บทที่ 4

คอมไพเลอร์ (COMPILER)

คอมไพเลอร์ทำหน้าที่ในการแปลภาษาต้นแบบไปเป็นรหัสกลาง (intermediate code) ในบทนี้จะอธิบายถึงโทเกนชนิดต่างๆ และการอ่านโทเกนของสแกนเนอร์ การออกแบบโครงสร้างข้อมูลของตารางสัญลักษณ์เพื่อจัดเก็บตัวระบุ การแปลนิพจน์,สเตตเมนต์, ฟังก์ชันและกระบวนการ นอกจากนี้ยังกล่าวถึงขั้นตอนการทำย้อนแก้ (backpatch) เพื่อให้คอมไพเลอร์สามารถแปลโปรแกรมต้นฉบับได้ในรอบเดียว

4.1 สแกนเนอร์ (Scanner)

การทำงานของสแกนเนอร์ จะเริ่มจากเปิดแฟ้มข้อมูล ของภาษาต้นแบบ เพื่อนำมาตรวจสอบเบื้องต้นครั้งละหนึ่งไบต์ โดยถ้า อักษรตัวแรกเป็นอักขระ (Character) ก็จะถือว่า โทเกนนั้นเป็นคำ ถ้าอักขระตัวแรกเป็นตัวเลข (Digit) หรือเครื่องหมายบวกหรือลบ (Sign) จะถือว่าโทเกน นั้นเป็นจำนวน (number) นอกนั้นจะถือว่าเป็นสัญลักษณ์พิเศษ (Special character)

Sign	หมายถึง+ , -
Digit	หมายถึง0 - 9
Character	หมายถึงA - Z , a - z , _ (underscore)

กลุ่มของโทเกนแบ่งออกได้ดังนี้

- คำ (Word) คือโทเกนที่ขึ้นต้นด้วย character หรือ under score โดยแบ่งเป็น
  - คำสงวน(Reserve word) คือคำที่กำหนดให้ใช้ในการเขียนภาษาตามที่ได้ออกแบบไว้
  - ตัวระบุ (Identifier) คือ คำอื่นๆ ที่ไม่ใช่คำสงวน เช่น ชื่อตัวแปร ชื่อฟังก์ชัน และชื่อของกระบวนการ เป็นต้น
- ตัวเลข (Number) ซึ่งในภาษานี้จะเป็นข้อมูลชนิดจำนวนเต็ม 16 บิตเท่านั้น
- สัญลักษณ์พิเศษ (Special character)

คำสงวน และตัวระบุ

ในกรณีที่อักขระตัวแรกเป็นคำสแกนเนอร์ จะอ่านข้อมูลจนพบวรรค (white space) หรือจนจบบรรทัด ก็จะได้โทเกนมา 1 คำ โทเกนนี้จะถูกนำไปเปรียบเทียบกับตารางคำสงวนหากพบคำนี้อยู่ในตารางคำสงวน โทเกนนี้จะเป็นคำสงวน หากไม่พบในตาราง จะถือว่าเป็นตัวระบุ โดยผู้ใช้สามารถกำหนดให้ตัวระบุมีความยาวตั้งแต่ 1 ถึง 255 ตัว

ตัวเลข (number)

ในภาษานี้ ข้อมูลชนิดตัวเลขจะเป็นข้อมูลแบบจำนวนเต็ม 16 บิต เท่านั้นดังนั้นในการอ่านโทเกนตัวเลข จะอ่าน อักขระทีละตัวจนกระทั่งไม่ใช่ข้อมูลชนิด digit

สัญลักษณ์พิเศษ (Special Symbol)

ในภาษานี้สัญลักษณ์พิเศษประกอบด้วย อักขระดังนี้คือ { } = < > : ; / ! * &

คอมเมนต์ (Comment)

ในกรณีที่โทเกนเป็นสัญลักษณ์พิเศษ // สแกนเนอร์จะถือว่าข้อความที่อยู่หลังเครื่องหมายนี้เป็นคอมเมนต์ทั้งบรรทัด และจะอ่านข้ามบรรทัดนั้นไป

4.2 ตารางสัญลักษณ์ (symbol table)

ในขั้นตอนการคอมไพเลอร์ สแกนเนอร์จะตรวจสอบว่าอักขระที่อยู่ในโปรแกรมเป็นโทเกนชนิดใดหากโทเกนที่พบเป็นตัวระบุ (identifier) จะถูกจัดเก็บในตารางสัญลักษณ์ ซึ่งตารางสัญลักษณ์จะเก็บชื่อ และ แอดเดรส (address) และข้อมูลที่จำเป็นของตัวระบุเหล่านี้

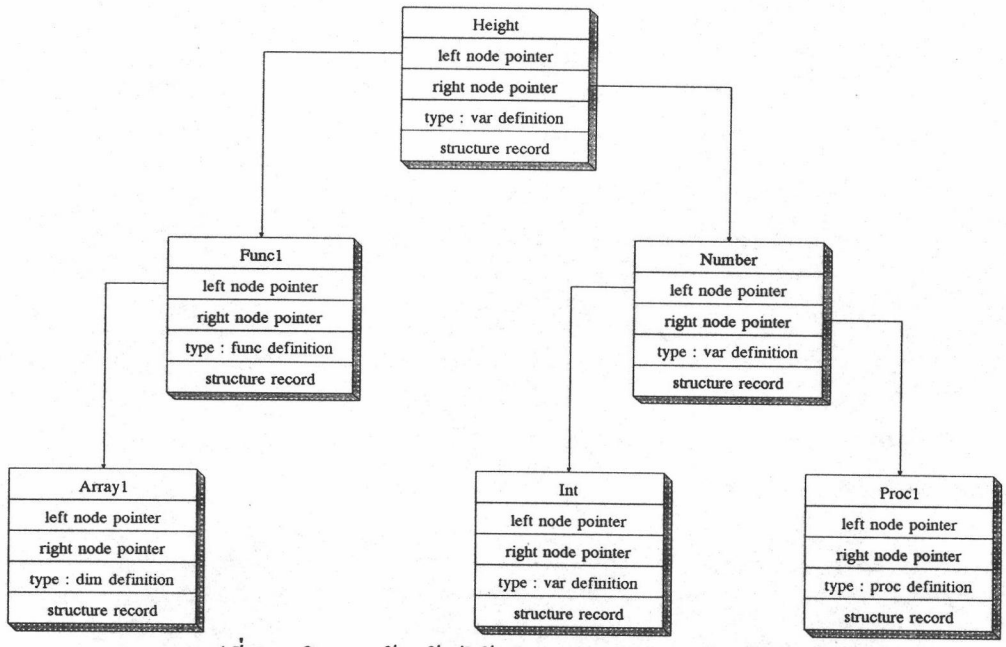
ในตารางสัญลักษณ์จะเก็บตัวระบุ ซึ่งแบ่งออกได้เป็น 4 ประเภท คือ

- Process definition
- function definition
- variable definition
- dimension definition

