บทที่ 7

การควบคุม (control)

7.1 คำสั่งงานการ์ด และเงื่อนไข

(Guarded Commands and Conditionals)

7.2 การวนซ้ำ และความหลากหลาย บน WHILE

(Loops and Variations on WHILE)

7.3 การโต้แย้งเกี่ยวกับ ข้อความสั่ง GOTO

(The GOTO Controversy)

7.4 โปรซีเดอร์ และ พารามิเตอร์

(Procedures and Parameters)

บทที่ 7
การควบคุม (Control)

ในบทนี้ เราจะอภิปราย การนิยามนามธรรม ของการควบคุม ผ่านการใช้โครงสร้างควบคุม และ โปรซีเดอร์ (structured control and procedures)

โครงสร้างควบคุม ชุดแรก ในภาษาโปรแกรม คือ GOTOs ซึ่งเลียนแบบ ข้อความสั่งกระโดด ของ ภาษาแอสเซมบลี ย้ายการควบคุมโดยตรง หรือ หลังจาก มีการทดสอบเงื่อนไข ไปยังตำแหน่งใหม่ ใน โปรแกรม

ภาษา Algol60 มีการปรับปรุง ให้เป็น **การควบคุมเชิงโครงสร้าง** (structured control) ซึ่งข้อความสั่งควบคุม ย้าย การควบคุมไปและกลับ จากลำดับ ของ ข้อความสั่งต่างๆ ซึ่งเป็น ทางเข้าหนึ่งทาง ทางออกหนึ่งทาง (single entry, single exit) นั่นคือ ลำดับของข้อความสั่งต่างๆ ซึ่งเข้าที่จุดเริ่มต้น และออกเมื่อจบ

ตัวอย่าง ตัวสร้างชนิด ทางเข้าหนึ่งทาง ออกหนึ่งทาง (single-entry, single exit constructs) ได้แก่ **บล็อก** (blocks) ของภาษา Algol60, Algol68, C และ Ada ซึ่งรวมการประกาศไว้ด้วย

ส่วน ข้อความสั่งประกอบ (compound statements) ของภาษา Pascal และ ลำดับข้อความสั่ง ของภาษา Modula-2 มีคุณสมบัติ ทางเข้าหนึ่งทาง และทางออกหนึ่งทาง เช่นกัน (แต่ ทั้งหมดนี้ ไม่รวม การประกาศ)

การเขียนโปรแกรมเชิงโครงสร้าง (structured programming) นำไปสู่ การปรับปรุงอย่างมากมาย ใน เรื่องการอ่านง่าย และ ความเชื่อถือได้ ของ โปรแกรม และตัวสร้างควบคุมเชิงโครงสร้างต่างๆ คือส่วนของ ภาษาหลักๆ ส่วนใหญ่ ในทุกวันนี้

บางภาษา ตัด GOTOs ออกไป ถึงแม้ว่าทุกวันนี้ยังมีพิสัยของการ โต้เถียงกัน เรื่องประโยชน์ของ GOTOs ภายในบริบท (context) ของ การเขียนโปรแกรมเชิงโครงสร้าง ในบทนี้สิ่งแรก จะอภิปรายเรื่อง**กลไกการควบคุมเชิงโครงสร้าง** (structured control mechanism) จากนั้นจะสรุป การใช้ ข้อความสั่ง GOTO

การขยาย ความคิด ของ บล็อก คือ โปรซีเดอร์ หรือ ฟังก์ชัน เป็น บล็อก ซึ่ง การกระทำการ ถูก อ้างถึง และ การเชื่อมประสานของมัน กำหนดไว้อย่างชัดเจน สิ่งเหล่านี้ เป็นโครงสร้างด้วยการนิยาม ทางเข้าและทางออก อย่างชัดเจน

หัวข้อสุดท้าย ซึ่งจะอภิปราย ในบทนี้ คือ สถานะการณ์ ซึ่งย้าย การควบคุม **ต้องเป็น** preempted : การจัดกระทำข้อยกเว้น สิ่งนี้ เป็น สถานะการณ์ ซึ่ง **ซับซ้อนมากกว่า เพราะว่า ข้อ** ยกเว้น เป็นสาเหตุให้ normal flow ของ การควบคุม ถูกรบกวน

7.1 <u>คำสั่งงานการ์ด และเงื่อนไข</u>

(Guarded Commands and Conditionals)

รูปแบบปกติส่วนใหญ่ของการควบคุมโครงสร้าง ได้แก่ การกระทำการ กลุ่มของ ข้อความสั่งเฉพาะ ภายใต้ เงื่อนไขที่แน่นอน

(The most typical form of stryctured control is execution of a group of statements only under certain conditions.)

