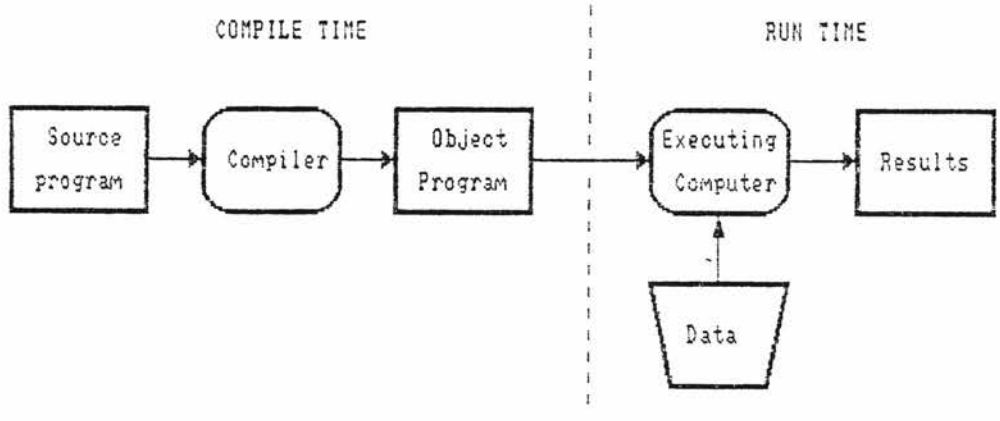
บทที่ 2

แนวความคิดพื้นฐานของตัวแปลโปรแกรมกับลักษณะภาษาโคบอล

ในบทนี้ จะนำเสนอพื้นฐานของเรื่องแนวความคิดทั่วไปในการสร้างตัวแปลโปรแกรม รวมทั้งส่วนประกอบต่าง ๆ ของตัวแปลโปรแกรม และลักษณะโดยทั่วไปของภาษาโคบอล โดยจะแบ่งออกเป็น 2 ส่วนใหญ่ ๆ คือ ส่วนแรกเป็นเรื่องแนวความคิดพื้นฐานของตัวแปลโปรแกรม และส่วนที่ 2 จะกล่าวถึงลักษณะของภาษาโคบอล

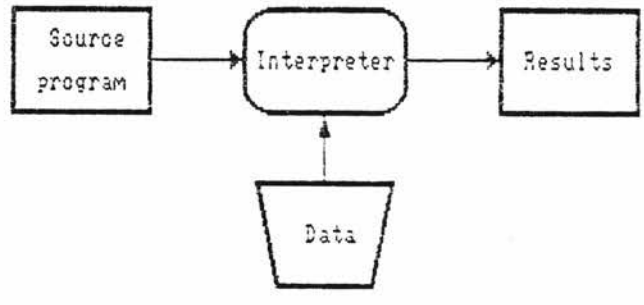
แนวความคิดพื้นฐานของตัวแปลโปรแกรม

อาจจะกล่าวกว้าง ๆ ได้ว่า ตัวแปลก็คือ ชุดโปรแกรมที่รับเอาโปรแกรมดิบ (Source program) เข้าไปแล้วทำการแปลเป็นโปรแกรมผล (Object program หรือ Target program)[1] ซึ่งถ้าหากโปรแกรมดิบเป็นภาษาแอสเซมบลี และโปรแกรมผลที่ได้รับเป็นภาษาเครื่อง ก็จะเรียกตัวแปลโปรแกรมนั้นว่า แอสเซมเบลอร์ ส่วนตัวแปลที่ทำหน้าที่แปลภาษาระดับสูง เช่น ภาษาฟอร์แทรน โคบอล ปาสคาล ฯลฯ ให้เป็นภาษาเครื่อง หรือแปลเป็นภาษาแอสเซมบลี สำหรับเครื่องคอมพิวเตอร์แต่ละเครื่องนั้น ก็จะเรียกตัวแปลชนิดนั้นว่า คอมไพเลอร์ (Compiler) ซึ่งในวิทยานิพนธ์ฉบับนี้ เมื่อกล่าวถึงตัวแปลโปรแกรม ก็จะหมายความถึง คอมไพเลอร์ นั่นเอง



รูปที่ 2-1 กระบวนการแปลโปรแกรม ของตัวแปลโปรแกรม
(Compilation Process)

ยังมีตัวแปลอีกประเภทหนึ่ง เรียกว่าตัวแปลภาษา (Interpreter) ซึ่งจะทำการแปลโปรแกรมดิบ และอ่านข้อมูลเข้ามาทำการประมวลผลในเวลาเดียวกัน ดังนั้นการแปลความหมายของโปรแกรมดิบก็จะเกิดขึ้นในช่วงเวลาการประมวลผล (Run Time) เลยทีเดียว โดยไม่มีการสร้างโปรแกรมผลเก็บไว้ก่อนการประมวลผล ดังแสดงในรูปที่ 2-2 ซึ่งตัวแปลภาษาบางอย่าง จะวิเคราะห์ประโยคคำสั่งของโปรแกรมดิบทุกครั้งที่เกิดการประมวลผล ทำให้ใช้เวลาในการประมวลผลมาก ซึ่งอาจจะมีทางแก้ไขให้ดีขึ้น โดยการแปลโปรแกรมดิบให้เป็นรหัสกลางชั่วคราว (Intermediate Code) ก่อน แล้วจึงมีตัวแปลภาษาอีกตัวหนึ่ง มาทำการแปลรหัสกลางนั้น ในแต่ละรอบของการประมวลผล ก็จะสามารถประหยัดเวลา ที่ใช้ในการประมวลผลได้ดีขึ้น

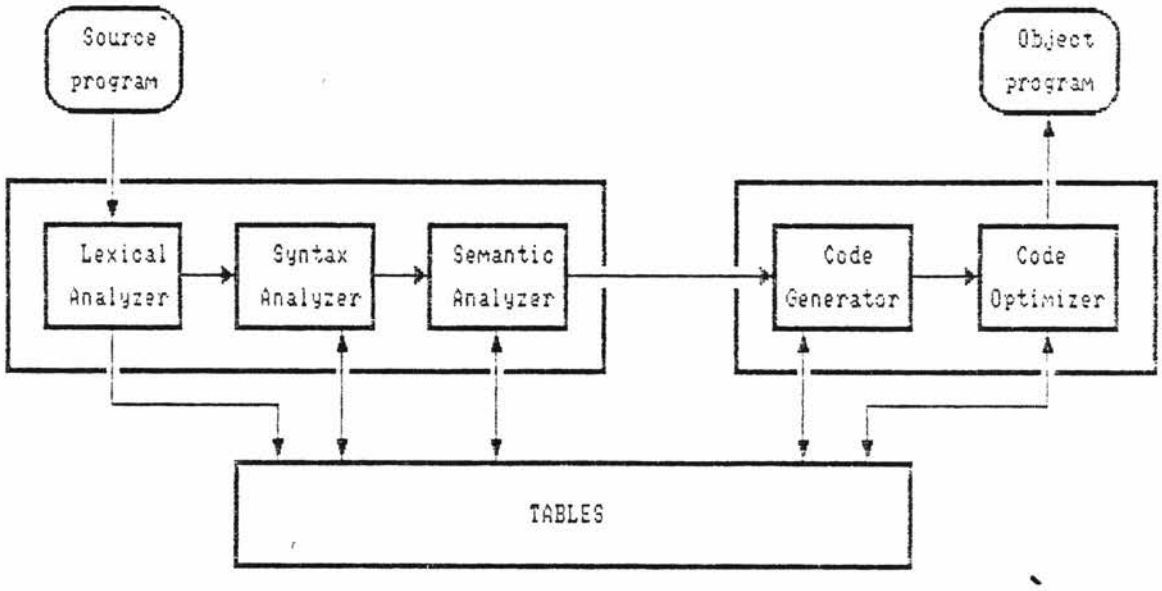


รูปที่ 2-2 กระบวนการแปลโปรแกรม ของตัวแปลภาษา
(Interpretation Process)