เนื่องจากคอมไพเลอร์จำเป็นจะต้องจัดการกับข้อมูลที่เกี่ยวข้องกับตัวระบุจำนวนมาก ดังนั้นเพื่อความรวดเร็วของการแปล จึงจำเป็นต้องออกแบบโครงสร้างของมูลที่ใช้ในการจัดการตัวระบุให้เหมาะสม โครงสร้างต้นไม้ทวิภาค (binary tree structure) เป็นโครงสร้างหนึ่งที่มีประสิทธิภาพในการจัดเก็บได้ง่าย และค้นหาได้อย่างรวดเร็ว

ตารางสัญลักษณ์นี้จะเก็บอยู่ในรูปของโครงสร้างต้นไม้ทวิภาค (binary tree structure) โดยจะจัดเรียงตามชื่อของตัวระบุ ตัวอย่างเช่นสแกนเนอร์พบโทเกนที่เป็นตัวระบุ ตามลำดับดังนี้

Height	variable definition
Func1	function definition
Array1	dimension definition
number	variable definition
int	variable definition
proc1	process definition



รูปที่ 4.1 โครงสร้างต้นไม้ทวิภาคของตารางสัญลักษณ์

structure record จะประกอบด้วยข้อมูลที่จำเป็นของตัวระบุ แต่ละชนิด

1.variable definition จะประกอบด้วย ชื่อและแอดเดรสของตัวแปร

2.dimension definition จะประกอบด้วย ชื่อ แอดเดรส และขนาด ของตัวแปร

3.function definition จะประกอบด้วยชื่อ,แอดเดรสและฟอร์มอลพารามิเตอร์ (formal parameter) ซึ่งฟอร์มอลพารามิเตอร์จะมีจำนวนกี่ตัว หรือ ไม่มีเลยก็ได้

4.Process definition จะประกอบด้วยชื่อ ,แอดเดรสและ ฟอร์มอลพารามิเตอร์ ซึ่งฟอร์มอลพารามิเตอร์ จะมีจำนวนกี่ตัว หรือ ไม่มีเลยก็ได้

ตารางสัญลักษณ์นี้จะมีสองระดับ หรือสองตารางเพื่อใช้เก็บตัวระบุ สองระดับได้แก่ตัวแปรส่วนกลาง และตัวแปรเฉพาะที่ เมื่อคอมไพเลอร์เริ่มทำงานจะในตารางสัญลักษณ์ส่วนกลาง หากมีโปรแกรมย่อยต่างๆเกิดขึ้น เช่น ฟังก์ชันหรือกระบวนการ จะเปลี่ยนไปใช้ตารางสัญลักษณ์ในระดับที่สองได้แก่ตารางสัญลักษณ์เฉพาะที่ และเมื่อเสร็จสิ้นจากกระบวนการย่อย จะกลับมาใช้ตารางสัญลักษณ์ส่วนกลางดังเดิม

ในการค้นหาตัวระบุในตารางสัญลักษณ์ หากคอมไพเลอร์ทำการแปลในโปรแกรมหลักจะเลือกใช้ตารางสัญลักษณ์ส่วนกลาง หากอยู่ระหว่างการคอมไพล์โปรแกรมย่อย จะค้นหาตัวระบุในตารางสัญลักษณ์เฉพาะที่ก่อน ถ้าไม่พบจึงจะไปค้นหาในตารางสัญลักษณ์ส่วนกลาง

4.3 LL grammar (Left to right Leftmost derivation)

ไวยากรณ์ LL คือไวยากรณ์ในการแปลที่จะแปลโดยเลือกอนุพันธ์จากซ้ายไปขวา เมื่อเลกซิคัลอนาไลเซอร์อ่านโทเกนจากซ้ายไปขวาแล้ว ส่งมาให้พาซเซอร์จะทำการเลือกโปรดักชันที่ใช้แปลจากโทเกนที่เข้ามาตามลำดับ และจะแทนที่สัญลักษณ์ที่ถูกแทนที่ได้จากซ้ายไปขวา ไวยากรณ์ LL(1) จะหมายถึงไวยากรณ์ที่แปลโดยไวยากรณ์ LL และกระจายประโยคสำหรับสัญลักษณ์ที่ถูกแทนที่ได้เพียงแบบเดียวเท่านั้น เช่นกำหนดไวยากรณ์

1. <expression> ::= OPEN_PAREN <expression> CLOSE_PAREN |
2. NUMBER MINUS <expression>

หากเลกซิคัลอนาไลเซอร์ส่งโทเกนตัวแรกเข้ามาเป็นโทเกนวงเล็บเปิด จะเลือกใช้โปรดักชันแรก หากโทเกนที่ได้เป็น NUMBER จะเลือกใช้โปรดักชันที่สอง ดังนั้นไวยากรณ์นี้จึงเป็นไวยากรณ์นี้เป็น LL (1) หากเราเพิ่มไวยากรณ์

3. <expression> ::= NUMBER PLUS <expression>

หากรับโทเกนเป็น NUMBER จะทำให้มีโปรดักชันในการแปลสองแบบ ดังนั้นไวยากรณ์นี้ไม่เป็นไวยากรณ์ LL(1)

การพัฒนาตอมไพเลอร์นี้ จะแปลแบบเรียกซ้ำตามลำดับขั้นซึ่งจัดเป็นการแปลที่ใช้ไวยากรณ์ LL(1) คือจะแทนที่สัญลักษณ์ที่แทนที่ได้จากตามซ้ายไปขวา และจะดำเนินการอนุพันธ์จากด้านซ้ายสุดก่อน

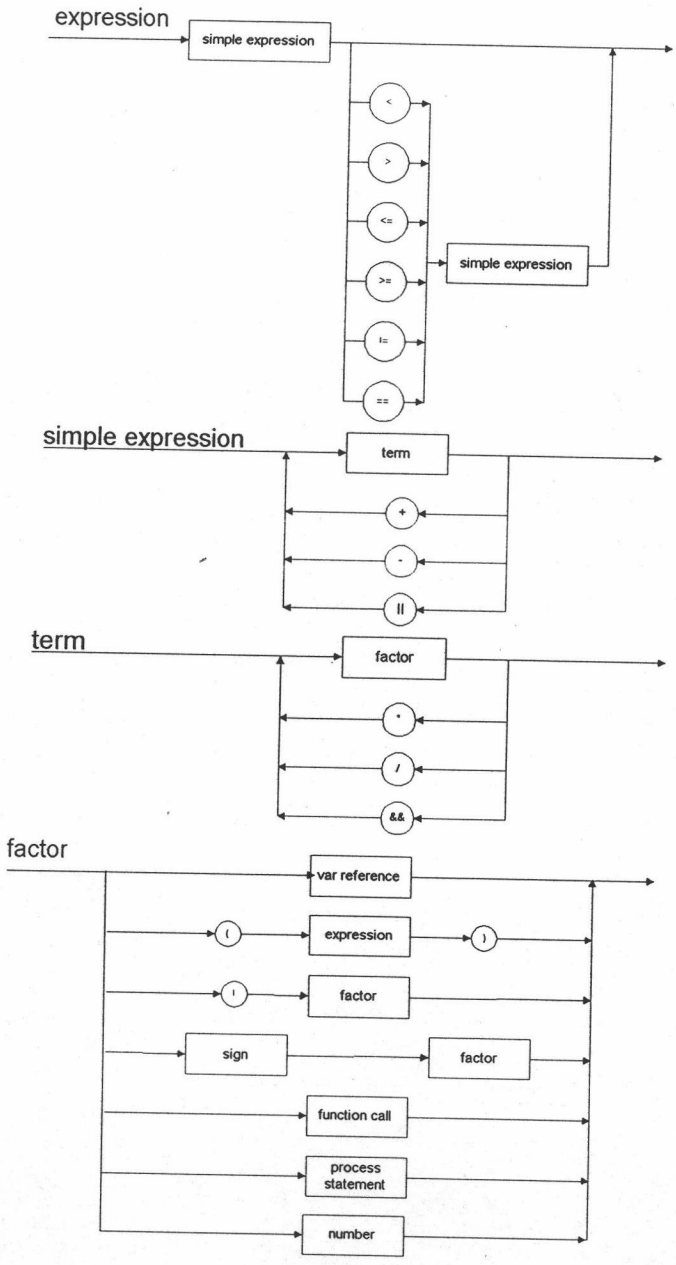
4.4 การแปลนิพจน์ (Parsing expression)