สิ่งนี้ เกี่ยวข้องกับ การทดสอบ แบบบูล หรือ แบบตรรกะ **ทดสอบก่อน** การเข้าไปยัง ลำดับของ ข้อความสั่งต่างๆ ตัวสร้างซึ่งเป็นที่นิยม ได้แก่ ตัวสร้าง **if-then-else** หมายถึง รูปแบบ ร่วม ส่วนใหญ่ ของ ตัวสร้างนี้

วิธีต่างๆ เช่น เงื่อนไข จะนำมาอภิปรายอย่างสั้นๆ

อันดับแรก เราต้องการอธิบาย รูปแบบทั่วไป ของ **ข้อความสั่งมีเงื่อนไข ซึ่งรวม ตัวสร้าง** แบบมีเงื่อนไขต่างๆ ทั้งหมด : (a general form of conditional statement that encompasses all the various conditional constructs :) ข้อความสั่ง **<u>guarded if</u>** **ผู้พัฒนาขึ้นมา** คือ E.W. Dijkstra เขียนดังนี้

```
if  B1 → S1
 |  B2 → S2
 |  B3 → S3

 |  Bn → Sn
fi
```

ความหมาย ของ ข้อความสั่งข้างต้นนี้ คือ :

Bi's ทุกตัวเป็นนิพจน์แบบบูล เรียกว่า **<u>การ์ด</u>** (guards)

Si's หมายถึงลำดับของ ข้อความสั่ง (are statement sequences)

ถ้า Bi ตัวใดตัวหนึ่ง ถูกประเมินผลแล้วมีค่าเป็นจริง ลำดับของข้อความสั่ง Si ซึ่งสมนัยกัน จะถูก กระทำการ (If one of the Bi's evaluates to true, then the corresponding statement sequence Si is executed.)

ถ้ามี Bi's มากกว่าหนึ่งตัว ที่มีค่าเป็นจริง แล้ว จะมี Si's ซึ่งสมนัย กัน เพียงหนึ่งชุดเท่านั้น ถูกเลือก ให้กระทำการ (If more than one of the Bi's is true, then one and only one of the corresponding Si's is selected for execution.)

ถ้า Bi's ทุกตัว มีค่าเป็นเท็จหมด จะเกิดข้อผิดพลาดขึ้น

(If none of the Bi's is true, then an error occurs.)

มีคุณสมบัติ ซึ่งน่าสนใจ หลายอย่าง ในคำอธิบายนี้

ข้อแรก คำอธิบาย ไม่ได้พูดว่า Bi ตัวแรก ซึ่ง ประเมินผลแล้ว ได้ค่าเป็นจริง จะเป็นตัวถูกเลือก ดังนั้น guarded if จึงเป็นการแนะนำชนิด **การไม่กำหนด (nondeterminism)** ให้กับการเขียนโปรแกรม คุณสมบัตินี้ กลายเป็น สิ่งที่ เป็นประโยชน์มาก ใน การเขียนโปรแกรมพร้อมกัน (concurrent programming)

ข้อที่สอง มันทิ้งไว้โดยไม่ได้ระบุว่า guards ทั้งหมด จะถูกประเมินผล หรือไม่ ดังนั้น ถ้าการประเมินผลของ Bi หนึ่งตัว มี side effect ผลลัพธ์ ของการกระทำการ guarded if อาจจะไม่ทราบค่า (be unknown) ซึ่งการทำให้เกิดผลเชิงกำหนดโดยปกตินั้น ข้อความสั่งเช่นนี้ จะประเมินผล Bi's อย่างสืบเนื่อง จนกระทั่งพบว่า มี Bi หนึ่งตัว มีค่าเป็นจริง จากนั้น Si ซึ่งสมนัยกัน จะถูกกระทำการ และการควบคุม เปลี่ยนไปยังจุด ซึ่ง ตามหลัง ข้อความสั่ง guarded

วิธีหลัก 2 วิธี ซึ่ง ภาษาโปรแกรม implement ข้อความสั่ง แบบมีเงื่อนไข เหมือนกับ guarded if ได้แก่ ข้อความสั่ง if และ ข้อความสั่ง case

7.1.1 **ข้อความสั่ง if** (If -statements)

รูปแบบพื้นฐาน ของ ข้อความส่ง if ถูกกำหนด ใน กฎ EBNF ของ Pascal's ดังนี้

<if-statement> ::= if<Boolean-expresion> then<statement> [else <statemnt>]

เมื่อ

<statement> อาจจะเป็น single statement หรือ a sequence of statements ปิดล้อมด้วยคู่ begin-end

ตัวอย่าง

```
if x <> 0.0 then y := 1.0/x
```

```
else  begin
          x := 2.0;
          y := 1.0/z
      end;
```

