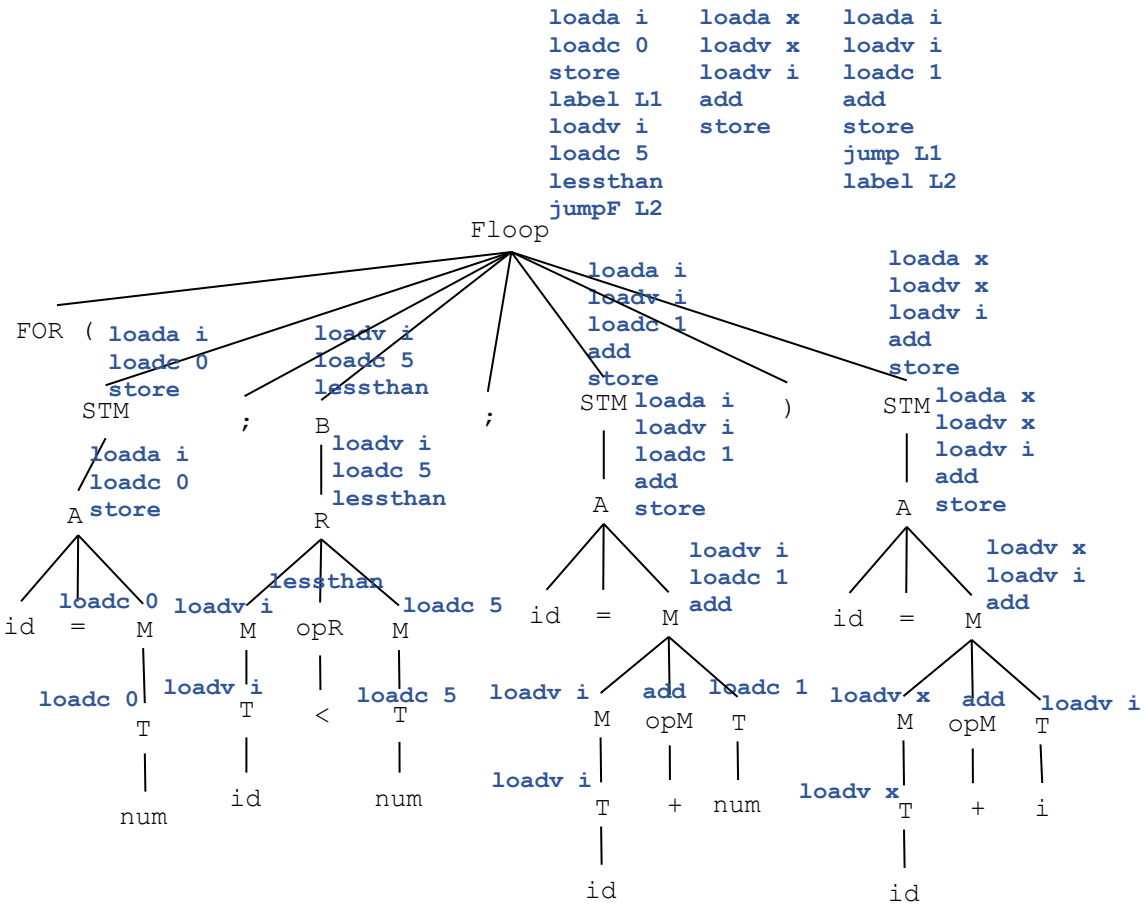


ไวยากรณ์ลักษณะประจำที่สร้าง P-code สำหรับคำสั่ง for

กำหนดไวยากรณ์สำหรับคำสั่ง for ในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ การสร้าง p-code สำหรับคำสั่ง for คล้ายกับคำสั่ง while แต่มีคำสั่งที่ทำแรกสุดและคำสั่งที่ทำก่อนวนกลับไปทำซ้ำแต่ละรอบ

	Grammar rule	Semantic rule
28	Floop → **FOR** (STM$_1$; B; STM$_2$) STM$_3$	L1=newLabel() L2=newLabel() Floop.code = STM$_1$.code+"label "+L1+ B.code+"jumpF "+L2 +STM$_3$.code +STM$_2$.code+"jump "+L1+"label "+L2

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง `FOR (i=0; i<5; i=i+1) x=x+i`



5.2.2 ไวยากรณ์ลักษณะประจำที่สร้าง 3-address code

ในหัวข้อนี้เราใช้ไวยากรณ์และลักษณะประจำ `code` จากหัวข้อ 5.2.1 ลักษณะประจำ `code` แสดงในต้นไม้แจงส่วนด้วยตัวอักษรสีฟ้า นอกจากนั้นยังมีลักษณะประจำ `var` ที่เก็บชื่อตัวแปรที่เก็บค่าที่คำนวณได้และต้องนำไปใช้ในส่วนอื่นของโปรแกรม ลักษณะประจำ `var` แสดงในต้นไม้แจงส่วนด้วยตัวอักษรสีแดง

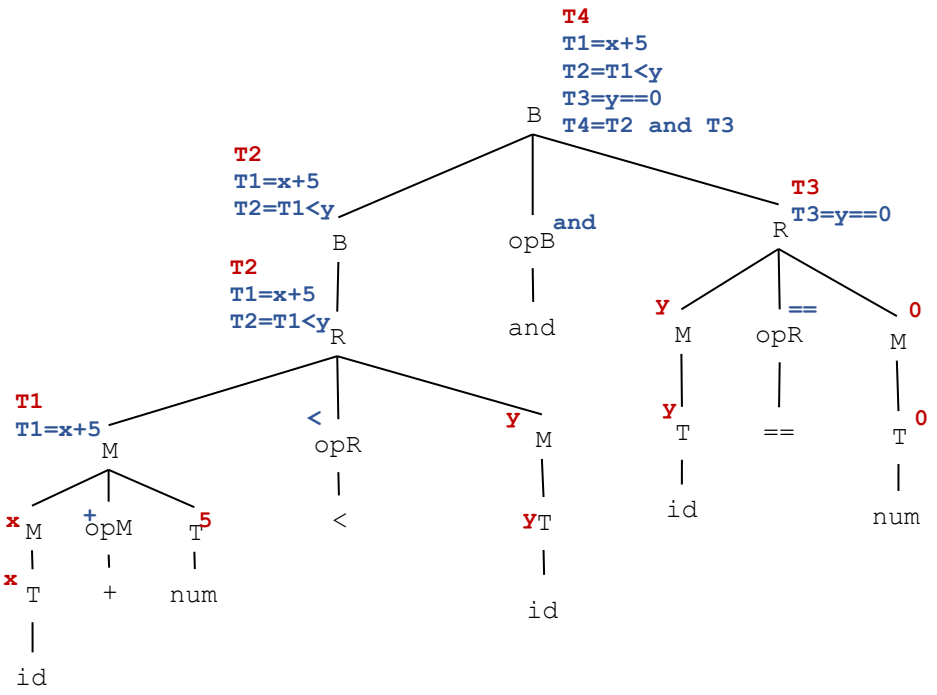
ไวยากรณ์ลักษณะประจำที่สร้าง 3-address code สำหรับนิพจน์

ในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ กฎของความหมายสร้าง 3-address code ที่คำนวณนิพจน์และเก็บในลักษณะประจำ `code` โดยสร้าง 3-address code ของนิพจน์ย่อยแล้วนำมาประกอบกันเป็น 3-address code สำหรับนิพจน์รวม ลักษณะประจำ `var` เป็นชื่อของตัวแปรชั่วคราวที่ใช้เก็บค่าที่คำนวณได้จากนิพจน์นั้น

	Grammar rule	Semantic rule
1	B_1 → B_2 opB R	B_1.var = newVar() B_1.code = B_2.code+R.code+B_1.var+"="+B_2.var+ opB.code+R.var
2	B_1 → **(** B_2 **)**	B_1.var = B_2.var B_1.code = B_2.code
3	B → R	B.var = R.var B.code = R.code
4	opB → **and**	opB.code = " and "
5	opB → **or**	opB.code = " or "
6	R → M_1 opR M_2	R.var = newVar() R.code = M_1.code+M_2.code+ R.var+"="+M_1.var+opR.code+ M_2.var
7	R_1 → **(** R_2 **)**	R_1.var = R_2.var R_1.code = R_2.code
8	opR → **<**	opR.code = " < "
9	opR → **==**	opR.code = " == "
10	M_1 → M_2 opM T	M_1.var = newVar() M_1.code = M_2.code+T.code+ M_1.var+"="+M_2.var+opM.code+T.var
11	M → T	M.var = T.var M.code = T.code
12	opM → **+**	opM.code = " + "
13	opM → **-**	opM.code = " - "
14	T → **(** M **)**	T.var = M.var T.code = M.code
15	T → **id**	T.var = getName(**id**) T.code = ""
16	T → **num**	T.var = getNum(**num**) T.code = ""

รูปข้างล่างนี้เป็นต้นไม้แจงส่วนของนิพจน์ `x+5 < y and y==0` จากกฎข้อ 15 และ 16 จะเห็นว่าลักษณะประจำ `code` ของโหนด `T` ที่มีลูกเป็น `id` หรือ `num` มีค่าเป็นสตริงว่างเนื่องจากเราสามารถอ้างถึงตัวแปรหรือค่าได้โดยตรง แต่ลักษณะประจำ `var` ของโหนดเหล่านี้ระบุชื่อของตัวแปรนั้น (จาก `id`) หรือสตริงที่ใช้อ้างถึงค่า `num` ลักษณะประจำ `var` นี้ใช้สร้าง 3-address code ของโหนดที่สูงขึ้นไป

จากกฎข้อ 6 โหนด `R` ในด้านขวาของต้นไม้มีลักษณะประจำ `var` เป็นตัวแปร `T3` ที่ใช้เก็บค่าที่ได้จากนิพจน์ในต้นไม้ย่อยนี้ ตัวแปรนี้สร้างจากฟังก์ชัน `newVar()` ส่วนลักษณะประจำ `code` ของโหนดนี้สร้างจาก M_1`.code` และ M_2`.code` ต่อด้วยคำสั่ง `T3 = y==0` ซึ่ง `y` และ `0` ในคำสั่งนี้ได้มาจากลักษณะประจำ `var` ของโหนดลูก

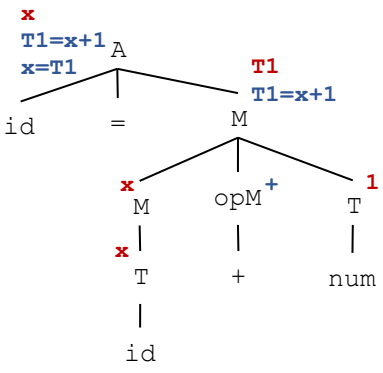


ไวยากรณ์ลักษณะประจำที่สร้าง 3-address code สำหรับคำสั่งกำหนดค่าตัวแปร

ในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ กฎของความหมายที่สร้าง 3-address code ระบุว่า 3-address code ของคำสั่งนี้ประกอบด้วยคำสั่งที่คำนวณค่าของนิพจน์ทางขวาของเครื่องหมาย = (คือ `M.code` หรือ `B.code`) เก็บในตัวแปรชั่วคราว ตามด้วยคำสั่งกำหนดค่าตัวแปรชั่วคราวนี้ให้ตัวแปรทางซ้ายของเครื่องหมาย = ลักษณะประจำ `var` เก็บชื่อตัวแปรชั่วคราวที่เก็บค่าที่คำนวณได้นี้