ในการใช้ไมโครคอมพิวเตอร์โดยทั่วไปแล้ว จะพบว่าตัวแปลภาษาเป็นที่นิยมใช้กันอย่างกว้างขวางมากกว่าตัวแปลโปรแกรม เหตุผลอย่างหนึ่งก็คือ การใช้ตัวแปลภาษาสามารถตอบโต้กับผู้ใช้ได้ทันทีในการออกคำสั่งแต่ละคำสั่ง โดยไม่จำเป็นต้องสร้างชุดคำสั่งให้ครบแล้วจึงแปลโดยตัวแปลโปรแกรม ซึ่งต้องมีขั้นตอนหลายขั้นตอน และค่อนข้างยุ่งยากสำหรับผู้ที่ไม่ชำนาญหรือไม่คุ้นเคย จะเห็นได้ว่าภาษาที่ได้รับความนิยม เช่น ภาษาเบสิก, เอ พี แอล (APL) มักจะถูกสร้างขึ้นในลักษณะของตัวแปลภาษาทั้งสิ้น

รูปแบบทั่วไปของตัวแปลโปรแกรม

ความยุ่งยากในการสร้างตัวแปลโปรแกรม ขึ้นอยู่กับลักษณะของภาษาระดับสูงที่จะแปลเป็นภาษาเครื่อง จะเห็นว่าภาษาระดับสูงที่ถูกออกแบบและสร้างขึ้นในระยะหลัง จะเอื้ออำนวยต่อการสร้างตัวแปลโปรแกรมได้โดยไม่ยากมากนัก เช่น ภาษาปาสคาล, ภาษาอัลกอ เป็นต้น ต่างกับภาษาระดับสูงในรุ่นแรก ๆ เช่น ภาษาโคบอล, ภาษาฟอร์แทรน ซึ่งการสร้างตัวแปลโปรแกรมเพื่อแปลโปรแกรมภาษาระดับสูงเหล่านี้ค่อนข้างยุ่งยาก แต่อย่างไรก็ดี รูปแบบพื้นฐานของตัวแปลโปรแกรมสามารถแสดงได้ ดังรูป 2-3



รูปที่ 2-3 ส่วนประกอบของตัวแปลโปรแกรม

จากรูปที่ 2-3 จะเห็นได้ว่าตัวแปลโปรแกรมจะทำหน้าที่หลักที่สำคัญ 2 ประการ คือ

1. วิเคราะห์โปรแกรมดิบ
2. สังเคราะห์โปรแกรมผลจากการวิเคราะห์โปรแกรมดิบ

หลักการง่าย ๆ ของการวิเคราะห์โปรแกรมดิบ คือการแยกย่อยส่วนของโปรแกรมภาษาระดับสูง (โปรแกรมดิบ) ออกเป็นส่วนประกอบย่อยเบื้องต้นแล้วจึงนำส่วนย่อย ๆ เหล่านี้ไปใช้ในการสังเคราะห์โปรแกรมผลขึ้นมา ซึ่งในการวิเคราะห์โปรแกรมดิบและการสังเคราะห์

โปรแกรมผลนี้ มีเครื่องมือที่ช่วยมากที่สุด ก็คือการใช้ตารางต่าง ๆ เพื่อเก็บค่าที่เกิดขึ้น และจำเป็นต้องใช้ในระหว่างการวิเคราะห์และการสังเคราะห์โปรแกรมดังกล่าว

ถ้าพิจารณาในส่วนของโปรแกรมดิบ จะพบว่าโปรแกรมดิบ จะประกอบกันขึ้นมาจากสายวลี (String) ของสัญญลักษณ์ต่าง ๆ ได้แก่ ตัวอักษร, ตัวเลข, อักขระพิเศษ ซึ่งส่วนย่อยเหล่านี้ จะประกอบกันขึ้นเป็นองค์ประกอบของโปรแกรม ได้แก่ นิพจน์ทางคณิตศาสตร์, ตัวแปร, ค่าคงที่, คำสงวนต่าง ๆ (ซึ่งคำสงวนในภาษาโคบอลจะมีความหมายเฉพาะ เช่น เป็นคำสั่ง ในการเตรียมเนื้อที่ในหน่วยความจำ, เป็นคำสั่งที่กำหนดค่าเริ่มต้นของตัวแปร และอื่น ๆ) กระบวนการแยกสายวลีของสัญญลักษณ์เหล่านี้ให้เป็นหน่วยย่อยเพื่อนำไปสังเคราะห์ให้เป็น โปรแกรมผล เรียกว่า กระบวนการวิเคราะห์เลกซิเคิล (Lexical Analysis)

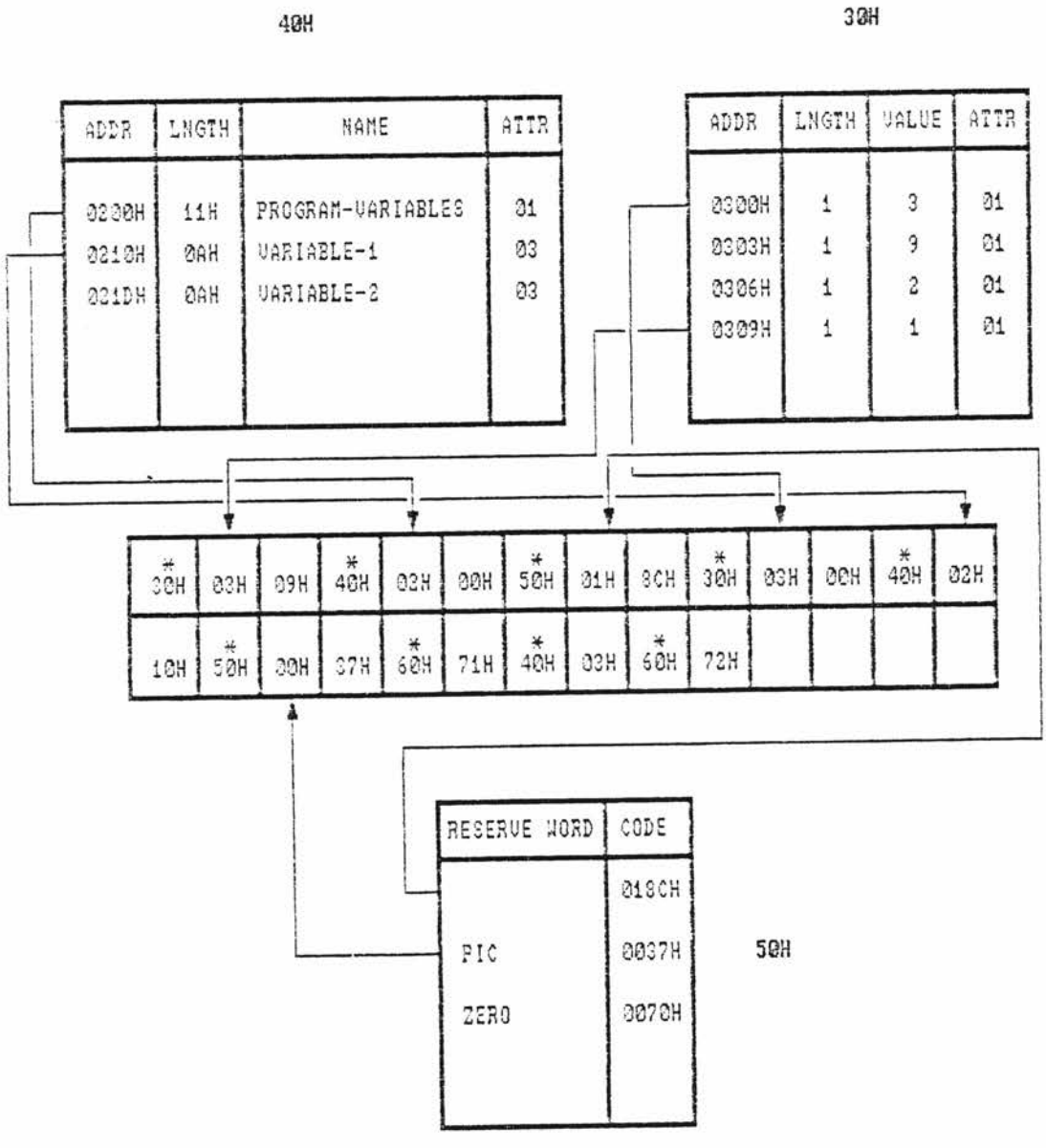
โปรแกรมดิบจะเป็นข้อมูลนำเข้าสู่ตัววิเคราะห์เลกซิเคิล เพื่อแยกย่อยเป็น โทเกน (Token) อันได้แก่ ค่าคงที่, ตัวแปร, คำสงวน, เครื่องหมายคณิตศาสตร์, เครื่องหมายวรรคตอน เป็นต้น ซึ่งโทเกน ดังกล่าวอาจจะถูกแปรเปลี่ยนให้เป็นรูปแบบที่ง่ายต่อการทำงานในขั้นตอนต่อไปของตัวแปลก็ได้โดยการเปลี่ยนเป็นรหัสแทนข้อมูล ดังตัวอย่างต่อไปนี้