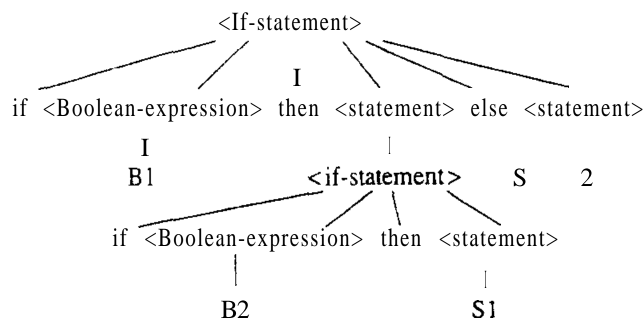
รูปแบบนี้ ของ if (ซึ่งมีอยู่ในภาษา Algol และ C) มีปัญหา คือ มันกำกวม ใน แง่วากยสัมพันธ์ ดังที่ได้อธิบายมาแล้ว ในบทที่ 4

ข้อความสั่ง ข้างล่างนี้

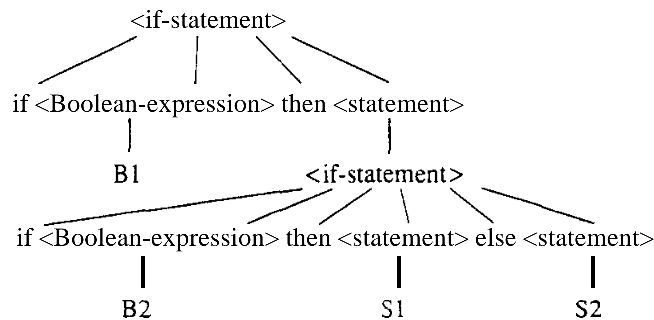
```
if B1 then if B2 then S1 else S2
```

มีต้นไม้วิภัชได้สองรูป ซึ่งแตกต่างกัน เป็นไปตามกฎ BNF :

รูปที่ 1



และ **รูปที่ 2**



ความกำกวมเช่นนี้ เรียกว่า **dangling-else** problem ในวากย**สัมพันธ์ ของ**ภาษา Pascal และ Algol60 ไม่ได้บอกว่า else ใน **ข้อความสั่ง** if **ข้างต้นนี้ เกี่ยวข้องกับ** if **ตัวแรก หรือ if ตัวที่**สอง ภาษา Pascal แก้ปัญหา โดยกล่าวไว้ใน **กฎไม่กำกวม** (disambiguatting rule) **ดังนี้**

ในกรณีที่ ส่วนที่เป็น else มีไม่ครบตามจำนวน if ให้จับคู่ else กับ if **ตัวใกล้ที่สุด ซึ่งยังไม่มีส่วนของ** else (the else is to be associated with the closest if that does not already have an else part.) กฎนี้ มีชื่อเรียกอีกอย่างหนึ่งว่า **กฎซ้อนในใกล้ที่สุด (the most closely nested rule)** สำหรับ ข้อความสั่ง if ซึ่งหมายความว่าในตัวอย่างข้างต้นนั้น ต้นไม้วิภัช รูปที่สอง เป็นชุดที่ถูกต้อง

ปัญหา dangling-else หมายถึง ปัญหา ของการออกแบบภาษา ซึ่งมองได้สองจุดคือ

(1) มันทำให้ ต้องมีการตั้งกฎใหม่ เพื่ออธิบายว่า อะไรคือคุณสมบัติเชิงวากยสัมพันธ์ ที่สำคัญ คืออะไร

(2) มันทำให้ ผู้อ่าน ตีความหมาย ของ ข้อความสั่ง if ยากมากขึ้น นั่นคือ มันฝ่าฝืน หลักเกณฑ์ ในการออกแบบภาษา ที่ว่า อ่านง่าย (readability) เช่นที่แสดงให้เห็นในตัวอย่างข้างต้น ถ้าเราต้องการ จับคู่ else ให้กับ if ตัวแรก จะต้องเขียน ในรูปแบบ อย่างใดอย่างหนึ่ง ดังนี้

```
if B1 then begin if B2 then S1 end else S2
```

หรือ

```
if B1 then if B2 then S1 else else S2
```

ยังมีวิธีอื่นๆ ให้ใช้ แก้ปัญหา dangling-else อีก นอกเหนือไปจาก กฎความไม่กำกวม จริงๆ แล้ว มันเป็นไปได้ ที่จะเขียน กฎ BNF ซึ่งกำหนด การสลับที่อย่างถูกต้อง แต่กฎเหล่านี้ ซับซ้อน วิธีซึ่งดีกว่า คือ ใช้ **<u>คำหลักปิดกลุ่ม</u> (bracketing keyword)** สำหรับข้อความสั่ง if ตัวอย่างเช่น กฎของภาษา Algol68 เขียนดังนี้

```
<if-statement> ::= if <Boolean-expression> then <statement> [else <statement>] fi
```

คำว่า fi สะกดคำย้อนกลับ เพื่อใช้ปิดข้อความสั่ง if และลบทิ้งความกำกวม เพราะว่า ขณะนี้ จากตัวอย่างข้างต้น สำหรับต้นไม้วิภัชชุดที่หนึ่ง จะต้องเขียนข้อความสั่ง if ดังนี้

```
if B1 then if B2 then S1 fi else S2 fi
```