ภาษาต้นแบบกำหนดให้ตัวดำเนินการ (operator) มีลำดับความสำคัญที่แตกต่างกัน ดังนี้

operator	precedence	categories
& !	4	unary operators
* / &&	3	multiplying operators
+ - ||	2	adding operators
== != < <= > >=	1	relational operators

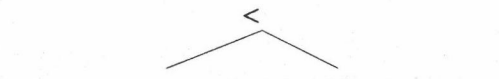
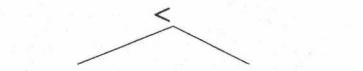
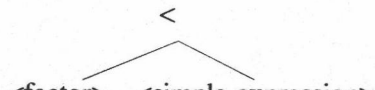
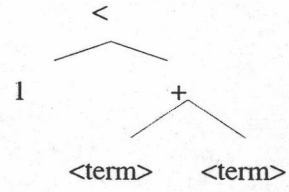
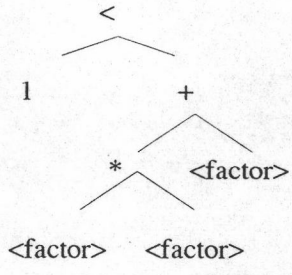
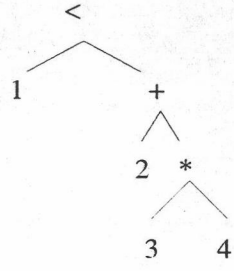
ตารางแสดง precedence of operators

เนื่องจากตัวดำเนินการที่มีลำดับความสำคัญสูงกว่าตัวดำเนินการตัวอื่นจะถูกดำเนินการก่อน เมื่อพิจารณาจากวิธีที่ใช้ในการแปลคือเทคนิคการแปลแบบเรียกซ้ำ จะพบว่าสัญลักษณ์ที่ถูกแทนที่ได้ (nonterminal) ที่ถูกเรียกก่อนจะถูกแปลเป็นตัวสุดท้าย ในขณะที่สัญลักษณ์ที่ถูกเรียกซ้ำจะถูกแปลก่อน เมื่อเรียงตามลำดับความสำคัญของตัวดำเนินการ เราจะสร้างสัญลักษณ์ที่ถูกแทนที่ได้ 4 ระดับ คือ expression เพื่อใช้แปลในระดับตัวดำเนินการเปรียบเทียบ (relational operation) ,simple expression เพื่อใช้แปลในระดับตัวดำเนินการบวก (adding operation) , term เพื่อใช้แปลในระดับตัวดำเนินการคูณ (multiplying operation) , และ factor เพื่อใช้แปลในระดับย่อยที่สุด ดังนั้นไวยากรณ์ของนิพจน์จะเรียงตามลำดับการเรียกแทนที่คือ expression จะเรียกซ้ำไปยัง simple expression ,term, และ factor ตามลำดับ ดังนี้



รูปที่ 4.2 ไวยากรณ์ของ expression, simple expression, term และ factor

ตัวอย่างเช่นนิพจน์ 1 < 2 + 3 * 4 จะแปลดังนี้

ขั้นตอน	พาซทรี	ไวยากรณ์ที่ใช้แปล
1	<expression>	
2		expression
3		simple expression
4		term
5		factor
6		simple expression
7		term
8		factor

จะได้รหัสกลาง

literal 1

literal 2

literal 3

multiply

literal 4

add

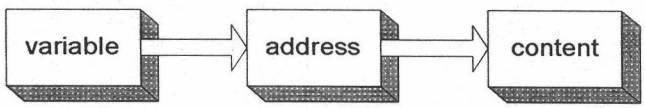
L.T.

4.5 การแปลสเตตเมนต์ (Parsing statement)

4.5.1 การอ้างอิงตัวแปร (variable reference)

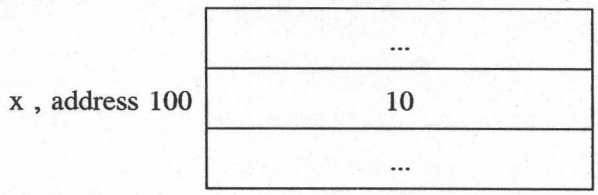
- ค่าทางด้านซ้ายและค่าทางด้านขวา (R-value vs. L-value)

ในการอ้างอิงหน่วยความจำเพื่อความง่ายในการเข้าถึงข้อมูล เรามักกำหนดตัวแปรในการอ้างอิงถึงตำแหน่งบนหน่วยความจำ ซึ่งแอดเดรส นี้จะเป็นตัวชี้ไปสู่ข้อมูลจริงที่อยู่บนหน่วยความจำ

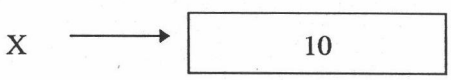


รูปที่ 4.3 การอ้างอิงตัวแปร

เช่น ตัวแปร X อยู่ในตำแหน่งที่ 100 ซึ่งเก็บค่า 10 อยู่ แสดงดังรูป



ในภาษาต้นแบบ หากกำหนด สเตตเมนต์กำหนดค่า (assignment statement) X = 10 หมายถึงนำค่า 10 ไปเก็บไว้ที่ตำแหน่งที่ X ชี้



เรากำหนดให้ R-value จะหมายถึง content ส่วน L-value จะหมายถึง address ของตัวแปรนั้น ดังนั้น สเตตเมนต์กำหนดค่า

```
X = 10
Y = X
```

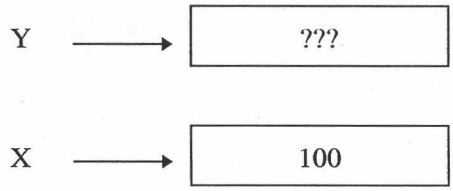
จะแปลเป็น

```
L-VALUE     X
LITERAL     10
SET
L-VALUE     Y
R-VALUE     X
SET
```