จากคำสั่งภาษาโคบอล

```
01  PROGRAM-VARIABLES.
    03  VARIABLE-1          PIC X(03) VALUE SPACE.
    03  VARIABLE-2          PIC 9(02) VALUE ZERO.
```

อาจจะถูกนำเข้าเก็บในตารางต่าง ๆ ดังรูป 2-4

การเก็บเป็นรหัสต่าง ๆ เหล่านี้ เพื่อให้ง่ายแก่การใช้ในขั้นตรวจสอบและวิเคราะห์ความถูกต้องของไวยากรณ์ภาษา (Syntactical Analysis) ซึ่งทำหน้าที่โดยส่วนของตัวแปลโปรแกรมที่เรียกว่า พาสเซอร์ (Parser)



* - เป็นสัญญลักษณ์บอกชนิดของ โทเกน ที่ปรากฏในตารางเก็บ โทเกน

40H - จากตารางเก็บสัญญลักษณ์ (Symbol table)

30H - จากตารางเก็บค่าคงที่ (Literal table)

50H - จากตารางเก็บคำสงวน (Reserved word table)

60H - จากการลงรหัสในโปรแกรม (Hard-coding)

รูปที่ 2-4 แสดงการเก็บค่าต่าง ๆ เข้าสู่ตาราง

อย่างไรก็ดี ในขั้นตอนการเปลี่ยนรูปร่างของ โทเกน ให้กลายเป็นรหัสแทนข้อมูลดังกล่าวนั้น จะกระทำในตัววิเคราะห์เลกซิเคิล หรือ พาสเซอร์ ก็ได้แล้วแต่การออกแบบ แต่มีข้อพึงสังวรอยู่ว่าถ้าทำงานหนักในส่วนตัววิเคราะห์เลกซิเคิล แล้ว พาสเซอร์ ก็อาจจะสร้างง่ายขึ้น แต่มีบางกรณีอาจจะเป็นไปในทางกลับกัน ทั้งนี้ขึ้นอยู่กับลักษณะของภาษานั้น ๆ

หน้าที่หลักโดยทั่วไปของ พาสเซอร์ และประเภทของ พาสเซอร์

ในส่วนของการวิเคราะห์โปรแกรมดิบที่สำคัญอีกส่วนหนึ่ง ก็คือ การตรวจสอบและวิเคราะห์ความถูกต้องของโปรแกรมที่เขียนขึ้นตามหลักไวยากรณ์ของภาษา (Syntactical Analysis) ซึ่งหน้าที่ดังกล่าวนั้น กระทำโดยส่วนของตัวแปลโปรแกรม หรือตัวแปลภาษาในส่วนที่เรียกว่าพาสเซอร์

โดยปรกติแล้ว ในตัวแปลโปรแกรมหรือตัวแปลภาษาโดยทั่วไป พาสเซอร์ จะทำงานต่อจากส่วนแรกคือตัววิเคราะห์เลกซิเคิล ดังนั้นส่วนนำเข้า (Input) ของ พาสเซอร์ ก็คือผลลัพธ์ที่ได้จากการทำการวิเคราะห์เลกซิเคิล อันหมายถึง โทเกน นั่นเองและเมื่อ พาสเซอร์นำ โทเกน นั้นมาผ่านกระบวนการตรวจสอบความถูกต้องของไวยากรณ์แล้ว ผลลัพธ์ที่ได้อาจอยู่ในรูปต่าง ๆ แต่ที่นิยมกันมากจะสร้างในแบบที่เรียกว่า พาสทรี (Parse Tree)[2]

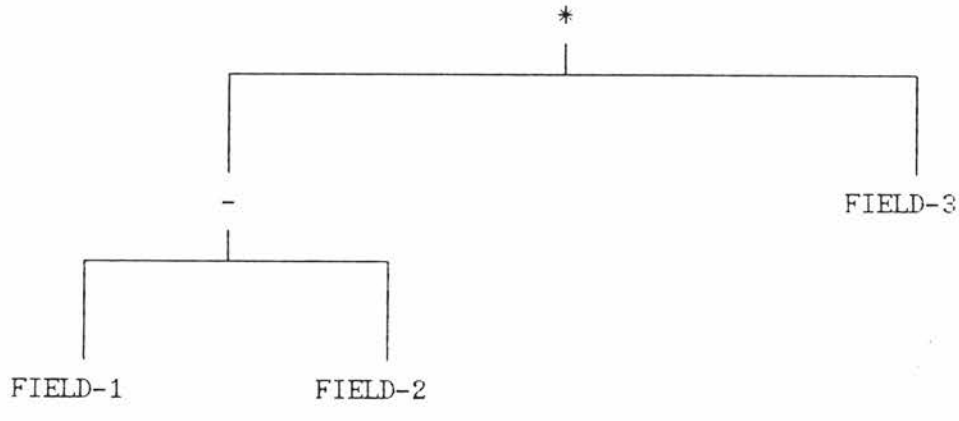
พาสเซอร์ นั้นมีการจำแนกออกเป็นประเภทใหญ่ ๆ ได้ 2 ประเภท คือ

1. Operator Precedence
2. Recursive Descent

Operator Precedence ใช้หลักการตรวจสอบลำดับความสำคัญของเครื่องหมายต่าง ๆ หรือตรวจสอบลำดับความสำคัญของ โทเกน ที่ประกอบกันเป็นประโยคคำสั่งนั่นเอง ในวิธีการแบบนี้ เราจะพบเห็นการใช้งานในการตรวจพวกนิพจน์ทางคณิตศาสตร์ (Arithmetic Expression) มากกว่าใช้ตรวจคำสั่งธรรมดาทั่ว ๆ ไป ตัวอย่างการพาส ของนิพจน์ทางคณิตศาสตร์ แสดงในรูปที่ 2-5

Recursive Descent เป็นวิธีการที่เกิดขึ้นด้วยการสร้างส่วนการทำงานที่สามารถเรียกใช้ตัวเอง (Recursive) เพื่อตรวจสอบว่าในคำสั่งนั้น ๆ พบกับชนิดของโทเกน

```
COMPUTE RESULT-FIELD = (FIELD-1 - FIELD-2) * FIELD-3.
```