ส่วน ต้นไม้วิภัช ชุดที่สอง เขียนข้อความสั่ง if ดังนี้

```
if B1 then if B2 then S1 else S2 fi fi
```

เพื่อบอกว่า if ตัวใด คู่กับ ส่วนที่เป็น else สิ่งนี้ จึงไม่มีความจำเป็น ต้องใช้ คู่ begin-end เพื่อเปิด ลำดับชุดใหม่ ของ ข้อความสั่ง

ข้อความสั่ง if สามารถเปิดลำดับ ของ ข้อความสั่ง ของมันเอง และกลายเป็น โครงสร้างที่บริบูรณ์ ดังนี้

ตัวอย่าง

```
if x > 0.0 then
        y := 1.0/x;
    done: = true:
else
        x := 1.0:
        y := 1.0/z;
    done := false:
fi
```

ภาษา Ada มีวิธีการเข้าถึง คล้ายกัน โดย มีคำสงวน สองคำ i end if ในสำหรับปิด ข้อความสั่ง if

ภ าษา FORTRAN77 คำหลักปิดกลุ่มที่ใช้ คือ ENDIF

ภาษา Modula-2 คำหลักปิดกลุ่ม คือ END

ตัวอย่าง ภาษา Modula-2

```
IF x> 0.0 THEN
        y := 1.0/x;
    done := true:
ELSE!
        x := 1.0;
        y := 1.0/z;
    done := false;
END
```

ในการแก้ปัญหาของภาษา Modula-2 ทำให้เกิดปัญหาใหม่ขึ้น กล่าวคือ คำสงวน END นั้น ใช้ จบ (terminate) โครงสร้างอื่นๆ จำนวนมากเช่นกัน ดังนั้น วากยสัมพันธ์ ที่ทำให้ การใช้ END เพื่อการจบโครงสร้าง จึงไม่ง่าย ส่วนภาษา Algol68 และภาษา Ada มีผลเฉลยที่ดีกว่าเล็กน้อย

ส่วนขยายของ ข้อความสั่ง if เมื่อมีทางเลือกจำนวนมาก(many alternatves) ทำให้ง่ายขึ้น ภาษา Modula-2 อาจจะมีการเขียน else หลายๆ ตัว ดังนี้

```
IF B1 THEN
   S1
ELSE
   IF B2 THEN
      s2
   ELSE
      IF B3 THEN
         s3
      ELSE
         S4
      END
   END
END
```

โดยมี END จำนวนมาก มากองอยู่ที่ตอนจบ เพื่อปิด IF ทั้งหมด แต่การใช้ ELSE IF ถูกทำให้รัดกุมขึ้น โดยใช้คำสงวน ตัวใหม่ ELSIF แทน เพื่อเปิด ลำดับชุดใหม่ ของ ข้อความสั่ง ที่ระดับเดียวกัน ตัวอย่างข้างต้น จึงเขียนใหม่ ดังนี้

```
IF B1 THEN
   S1
ELSIF B2 THEN
   S2
ELSIF B3 THEN
   s3
ELSE S4
END
```

ภาษา Ada มี ตัวสร้าง (construct) เหมือนกัน และในภาษา Algol68 elseif เรียกว่า elif

7.1.2 ข้อความสั่ง Case (Case-statements)

ผู้ที่ ประดิษฐ์ ข้อความสั่ง case คือ C.A.R. Hoare ข้อความสั่ง case เป็น guard if พิเศษชนิดหนึ่ง คือแทนที่ guards จะเป็นนิพจน์แบบบูล กับถูกแทนที่ด้วย ค่าเชิงเลข (ordinal values) ซึ่งถูกเลือกโดย นิพจน์เชิงเลข (ordinal expression)

ตัวอย่าง ภาษา Modula-2

```
CONST  n = 5;
VAR    x : [1 . . 10];
CASE  x - 1  OF
   0 :
        y := 0;
        z := 2;
| 2, 3 . . n :
        y := 3;
        z := 1;
| 7, 9 :
        z := 10;
ELSE
   (* do nothing *)
END  (* case *)
```

โปรดสังเกตว่า cases อาจจะเป็น ค่าคงที่ (constants) นิพจน์แบบคงที่ (constant expressions) หรือ พิสัยแบบคงที่ (constant ranges) หรือ รายการของสิ่งเหล่านี้ (ภาษา Pascal มาตรฐาน ตัด พิสัย ออกจาก บทนิยาม ของ ข้อความสั่ง case)

ส่วนที่เป็น ELSE ทั้งหมด ซึ่งจับคู่กัน อาจละเว้นได้ (optional) แต่ถ้า มี case เกิดขึ้น ซึ่งไม่มีรายการไว้ และไม่มีส่วนของ ELSE จะเกิดข้อความผิดพลาดขึ้น และค่าที่เหมือนกัน จะเป็นส่วนของ case specifiers สองชุดไม่ได้