- แอดเดรส (ADDRESS ,&)

ในสเตตเมนต์กำหนดค่า จะพบว่าตัวแปรที่ปรากฏทางด้านซ้าย จะหมายถึง L-value หรือ address หากพบตัวแปรอยู่ทางด้านขวาจะหมายถึง R-value หรือ content แต่ถ้าพบ & ณ.ตัวแปรทางด้านขวาจะหมายถึง address ของตัวแปรนั้นเช่น

X = &Y สมมติให้ Y อยู่ที่ตำแหน่ง 100 หมายถึง



หรือเขียนเป็นรหัสกลาง ได้ดังนี้

```
L-VALUE     X
L-VALUE     Y
SET
```

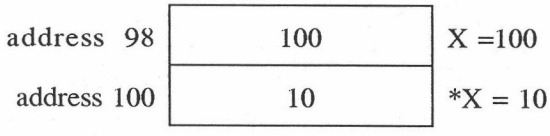
เราจะไม่พบคำสั่ง & ณ.ตัวแปรด้านซ้ายของสเตตเมนต์กำหนดค่า (assignment statement)

- Indirect address , (*)

หากพบเครื่องหมาย * (asterisk) อยู่ทางด้านซ้ายของตัวแปรในสเตตเมนต์กำหนดค่า จะหมายถึง content ของตัวแปรนั้น เช่น

X = 100 สมมติให้ xอยู่ที่ address 98

*X = 10



หากพบคำสั่ง * อยู่ทางด้านขวาเช่น

X = 100

Y = *X + 1

สมมติให้ X,Y อยู่ที่ address 96 และ และ 98 ตามลำดับ และที่ address 100 มี content เท่ากับ 1

address 96	100	X=100
address 98	2	
address 100	1	

ค่าของ content ที่ content ของ X ชี้อยู่ (*X) จะหมายถึง 1

Y = *X จะหมายถึง

```
L-VALUE     Y
R-VALUE     X
FETCH
SET
```

ประโยชน์ของ & ,* จะใช้เมื่อจำเป็นต้อง call by reference เพื่อใช้ในการส่งค่า parameter ออกมาสู่นอกโปรแกรมย่อย เช่น

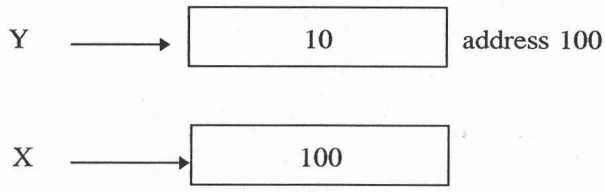
```
function inc(X)
{
        *X = *X + 1
}
run
{
        y = 10
        inc(&y)
}
```

โปรแกรมนี้มีจุดประสงค์ให้ passing parameter Y ภายหลังการเรียกฟังก์ชันย่อยมีค่าเพิ่มขึ้น(จาก 10 ไปเป็น 11)

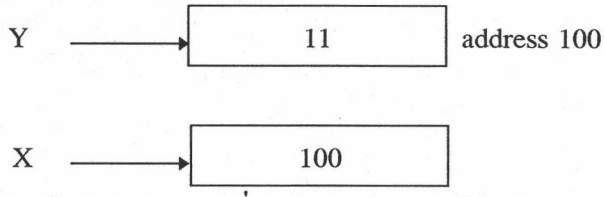
เมื่อแปลฟังก์ชันเป็นรหัสกลาง จะได้

```
1:      R-VALUE     X
2:      R-VALUE     X
3:      FETCH
4:      LITERAL     1
5:      PLUS
6:      SET
```

เมื่อทำการ execution inc(&Y) จะได้ ค่า passing parameter x จะถูก copy จาก &Y คือค่า 100 ดังรูป



เมื่อดำเนินการ บรรทัดที่ 1 และ 2 จะหมายถึงค่า content ของ X คือ 100 (เท่ากับaddress ของ Y) เมื่อพบบรรทัดที่ 3 จะ fetch ค่าจาก address 100 จะได้ค่า 10 ในบรรทัดที่ 4-5-6 จะเพิ่มค่าและ set ค่าที่ address 100 เป็นค่า 11 ดังรูป



ค่าที่เปลี่ยนแปลงคือค่า content ที่ address 100 (ตัวแปร Y) ซึ่งเมื่อสิ้นสุดการเรียกฟังก์ชันค่า x จะหายไป แต่ในขณะที่ค่า y จะยังคงมีค่าเท่ากับ 11

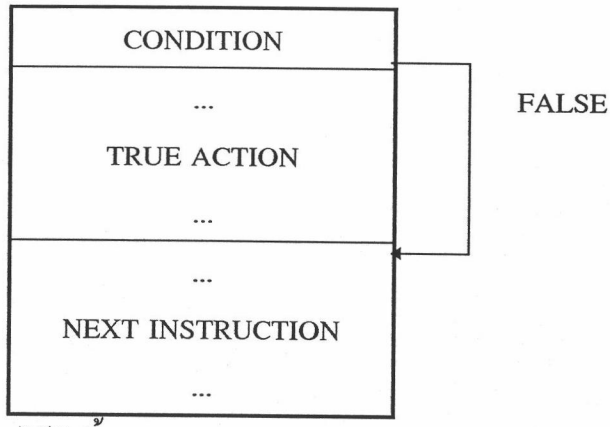
4.5.2 สเตตเมนต์เงื่อนไข (conditional statement)

ในภาษาที่กำหนด เรากำหนดให้มีคำสั่ง IF เป็น สเตตเมนต์เงื่อนไข ซึ่งคำสั่ง IF มีไวยากรณ์ 2 แบบ ดังนี้ คือ

ก ร ณี ที่ 1

IF (condition)
true action

โดย condition ที่ใช้ในการทดสอบ จะเป็น expression ที่ให้ผลลัพธ์เป็นค่าจำนวนจริง ซึ่งจะนำไปใช้ทดสอบว่า ถ้า condition ที่ได้ไม่เท่ากับศูนย์ จะทำ true action ดังรูป



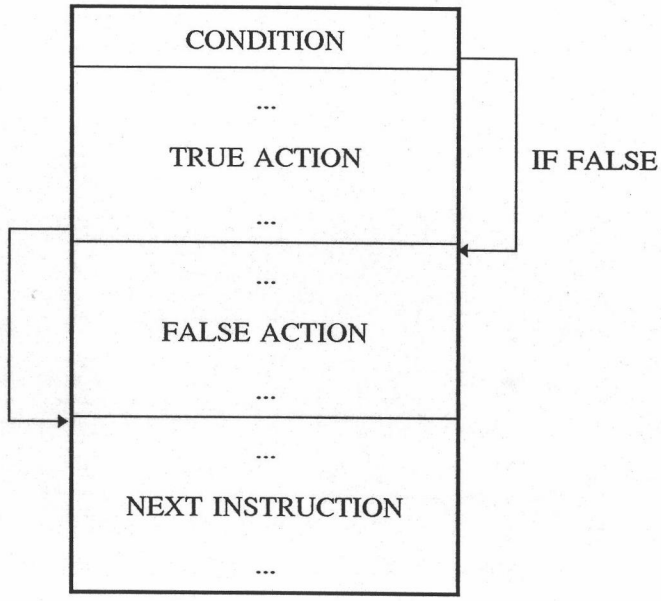
เมื่อแปลเป็นรหัสไบท์ จะได้ดังนี้

```
condition
JZ  next instruction          // Jump if zero
true action
next instruction
```