รูปที่ 2-5 แสดงการสร้างพาสทรีจากนิพจน์ทางคณิตศาสตร์

ที่คาดหวังไว้หรือไม่ นิยมใช้ หลักการแบบนี้ เพื่อตรวจสอบประโยคคำสั่งโดยทั่ว ๆ ไป นอกเหนือไปจากพวกนิพจน์ทาง คณิตศาสตร์

เราจะพบว่าตัวแปลโปรแกรมที่ถูกสร้างขึ้นในสมัยเก่า ๆ (ราว ๆ ทศวรรษที่ 1970) ทั้ง 2 วิธีการดังกล่าวจะถูกใช้เพื่อการตรวจสอบไวยากรณ์ของโปรแกรม เนื่องจากเป็นวิธีการที่ค่อนข้างง่ายต่อการสร้าง แต่มีข้อบกพร่องหลายประการ ดังนั้นจึงมีการประยุกต์จาก 2 หลักการดังกล่าวข้างต้นนั้น มาเป็นวิธีการใหม่ 2 วิธี ได้แก่

1. LL Parsing
2. LR Parsing

LL Parsing เป็นวิธีการสร้างผลลัพธ์จาก พาสเซอร์ เป็น พาสทรี ในลักษณะการสร้างแบบบนลงล่าง (Top Down Parsing) หมายความว่าส่วนของราก (Root) จะถูกสร้างก่อน แล้วจึงสร้างโยง เชื่อมเข้ากับส่วนของลูก (Leaf)

LR Parsing คือการสร้าง พาสทรี แบบล่างขึ้นบน (Bottom up) ใช้แนวความคิดของโครงสร้างข้อมูลแบบแสตก (Stack) โดยลดในส่วนที่สามารถแทนที่กันได้ตามกฎเกณฑ์ที่กำหนดขึ้นในตอนต้น สร้างส่วนของลูก (Leaf) แล้วลูกอาจถูกแทนที่ด้วยกฎเกณฑ์ดังกล่าวลดรูปให้เล็กลง แล้วจึงสามารถนำมาเชื่อมโยงเข้ากับส่วนของราก กลายเป็น พาสทรีได้

ในตัวแปลโปรแกรม 2 ส่วนที่กล่าวมาแล้วนั้น ได้แก่ตัววิเคราะห์เลกซิเคิล และพาสเซอร์ นับว่าเป็นเพียงส่วนตรวจสอบความถูกต้องด้านไวยากรณ์ภาษาของโปรแกรมเท่านั้น นอกเหนือไปจากส่วนดังกล่าวแล้ว ยังจะต้องมีอีกส่วนสำคัญคือส่วนของการสังเคราะห์โปรแกรมผลจากการวิเคราะห์โปรแกรมดิบ โดยสามารถแบ่งส่วนการสังเคราะห์โปรแกรมผล ออกเป็นส่วนย่อย ๆ ได้ดังต่อไปนี้

1. การสร้างรหัสชั่วคราว (Intermediate Code Generation)
2. การวิเคราะห์และสังเคราะห์รหัสที่เหมาะสม (Code Optimization)
3. การสร้างรหัสเครื่อง (Code Generation)

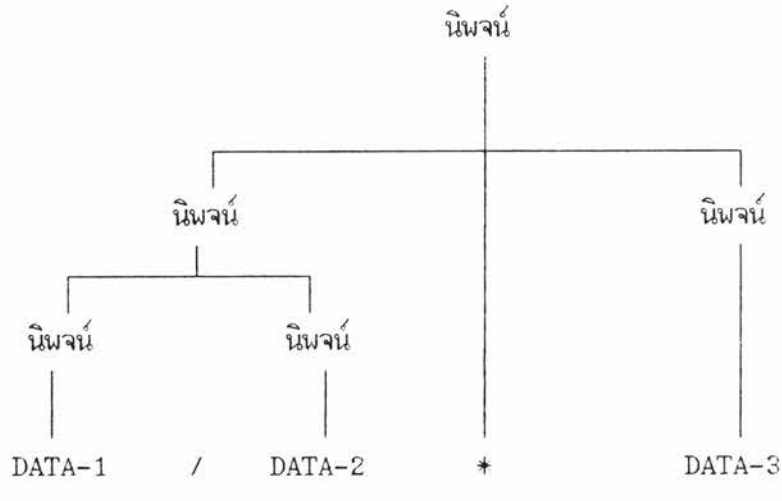
การสร้างรหัสกลางชั่วคราว (Intermediate Code Generation)

ดังได้กล่าวมาแล้วว่าจากขั้นตอนการทำพาสซิงหรือการทำการวิเคราะห์ความถูกต้องด้านไวยากรณ์ภาษาของโปรแกรมนั้น ผลลัพธ์ที่ได้คือ พาสทรี ซึ่งเป็นส่วนนำเข้าถัดมา ของส่วนการสร้างรหัสกลางชั่วคราว (Intermediate Code Generation)

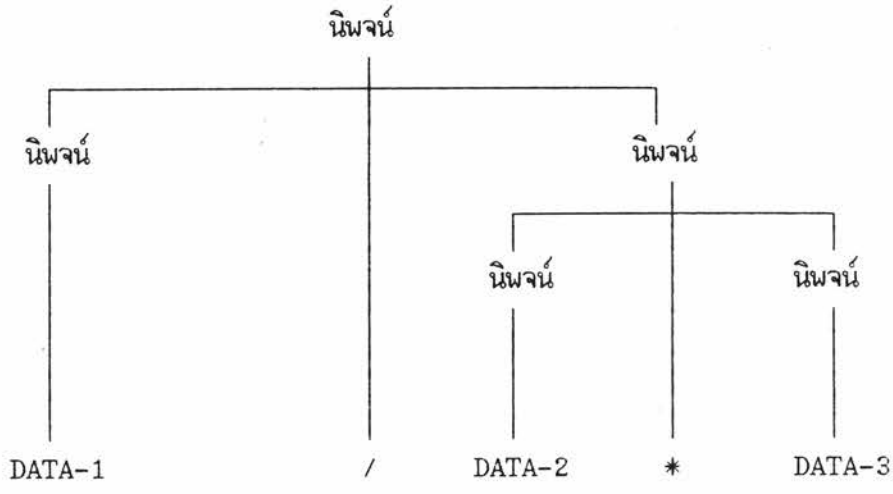
หน้าที่ของการสร้างรหัสกลาง คือเปลี่ยนแปลง พาสทรี ให้อยู่ในรูปแบบรหัสที่ ง่ายต่อการสร้างเป็นรหัสเครื่อง เนื่องจากรหัสต่าง ๆ ที่อยู่ในลักษณะของ พาสทรี นั้น ถูกสร้างขึ้นเพื่อแสดงความหมายที่ผู้เขียนโปรแกรมต้องการให้คำสั่งที่เขียนขึ้นนั้นทำอะไร ตัวอย่างเช่น

```
COMPUTE RESULT-VALUE = DATA-1 / DATA-2 * DATA-3.
```

ส่วนของนิพจน์ทางด้านขวามือสามารถนำมาสร้างเป็น พาสทรี ได้ดังนี้



(ก)



(ข)

วิธีการที่นิยมมากในการสร้างรหัสกลางชั่วคราวคือวิธีการสร้างรหัส 3 ตัวประกอบ (Three-Address Code) กล่าวคือจะนำพาสทรีมาทำการเดินผ่านเก็บผล (Traverse) ย่อยพาสทรี 1 ต้นให้เหลือคำสั่งหลายคำสั่งที่เกือบเทียบเท่ากับความสามารถของคำสั่งเครื่อง (Machine Code) หรือภาษาสัญญลักษณ์ของเครื่อง (Symbolic code)