ภาษาสมัยใหม่ ส่วนใหญ่ มี ข้อความสั่ง case คล้ายกับภาษา Modula-2 ด้วยข้อจำกัดคล้ายกัน (cases คาบเกี่ยวกันไม่ได้ ถ้ามี cases ซึ่งไม่ได้กำหนดไว้ เกิดขึ้น จะเป็นข้อผิดพลาด)

ภาษา Pascal ละทิ้ง (omits) else ทั้งหมด ที่จับคู่กัน (catch-all else) โดยบังคับ ให้ cases ทั้งหมด ต้อง มีรายการไว้ (การ implement ส่วนใหญ่ ต้องใส่ไว้ อย่างไรก็ตาม อาจจะเป็น else หรือ otherwise หนึ่งตัว)

ภาษา Algol68 มีโครงสร้างคล้ายกัน แต่ตัวเลือก (selectors) ถูกจำกัด ให้เป็นค่า จำนวนเต็ม (integer values) รายการของ case ไม่จำเป็นต้อง มีทั้งหมด : ถ้า case ที่เกิดขึ้น ไม่มีรายการไว้ การควบคุม จะส่งไปยัง ข้อความสั่ง หลัง (after) ข้อความสั่ง case ไม่ใช่เกิดข้อผิดพลาด

ภาษา C มีความหลากหลาย บน ข้อความสั่ง case เรียกว่า ข้อความสั่ง **switch** เขียนดังนี้

```
switch ( x - 1)
{case 0 :
        y = 0;
        z = 2;
        break;
 case 2 :
 case 3 :
 case 4 :
 case 5 :
        y = 3;
        z = 1;
        break;
 case 7 :
 case 9 :
        z = 10;
        break;
default : ;
        /* do nothing */
}
```

ภาษา Algol68, cases ต่างๆ ต้องเป็น ค่าจำนวนเต็ม (integer-valued) เท่านั้น และต้อง

เขียนรายการแยกต่างหากจากกัน break ใช้สำหรับ ออกจาก (exit) ข้อความสั่ง ถ้าโปรแกรมเมอร์ไม่ต้องการ กระทำการ ข้อความสั่งส่วนที่เหลือทั้งหมดต่อไป สิ่งนี้ทำให้ cases มีขนาดใหญ่มาก เมื่อรวมกับ list ของ case 2 จนถึง case 5 แต่หมายความว่า **break** ต้องนำมาใช้ เพื่อ จบ case แต่ละชุดที่แยกต่างหากจากกัน

ตัวสร้าง if และตัวเลือกอื่นๆ อาจจะเป็นนิพจน์ ซึ่ง ส่งกลับ ค่าได้เช่นเดียวกับ ข้อความสั่ง ในกรณีนี้ มี ข้อจำกัดเรื่องชนิด เพราะว่า ชนิดผลลัพธ์ สำหรับ นิพจน์ จะต้องถูกอ้างถึงได้

<u>ตัวอย่าง</u> นิพจน์ ใน ภาษา Algol68

```
if B then exp1 else exp2 fi
```

ส่งกลับ (returns) ค่าของ exp1 หรือ exp2 ขึ้นอยู่กับค่าของ B ดังนั้น exp1 และ exp2 ต้องเป็นชนิดเดียวกัน หรือ ชนิดที่แทนกันได้ (compatible types) เพื่อให้ **ชนิดของผลลัพธ์** ร่วม สามารถส่งกลับได้

ภาษา C มี นิพจน์แบบมีเงื่อนไข ใส่เพิ่ม ให้กับ ข้อความสั่งแบบมีเงื่อนไข

<u>ตัวอย่าง</u> นิพจน์มีเงื่อนไข ของภาษา C

```
e1 ? e2 : e3
```

เมื่อ e1 ถูกประเมินผล ถ้าไม่ใช่ค่าศูนย์ e2 จะถูกประเมินผล และค่าส่งกลับของมัน คือ ค่าของนิพจน์

กรณีอื่นๆ (otherwise) คือ ถ้า e1 ถูกประเมินผลแล้ว มีค่าเป็นศูนย์ ค่าของ e3 จะถูกส่งกลับ

ในที่นี้ นิพจน์ e2 และ e3 ต้องเป็นชนิดที่แทนกันได้ ชนิดของผลลัพธ์ คำนวณจาก **กฎการแปลงรูป** (type conversion rules) ของภาษา C

7.2 <u>การวนซ้ำ และความหลากหลายของ WHILE</u>

(Loops and Variations on WHILE)