กรณีที่ 2

```
IF  ( expression )
        true action
ELSE
        false action
```

ถ้า condition ไม่เท่ากับศูนย์ จะทำ true action หากมิฉะนั้น จะทำ false action

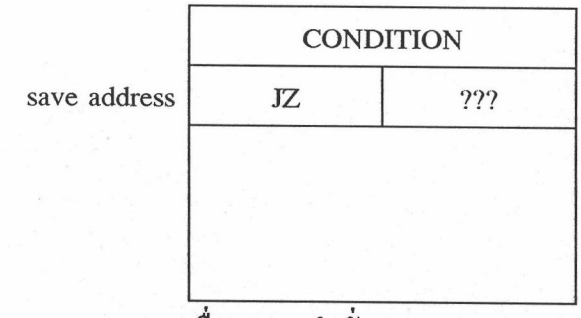


เมื่อแปลเป็นรหัสไบท์ จะได้ดังนี้

```
condition
JZ false action                    // Jump if zero
true action
jump next instruction              // jump end
false action
next instruction
```

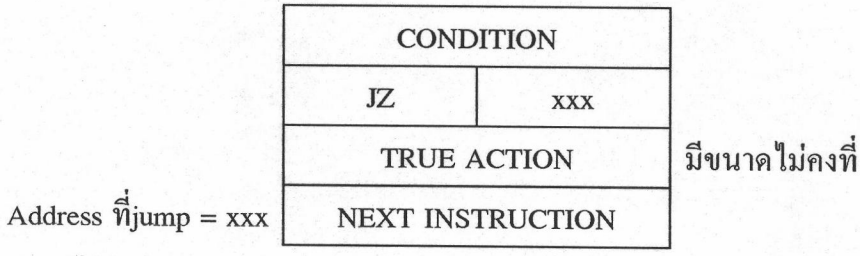
อย่างไรก็ตามในการ JUMP ทั้ง 2 กรณี คือ JUMP, JZ เราจะพบว่าในการสร้าง byte code จาก parse tree ยังไม่สามารถระบุตำแหน่งของแอดเดรส ในการ jump ได้ เพราะว่า จะไม่ทราบความยาวของ byte code ในส่วนที่เป็นความยาวของ true action หรือ ความยาวของ byte code ของ false action ดังนั้นในการสร้างรหัสต้องทำการสร้าง byte code ของคำสั่ง JUMP,JZ และเตรียมช่องว่างสำหรับแอดเดรส ที่ JUMP ไว้ก่อน และเก็บ save address ไว้ เพื่อเก็บค่าตำแหน่งของคำสั่ง JUMP ไว้ก่อน จากนั้นเมื่อ generate code ในส่วนของ true action และ false action เราจะทราบตำแหน่งของการ jump ที่แท้จริง~และจะย้อนกลับไปปรับปรุงตำแหน่งที่ใช้ JUMP โดยใช้ save address ที่เก็บไว้

กรณีที่ 1



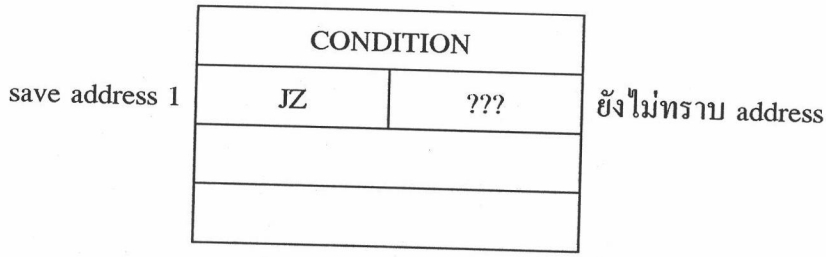
เมื่อ parse คำสั่ง IF (expression)

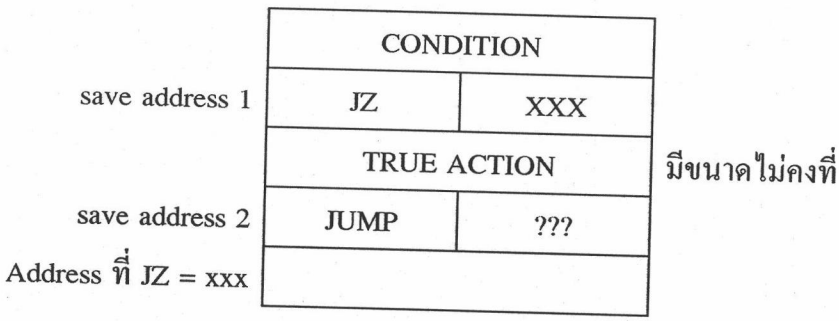
~

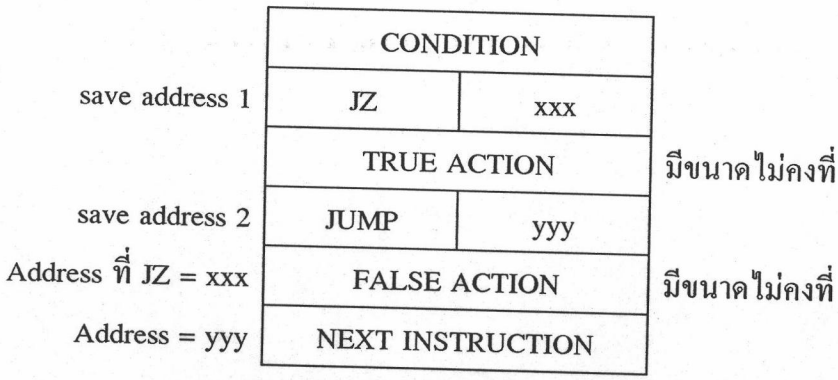


เมื่อ parse คำสั่ง true action ก็จะสามารถระบุตำแหน่งของการ jump ได้ และจะย้อนกลับไปยัง save address เพื่อบันทึกตำแหน่งที่ต้องการ jump

กรณีที่ 2







4.5.3 สเตตเมนต์ทำซ้ำ (Repetition statement)