(ก)
```
T1 = DATA-1 / DATA-2
T2 = T1 * DATA-3
```

(ข)
```
T1 = DATA-2 * DATA-3
T2 = DATA-1 / T1
```

เราจะพบว่าในระดับของรหัสกลางชั่วคราวนี้ ใกล้เคียงกับประโยคคำสั่งของภาษาแอสเซมบลีของเครื่องมาก ดังนั้นจึงเป็นความสะดวกที่จะแปลง พาสทรี ลงมาเป็น รหัสกลางชั่วคราวก่อนที่จะนำไปสร้างเป็นรหัสเครื่องที่จะใช้ทำงานได้จริงอีกครั้งหนึ่ง

ในความเป็นจริงแล้ว มีตัวแปลโปรแกรมจำนวนไม่น้อยที่ไม่แยกการสร้างพาสทรีออกมาเป็นผลลัพธ์ของ พาสเซอร์ แต่กลับทำการสร้างรหัสกลางชั่วคราวเลย ทั้งนี้ความเหมาะสมในการสร้างรหัสกลางชั่วคราวหลังจากการสร้าง พาสทรี หรือไม่มีการสร้าง พาสทรี ขึ้นอยู่กับความซับซ้อนของไวยากรณ์ภาษาระดับสูงนั้น ๆ ว่าจะซับซ้อนมากเพียงไหนอันชี้ถึงความยุ่งยากสับสน ในการวิเคราะห์ความถูกต้อง ทางด้านไวยากรณ์ภาษาของ โปรแกรมว่าทำได้ยากง่ายเพียงใด และถ้าหากจะสังเคราะห์รหัสกลางชั่วคราวในขณะนั้นเลย ทีเดียว จะนำความสับสนมาสู่การสร้างตัวแปลโปรแกรมมากเกินไปหรือไม่ เหล่านี้เป็นปัจจัยที่ต้องพิจารณาร่วมกันในการออกแบบสร้างตัวแปลโปรแกรมแต่ละตัวขึ้นมา

<u>การวิเคราะห์และสังเคราะห์รหัสที่เหมาะสม</u> (Code Optimization)

โปรแกรมผล (Object Program) ที่ถูกใช้งานบ่อย ๆ ควรจะมีขนาดเล็ก และสามารถดำเนินการได้เร็ว ดังนั้น ตัวแปลโปรแกรมบางชนิด จึงมีการสร้างส่วนประกอบที่ทำหน้าที่ลดขนาดของรหัสกลางชั่วคราว หรือปรับปรุงรหัสกลางชั่วคราวให้มีขนาดเล็กลงกว่าเดิม สามารถดำเนินการได้เร็วขึ้น เช่นตัวแปลโปรแกรมภาษา พีแอลวัน ซึ่งส่วนประกอบที่มีหน้าที่นั้นเรียกว่า การวิเคราะห์และสังเคราะห์รหัสที่เหมาะสม (Optimization Phase)

ตัวอย่างของรหัสกลางชั่วคราวบางอย่างที่ขาดประสิทธิภาพ สามารถปรับปรุงใหม่

```
LBL1:  IF A > B  GO TO  LBL2
       GO TO  LBL3
LBL2:  T1 = 2 * B
       T2 = T1 - 5
       IF A <= T2  GO TO  LBL4
       GO TO  LBL3
LBL4:  A = A + B
       GO TO  LBL1
LBL3:  ..................
       ..................
```

จากตัวอย่างนี้ จะเห็นได้ว่าความจริงแล้ว คือโครงสร้างของคำสั่งแบบวนทำในขณะที่เงื่อนไขยังเป็นจริงอยู่ (DO-WHILE loop) จากคำสั่ง

```
PERFORM EXECUTION-LOOP UNTIL (A NOT > B) OR
        (A > ((2 * B) - 5)).
EXECUTION-LOOP.
    ADD B TO A.
```

เราจะเห็นได้ว่าจากรหัสกลางชั่วคราวมีการสั่งให้กระโดดข้ามมากเกินความจำเป็นที่

```
IF A > B  GO TO  LBL2
GO TO  LBL3                    .....................(1)
```

ซึ่งอาจจะแก้ไขใหม่เป็น

```
IF A <= B  GO TO  LBL3    .....................(2)
```

เนื่องจาก (1) สามารถสร้างขั้นตอนการทำงานของรหัสเครื่อง ได้ดังนี้

1. เปรียบเทียบค่า A กับค่า B เพื่อจัดรหัสแสดงเงื่อนไข (Set Condition Code)
2. กระโดดไปที่ LBL2 ถ้ารหัสแสดงเงื่อนไข แสดงค่าความจริงเป็นจริง (Condition Code ถูก Set)
3. กระโดดไปที่ LBL3

ในขณะที่ (2) สามารถสร้างรหัสเครื่องเพียง 2 ขั้นตอน

1. เปรียบเทียบค่า A กับ B เพื่อจัดรหัสแสดงเงื่อนไข (Set Condition Code)
2. กระโดดไปที่ LBL3 ถ้ารหัสแสดงเงื่อนไขแสดงว่า < หรือ =

อีกตัวอย่างหนึ่ง ที่สามารถแสดงได้ คือ

```
A = B + C + D
E = B + C + F
```

โดยทั่วไป อาจสร้างรหัสชั่วคราวเป็น

```
T1 = B + C
A  = T1 + D
T1 = B + C
E  = T1 + F
```

ซึ่งสามารถปรับปรุงใหม่ให้เป็น

```
T1 = B + C
A  = T1 + D
E  = T1 + E
```

<u>การสร้างรหัสเครื่อง</u> (Code Generation)

ในขั้นตอนนี้ เป็นการเปลี่ยนรหัสกลางชั่วคราวให้กลายเป็น คำสั่งรหัสเครื่องที่สามารถทำงานได้จริง ๆ โดยวิธีการที่ง่ายที่สุด คือถอด (Map) จากรหัสกลางชั่วคราวออกมาเป็น รหัสภาษาเครื่องโดยตรงเลย เช่น

```
จากรหัสกลางชั่วคราว      A = B + C
อาจแปลงเป็นรหัสเครื่อง       LOAD  B
                             ADD   C
                             STORE A
```

จะเห็นว่าการถอดแบบตรงไปตรงมานี้ สามารถทำได้ง่าย แต่ประสิทธิภาพของรหัสเครื่องแบบนี้ อาจจะไม่ดีนัก เพราะเป็นคำสั่ง LOAD และ STORE ในหน่วยความจำทั้งสิ้น ซึ่ง

ทำได้ช้ากว่ารีจิสเตอร์ (Register) ในหน่วยประมวลผลกลาง ดังนั้นการสร้างรหัสเครื่องที่ดีจึงมักมีการสร้างรหัสให้ใช้ประโยชน์จากรีจิสเตอร์ ในหน่วยประมวลผลกลางดังกล่าวมากที่สุด โดยหลีกเลี่ยงคำสั่ง LOAD และ STORE ในหน่วยความจำ แต่อย่างไรก็ดี สถาปัตยกรรมของหน่วยประมวลผลกลางของคอมพิวเตอร์บางระบบ มีจำนวนรีจิสเตอร์ ไม่มากนัก ดังนั้นเป็นหน้าที่ของ ตัวสร้างรหัสเครื่องที่ดี ที่จะใช้ประโยชน์จากรีจิสเตอร์ที่มีจำกัดนั้นให้มากที่สุด เท่าที่จะเป็นไปได้ตามหลักการของการจัดสรรรีจิสเตอร์ (Register Allocation)

การจัดการเกี่ยวกับความผิดพลาดที่เกิดขึ้น (Error Handling)

หน้าที่ที่สำคัญอีกประการหนึ่งของตัวแปลโปรแกรม คือ ตรวจพบความผิดและรายงานความผิดที่พบนั้นให้ผู้เขียนโปรแกรมได้ทราบอย่างชัดเจน ซึ่งความผิดพลาดนั้นอาจจะพบในช่วงเวลาต่าง ๆ กัน เช่น