การวนซ้ำ และการใช้การวนซ้ำ เพื่อกระทำ การดำเนินการซ้ำๆกัน (repetitive operations) โดยเฉพาะ การใช้แถวลำดับ เป็นคุณสมบัติที่สำคัญอย่างหนึ่ง ของการเขียนโปรแกรมคอมพิวเตอร์ ตั้งแต่เริ่มต้นแล้ว ในแง่ที่ว่า เครื่องคอมพิวเตอร์ ถูกประดิษฐ์ขึ้นมาเพื่อให้ทำงาน ซึ่งกระทำการดำเนินการซ้ำๆกัน ง่ายขึ้นและเร็วขึ้น รูปแบบทั่วไป สำหรับตัวสร้างการวนซ้ำ (loop construct) ซึ่ง กำหนดโดย โครงสร้างซึ่งสมนัยกับ guarded if ของ Dijkstra เรียกว่า **guarded do** เขียนดังนี้

```
do   B1 → S1
     B2 → S2
I    B3 → s3
     ...
   Bn → Sn
od
```

ข้อความสั่ง ข้างต้นนี้ ทำซ้ำๆกัน จนกระทั่ง Bi's ทุกตัวเป็นเท็จ ที่แต่ละขั้นตอน Bi's ซึ่งเป็นจริงหนึ่งตัว จะถูกเลือก โดยไม่กำหนด และ Si ซึ่งสมนัยกัน จะถูกกระทำการ

(This statement is repeated until all the Bi's are false. At each step, one of the true Bi's is selected nondeterministicaly, and the corresponding Si is executed.)

รูปแบบมาตรฐานของ ตัวสร้างการวนซ้ำ ซึ่งเป็น guarded do ที่สำคัญ จะมีเพียง หนึ่ง guard เท่านั้น (จึงไม่มี nondeterminism)

คือ while-loop ของ ภาษา Algol68

```
while B do S od
```

หรือ while-loop ของ ภาษา Modula-2

```
WHILE B DO S END
```

ใน ข้อความสั่งเหล่านี้ B เป็น นิพจน์แบบบูล ซึ่งจะถูกประเมินผลเป็นอันดับแรก ถ้ามีค่าเป็นจริง ข้อความสั่ง (หรือบล็อก หรือลำดับข้อความสั่ง) S จะถูกกระทำการ จากนั้น B จะถูกประเมินผลอีก ทำเช่นนี้เรื่อยไป โปรดสังเกตว่า ถ้า B เป็นเท็จ ตั้งแต่ต้น แล้ว S จะไม่ถูกกระทำการใดๆ ทั้งสิ้น บางภาษา มี ข้อความสั่งเลือก ซึ่ง ทำให้เชื่อมั่นได้ว่า S จะถูกกระทำการอย่างน้อยที่สุด หนึ่งครั้ง ในภาษา Pascal สิ่งนี้คือ ข้อความสั่ง **repeat** (ภาษา Modula-2 มีข้อความสั่งเหมือนกัน) มีรูปแบบดังนี้

20-2

```
repeat S until B
```

ซึ่ง มีความหมายเหมือนกับ รหัส ต่อไปนี้

```
S
while not B do S
```

ดังนั้น ข้อความสั่ง repeat คือ "**syntactic sugar**" : หมายถึง ตัวสร้างภาษา ซึ่งสามารถแสดงออกอย่างบริบูรณ์ ในเทอมของตัวสร้างอื่นๆ (a language construct that is completely expressible interms of other constructs) แต่แสดงออกได้เฉพาะสถานะการณ์ร่วม หนึ่งอย่าง

ตัวสร้าง while และตัวสร้าง repeat มีคุณสมบัติว่า การจบส่วนวนซ้ำ เกิดขึ้น ณ หนึ่งจุดเท่านั้น และ S จะไม่สามารถ ออกไป ณ จุดตอนกลางได้ ยกเว้น ตัวแปรแบบบูลพิเศษ ซึ่งกล่าวถึงตอนต้น และถูกทดสอบ ใน B บางครั้ง มันจะมีประโยชน์ ถ้าสามารถทำให้ ออกจาก การวนซ้ำ จากจุดภายใน จุดใดจุดหนึ่งได้ รูปแบบทั่วไป ที่มีมากกว่า ของการวนซ้ำ บางครั้งรวมอยู่ในภาษาด้วย เช่น LOOP-EXIT ของภาษา Modula-2 ดังตัวอย่างข้างล่างนี้

```
LOOP

    IF B1 THEN EXIT END;

    IF B2 THEN EXIT END:

END:    (* loop *)
```

ข้อความสั่ง LOOP-EXIT ทำให้ มีทางออก มากมาย ที่ใดก็ได้ภายใน body ของมัน

ข้อความสั่ง LOOP-EXIT อาจจะไม่มี ข้อความสั่ง EXIT ใดๆ ได้เช่นกัน กรณีเช่นนี้ การวนซ้ำ จะไม่มีการหยุด (loops forever)

ภาษา PL/1 และ C ใช้ข้อความสั่ง break แทน EXIT (exit ที่ใช้ในภาษา C ทำให้ การกระทำของ โปรแกรมทั้งหมด จบลง)

การวนซ้ำ ด้วย exits สามารถ ถูกจำลองไว้ (be simulated) ด้วยข้อความสั่ง while หรือข้อความสั่ง repeat