	Grammar rule	Semantic rule
17	`A → id = M`	`A.var = getName(id)` `A.code = M.code+ A.var +"="+M.var`
18	`A → id = B`	`A.var = getName(id)` `A.code = B.code+ A.var +"="+B.var`

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง `x=x+1` ลักษณะประจำ `code` ที่แสดงในต้นไม้นี้สร้างจากไวยากรณ์ลักษณะประจำข้างบน ตัวแปรชั่วคราว `T1` เก็บค่าของนิพจน์ `x+1` และคำสั่ง `x=T1` เก็บค่าของ `x+1` ในตัวแปร `T1` ในตัวแปร `x`

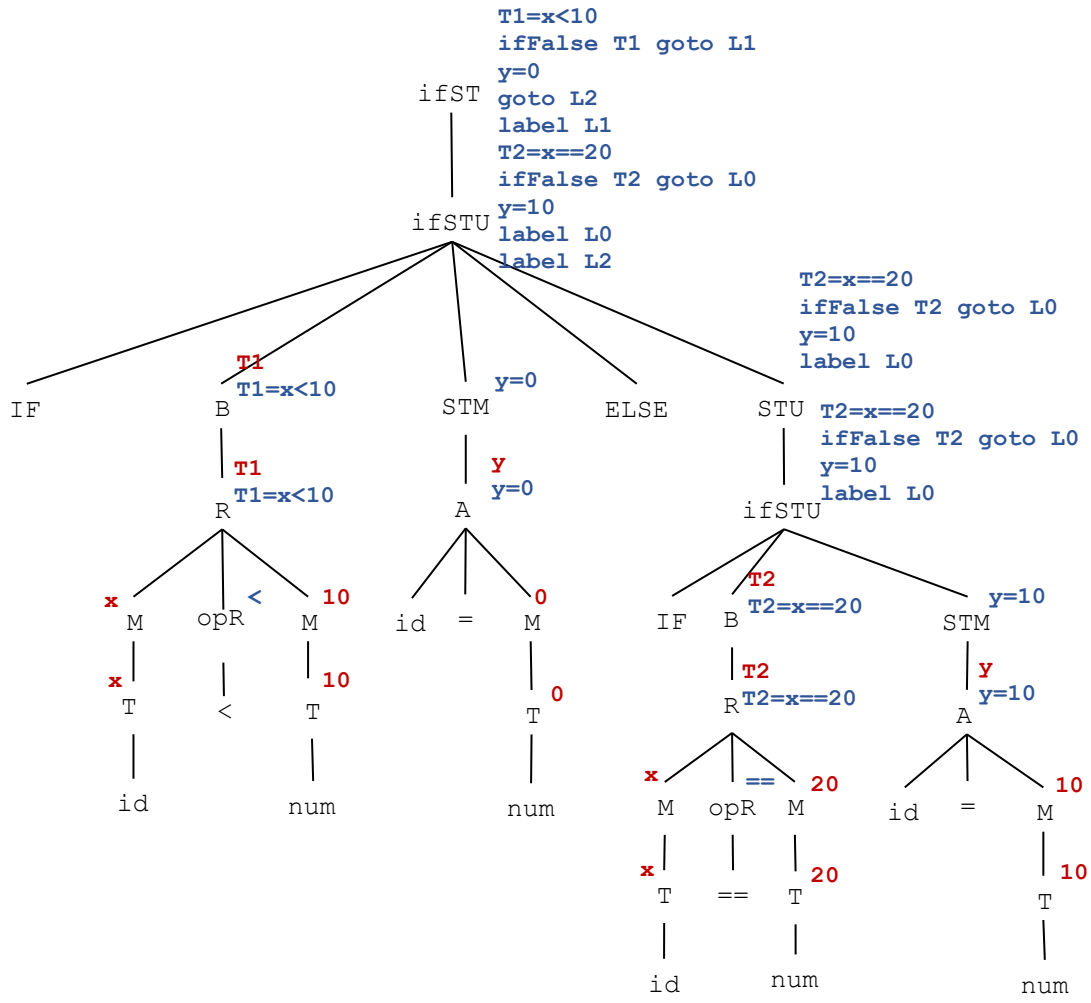


ไวยากรณ์ลักษณะประจำที่สร้าง 3-address code สำหรับคำสั่ง if

จากตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ กฎของความหมายในข้อ 24-26 ระบุว่า 3-address code สำหรับคำสั่ง if สร้างจาก 3-address code ที่คำนวณเงื่อนไขตามไวยากรณ์ของนิพจน์ที่กล่าวไปแล้ว ค่าของเงื่อนไขนี้เก็บในตัวแปรที่เก็บชื่อไว้ในลักษณะประจำ `var` จากนั้นใช้คำสั่ง `ifFalse … goto …` ตรวจสอบค่าในตัวแปรนี้ ถ้าเป็นเท็จจะกระโดดไปทำงานในส่วน else โดยตำแหน่งที่จะกระโดดไปทำงานนี้สร้างด้วยฟังก์ชัน `newLabel()` 3-address code ของส่วน then จะอยู่ต่อจากคำสั่ง `ifFalse` และต้องตามด้วยคำสั่ง `goto` ที่จะทำให้กระโดดข้ามส่วน else ไป ในกฎข้อ 24 และ 26 ตำแหน่ง `L1` เป็นส่วนของ else และ ตำแหน่ง `L2` เป็นคำสั่งหลังคำสั่ง if-then-else ในกฎข้อ 25 ตำแหน่ง `L` เป็นตำแหน่งของคำสั่งหลังคำสั่ง if-then

	Grammar rule	Semantic rule
19	STM → ifSTM	STM.code = ifSTM.code
20	STM → A	STM.code = A.code
21	STU → ifSTU	STU.code = ifSTU.code
22	ifST → ifSTM	ifST.code = ifSTM.code
23	ifST → ifSTU	ifST.code = ifSTU.code
24	ifSTM → **IF** B STM_1 **ELSE** STM_2	EL=newLabel() OT=newLabel() ifSTM.code = B.code+"ifFalse "+B.var+" goto "+EL+ STM_1.code+"goto "+OT+ "label "+EL+ STM_2.code+"label "+OT
25	ifSTU → **IF** B STM	OT=newLabel() ifSTM.code = B.code+"ifFalse "+B.var+" goto "+OT+ STM.code+"label "+OT
26	ifSTU → **IF** B STM_1 **ELSE** STU_2	EL=newLabel() OT=newLabel() ifSTM.code = B.code+"ifFalse "+B.var+" goto "+EL+ STM_1.code+"goto "+OT+ "label "+EL+ STU_2.code+"label "+OT

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง `IF x<10 y=0 ELSE IF x==20 y=10` ลักษณะประจำ `code` ที่แสดงในต้นไม้นี้สร้างจากไวยากรณ์ลักษณะประจำข้างบน ในการสร้างลักษณะประจำ `code` ของโหนด `ifSTU` ในต้นไม้ย่อยด้านขวาล่าง เราต้องสร้างชื่อที่ระบุตำแหน่งในโปรแกรมก่อนด้วยฟังก์ชัน `newLabel()` เก็บในตัวแปร `EL` และ `OT` ในตัวอย่างนี้ สมมติให้ `OT` เก็บสตริง `"L0"` จากนั้น `code` ของโหนดนี้สร้างจาก `code` ของโหนดลูก `B` ที่เป็นเงื่อนไขของคำสั่ง if ต่อด้วยคำสั่ง `ifFalse … goto …` ที่กระโดดไปยังตำแหน่งในโปรแกรมที่ตั้งชื่อไว้ด้วยฟังก์ชัน `newLabel()` หลังคำสั่งนี้เป็น `code` ของโหนดลูก `STM` ที่เป็นส่วน `then` จากนั้นตามด้วยคำสั่งท้ายสุดเป็นการระบุตำแหน่ง



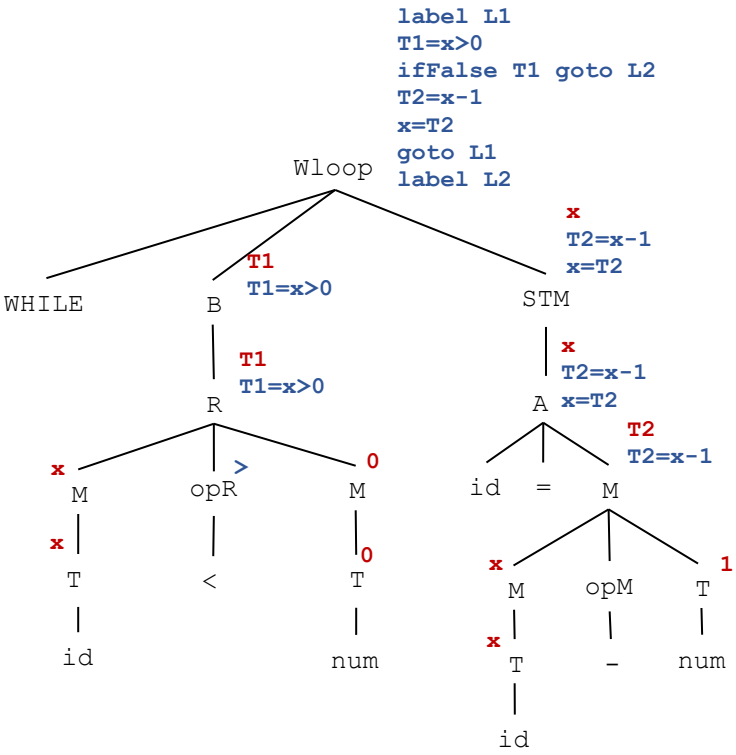
ไวยากรณ์ลักษณะประจำที่สร้าง 3-address code สำหรับคำสั่ง while

3-address code สำหรับคำสั่ง while ต้องระบุตำแหน่งของคำสั่งแรกที่ตรวจสอบเงื่อนไข (L1 ในกฎของความหมาย) เพื่อให้วนกลับไปทำงานซ้ำ และตำแหน่งของคำสั่งแรกที่อยู่ต่อจาก while loop (L2 ในกฎของความหมาย) เพื่อให้กระโดดออกจากการวน กฎของความหมายที่สร้าง 3-address code คล้ายกับการสร้าง p-code แต่ต้องรับค่าของเงื่อนไขของการวนซ้ำจากตัวแปรที่เก็บใน B.var เพื่อใช้ในคำสั่ง ifFalse …

	Grammar rule	Semantic rule
27	Wloop → **WHILE** B STM	ST=newLabel() OT=newLabel() Wloop.code = "label "+ST+B.code+"ifFalse "+B.var+"goto "+OT +STM.code+"goto "+ST+"label "+OT

รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง WHILE x>0 x=x-1 ในการสร้างลักษณะประจำ code ของโหนด Wloop ใช้ตำแหน่ง L1 และ L2 ที่สร้างด้วยฟังก์ชัน newLabel() ดังนั้น ST="L1" และ OT="L2" นอกจากนั้น T1 เป็นตัวแปรที่เก็บค่าของเงื่อนไขการวนซ้ำที่รับมาจาก จาก B.var ดังนั้น code ของโหนด Wloop เริ่มต้นด้วยคำสั่ง label L1 ที่ระบุตำแหน่งเริ่มต้นการวนซ้ำ จากนั้นจึงต่อด้วย code ของโหนดลูก B ที่เป็นเงื่อนไขของคำสั่ง while ที่ต่อด้วยคำสั่ง ifFalse T1 goto L2 ที่ทำให้กระโดดออกจาก

การวนซ้ำ ต่อจาก code ของโหนด STM เป็นคำสั่ง goto L1 ที่ทำให้วนไปตรวจสอบเงื่อนไขการวนซ้ำ สุดท้ายเป็นการระบุตำแหน่งคำสั่งแรกนอกโครงสร้าง while ที่สร้างไว้แล้วด้วยคำสั่ง label L2

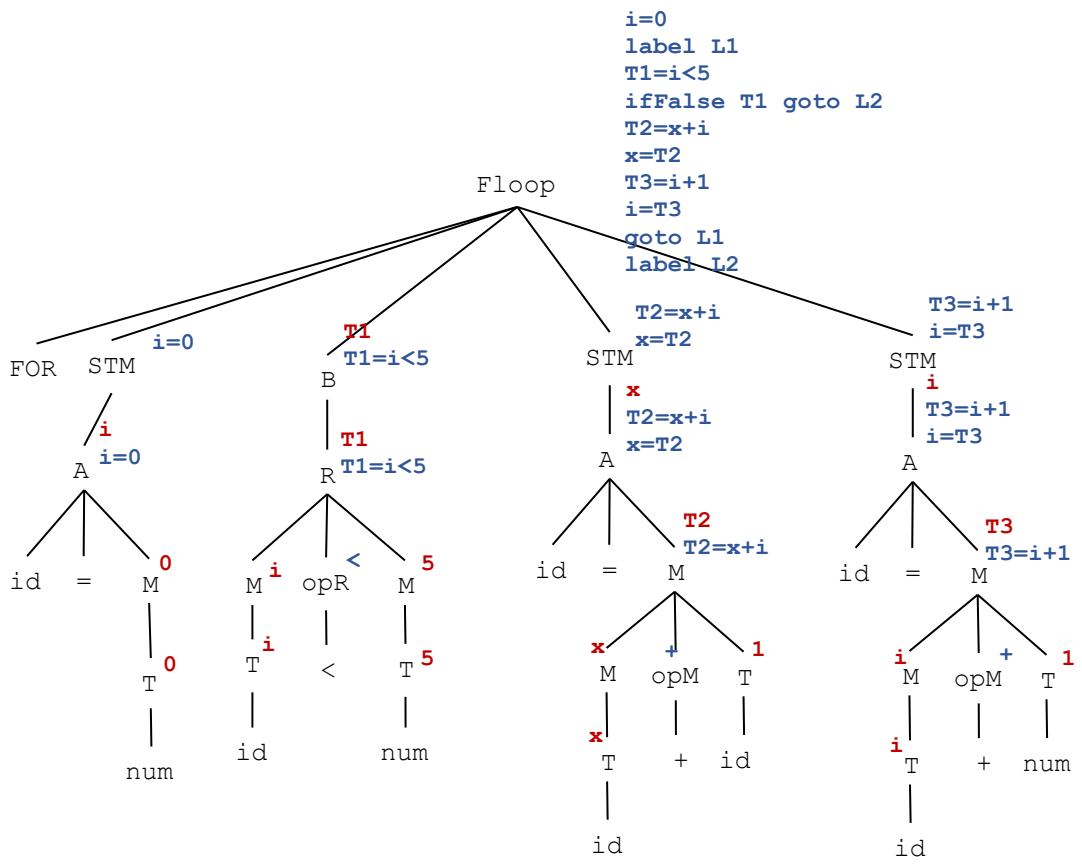


ไวยากรณ์ลักษณะประจำที่สร้าง 3-address code สำหรับคำสั่ง for

กำหนดไวยากรณ์สำหรับคำสั่ง for ในตารางไวยากรณ์ลักษณะประจำข้างล่างนี้ การสร้าง 3-address code สำหรับคำสั่ง for คล้ายกับคำสั่ง while แต่มีคำสั่งที่ทำแรกสุดและคำสั่งที่ทำก่อนวนกลับไปทำซ้ำแต่ละรอบ

	Grammar rule	Semantic rule
28	Floop → **FOR** (STM$_1$; B; STM$_2$) STM$_3$	ST=newLabel() OT=newLabel() Floop.code = STM$_1$.code+"label "+ST+ B.code+ "ifFalse "+B.var+"goto "+OT+ STM$_3$.code +STM$_2$.code+"goto "+ST+ "label "+OT

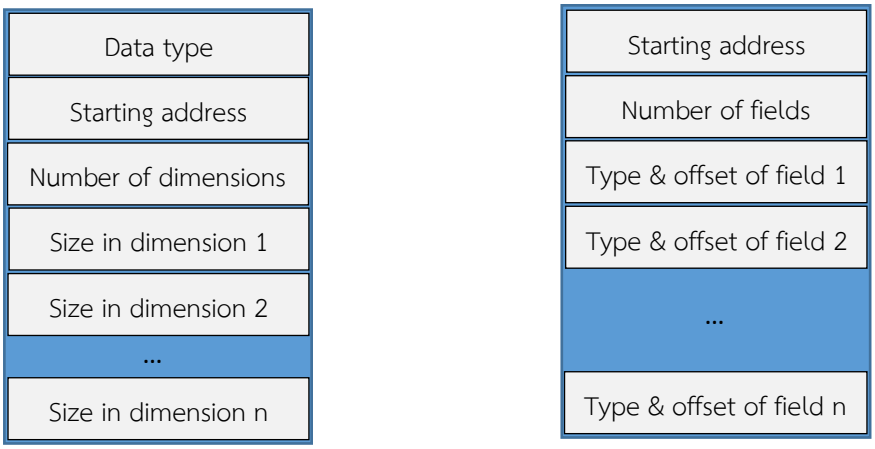
รูปข้างล่างนี้แสดงต้นไม้แจงส่วนของคำสั่ง FOR (i=0; i<5; i=i+1) x=x+i โดยเพิ่ม 3-address code ของคำสั่ง i=0 ไว้ก่อนการตรวจสอบเงื่อนไข และ 3-address code ของคำสั่ง i=i+1 ไว้ก่อนคำสั่ง goto L1



5.3 ชนิดข้อมูลและไวยากรณ์ลักษณะประจำสำหรับเข้าถึงข้อมูลชนิดต่างๆ

ภาษาระดับสูง (high-level language) มีชนิดข้อมูลพื้นฐาน (primitive data type) เช่น จำนวนเต็ม (integer) จำนวนจริง (floating-point number) ตัวอักขระ (character) เมื่อใช้รหัสเครื่อง เราสามารถอ้างถึงข้อมูลพื้นฐานเหล่านี้ด้วยเลขที่อยู่ของข้อมูลในหน่วยความจำซึ่งเก็บไว้ในตารางสัญลักษณ์ เมื่อใช้รหัสกลาง เราสามารถใช้ตัวแปรอ้างถึงข้อมูลพื้นฐานเหล่านี้โดยตรง

อย่างไรก็ตาม ภาษาระดับสูงยังมีข้อมูลที่มีโครงสร้างข้อมูลที่ซับซ้อนมากขึ้น เช่น ข้อมูลแถวลำดับ (array) ข้อมูลแบบระเบียน (record) ข้อมูลเหล่านี้เก็บในหน่วยความจำที่ต่อเนื่องกันทั้งชุดและเก็บเลขที่อยู่ของตำแหน่งแรกในหน่วยความจำที่ใช้เก็บข้อมูลนั้น เมื่อใช้รหัสเครื่องหรือรหัสกลาง เราต้องการอ้างถึงข้อมูลพื้นฐานแต่ละตัวในโครงสร้างนี้ด้วย ดังนั้นตัวสร้างรหัสต้องคำนวณเลขที่อยู่ของข้อมูลพื้นฐานแต่ละตัวในโครงสร้างได้ ทั้งนี้ตัวแปลภาษาต้องเก็บรายละเอียดของโครงสร้างที่จำเป็นสำหรับการคำนวณเลขที่อยู่ของข้อมูลแต่ละตัวในตารางสัญลักษณ์ รายละเอียดที่จำเป็นสำหรับข้อมูลแถวลำดับ คือ ชนิดของค่าในแถวลำดับ เลขที่อยู่ของหน่วยความจำที่เก็บข้อมูลแถวลำดับ จำนวนมิติ (dimension) ของข้อมูลแถวลำดับ และ ขนาดของข้อมูลแถวลำดับในแต่ละมิติ ดังแสดงในรูปข้างล่างรูปซ้าย รายละเอียดที่จำเป็นสำหรับข้อมูลแบบระเบียน คือ เลขที่อยู่ของหน่วยความจำที่เก็บข้อมูลแบบระเบียน จำนวนฟิล์ด (field), ชนิดของข้อมูลแต่ละฟิล์ด และระยะห่างของแต่ละฟิล์ดจากเลขที่อยู่แรกของตัวแปร ดังแสดงในรูปต่อไปนี้

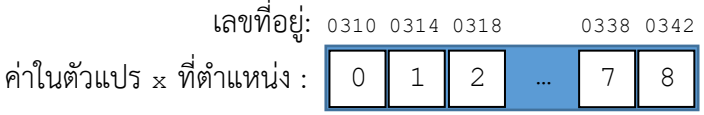


รายละเอียดที่จำเป็นสำหรับข้อมูลแถวลำดับ รายละเอียดที่จำเป็นสำหรับข้อมูลแบบระเบียน

หัวข้อนี้อธิบายการสร้างรหัสกลางคำนวณเลขที่อยู่ของข้อมูลแต่ละตัวในโครงสร้าง 2 แบบนี้ หัวข้อ 5.3.1 แสดงการสร้างรหัสกลางคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแถวลำดับ หัวข้อ 5.3.2 แสดงการสร้างรหัสกลางคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแบบระเบียน

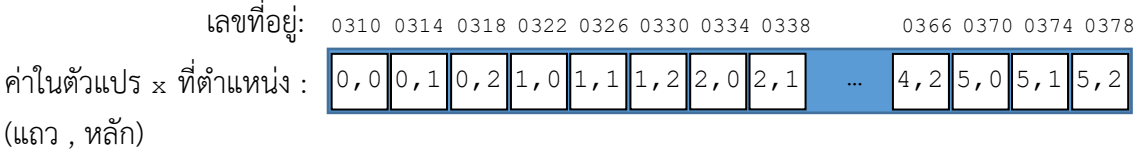
5.3.1 การสร้างรหัสกลางคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแถวลำดับ

ข้อมูลในตัวแปรแถวลำดับถูกจัดเก็บในหน่วยความจำที่ต่อกันโดยเรียงตามตำแหน่ง สมมติให้ `x` เป็นตัวแปรแถวลำดับ 1 มิติขนาด 9 ตัว แต่ละตัวใช้เนื้อที่ 4 ไบท์ ถ้าตัวแปรนี้ถูกเก็บที่หน่วยความจำเลขที่ 0310 เป็นต้นไป ค่าแต่ละตำแหน่งในตัวแปร `x` เก็บในตำแหน่งที่แสดงในรูปข้างล่างนี้ ดังนั้นถ้า `x` เป็นตัวแปรแถวลำดับ 1 มิติขนาด `n` ตัว เลขที่อยู่ของ `x[i]` เป็น `&x+is` เมื่อ `&x` เป็นเลขที่อยู่ของหน่วยความจำที่เก็บ `x` และ `s` เป็นขนาดของหน่วยความจำที่ใช้เก็บค่าหนึ่งตัว

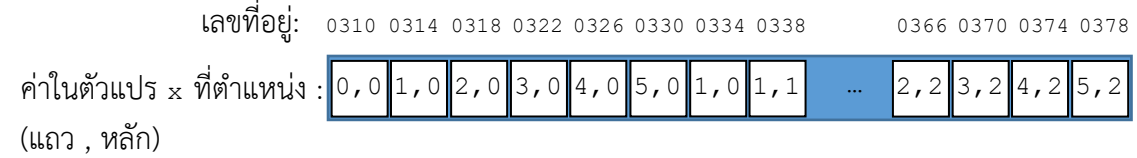


สำหรับตัวแปรแถวลำดับหลายมิติ เมื่อนำมาเก็บเรียงในหน่วยความจำที่จัดเรียงตามเลขที่อยู่แบบหนึ่งมิติ เราต้องเลือกว่าจัดลำดับตามมิติใดก่อน ในการจัดแบบแถวเป็นหลัก (row-major order) จะเรียงค่าทีละแถวเรียงจากแถวแรกไปยังแถวสุดท้าย ส่วนการจัดแบบคอลัมน์เป็นหลัก (column-major order) จะเรียงค่าทีละคอลัมน์เรียงจากคอลัมน์แรกไปยังคอลัมน์สุดท้าย

ในการจัดเรียงแบบแถวเป็นหลัก ถ้า `x` เป็นตัวแปรแถวลำดับ 2 มิติขนาด 6 แถว 3 คอลัมน์ แต่ละตัวใช้เนื้อที่ 4 ไบท์ ถ้าตัวแปรนี้ถูกเก็บที่หน่วยความจำเลขที่ 0310 เป็นต้นไป ค่าแต่ละตำแหน่งในตัวแปร `x` เก็บในตำแหน่งที่แสดงในรูปข้างล่างนี้ ดังนั้นถ้า `x` เป็นตัวแปรแถวลำดับ 2 มิติขนาด `m` แถว `n` คอลัมน์ เลขที่อยู่ของ `x[i][j]` เป็น `&x+(i*n+j)*s` เมื่อ `&x` เป็นเลขที่อยู่ของหน่วยความจำที่เก็บ `x` และ `s` เป็นขนาดของหน่วยความจำที่ใช้เก็บค่าหนึ่งตัว



ในการจัดเรียงแบบคอลัมน์เป็นหลัก ถ้า x เป็นตัวแปรแถวลำดับ 2 มิติขนาด 6 แถว 3 คอลัมน์ แต่ละตัวใช้เนื้อที่ 4 ไบท์ ถ้าตัวแปรนี้ถูกเก็บที่หน่วยความจำเลขที่ 0310 เป็นต้นไป ค่าแต่ละตำแหน่งในตัวแปร &x เก็บในตำแหน่งที่แสดงในรูปข้างล่างนี้ ดังนั้นถ้า x เป็นตัวแปรแถวลำดับ 2 มิติขนาด m แถว n คอลัมน์ เลขที่อยู่ของ x[i][j] เป็น &x+(j*m+i)*s เมื่อ &x เป็นเลขที่อยู่ของหน่วยความจำที่เก็บ x และ s เป็นขนาดของหน่วยความจำที่ใช้เก็บค่าหนึ่งตัว



การคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแถวลำดับด้วย P-code

กำหนดให้ x เป็นตัวแปรแถวลำดับ 1 มิติ, i เป็นตัวแปรที่เก็บตำแหน่งของค่าในแถวลำดับ และ s เก็บขนาดของค่าหนึ่งค่าในตัวแปร (เช่น จำนวนจริงมีขนาด 4 ไบท์ ตัวอักขระมีขนาด 1 ไบท์) &x เป็นเลขที่อยู่ของค่าแรกในแถวลำดับ (ในลักษณะเดียวกับที่ใช้ในภาษาซี) ส่วนของ P-code ต่อไปนี้คำนวณเลขที่อยู่ของ x[i] ด้วยคำสั่ง ixa

```
loada x      // stack content : &x
loadv i      // stack content : &x | i
ixa s        // stack content : &x+is
```

นอกจากนั้นยังใช้คำสั่ง ind k ที่คำนวณเลขที่อยู่ในหน่วยความจำจากค่าบนสุดของสแตคบวก k และเก็บในสแตค ตัวอย่างข้างล่างนี้แสดงการกำหนดค่าตัวหนึ่งในตัวแปรแถวลำดับด้วยคำสั่ง x[i]=0 และนำค่าตัวหนึ่งในตัวแปรแถวลำดับมาเก็บในอีกตัวแปรด้วยคำสั่ง y=x[i]

P-code สำหรับคำสั่ง x[i]=0

```
loada x
loadv i
ixa s
loadc 0
store
```

P-code สำหรับคำสั่ง y=x[i]

```
loada y
loada x
loadv i
ixa s
ind 0
store
```

ในตัวอย่างนี้ให้ตัวแปรแถวลำดับเก็บแบบแถวเป็นหลักและให้ x เป็นตัวแปรแถวลำดับ 2 มิติที่มีขนาด m แถว n คอลัมน์ โดย i และ j เป็นตัวแปรที่เก็บแถวและคอลัมน์ของค่าใน x และ s เก็บขนาดของค่าหนึ่งค่าในตัวแปร &x เป็นเลขที่อยู่ของค่าแรกในแถวลำดับ ส่วนของ P-code ต่อไปนี้คำนวณเลขที่อยู่ของ x[i][j] คือ x+(i*n+j)*s ด้วยคำสั่ง ixa

```
loada x      // stack content : x
loadv j      // stack content : x | j
loadv i      // stack content : x | j | i
ixa n        // stack content : x | j+i*n
ixa s        // stack content : x+(j+i*n)s
```

ตัวอย่างข้างล่างนี้แสดงการกำหนดค่าตัวหนึ่งในตัวแปรแถวลำดับด้วยคำสั่ง `x[i][j]=0` และนำค่าตัวหนึ่งในตัวแปรแถวลำดับมาเก็บในอีกตัวแปรด้วยคำสั่ง `y=x[i][j]`

P-code สำหรับคำสั่ง `x[i][j]=0`

```
loada x
loadv j
loadv i
ixa n
ixa s
loadc 0
store
```

P-code สำหรับคำสั่ง `y=x[i][j]`

```
loada y
loada x
loadv j
loadv i
ixa n
ixa s
ind 0
store
```

การคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแถวลำดับด้วย 3-address code

กำหนดให้ `x` เป็นตัวแปรแถวลำดับ 1 มิติ `i` เป็นตัวแปรที่เก็บตำแหน่งของค่าในแถวลำดับ และ `s` เก็บขนาดของค่าหนึ่งค่าในตัวแปร (เช่น จำนวนจริงมีขนาด 4 ไบท์ ตัวอักขระมีขนาด 1 ไบท์) `&x` เป็นเลขที่อยู่ของค่าแรกในแถวลำดับ ส่วนของ 3-address code ต่อไปนี้คำนวณเลขที่อยู่ของ `x[i]` คือ `&x+is`

```
t1 = i*s
t2 = &x+t1
```

ตัวอย่างข้างล่างนี้แสดงการกำหนดค่าตัวหนึ่งในตัวแปรแถวลำดับด้วยคำสั่ง `x[i]=0` และนำค่าตัวหนึ่งในตัวแปรแถวลำดับมาเก็บในอีกตัวแปรด้วยคำสั่ง `y=x[i]`

3-address code สำหรับคำสั่ง `x[i]=0`

```
t1 = i*s
t2 = &x+t1
*t2 = 0
```

3-address code สำหรับคำสั่ง `y=x[i]`

```
t1 = i*s
t2 = &x+t1
y = *t2
```

ในตัวอย่างนี้ให้ตัวแปรแถวลำดับเก็บแบบแถวเป็นหลักและให้ `x` เป็นตัวแปรแถวลำดับ 2 มิติที่มีขนาด `m` แถว `n` คอลัมน์ โดย `i` และ `j` เป็นตัวแปรที่เก็บแถวและคอลัมน์ของค่าใน `x` และ `s` เก็บขนาดของค่าหนึ่งค่าในตัวแปร `&x` เป็นเลขที่อยู่ของค่าแรกในแถวลำดับ ส่วนของ 3-address code ต่อไปนี้คำนวณเลขที่อยู่ของ `x[i][j]` คือ `x+(i*n+j)s`

```
t1 = i*n
t2 = t1+j
t3 = t2*s
t4 = &x+t3
```

ตัวอย่างข้างล่างนี้แสดงการกำหนดค่าตัวหนึ่งในตัวแปรแถวลำดับด้วยคำสั่ง `x[i][j]=0` และนำค่าตัวหนึ่งในตัวแปรแถวลำดับมาเก็บในอีกตัวแปรด้วยคำสั่ง `y=x[i][j]`

3-address code สำหรับคำสั่ง `x[i][j]=0`

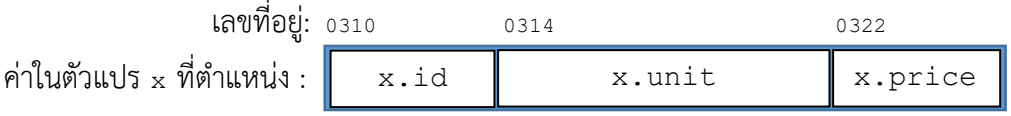
```
t1 = i*n
t2 = t1+j
t3 = t2*s
t4 = &x+t3
*t4 = 0
```

3-address code สำหรับคำสั่ง `y=x[i][j]`

```
t1 = i*n
t2 = t1+j
t3 = t2*s
t4 = &x+t3
y = *t4
```

5.3.2 การสร้างรหัสกลางคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแบบระเบียน

ข้อมูลในตัวแปรแบบระเบียนถูกจัดเก็บในหน่วยความจำที่ต่อกันโดยเรียงตามลำดับในระเบียน สมมติให้ `x` เป็นตัวแปรแบบระเบียนที่ประกอบด้วยฟิล์ด `id` ที่เป็นจำนวนเต็มขนาด 4 ไบท์, `price` ที่เป็นจำนวนจริงขนาด 8 ไบท์ และ `unit` ที่เป็นจำนวนเต็มขนาด 4 ไบท์ ถ้าตัวแปรนี้ถูกเก็บที่หน่วยความจำเลขที่ 0310 เป็นต้นไป ค่าของแต่ละฟิล์ดเก็บในตำแหน่งที่แสดงในรูปข้างล่างนี้ ดังนั้นถ้า `x` เป็นตัวแปรแถวลำดับ 1 มิติขนาด `n` ตัว เลขที่อยู่ของ `x[i]` เป็น `&x+is` เมื่อ `&x` เป็นเลขที่อยู่ของหน่วยความจำที่เก็บ `x` และ `s` เป็นขนาดของหน่วยความจำที่ใช้เก็บค่าหนึ่งตัว



การคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแถวลำดับด้วย P- code

กำหนดให้ `x` เป็นตัวแปรแบบระเบียน, `f` เป็นชื่อฟิล์ด, `&x` เป็นเลขที่อยู่ของค่าแรกในตัวแปร, `k` เป็นระยะห่างของฟิล์ด `f` จากเลขที่อยู่ของค่าแรกในตัวแปร ส่วนของ P-code ต่อไปนี้คำนวณเลขที่อยู่ของ `x.f`

```
loada x      // stack content : &x
loadc k      // stack content : &x | k
add          // stack content : &x+k
```

ตัวอย่างข้างล่างนี้แสดงการกำหนดค่าตัวหนึ่งในตัวแปรแถวลำดับด้วยคำสั่ง `x.f=0` และนำค่าตัวหนึ่งในตัวแปรแถวลำดับมาเก็บในอีกตัวแปรด้วยคำสั่ง `y=x.f`

p-code สำหรับคำสั่ง `x.f=0`

```
loada x
loadc k
add
loadc 0
store
```

P-code สำหรับคำสั่ง `y=x.f`

```
loada y
loada x
ind k
store
```

การคำนวณเลขที่อยู่ของข้อมูลในตัวแปรแถวลำดับด้วย 3-address code

กำหนดให้ `x` เป็นตัวแปรแบบระเบียนดังที่กล่าวไป ส่วนของ 3-address code ต่อไปนี้คำนวณเลขที่อยู่ของ `x.f`

```
t1 = &x
t2 = t1+k
```

ตัวอย่างข้างล่างนี้แสดงการกำหนดค่าตัวหนึ่งในตัวแปรแถวลำดับด้วยคำสั่ง `x.f=0` และนำค่าตัวหนึ่งในตัวแปรแถวลำดับมาเก็บในอีกตัวแปรด้วยคำสั่ง `y=x.f`

3-address code สำหรับคำสั่ง `x.f=0`

```
t1 = &x
t2 = t1+k
*t2 = 0
```

3-address code สำหรับคำสั่ง `y=x.f`

```
t1 = &x
t2 = t1+k
y = *t2
```

6. สภาพแวดล้อมเวลาดำเนินงาน

การทำงานของโปรแกรมมักเกี่ยวข้องกับตัวแปรซึ่งมีหลายชนิด ตัวแปรบางตัวถูกกำหนดให้ใช้ได้ตั้งแต่เริ่มต้นจนจบการทำงานของโปรแกรม ตัวแปรบางตัวใช้ในระหว่างที่ฟังก์ชันหรือส่วนย่อยของโปรแกรมทำงานเท่านั้น ตัวแปลภาษาต้องจัดการเนื้อที่ในหน่วยความจำให้ตัวแปรเหล่านี้อย่างเหมาะสม หัวข้อนี้อธิบายการจัดการหน่วยความจำในโปรแกรมสำหรับตัวแปรชนิดต่างๆ หัวข้อ 6.1 อธิบายตัวแปรแบบต่าง ๆ ที่ใช้ในภาษาโปรแกรม หัวข้อ 6.2 อธิบายการจัดหน่วยความจำสำหรับตัวแปรชนิดต่าง ๆ หัวข้อ 6.3 อธิบายคำสั่งที่ตัวแปลภาษาต้องเพิ่มในรหัสกลางหรือรหัสเครื่องเพื่อจัดการเกี่ยวกับฟังก์ชัน

6.1 ชนิดของตัวแปรในภาษาโปรแกรม

ตัวแปรในภาษาโปรแกรมแบ่งตามการผูก (binding) กับหน่วยความจำและช่วงชีวิต (lifetime) ของตัวแปรได้เป็น 4 ชนิดคือ ตัวแปรแบบสถิต (static variable) ตัวแปรในสแตคแบบพลวัต (stack-dynamic variable) ตัวแปรในฮีปแบบกำหนดโดยปริยาย (implicit heap dynamic variable) และ ตัวแปรในฮีปแบบกำหนดชัดแจ้ง (explicit heap dynamic variable)

6.1.1 ตัวแปรแบบสถิต (static variable)

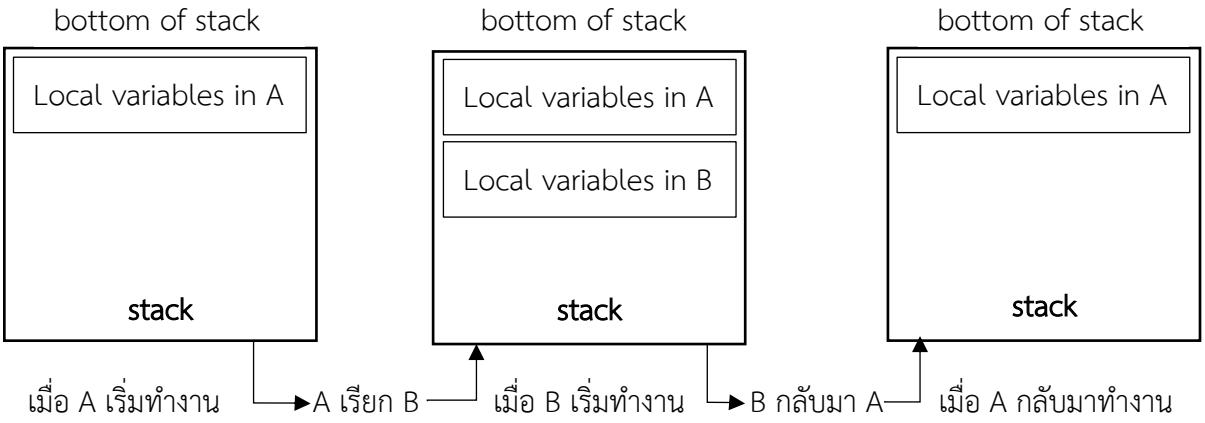
ตัวแปรแบบสถิตเป็นตัวแปรที่มีช่วงชีวิตยาวตลอดการทำงานของโปรแกรม ตัวแปรในภาษาซีที่ประกาศโดยมีคำว่า `static` กำกับเป็นตัวแปรแบบสถิต เช่น `static int x` ตัวแปลภาษาต้องกำหนดที่อยู่ในหน่วยความจำให้ตัวแปรชนิดนี้ตั้งแต่โปรแกรมเริ่มทำงานและตัวแปรนั้นจะผูกกับเลขที่อยู่ในหน่วยความจำที่ตำแหน่งนั้นตลอดการทำงาน ดังนั้นจึงสามารถอ้างถึงตัวแปรนี้ในรหัสเครื่องด้วยเลขที่อยู่นั้นตลอดการทำงานของโปรแกรม เช่น ถ้าประกาศตัวแปร `x` ดังที่ยกตัวอย่างไปแล้ว ตัวแปลภาษากำหนดเลขที่อยู่ในหน่วยความจำที่เก็บตัวแปร `x` ตั้งแต่โปรแกรมเริ่มทำงาน ในที่นี้สมมุติให้เป็นเลขที่อยู่ 0326 จากนั้นจะใช้เลขที่อยู่ 0326 นี้อ้างถึงตัวแปร `x` ในโปรแกรมได้เสมอ

6.1.2 ตัวแปรในสแตคแบบพลวัต (stack-dynamic variable)

ในภาษาโปรแกรมมีตัวแปรที่ใช้ได้เฉพาะในฟังก์ชันและมีช่วงชีวิตอยู่ในช่วงที่ฟังก์ชันนั้นทำงานเท่านั้น ตัวอย่างของตัวแปรประเภทนี้ ได้แก่ ตัวแปรเฉพาะที่ (local variable) ซึ่งประกาศไว้ในฟังก์ชันในภาษาซีและภาษาจาวา เนื่องจากตัวแปรประเภทนี้มีชีวิตอยู่เมื่อฟังก์ชันนั้นทำงาน แต่ฟังก์ชันหนึ่งอาจเรียกอีกฟังก์ชันหนึ่งทำงาน เช่น ฟังก์ชัน A เรียกฟังก์ชัน B ให้ทำงานซึ่งทำให้ฟังก์ชัน A ต้องหยุดการทำงานไปเมื่อฟังก์ชัน B เริ่มทำงาน ฟังก์ชัน A จะกลับมาทำงานอีกครั้งเมื่อฟังก์ชัน B ทำงานเสร็จ ดังนั้นตัวแปรในฟังก์ชัน A ต้องถูกเก็บไว้ เมื่อฟังก์ชัน B ทำงานเสร็จ ฟังก์ชัน A จะกลับมาทำงานต่อโดยต้องนำตัวแปรในฟังก์ชัน A กลับมาใช้ได้โดยไม่มีการเปลี่ยนค่าไปจากตอนที่ฟังก์ชัน A หยุดทำงาน

เนื่องจากการเรียก (call) ฟังก์ชันและการกลับ (return) จากฟังก์ชันมีลักษณะเหมือนการทำงานของสแตค คือ ฟังก์ชันที่ถูกเรียกหลังจะจบการทำงานก่อนในลักษณะเข้าหลัง-ออกก่อน (last-in, first-out) ตัวแปลภาษาจึงสามารถจัดการที่เก็บตัวแปรที่ใช้ในฟังก์ชันแบบสแตคได้เช่นกัน ตัวแปลภาษาจัดแบ่งที่ใน