1. สะกดผิด ทำให้ตัวสำคัญบางคำผิดพลาดไป
2. ตัววิเคราะห์ไวยากรณ์ (Syntax Analyzer) ทำงานต่อไม่ได้ เนื่องจากวงเล็บไม่ครบ
3. การสร้างรหัสกลางชั่วคราวทำไม่ได้เนื่องจากตัวกระทำ (Operand) 2 ตัวที่มากระทำกันเป็นคนละชนิดกัน
4. ตัววิเคราะห์และสังเคราะห์รหัสที่เหมาะสม (Code Optimizer) พบว่ารหัสบางอันที่ถูกสร้างขึ้นทำงานแบบวนทำโดยไม่มีจุดสิ้นสุด
5. ตัวสร้างรหัสเครื่อง (Code Generator) สร้างรหัสเครื่องไม่ได้ เพราะค่าคงที่ที่ใช้มีขนาดใหญ่เกินกว่าที่สถาปัตยกรรมของเครื่องจะรองรับได้
6. ตัวแปรที่สร้างขึ้นไม่เป็นหนึ่งเดียว (Unique) ในขณะที่สร้างตารางสัญญลักษณ์ (Symbol Table) พบความผิดพลาดดังกล่าว

ไม่ว่าความผิดพลาดจะถูกพบในช่วงใดก็ตาม จะต้องมีการรายงานความผิดพลาดนั้น โดยมีส่วนของตัวแปลโปรแกรมที่เรียกว่าตัวจัดการความผิดพลาด (Error Handler) ทำหน้าที่สร้างข่าวสารที่ชัดเจนให้ผู้เขียนโปรแกรมได้ทราบ และต้องทำการ "ซ่อม" ความผิดพลาดนั้น เพื่อให้การแปลโปรแกรม ดำเนินต่อไปได้ อย่างน้อยเพื่อให้ความผิดพลาดทั้งหมดถูกจับได้ตลอด

ทั้งโปรแกรม ด้วยการแปล โปรแกรมเพียงครั้งเดียว ดังนั้นอาจสรุปแยกหน้าที่สำคัญของตัวจัดการความผิดพลาด ได้เป็น

1. ตรวจสอบความผิดพลาด (Error Detection)
2. รายงานความผิดพลาด (Error Diagnostic Report)
3. แก้ไขความผิดพลาดเพื่อดำเนินต่อ (Error Recovery)

ภาษาคอมพิวเตอร์ระดับสูง (High Level Language)

จากองค์ประกอบต่าง ๆ ของตัวแปลโปรแกรม ดังได้กล่าวมาแล้วนั้น จะพบว่าตัวแปลโปรแกรมเป็นตัวกลาง (Interface Machine) ที่ทำหน้าที่เป็นสื่อติดต่อให้ผู้ใช้เครื่องคอมพิวเตอร์กับเครื่อง เข้าใจกันโดยใช้ภาษาคอมพิวเตอร์เป็นเครื่องมือ โดยสามารถกำหนดขั้นตอนอย่างละเอียดของงานที่ต้องการได้ ซึ่งเราเรียกลักษณะของภาษาคอมพิวเตอร์แบบนี้ว่าภาษากำหนดกระบวนการ (Procedural Language)

ข้อดีที่ทำให้ภาษาระดับสูงนี้เป็นที่นิยมมากกว่าภาษาระดับต่ำ เนื่องจาก

1. เข้าใจง่าย เพราะคำสั่งที่เขียนขึ้นใกล้เคียงกับภาษามนุษย์ ในหมวกภาษาระดับสูงด้วยกันเอง ยังอาจจะพบว่า บางภาษาง่าย บางภาษาไม่ง่ายต่อการเขียนโปรแกรมนัก ซึ่งหากกล่าวโดยกว้าง ๆ อาจกล่าวได้ว่า ภาษาระดับสูงที่ดี ควรจะมีลักษณะที่ง่าย ต่อการออกแบบโปรแกรมเป็นส่วน ๆ (Modular - Design) ทั้งยังต้องคำนึงถึงชนิดของข้อมูลที่ถูกเก็บ รวมทั้งเครื่องหมาย และคำสำคัญที่ใช้ ควรง่ายต่อการเข้าใจ
2. ธรรมชาติพื้นฐานของภาษา เนื่องจากภาษาระดับสูงส่วนใหญ่ถูกสร้างขึ้นจากความเข้าใจในการแสดงแนวความคิด เพื่อแก้ปัญหาต่าง ๆ กัน เช่น ภาษาโคบอลถูกสร้างขึ้นในลักษณะคำสั่งภาษาอังกฤษอย่างตรงไปตรงมา เหมือนกับการสั่งงานในประจำวันของคน
3. โยกย้ายได้ กล่าวคือโปรแกรมที่ถูกเขียนขึ้นโดยภาษาระดับสูง เช่น โคบอล สามารถนำไปประมวลผลบนเครื่องคอมพิวเตอร์เครื่องใดก็ได้ โดยแทบจะไม่ต้องมีการแก้ไขเปลี่ยนแปลงโปรแกรมเลย หากโปรแกรมนั้นเขียนขึ้นตาม

มาตรฐานสากลที่กำหนด เนื่องจากเป็นภาษาที่มีการกำหนดรูปแบบมาตรฐานยอมรับกันโดยทั่วไป

อย่างไรก็ดี ในเรื่องที่จะต้องนำมาพิจารณาถึงการกำหนดรูปแบบของภาษาระดับสูงใด ๆ ว่าสามารถใช้งานได้ดี หรือง่ายเพียงใดนั้น จะพิจารณาในเรื่องต่าง ๆ ดังนี้

1. โครงสร้างข้อมูล ให้ใช้การกำหนดรูปแบบข้อมูลที่เก็บแบบใดได้บ้าง เช่น เก็บแบบ ตัวอักษร, สายวลี, เลขฐานสอง เป็นต้น
2. กฎเกณฑ์ที่กำหนดเป็นไวยากรณ์ของภาษา
3. วิธีการควบคุมลำดับขั้นตอนการทำงานของโปรแกรม (Flow of Control)

ลักษณะโดยทั่วไปของภาษาโคบอล

ภาษาโคบอล เป็นภาษาคอมพิวเตอร์ระดับสูง ที่ถูกสร้างขึ้นเพื่อการใช้งานด้านการประมวลผลที่ไม่ต้องมีความซับซ้อนยุ่งยากในเชิงคำนวณ แต่เหมาะกับการใช้งานเกี่ยวกับการจัดการแฟ้มข้อมูล ตลอดจนการออกรายงานทางด้านธุรกิจ โปรแกรมภาษาโคบอลจะประกอบกันในลักษณะของลำดับขั้น ดังแสดงในรูปที่ 2-6

หลักการโดยทั่วไปของภาษาโคบอล

1. มีการกำหนดชื่อโปรแกรมใด ๆ ที่เขียนขึ้น ภายใต้ย่อหน้า (Paragraph) PROGRAM-ID ในส่วนของ IDENTIFICATION DIVISION เพื่อนำไปสร้างเป็นชื่อที่ระบบใช้อ้างอิง (System Alias) ในการอ้างถึงโปรแกรมนั้น

2. การควบคุมติดต่อกับอุปกรณ์รับส่งข้อมูลภายนอก กระทำภายใต้ ENVIRONMENT DIVISION

   2.1 การควบคุมเครื่องพิมพ์ (Printer Channel) ทำได้โดยการกำหนดชื่อพิเศษเพื่ออ้างถึงในขณะดำเนินงานโดยย่อหน้า SPECIAL-NAMES

   2.2 กำหนดอุปกรณ์รับส่งข้อมูลที่ใช้ในการบันทึกข้อมูลให้สอดคล้องกับแฟ้มข้อมูลที่ต้องการใช้ภายใต้ย่อหน้า FILE-CONTROL โดยการใช้วลี SELECT

   2.3 อาจจะกำหนดจำนวนบัฟเฟอร์ (Buffer) ทั้งสำหรับรับและส่งข้อมูล

ระหว่างหน่วยความจำกับอุปกรณ์รับส่งข้อมูลโดยปรกติแล้วจำนวนบัฟเฟอร์จะถูกกำหนดเป็นบัฟเฟอร์คู่เสมอ (Double Buffer)

3. ในส่วนของ DATA DIVISION สามารถกำหนดรูปแบบของโครงสร้างข้อมูลได้หลายประเภท ได้แก่

3.1 รูปแบบ ตัวอักษรธรรมดา ซึ่งเป็นค่าทั่วไป (Default Value)

PIC X( ) หรือ

PIC 9( ) USAGE IS DISPLAY.