ตัวอย่าง การวนซ้ำ ที่มีทางออก สองทาง เขียนดังนี้

```
Done1 := FALSE;
Done2 := FALSE;
REPEAT
   . .
   IF B1 THEN Done1 := TRUE
   ELSE
      .
      IF B2 THEN Done2 := TRUE
      ELSE
         . .
      END (* ifB2 *)
   END (* if B1 *)
UNTIL Done1 OR Done2;
```

แต่สิ่งนี้ ทำให้ โปรแกรมอ่านยากกว่า การใช้ ข้อความสั่ง LOOP ในที่นี้ ตัวแปรแบบบูล ที่เพิ่มขึ้นคือ Done1 และ Done2 เกี่ยวข้องกันในแง่ที่ว่า ข้อความสั่ง เข้าหนึ่งทาง (single-entry) ออกหนึ่งทาง (single-exit) เหมือนกับ ข้อความสั่ง repeat และ ข้อความสั่ง while มีกำลัง (powerful) น้อยกว่า การวนซ้ำ ที่มี ทางออก หลายทาง

กรณีพิเศษ ร่วมอย่างหนึ่ง ของตัวสร้างการวนซ้ำ ได้แก่ **for-loop**

ตัวอย่าง ภาษา Modula-2

```
FOR I := 0 TO 2*Nmax STEP 2 DO
   . . .
END;
```

หรือ DO loop ของภาษา FORTRAN เขียนดังนี้

```
      DO 20 I = 0, 2*Nmax
         . . .
   20 CONTINUE
```

ในการวนซ้ำของ Modula-2 นั้น I คือ**ตัวแปรควบคุม** (control vaiable) ในกรณีนี้ 0 และ 2*Nmax เป็น **นิพจน์กำหนดขอบเขต** (bound expressions) การกระทำการ ของ loop สิ่งแรก ประเมินผล bound expressions และกำหนดให้ I มีค่าเท่ากับ first bound และเพิ่มค่าของ I โดย นิพจน์ หลัง STEP (กรณีนี้คือ 2) หลังจาก การกระทำการแต่ละครั้งของการวนซ้ำ รูปแบบของ loop เช่นนี้ มีรวมอยู่ในภาษาต่างๆ เพราะว่า มีประสิทธิผลสูงสุดมากกว่า ตัวสร้าง loop อื่นๆ ตัวอย่างเช่น ตัวแปรควบคุม (บางที เป็น final bound) อาจใส่ใน เรจิสเตอร์ (registers) ทำให้การดำเนินการเร็วมาก

ตัวประมวลผล จำนวนมาก มี หนึ่งคำสั่ง ซึ่ง สามารถเพิ่มค่า เรจิสเตอร์ ทดสอบค่าในเรจิสเตอร์ และแตกกิ่ง ดังนั้น การควบคุม การวนซ้ำ และเพิ่มค่า สามารถเกิดขึ้น ใน คำสั่งเครื่องหนึ่งคำสั่ง ทำให้ได้รับประสิทธิภาพ เพิ่มขึ้น อย่างไรก็ตาม มี ข้อจำกัด จำนวนมาก ใน ข้อความสั่ง for ข้อจำกัดส่วนใหญ่ เกี่ยวข้องกับ ตัวแปรควบคุม ได้แก่

- ภายใน body ของการวนซ้ำ ค่าของ I เปลี่ยนแปลงไม่ได้

  (The value of I cannot be changed within the body of the loop.)

- หลังจาก จบ การวนซ้ำ ค่าของ I จะไม่ทราบค่า

  (The value of I is undefined after the loop terminates.)

- I ต้องถูกจำกัดด้วย ชนิด และอาจถูกประกาศ ใน วิธีต่างๆไม่ได้

  (I must be of restricted type and may not be declared in certain ways.)

ตัวอย่างเช่น เป็นพารามิเตอร์ ของ โปรซีเดอร์ หรือเป็น เขตระเบียน (record field) หรือต้องเป็นตัวแปรเฉพาะที่ (local variable)

บางภาษา มีข้อจำกัดว่า ตัวแปรควบคุม ต้องเป็น ชนิด integer ส่วนภาษาอื่นๆ เช่น Pascal และ Modula-2 อนุญาต ให้ เป็น ชนิดเชิงตัวเลขใดๆ ก็ได้ (any ordinal type)

ภาษา Ada มี loop ซึ่ง นิยาม ตัวแปรควบคุม ของมันเอง ซึ่งชนิด ของ ตัวแปรควบคุม ได้มาจาก bound expressions และ นำไปใช้นอก loop ไม่ได้

คำถามเพิ่มเติม เกี่ยวกับ พฤติกรรมของ การวนซ้ำ ได้แก่

(Further questions about the behaviour of loops include.)

(1) bounds ถูกประเมินผลเพียงครั้งเดียว ใช่หรือไม่? ถ้าใช่ หลังจากนั้น bounds อาจจะไม่เปลี่ยนแปลง หลังจาก เริ่มการกระทำการ (Are the bounds evaluated only once? If so, then the bounds may not change after execution begins.)