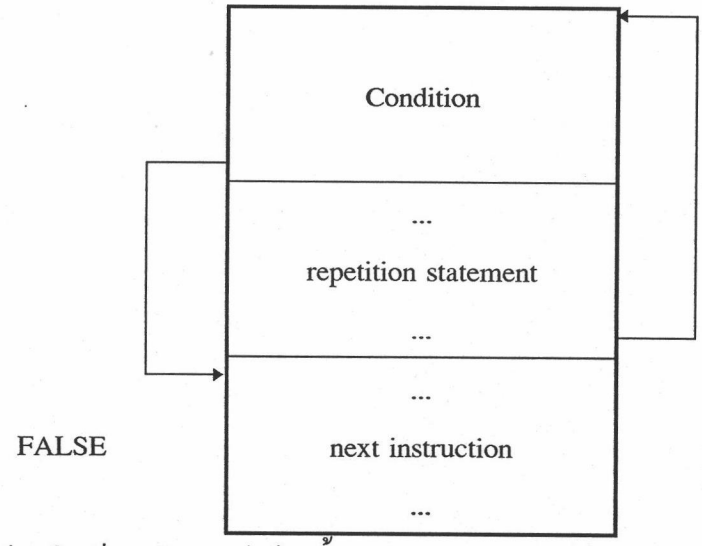
WHILE จะใช้ในการทำซ้ำ โดยคำสั่ง WHILE จะมีไวยากรณ์ดังนี้คือ

WHILE (condition)

statement ;

โดย condition ที่ใช้ในการทดสอบ จะเป็น expression ที่ให้ผลลัพท์เป็นค่าจำนวนจริง ซึ่งจะนำไปใช้ทดสอบว่า ถ้า condition ที่ได้ไม่เท่ากับศูนย์ จะทำการประมวลผลสเตตเมนต์

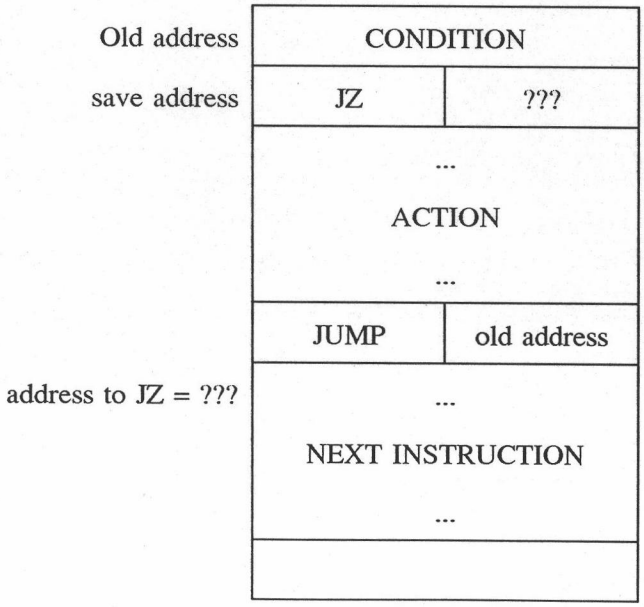
และจะย้อนกลับไปตรวจสอบเงื่อนไขใหม่ หรือหากพบว่า CONDITION เท่ากับศูนย์ จะดำเนินการที่คำสั่งถัดไป



ซึ่งเขียนในเป็นรหัสกลางได้ดังนี้

```
condition
JZ  next instruction          // jump if zero
repetition statement
Jump to condition
next instruction
```

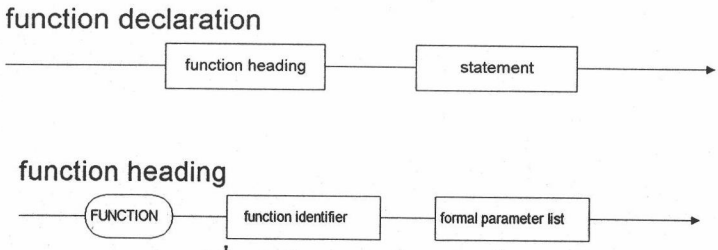
การ JZ จะเกิดปัญหาเช่นเดียวกับ if statement เพราะยังไม่ทราบตำแหน่งของ JZ จึงต้องใช้เทคนิค Back patch ในการบันทึกตำแหน่งคำสั่ง jump จะใช้ตำแหน่งของ instruction ก่อนการพบคำสั่ง WHILE



4.6 การแปลฟังก์ชัน (parsing function)

4.6.1 ส่วนประกาศฟังก์ชัน (function declaration)

ฟังก์ชันเป็นขั้นตอนในการกำหนดให้โปรแกรมสามารถแยกปัญหาเป็นส่วนย่อยๆ ในการประกาศฟังก์ชัน (function declaration) จะมีส่วนประกอบสองส่วนคือ ส่วนหัวของฟังก์ชัน และส่วนของสเตตเมนต์ ดังรูป



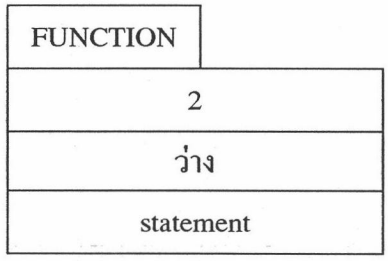
รูปที่ 4.4 ไวยากรณ์ของฟังก์ชัน

ซึ่งส่วนหัวของฟังก์ชันจะประกอบด้วยคำสงวน FUNCTION และตามด้วยฟอร์มอลพารามิเตอร์ (formal parameter) โดยฟอร์มอลพารามิเตอร์คือชุดของตัวแปรที่กำหนดขึ้นในส่วนประกาศของฟังก์ชัน ในฟังก์ชันหนึ่งอาจจะมีฟอร์มอลพารามิเตอร์หลายตัวหรือไม่มีเลยก็ได้

เมื่อคอมไพเลอร์พบคำสงวน FUNCTION จะทำการอ่านชื่อของฟังก์ชันและตรวจสอบกับตัวระบุ (identifier) อื่นในตารางสัญลักษณ์ส่วนกลาง (global scope) โดยชื่อของฟังก์ชันจะต้องไม่ซ้ำกับตัวระบุที่ได้ประกาศไว้แล้วก่อนหน้านี้ หากชื่อฟังก์ชันไม่ซ้ำกับตัวระบุอื่นใด คอมไพเลอร์จะเพิ่มฟังก์ชันนี้เข้าไปในตารางสัญลักษณ์จากนั้นจะนับจำนวนฟอร์มอลพารามิเตอร์ หากกำหนดฟังก์ชันในโปรแกรมต้นฉบับดังนี้

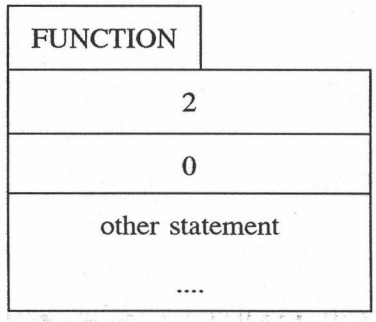
```
FUNCTION max (a,b)
{
        IF (a > b)
          RETURN  (a)
        ELSE
          RETURN (b);
}
```