หน่วยความจำไว้ใช้เป็นสแตคที่เก็บตัวแปรเหล่านี้ จากรูปข้างล่างนี้ ตัวแปลภาษาจะใช้ที่ว่างบนสแตคเพื่อเก็บตัวแปรในฟังก์ชัน A เมื่อฟังก์ชัน A เริ่มทำงาน หากฟังก์ชัน A มีการเรียกใช้ฟังก์ชัน B ต่อไปก็จะใช้ที่ถัดไปบนสแตคเพื่อเก็บตัวแปรในฟังก์ชัน B ดังนั้นหากฟังก์ชัน A ยังทำงานไม่เสร็จตัวแปรของฟังก์ชัน A จะยังถูกเก็บไว้ในสแตค ในขณะที่ด้านบนเป็นที่เก็บตัวแปรของฟังก์ชัน B เมื่อฟังก์ชัน B ทำงานเสร็จแล้ว ตัวแปรของฟังก์ชัน B ที่อยู่ด้านบนก็จะถูกนำออกไปจากสแตคซึ่งทำให้ตัวแปรของฟังก์ชัน A ปรากฏอยู่บนสุดของสแตคแทนและถูกนำกลับมาใช้ในฟังก์ชัน A ได้โดยมีค่าเดียวกับตอนที่ฟังก์ชันหยุดทำงาน ดังนั้นเลขที่อยู่ในหน่วยความจำที่ผูกกับตัวแปรจะไม่เปลี่ยนไประหว่างการทำงาน



6.1.3 ตัวแปรในฮีปแบบกำหนดชัดแจ้ง (explicit heap dynamic variable)

นอกจากตัวแปรสองชนิดที่ได้กล่าวไปแล้ว ภาษาโปรแกรมยังมีตัวแปรที่ผู้เขียนโปรแกรมสามารถระบุให้สร้างขึ้นมาหรือให้ลบตัวแปรนั้นทิ้งในขณะที่โปรแกรมทำงานได้ เช่น ในภาษาจาวา ผู้เขียนโปรแกรมสามารถใช้คำสั่ง `new` เพื่อขอเนื้อที่ในหน่วยความจำสำหรับตัวแปรหนึ่งได้ และใช้คำสั่ง `free` เพื่อเลิกใช้หน่วยความจำสำหรับตัวแปรนั้น ดังนั้นช่วงชีวิตของตัวแปรนั้นถูกกำหนดโดยผู้เขียนโปรแกรมซึ่งทำให้ตัวแปลภาษาไม่สามารถกำหนดช่วงชีวิตของตัวแปรนั้นได้ ตัวแปลภาษาจะแบ่งเนื้อที่ในหน่วยความจำสำหรับตัวแปรชนิดนี้แยกไว้ เมื่อมีคำสั่งขอใช้ที่สำหรับตัวแปรแบบนี้ก็จะขอเนื้อที่ในหน่วยความจำส่วนนี้มาใช้ พื้นที่ในหน่วยความจำส่วนนี้ เรียกว่า ฮีป (heap)

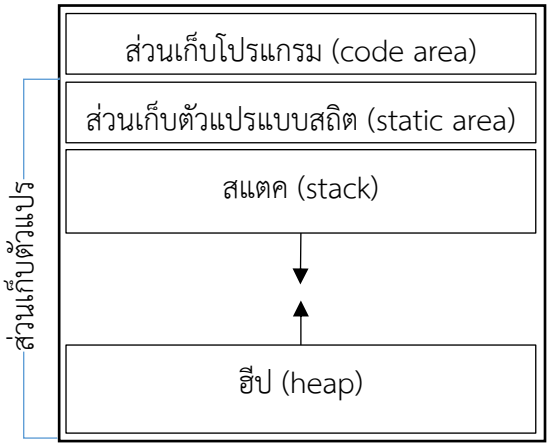
6.1.4 ตัวแปรในฮีปแบบกำหนดโดยปริยาย (implicit heap dynamic variable)

ในภาษาโปรแกรมที่อนุญาตให้ตัวแปรเปลี่ยนชนิดในขณะที่โปรแกรมทำงานได้ เมื่อชนิดของตัวแปรเปลี่ยนไป ขนาดของหน่วยความจำที่ต้องใช้ในการเก็บตัวแปรนั้นอาจเปลี่ยนไปด้วย เช่น เมื่อตัวแปร `x` มีชนิดเป็นจำนวนเต็มด้วยคำสั่ง `x=0` ตัวแปร `x` ใช้เนื้อที่ 4 ไบท์ ต่อมาเมื่อใช้คำสั่ง `x=[1,2,3,4]` `x` เป็นตัวแปรแถวลำดับที่เก็บจำนวนเต็ม 4 ค่าและต้องใช้เนื้อที่ 16 ไบท์ ดังนั้นตัวแปลภาษาต้องจัดหาเนื้อที่ในฮีปให้ใหม่เพื่อเก็บตัวแปร `x` การขอที่ในฮีปของตัวแปรแบบนี้ต่างกับตัวแปรในฮีปแบบกำหนดชัดแจ้ง คือ ผู้เขียนโปรแกรมไม่ได้กำหนดการขอใช้เนื้อที่หน่วยความจำ แต่ตัวแปลภาษาขอใช้เนื้อที่ให้โดยปริยายเมื่อมีการเปลี่ยนชนิดของตัวแปร

หัวข้อต่อไปอธิบายการจัดการหน่วยความจำสำหรับตัวแปรที่ใช้ในขณะที่โปรแกรมทำงาน

6.2 การจัดหน่วยความจำสำหรับตัวแปร

เนื้อที่ในหน่วยความจำที่ใช้ระหว่างการทำงานของโปรแกรมแบ่งเป็นส่วนที่เก็บโปรแกรมและส่วนที่เก็บตัวแปร ตัวแปลภาษาสามารถบอกได้ว่าเนื้อที่ส่วนเก็บโปรแกรมต้องมีขนาดเท่าไรเพราะส่วนนี้คือรหัสเครื่องที่สร้างโดยตัวแปลภาษาเอง แต่ตัวแปลภาษาไม่สามารถบอกได้ว่าหน่วยความจำส่วนที่ใช้เก็บตัวแปรของโปรแกรมต้องมีขนาดเท่าไรเพราะตัวแปรภายในฟังก์ชันจะถูกสร้างขึ้นเมื่อฟังก์ชันถูกเรียกใช้และตัวแปลภาษาไม่อาจบอกได้ก่อนโปรแกรมเริ่มทำงานว่า ฟังก์ชันใดจะถูกเรียกใช้กี่ครั้ง แต่ตัวแปลภาษาบอกได้ว่าตัวแปรแบบสถิตในโปรแกรมต้องการเนื้อที่ในหน่วยความจำเท่าไร ดังนั้นตัวแปลภาษาจะจัดเนื้อที่ในหน่วยความจำสำหรับโปรแกรมหนึ่งดังแสดงในรูปข้างล่างนี้



หน่วยความจำส่วนแรก คือ code area ใช้เก็บโปรแกรมที่จะทำงาน ส่วนถัดมา คือ static area ใช้เก็บตัวแปรแบบสถิต ขนาดของเนื้อที่สำหรับสองส่วนนี้ไม่เปลี่ยนแปลงระหว่างที่โปรแกรมทำงาน ส่วนถัดมาของหน่วยความจำใช้สำหรับตัวแปรในฟังก์ชันที่ถูกเก็บในสแตคซึ่งอาจมีการเพิ่มหรือลดขนาดของสแตคตามการเรียกใช้ฟังก์ชัน ส่วนสุดท้ายเป็นฮีปที่ใช้เก็บตัวแปรที่สร้างระหว่างการทำงานแต่ไม่ได้อยู่ในสแตค เนื่องจากเนื้อที่ของฮีปและสแตคมีการเพิ่มหรือลดระหว่างที่โปรแกรมทำงาน ตัวแปลภาษาจึงจัดให้สแตคและฮีปใช้หน่วยความจำที่ต่อกัน สแตคใช้เนื้อที่เริ่มจากเลขที่อยู่ต่ำและสามารถเพิ่มขนาดไปยังเลขที่อยู่สูงขึ้นได้ ส่วนฮีปใช้เนื้อที่เริ่มจากเลขที่อยู่สูงและสามารถเพิ่มขนาดไปยังเลขที่อยู่ต่ำลง ทั้งนี้ทำให้โปรแกรมใช้เนื้อที่สำหรับสแตคและฮีปร่วมกัน เช่น สแตคอาจใช้หน่วยความจำส่วนนี้ทั้งหมดในช่วงที่โปรแกรมมีการเรียกใช้ฟังก์ชันเวียนเกิดหลายครั้ง แต่เมื่อฟังก์ชันทำงานจบไปแล้วเนื้อที่ในสแตคที่ใช้สำหรับตัวแปรในฟังก์ชันจะถูกคืนกลับไป โปรแกรมอาจขอเนื้อที่สำหรับตัวแปรในฮีปแบบกำหนดชัดแจ้งที่เป็นตัวแปรแถวลำดับขนาดใหญ่โดยขอหน่วยความจำจากฮีปซึ่งอาจเป็นส่วนที่เคยใช้เก็บตัวแปรของฟังก์ชันมาก่อน นั่นคือหน่วยความจำส่วนหนึ่งอาจถูกใช้สำหรับสแตคในเวลาหนึ่งแต่ใช้สำหรับฮีปในอีกเวลาหนึ่งก็ได้

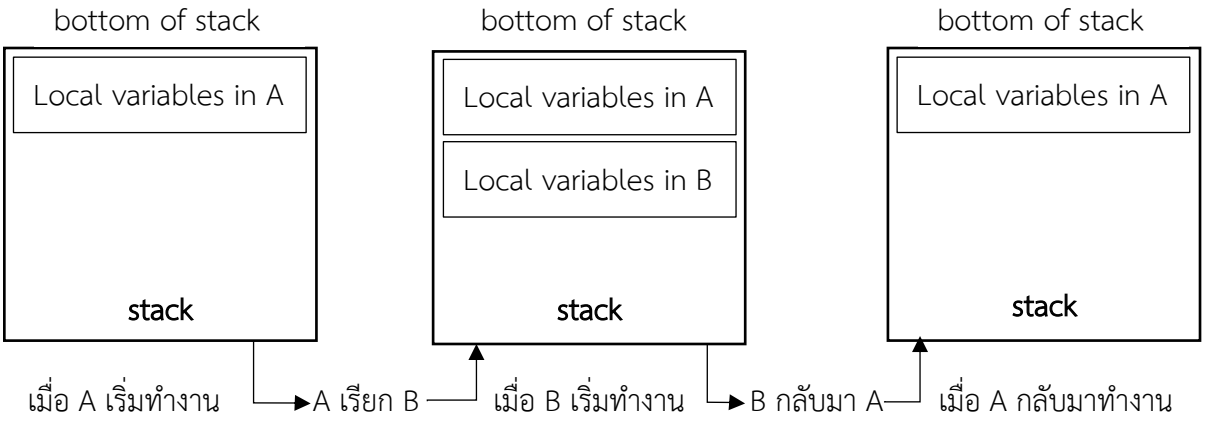
6.2.1 การจัดเก็บตัวแปรแบบสถิต

เนื่องจากตัวแปรแบบสถิตมีช่วงชีวิตตลอดการทำงานของโปรแกรม ตัวแปลภาษาจัดหน่วยความจำในเนื้อที่แบบสถิต (static area) และ กำหนดเลขที่อยู่ให้ตัวแปรแบบสถิตแต่ละตัวได้ตั้งแต่โปรแกรมเริ่มทำงาน