3.2 รูปแบบ การเก็บตัวเลขฐานสอง ใช้เนื้อที่ 2 ไบต์ 4 ไบต์ และ 8 ไบต์

PIC 9( ) USAGE IS COMPUTATIONAL.

3.3 รูปแบบ การเก็บตัวเลขแบบ Short-precision internal Floating point ใช้เนื้อที่ 4 ไบต์*3

PIC 9( ) USAGE IS COMPUTATIONAL-1.

3.4 รูปแบบ การเก็บตัวเลขแบบ Long-precision internal Floating point ใช้เนื้อที่ 8 ไบต์*3

PIC 9( ) USAGE IS COMPUTATIONAL-2.

3.5 รูปแบบ การเก็บตัวเลขแบบ Packed Decimal

PIC 9( ) USAGE IS COMPUTATIONAL-3.

3.6 รูปแบบ การเก็บตัวเลขแบบเลขฐานสองใช้เนื้อที่ 2 ไบต์ 4 ไบต์ 8 ไบต์

PIC 9( ) USAGE IS COMPUTATIONAL-4.**

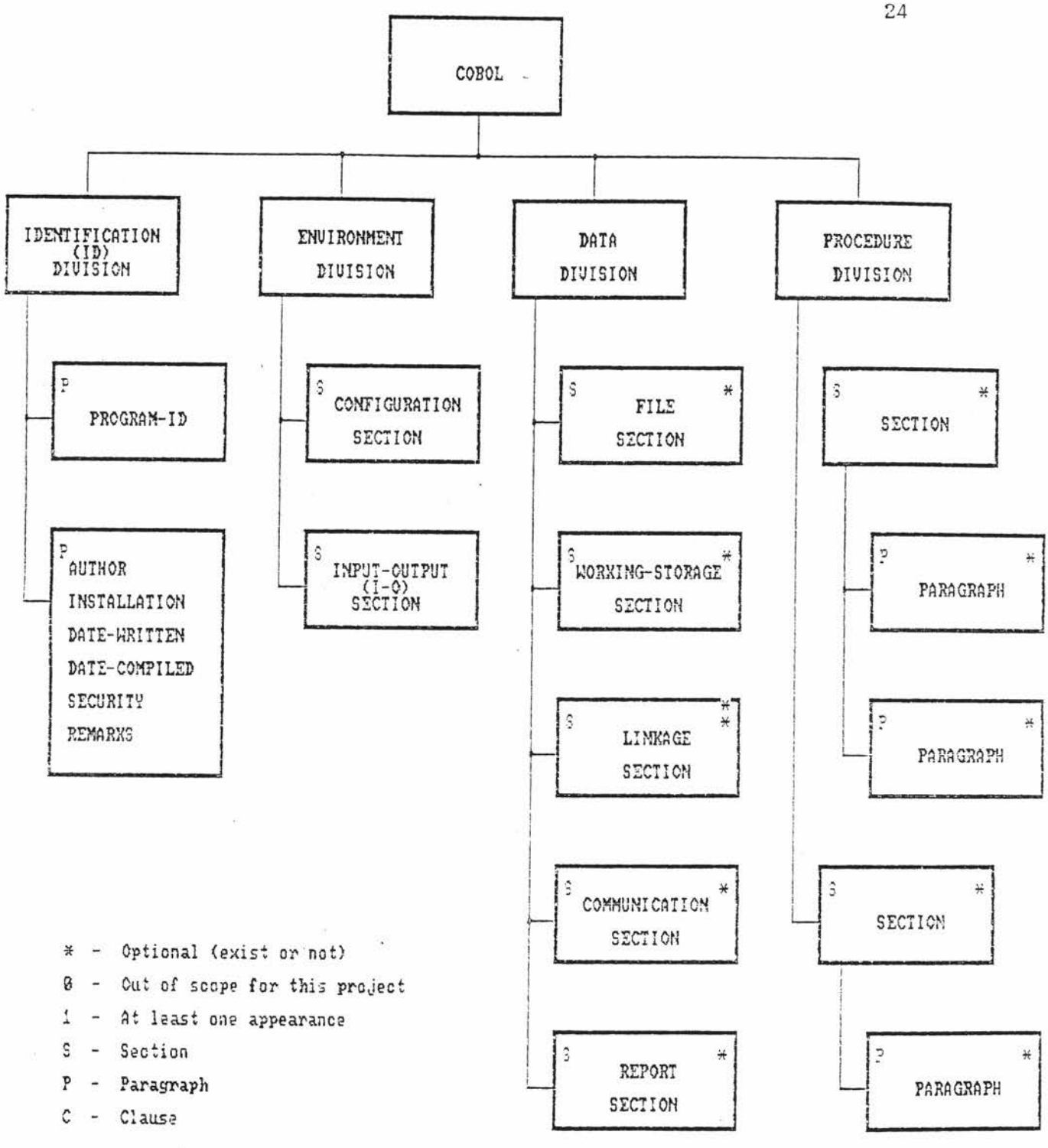
3.7 การกำหนด DSECT ทำได้โดยใช้วลี REDEFINES