เมื่อคอมไพเลอร์พบคำสั่ง FUNCTION max(a,b) จะสร้างรหัสกลาง FUNCTION และจะเขียนจำนวนของตัวแปรฟอร์มอลพารามิเตอร์ ซึ่งในตัวอย่างนี้มีสองตัวคือ a และ b แต่ในขั้นตอนนี้ยังไม่ทราบถึงจำนวนตัวแปรเฉพาะที่ ที่ปรากฏอยู่ในฟังก์ชันนี้ ดังนั้นจะสำรองพื้นที่ไว้สองไบต์เพื่อใช้ในการเก็บจำนวนตัวแปรเฉพาะที่



จำนวนของฟอร์มอลพารามิเตอร์
สำหรับจำนวนตัวแปรเฉพาะที่

ในขั้นตอนต่อมาคอมไพเลอร์สร้างตารางสัญลักษณ์ตารางใหม่ เพื่อใช้เก็บตัวแปรเฉพาะที่ และจะแปลส่วนของสเตตเมนต์ของฟังก์ชันที่เหลือจนจบ และจะย้อนกลับไปบันทึกจำนวนตัวแปรเฉพาะที่ในตำแหน่งที่ได้สำรองไว้



จำนวนของฟอร์มอลพารามิเตอร์
จำนวนตัวแปรเฉพาะที่

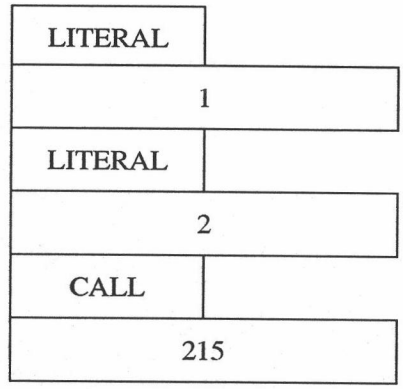
4.6.2 การคืนค่าของฟังก์ชัน (function return)

คำสั่ง RETURN จะมีสองรูปแบบคือ การคืนค่าแบบมีการส่งค่ากลับ และ คืนค่าแบบไม่มีการส่งค่ากลับ การแปลคำสั่งนี้คอมไพเลอร์จะต้องตรวจสอบว่าหลังคำสงวน RETURN เป็นนิพจน์หรือไม่ หากเป็นนิพจน์จะแปลนิพจน์จนเสร็จและปิดท้ายด้วยคำสั่ง RET1 หากไม่ใช่นิพจน์แปลเป็นรหัสกลาง RET0 และหากผู้ใช้ไม่กำหนดคำสั่ง return ไว้ที่คำสั่งสุดท้ายของสเตตเมนต์ คอมไพเลอร์จะปิดท้ายฟังก์ชันด้วยคำสั่ง RET0 เพื่อให้อินเตอร์พรีเตอร์รู้ว่าต้องกลับไปทำงานต่อในโปรแกรมที่เป็นผู้เรียกมา

4.6.3 การเรียกฟังก์ชัน (function call)

ในการเรียกใช้ฟังก์ชัน เช่น max (1,2); เมื่อสแกนเนอร์พบโทเกนสตริง max ซึ่งเป็นตัวระบุ (identifier) คอมไพเลอร์จะตรวจดูในตารางสัญลักษณ์ว่าเป็นตัวระบุประเภทใด เมื่อพบว่าเป็นชื่อของฟังก์ชัน จะอ่านค่าแอดเดรส และจำนวนของฟอร์มอลพารามิเตอร์ จากนั้นจะแปลนิพจน์ที่ปรากฏในฟังก์ชัน นิพจน์เหล่านี้เรียกว่า พารามิเตอร์จริง (actual parameter) และจะปิด

ท้ายด้วยรหัสกลาง call ตามด้วยแอดเดรสของฟังก์ชันเช่นสมมติว่าฟังก์ชันนี้อยู่ที่ตำแหน่ง 215 จะได้รหัสกลางดังนี้

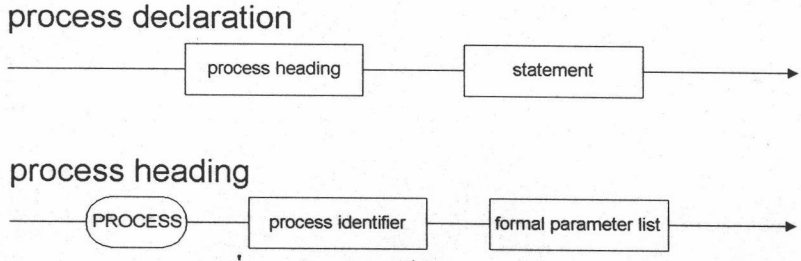


ตำแหน่งของฟังก์ชัน max

4.7 การแปลกระบวนการ (parsing process)

4.7.1 การประกาศประบวนการ (declaration process)

ประกาศกระบวนการ (process declaration) จะมีส่วนประกอบสองส่วนคือ ส่วนหัวของกระบวนการและส่วนของสเตตเมนต์ มีไวยากรณ์ดังนี้

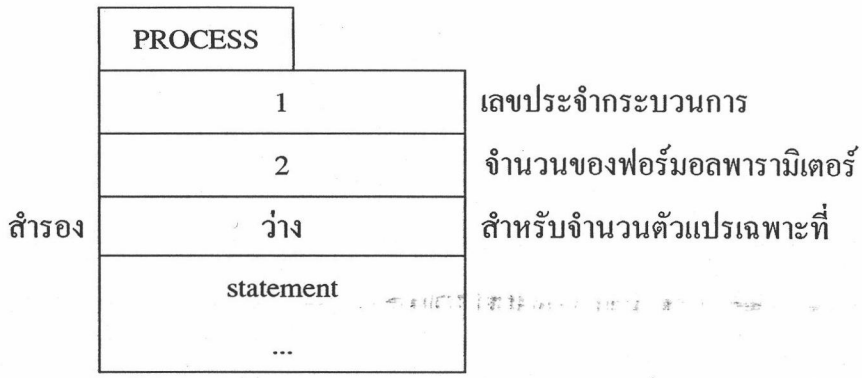


รูปที่ 4.5 ไวยากรณ์ของกระบวนการ

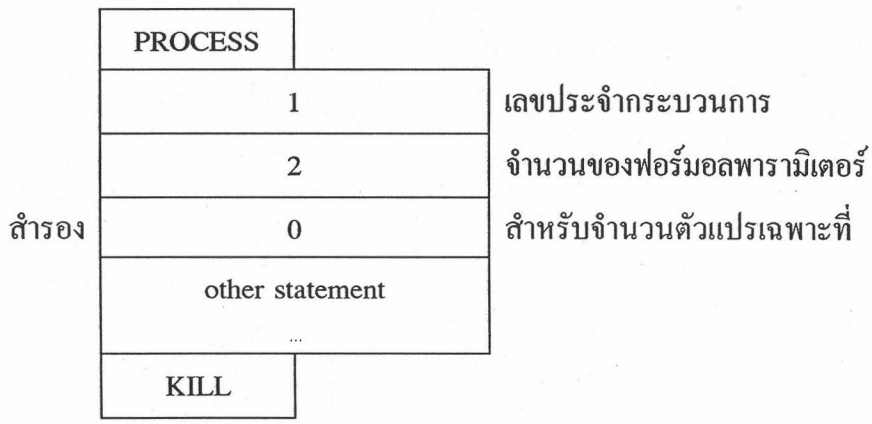
รหัสกลางของกระบวนการจะมีลักษณะคล้ายฟังก์ชันเป็นอย่างมาก แต่กระบวนการจะมีส่วนของเลขประจำกระบวนการเพิ่มเข้ามา โดยเลขประจำกระบวนการจะมีค่าเริ่มต้นตั้งแต่ค่าหนึ่งและกระบวนการอื่นที่เพิ่มเข้ามาในภายหลังจะมีค่าเพิ่มขึ้นตามลำดับ หากกำหนดกระบวนการในโปรแกรมต้นฉบับดังนี้

```
PROCESS P1(a,b) {
        IF (a > b) DoSomething();
}
```