จากนั้นตัวแปลภาษาสามารถใช้เลขที่อยู่นี้อ้างถึงตัวแปรแบบสถิตในรหัสเครื่องตลอดการทำงานของโปรแกรม ดังนั้นตัวแปลภาษาจะเก็บเลขที่อยู่นี้ไว้ในตารางสัญลักษณ์และใช้ในการสร้างรหัสเครื่อง

6.2.2 การจัดเก็บตัวแปรในสแตคแบบพลวัต

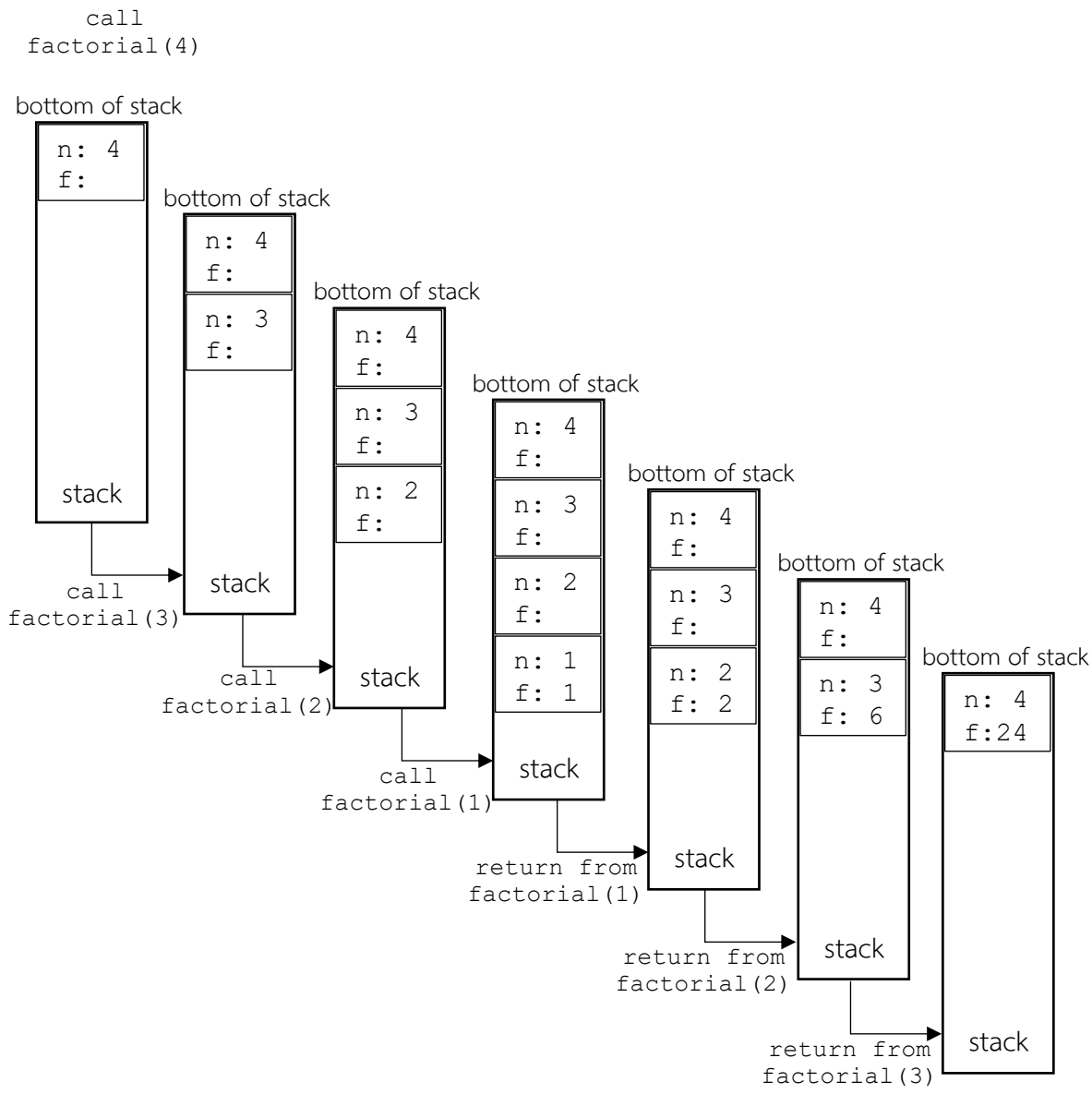
ตัวแปรเฉพาะที่ในฟังก์ชันถูกสร้างขึ้นเมื่อมีการเรียกใช้ฟังก์ชันและถูกเก็บไว้จนกระทั่งฟังก์ชันทำงานจบ ดังนั้นตัวแปรเหล่านี้ถูกเก็บไว้ในสแตค สมมติให้มีฟังก์ชัน A และ B ซึ่งฟังก์ชัน A เรียกใช้ฟังก์ชัน B เราเรียก A เป็นตัวเรียก (caller) และ B เป็นตัวถูกเรียก (callee) เมื่อฟังก์ชัน A ทำงาน ตัวแปรเฉพาะที่ของฟังก์ชัน A จะถูกเก็บไว้ในสแตค เมื่อฟังก์ชัน A เรียกใช้ฟังก์ชัน B ตัวแปรเฉพาะที่ของฟังก์ชัน A จะยังถูกเก็บไว้และตัวแปรของฟังก์ชัน B จะถูกสร้างเก็บไว้ด้านบนของสแตคดังแสดงนรูปข้างล่าง เมื่อฟังก์ชัน B จบการทำงานและย้อนกลับมาทำงานของฟังก์ชัน A ต่อ ตัวแปรเฉพาะที่ของฟังก์ชัน B ที่อยู่บนสุดในสแตคจะถูกป๊อบ (pop) ทิ้งและทำให้ตัวแปรเฉพาะที่ของฟังก์ชัน A กลับมาอยู่บนสุดในสแตคดังแสดงในรูปข้างล่าง พารามิเตอร์ (parameter) ของฟังก์ชันเป็นเหมือนตัวแปรเฉพาะที่ซึ่งต้องมีค่าส่งเข้ามาหรือนำค่าส่งออกไป ดังนั้นจึงต้องเก็บในสแตคในลักษณะเดียวกัน



ในการเรียกฟังก์ชันแบบเวียนเกิด (recursive function) ซ้ำ ตัวแปรเฉพาะที่ของฟังก์ชันนั้นต้องถูกสร้างขึ้นใหม่ทุกครั้งด้วย ตัวอย่าง เช่น ฟังก์ชัน `factorial` ข้างล่างนี้มีตัวแปรเฉพาะที่ `f`

```
int factorial(int n) {
   int f;
   if (n==0 or n==1) f=n;
   else f=n*factorial(n-1);
   return f;
}
```

หากโปรแกรมหลักเรียกฟังก์ชัน `factorial(4)` ตัวแปร `f` และพารามิเตอร์ `n` ของฟังก์ชัน `factorial` ต้องถูกเก็บไว้สำหรับการเรียกแบบเวียนเกิดทุกครั้งดังแสดงในรูปข้างล่างนี้



จากตัวอย่างที่กล่าวมานี้จะเห็นว่าเมื่อตัวแปลภาษาอ้างถึงตัวแปรเฉพาะที่ในการเรียกฟังก์ชันแต่ละครั้ง เลขที่อยู่ของตัวแปรเฉพาะที่นั้นจะต่างกัน เพราะอ้างถึงเนื้อที่ในสแตคที่สร้างขึ้นในการเรียกฟังก์ชันแต่ละครั้ง

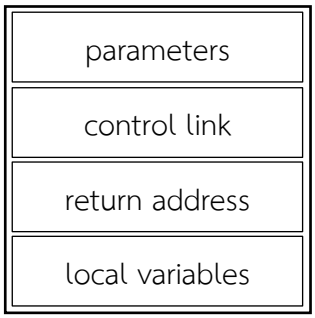
6.2.3 การจัดเก็บตัวแปรในฮีป

เมื่อโปรแกรมขอเนื้อที่ในฮีปทั้งแบบชัดแจ้งและโดยปริยาย ตัวแปลภาษาต้องตรวจสอบว่ามีเนื้อที่ว่างในฮีปหรือไม่และจัดเนื้อที่ให้หากมี ตัวแปลภาษาจะหาหน่วยความจำในส่วนฮีปที่ว่างและใช้เลขที่อยู่นี้สำหรับตัวแปรนั้น ดังนั้นตัวแปรนี้จะมีเลขที่อยู่จริงเมื่อโปรแกรมทำงานไปแล้ว

6.3 คำสั่งที่ใช้จัดการสแตคการเรียกและเลขที่อยู่ของตัวแปรเฉพาะที่

ในการเรียกใช้ฟังก์ชัน ตัวแปลภาษาต้องเตรียมเนื้อที่สำหรับตัวแปรเฉพาะที่และพารามิเตอร์ของฟังก์ชันดังที่กล่าวไปแล้ว นอกจากนั้นตัวแปลภาษายังต้องจัดให้เก็บเลขที่อยู่หลังคำสั่งเรียกใช้ฟังก์ชันที่เรียกว่า

เลขที่อยู่กลับ (return address) เพื่อให้สามารถกระโดดกลับมาทำงานที่คำสั่งนั้นเมื่อฟังก์ชันนั้นทำงานจบ ตัวแปลภาษาจะเก็บตัวชี้ที่เรียกว่า ลิงค์ควบคุม (control link) ที่ชี้ไปยังระเบียนการทำงานของฟังก์ชันที่เป็นตัวเรียก (caller) ด้วย ข้อมูลเกี่ยวกับฟังก์ชันที่กล่าวมานี้รวมเรียกว่า ระเบียนการทำงาน (activation record) หรือ เฟรม (frame) ดังแสดงในรูปข้างล่าง



ระเบียนการทำงาน (activation record)

6.3.1 คำสั่งจัดการเรียกฟังก์ชันและกลับจากฟังก์ชัน

เมื่อโปรแกรมเรียกใช้ฟังก์ชัน จะต้องมีการเก็บระเบียนการทำงานในสแตค เรียกว่า สแตคการเรียก (call stack) ตัวแปลภาษาสร้างคำสั่งที่เก็บระเบียนการทำงานในสแตคการเรียกโดยใช้ตัวชี้ของเฟรม FP (frame pointer) เพื่อบอกตำแหน่งของระเบียนการทำงานและตัวชี้ของสแตค SP (stack pointer) เพื่อบอกตำแหน่งบนสุดของสแตคการเรียก คำสั่งชุดนี้เรียกว่าลำดับการเรียก (calling sequence) ซึ่งประกอบด้วยการทำงานต่อไปนี้

1. จองที่ในสแตคสำหรับพารามิเตอร์ที่ส่งคืน (return parameter) และนำค่าของพารามิเตอร์ที่ส่งเข้ามาเก็บในสแตคการเรียก แล้วเลื่อน SP
2. เก็บลิงค์ควบคุมซึ่งได้จาก FP ที่ชี้ไปยังระเบียนการทำงานของฟังก์ชันเดิมลงในสแตค แล้วเลื่อน SP
3. เลื่อน FP มาชี้ที่เดียวกับ SP
4. เก็บเลขที่อยู่กลับลงในสแตคการเรียก แล้วเลื่อน SP
5. จองที่ในสแตคสำหรับตัวแปรเฉพาะที่ แล้วเลื่อน SP
6. กระโดดไปทำงานที่ฟังก์ชันที่ถูกเรียก