3.8 การสร้างโครงสร้างข้อมูลแบบ Array ทำได้โดยใช้วลี OCCURS

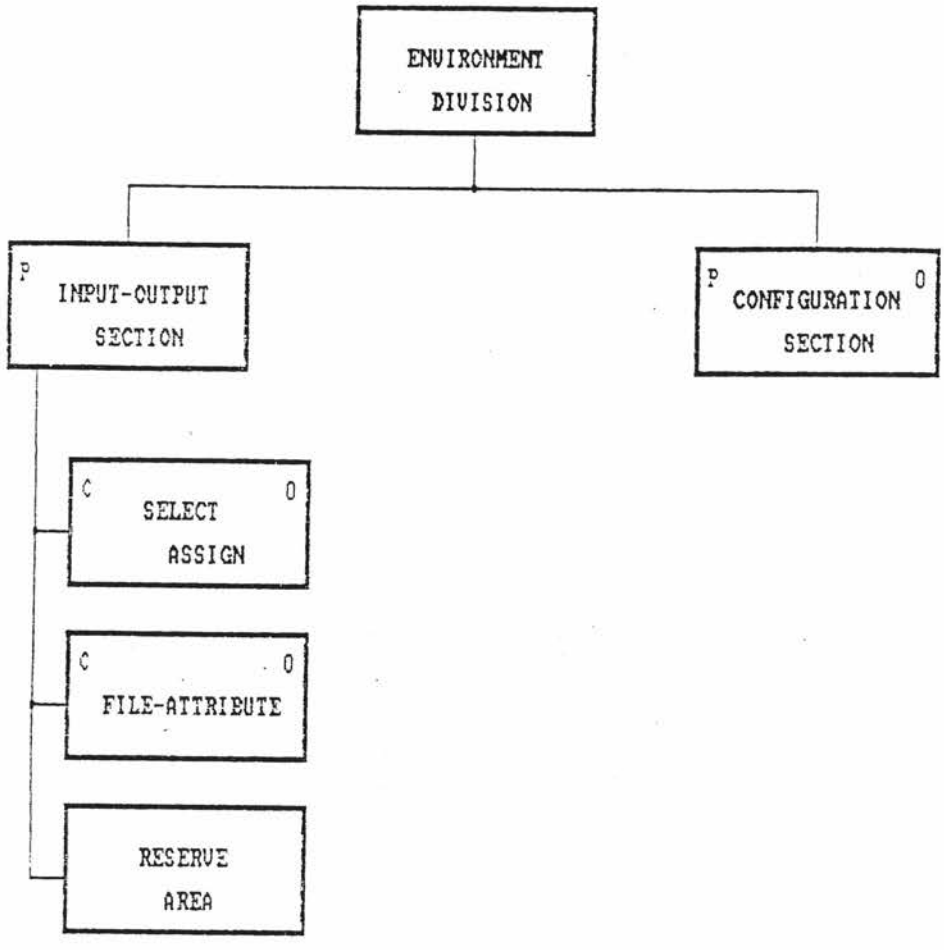
---

* เฉพาะ IBM OS/VS COBOL เท่านั้น

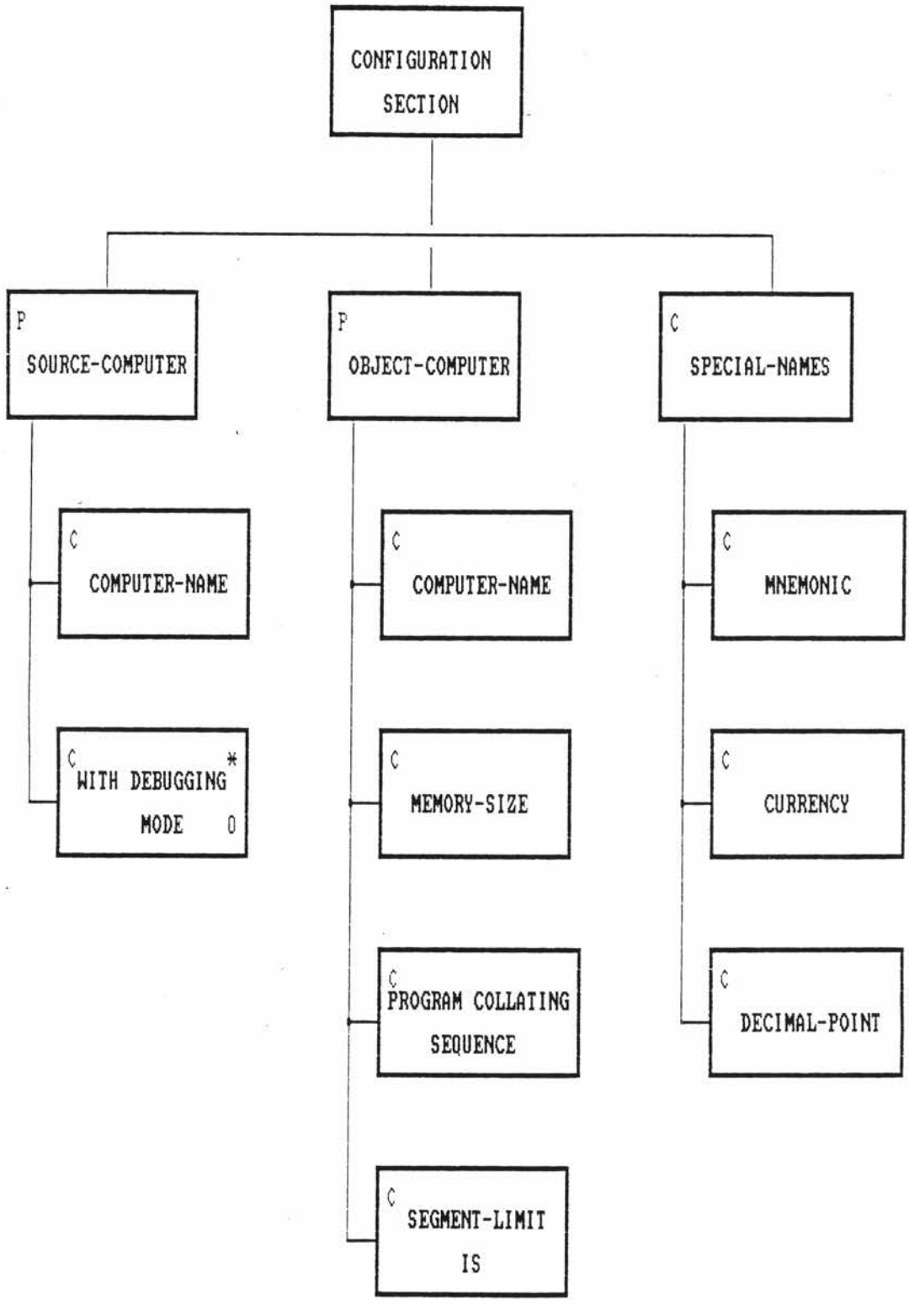
** ขนาดของเนื้อที่ ที่ใช้เก็บข้อมูล ขึ้นอยู่กับขนาดของข้อมูล ที่ต้องการเก็บ แต่ต้องถูกจัด (Align) ให้ลงตามขอบเขต (Boundary) ของ Halfword, Fullword, Doubleword เสมอ และในที่นี้สำหรับ OS/VS COBOL ทั้ง COMP และ COMP-4 จะมีการจัดเก็บข้อมูลเหมือนกันทุกประการ



รูปที่ 2-6 แสดงส่วนประกอบของโปรแกรมภาษาโคบอล



รูปที่ 2-7 แสดงส่วนประกอบภายใน ENVIRONMENT DIVISION



รูปที่ 2-8 แสดงส่วนประกอบภายใน CONFIGURATION SECTION

4. การควบคุมการทำงานของโปรแกรม (Flow of Control) ถูกกำหนดตามตรรกวิธีภายในส่วนของ PROCEDURE DIVISION ลักษณะการวนทำของภาษาโคบอล เป็นไปในลักษณะของการวนทำในขณะที่เงื่อนไขเป็นจริง (DO-WHILE Loop) คือประโยคคำสั่ง PERFORM.....UNTIL

เนื่องจากภาษาโคบอล ได้ถูกสร้างขึ้นเป็นเวลานานมาแล้ว ตั้งแต่ประมาณปี คศ. 1959 จากความร่วมมือของหลายฝ่าย โดยที่ในยุคนั้นวิชาการที่เกี่ยวกับหลักการสร้างตัวแปลโปรแกรมและตัวแปลภาษา ยังมิได้มีการกำหนด BNF (Backus Naur Form)*4

ด้วยเหตุดังกล่าว หลักการในการสร้างตัวแปลโปรแกรมบางอย่าง ไม่สามารถนำมาใช้สร้างตัวแปลโปรแกรมภาษาโคบอลได้โดยตรง ต้องมีการดัดแปลงมากมาย โดยเฉพาะอย่างยิ่งในขั้นตอน การวิเคราะห์ไวยากรณ์ภาษาของโปรแกรม ไม่อาจนำหลักการของ Finite Automata state**5 มาใช้ได้ดีนัก

การออกแบบเพื่อจัดสร้างตัวตรวจสอบไวยากรณ์โปรแกรมภาษาโคบอล จึงมีความเบี่ยงเบนไปจากหลักมาตรฐานต่าง ๆ ดังกล่าวมาแล้วไปบ้าง ดังจะนำเสนอต่อไป

---

* การรับรู้ (Recognize) ว่าสายอักขระที่พบ (Detect) นั้นจัดเป็นส่วน (Token) ที่ถูกต้องตามไวยากรณ์ของภาษาโดยความหมายที่อธิบายโดย BNF หรือไม่

** รูปแบบที่ตกลงกันเป็นมาตรฐาน ใช้ในการเขียนอธิบายลักษณะของไวยากรณ์ภาษา