เมื่อคอมไพเลอร์พบคำสั่ง PROCESS P1(a,b) จะสร้างรหัสกลาง



และเมื่อแปลสเตตเมนต์เสร็จแล้วจะย้อนมาบันทึกจำนวนตัวแปรเฉพาะที่ และจะปิดท้ายด้วยรหัสกลาง kill เพื่อให้อินเตอร์พรีเตอร์ทราบว่าจบกระบวนการแล้ว และลบกระบวนการนี้ออกจากระบบ



4.7.2 ฟังก์ชันอ่านค่าเลขกระบวนการ (function GetPID)

เป็นคำสั่งฟังก์ชันของระบบ (system function call) บรรจุไว้ในคอมไพเลอร์ ซึ่งทำหน้าที่ในการอ่านค่าเลขประจำกระบวนการ เมื่อพบคำสั่งนี้จะคืนค่าหมายเลขประจำกระบวนการให้กับอินเตอร์พรีเตอร์ ไวยากรณ์ของฟังก์ชันนี้จะมีพารามิเตอร์คือชื่อของกระบวนการ ดังนี้

GetPID (ProcessName);

แต่เนื่องจากเรากำหนดให้สามารถกำหนดสามารถเรียกใช้ฟังก์ชันนี้ ก่อนการประกาศกระบวนการได้ ดังนั้นหากมีการเรียกฟังก์ชันหลังการประกาศกระบวนการ จะสามารถรู้ค่าเลขกระบวนการได้ทันทีโดยค้นจากตารางสัญลักษณ์ แต่ในกรณีที่เรียกฟังก์ชันนี้ก่อนการประกาศกระบวนการ คอมไพเลอร์จะเพิ่มชื่อและเลขกระบวนการเข้าไปในตารางสัญลักษณ์ แต่จะยังไม่บันทึกค่าอื่นๆเช่นตำแหน่งของกระบวนการ, ฟอร์มอลพารามิเตอร์และจำนวนตัวแปรเฉพาะที่ จนกระทั่งคอมไพเลอร์พบส่วนประกาศของฟังก์ชันจึงจะปรับปรุงข้อมูลในตารางสัญลักษณ์อีกครั้ง

คำสั่ง GetPID จะช่วยให้สามารถอ้างอิงถึงกระบวนการในโปรแกรมได้โดยไม่จำเป็นต้องกำหนดเป็นค่าคงที่ ทั้งนี้เนื่องจากเลขประจำกระบวนการจะถูกสร้างเองในขั้นตอนการแปลโปรแกรม เช่นในกรณีที่ต้องการติดต่อระหว่างโปรแกรมโดยใช้คำสั่ง send ซึ่งมีไวยากรณ์คือ

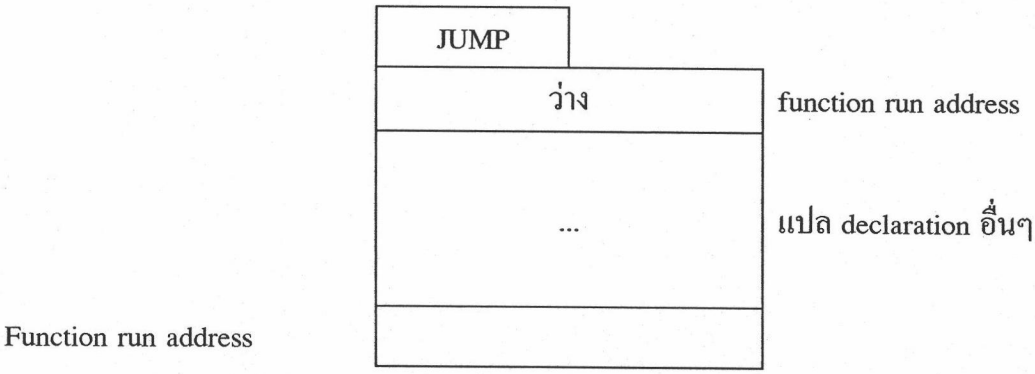
send (PID , Message);

โดยที่ PID คือเลขกระบวนการที่ต้องการติดต่อ และ Message คือ เลขจำนวนเต็มที่ต้องการส่ง ยกตัวอย่างเช่นเมื่อต้องการส่งค่า 0 ไปให้กระบวนการชื่อ process1 จะเขียนดังนี้

send (GetPID(process1) , 0);

4.8 การแปลฟังก์ชันรัน (run function)

ฟังก์ชันดำเนินการจะเป็นฟังก์ชันสุดท้ายที่ปรากฏอยู่ในโปรแกรมต้นฉบับ แต่จะเป็นกระบวนการแรกที่ถูกประมวลผล ดังนั้นในคำสั่งแรกของรหัสกลางจะเป็นคำสั่งที่กระโดดไปยังตำแหน่งที่ต้องการ แต่ในขณะที่ทำการคอมไพล์โปรแกรมจะต้องแปลส่วนประกาศ (declaration part) อื่นๆก่อน ดังนั้นจะต้องสำรองที่ว่างไว้สำหรับตำแหน่งของฟังก์ชันดำเนินการ



นอกจากนี้ในการกำหนดตัวแปรแบบอะเรย์ ซึ่งจะเป็นตัวแปรที่จัดเก็บอยู่ในเซกเมนต์ข้อมูล (ซึ่งจะกล่าวถึงโดยละเอียดในบทที่ห้า) การกำหนดค่าเริ่มต้นในกับอะเรย์ คอมไพเลอร์จะแทรกรหัสกลางเพื่อกำหนดค่าให้กับอะเรย์ โดยแทรกรหัสเพื่อกำหนดขนาดของอะเรย์

L-valueGV *address of array descriptor*
Literal *dimension of array*
Set GV
...

จากนั้นจึงจะแบบสเตตเมนต์อื่นๆในฟังก์ชันดำเนินการต่อไป

ทั้งหมดนี้คือขั้นตอนในการพัฒนาคอมไพลเลอร์เพื่อแปลภาษาต้นแบบไปเป็นภาษาเป้าหมายคือรหัสกลางซึ่งอยู่ในรูปรหัสไบต์ ซึ่งจะนำไปใช้ในการประมวลผลของอินเตอร์พรีเตอร์ ซึ่งจะอธิบายในบทต่อไป