ตัวแปลภาษาต้องเพิ่มคำสั่งที่ทำงานตามลำดับการเรียกเมื่อมีการเรียกฟังก์ชัน ตัวอย่างต่อไปนี้แสดงคำสั่งที่จัดการลำดับการเรียกใน 3-address code

ตัวอย่าง พิจารณาลำดับการเรียกของ `factorial(x)` ที่แสดงในโปรแกรมด้วยตัวหนาในโปรแกรมภาษาซีต่อไปนี้

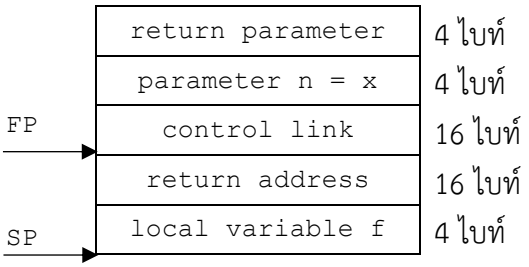
```
int factorial(int n){
    int f;
    if (n==0) f=1;
    else      f=factorial(n-1)*n;
    return f;
}
```

```
int main(void) {
    int x, y;
    x=3;
    …
    y=factorial(x);
}
```

ฟังก์ชัน `factorial` มี `n` เป็น input parameter และส่งคืน return parameter ที่กำหนดเป็น `int` ในหัวฟังก์ชัน ตัวแปลภาษาเพิ่มคำสั่งของ 3-address code เพื่อจัดการเรียกฟังก์ชัน `factorial(x)` ดังนี้

```
T  = SP+4        // make space for return parameter
SP = T           // update SP
*SP= x           // push x as parameter n
T  = SP+4        // size of integer is 4
SP = T           // update SP
*SP= FP          // push control link
T  = SP+16       // size of address is 16
SP = T           // update SP
FP = SP          // update FP
*SP= RTA         // push return address
T  = SP+16       // size of address is 16
SP = T           // update SP
T  = SP+4        // make space for local variable f
SP = T           // update SP
goto factorial   // jump to function
label RTA        // set return address
```

คำสั่งชุดนี้ทำให้ได้ระเบียนการทำงานข้างล่างนี้อยู่ในสแตคการเรียก



เมื่อการทำงานของฟังก์ชันจบลงและมีคำสั่ง return เพื่อให้ย้อนกลับมาทำงานต่อจากคำสั่งเรียกใช้ฟังก์ชันนั้น ตัวแปลภาษาต้องสร้างคำสั่งที่ทำให้ย้อนกลับมาทำงานต่อจากคำสั่งเรียกฟังก์ชัน พร้อมทั้งส่งพารามิเตอร์กลับ และคืนค่าตัวแปรเฉพาะที่ของฟังก์ชันนั้นกลับมา คำสั่งชุดนี้เรียกว่าลำดับการกลับ (return sequence) ซึ่งประกอบด้วยการทำงานต่อไปนี้

1. เก็บเลขที่อยู่กลับจากระเบียนการทำงานเพื่อกระโดดไปทำงานที่ฟังก์ชันที่เรียกตาม
2. เลื่อน SP ไปชี้ที่เดียวกับ FP (ป๊อปตัวแปรเฉพาะที่และเลขที่อยู่กลับทิ้ง)
3. เลื่อน FP ไปชี้ที่ระเบียนการทำงานของฟังก์ชันเดิม โดยนำลิงค์ควบคุมไปเก็บใน FP
4. เลื่อน SP ไปชี้ที่พารามิเตอร์ที่ส่งคืน
5. กระโดดไปทำงานคำสั่งที่เลขที่อยู่กลับ

เมื่อฟังก์ชันทำงานจบและกลับมาทำงานหลังคำสั่งเรียกฟังก์ชัน ตัวแปลภาษาต้องเพิ่มคำสั่งที่ทำงานตามลำดับการกลับ ตัวอย่างต่อไปนี้แสดงคำสั่งที่จัดการลำดับการกลับ

ตัวอย่าง พิจารณาลำดับการกลับของ `factorial(x)` ที่แสดงในโปรแกรมในตัวอย่างที่แล้ว ตัวแปลภาษาเพิ่มคำสั่งของ 3-address code เพื่อจัดการกลับจากฟังก์ชัน `factorial` ดังนี้

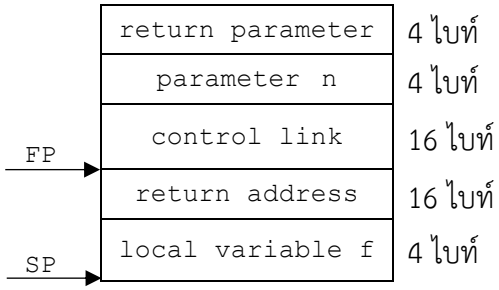
```
RET= *FP          // save return address
SP = FP           // pop local variable f and return address
T  = SP-16        // move SP down to control link
SP = T
FP = *SP          // move FP down to the caller's activation record
T  = SP-4         // pop input parameter
SP = T
T  = SP-4         // move SP down to return parameter
SP = T
goto RET          // jump back
```

6.3.2 การคำนวณเลขที่อยู่ของตัวแปรในฟังก์ชัน

สำหรับตัวแปรเฉพาะที่แต่ละตัว ตัวแปลภาษาจะเก็บตำแหน่งของตัวแปรในระเบียนการทำงานไว้ในตารางสัญลักษณ์เพื่อใช้หาเลขที่อยู่ของตัวแปรนั้น เมื่อต้องการอ้างถึงตัวแปรเฉพาะที่ในฟังก์ชันที่กำลังทำงานอยู่ ตัวแปลภาษาคำนวณเลขที่อยู่ของตัวแปรนั้นโดยนำค่าของตัวชี้ของระเบียนการทำงาน FP บวกกับตำแหน่งของตัวแปรเฉพาะที่จากตารางสัญลักษณ์ ตัวอย่างต่อไปนี้แสดงการคำนวณหาเลขที่อยู่ของตัวแปรเฉพาะที่ในฟังก์ชัน ตัวอย่างต่อไปนี้แสดงการคำนวณเลขที่อยู่ของตัวแปรในฟังก์ชัน

ตัวอย่าง พิจารณาฟังก์ชัน `factorial(x)` และระเบียนการทำงานของฟังก์ชัน `factorial` ข้างล่างนี้

```
int factorial(int n){
    int f;
    if (n==0) f=1;
    else      f=factorial(n-1)*n;
    return f;
}
```



เมื่อฟังก์ชัน `factorial` ทำงาน ระเบียนการทำงานของฟังก์ชัน `factorial` จะอยู่บนสุดของสแตค ดังนั้นเราสามารถคำนวณเลขที่อยู่ของค่าในระเบียนการทำงานได้ดังนี้

- เลขที่อยู่ของพารามิเตอร์ `n` คือ `FP-20`
- เลขที่อยู่ของพารามิเตอร์ที่ส่งกลับ คือ `FP-24`
- เลขที่อยู่ของตัวแปรเฉพาะที่ `f` คือ `FP+16`

ดังนั้น โปรแกรม 3-address code ของฟังก์ชัน factorial เป็นดังนี้

```
label factorial
fAddr = FP+16      // find the address of local variable f
nAddr = FP-16      // find the address of parameter n
c = *nAddr==0
ifFalse c goto Flabel    // if (n==0)
*fAddr = 1               // f=1
goto OUT
label Flabel             // else
p = *nAddr -1            // calculate n-1
// ---- call sequence
T  = SP+4
SP = T
*SP= p                   // set parameter as n-1
T  = SP+4
SP = T
*SP= FP                  // save control link
T  = SP+16
SP = T
FP = SP
*SP= RTA                 // save return address
T  = SP+16
SP = T
T  = SP+4
SP = T
// ---- call factorial
goto factorial
label RTA                // set return address
par = *SP                // get return parameter
N  = *nAddr
*fAddr = par*N           // f=factorial(n-1)*n
Label OUT
T  = FP-20               // return f; store f as return parameter
*T =*fAddr
// ---- return sequence
RET= *FP           // get return address
SP = FP            // pop local variables and return address
T  = SP-16
SP = T
FP = *SP           // move FP to caller's activation record
T  = SP-4
SP = T             // pop input parameter
T  = SP-4
SP = T             // move SP to return parameter
goto RET
```

จากตัวอย่างนี้จะเห็นได้ว่าการอ้างถึงตัวแปรเฉพาะที่ในฟังก์ชันมีขั้นตอนการคำนวนเลขที่อยู่มากกว่าการอ้างถึงตัวแปรแบบสถิต

7. สรุป

การแปลภาษาโปรแกรมแบ่งเป็นหลายขั้นตอน ในขั้นแรก สแกนเนอร์แบ่งสายของตัวอักขระเป็นโทเคนซึ่งเป็นหน่วยเล็กสุดที่มีความหมายในภาษาโปรแกรม จากนั้นตัวแจงส่วนนำสายของโทเคนมาตรวจสอบความสัมพันธ์ทางไวยากรณ์ที่กำหนดของภาษาและสร้างต้นไม้แจงส่วนที่แสดงความสัมพันธ์ทางไวยากรณ์ของโทเคนทั้งหมด ต่อมาตัววิเคราะห์ความหมายจึงตรวจสอบความถูกต้องทางความหมายซึ่งมักเกี่ยวข้องกับชนิดของข้อมูล ขั้นตอนทั้ง 3 นี้เรียกว่าเป็นส่วนหน้า (front-end) ของตัวแปลภาษาและทำหน้าที่ตรวจสอบความถูกต้องของโปรแกรมและสร้างโครงสร้างที่แสดงความสัมพันธ์ของโทเคน

ส่วนหลัง (back-end) ของตัวแปลภาษาทำหน้าที่สร้างโปรแกรมในภาษาที่ต้องการจากโครงสร้างที่ได้จากส่วนหน้า ส่วนหลังของตัวแปลภาษาอาจเป็นตัวก่อกำเนิดรหัสที่ทำหน้าที่สร้างโปรแกรมในภาษาที่ต้องการจากโครงสร้างที่ได้จากส่วนหน้า แต่การทำงานของส่วนหลังอาจสร้างโปรแกรมที่เป็นรหัสกลางก่อนแล้วจึงแปลเป็นภาษาเครื่องที่ต้องการ

นอกจากนั้นตัวแปลภาษาอาจมีตัวปรับประสิทธิภาพของรหัส (code optimization) ที่ปรับโปรแกรมให้ทำงานเร็วขึ้นซึ่งไม่ได้กล่าวถึงในที่นี้ การปรับประสิทธิภาพนี้อาจทำบนต้นไม้แจงส่วน รหัสกลาง หรือ รหัสเครื่อง ก็ได้