(2) ถ้า lower bound มีค่ามากกว่า upper bound การวนซ้ำ จะถูกกระทำการหรือไม่? ภาษาโปรแกรมสมัยใหม่ ส่วนใหญ่ กระทำการทดสอบ bound ที่ตอนเริ่มต้น ไม่ใช่ ตอนจบของการวนซ้ำ ดังนั้น การวนซ้ำ จึงเหมือนกับ while-loop
อย่างไรก็ตาม การ implement ของภาษา FORTRAN ชุดเก่า บางตัว มี loops ซึ่ง ต้องกระทำการอย่างน้อยที่สุด หนึ่งครั้งเสมอ

(3) ค่าของตัวแปรควบคุม จะ undefined หรือไม่? ถ้า ข้อความสั่ง GOTO หรือ EXIT ถูกนำมาใช้เพื่อออกจาก loop ก่อนการจบ (before termination) บางภาษา ยอมให้ค่านำมาใช้ได้ บน "abnormal termination" แต่บางภาษา ไม่ยอม ให้นำมาใช้

(4) ตัวแปลภาษา อะไร ซึ่ง ตรวจสอบการกระทำ บนโครงสร้างการวนซ้ำ บทนิยาม ของบางภาษา จะไม่มี ตัวแปลภาษา ซึ่ง จับ (catch) การกำหนดค่า ให้กับ ตัวแปรควบคุม ซึ่งจะเป็นเหตุให้เกิดผลลัพธ์ ซึ่งคาดการณ์ล่วงหน้าไม่ได้

บางภาษา เช่น CLU มีรูปแบบทั่วไป ของ ตัวสร้าง for-loop ซึ่งเกี่ยวกับ **<u>วัตถุของภาษาตัวใหม่</u>** (a new language object) เรียกว่า **<u>ตัวทำซ้ำ</u>** (iterator) โดยนามธรรม ตัวทำซ้ำ ต้องจัดหาบทนิยาม ให้กับตัวแปรควบคุม โครงร่างของการทำซ้ำ สำหรับตัวแปรควบคุมเหล่านี้ นั่นคือ วิธีของการกำหนดค่าใหม่ ให้กับ ค่าปัจจุบันของมัน (that is, a way of assigning a new value given its current value) และ สิ่งอำนวยความสะดวก สำหรับทำการทดสอบ การจบ ดังนั้น ตัวทำซ้ำ จึงกลายเป็น บางสิ่ง ซึ่งเหมือนกับการประกาศชนิดใหม่ (และใส่เข้าไปใน โครงร่างของการนิยามนามธรรม แบบชนิดข้อมูล)

<u>ตัวอย่าง</u>

```
numcount = proc(s : string) returns(int);
    count : int = 0;
    for c : char in stringchars(s) do
        if numneric(c) then
            count := count + 1:
        end;
    end,
    return(count);
end numcount;
```

```
stringchars = iter(s : string) yields(char);
    index : int := 1;
    limit : int := string$size(s);
    while index <= limit do
        yield(string$fetch(s, index))
        index := index + 1;
    end:
end stringchars;
```

ในที่นี้ ตัวทำซ้ำ คือ stringchars ตัวทำซ้ำ ถูกนิยามเหมือนกับ ฟังก์ชัน คือ มีค่าส่งกลับ (ในที่นี้คือ char) อย่างไรก็ตาม ผลกระทบ ของการเรียก ตัวทำซ้ำ เป็นดังนี้

ครั้งแรก เมื่อตัวทำซ้ำถูกเรียก ค่าของพารามิเตอร์ของมัน จะถูกเก็บไว้ จากนั้น ตัวทำซ้ำเริ่มต้นกระทำการ จนกระทั่ง ถึง ข้อความสั่ง **yield** เมื่อมัน หยุดการกระทำการ และส่งกลับ ค่าของ นิพจน์ ใน yield การเรียกครั้งต่อไป มันกระทำการอีก หลังจาก yield การหยุด เมื่อใดก็ตามที่ถึง yield อีกชุดหนึ่ง จนกระทั่งมันออกไป ซึ่ง การวนซ้ำ จากที่ซึ่งมันถูกเรียก จนถึง จบ

7.3 <u>การโต้แย้งเรื่อง GOTO</u> (The GOT0 controversy)

ข้อความสั่ง GOTOs ยังคงมีอยู่ ในภาษาโปรแกรมจำนวนหนึ่ง เช่น ภาษา FORTRAN และ BASIC

ตัวอย่าง รหัสภาษา FORTRAN77

```
10 IF (A(I).EQ.O) GOT0 20
      .
      I = I + 1
      GOT0 10
20 CONTINUE
```

ซึ่งมีความหมายเหมือนกับ ภาษา Pascal ดังนี้

```
while a[i] <> 0 do
```

```
begin
    .  .
    i := i + 1
end:
```

นับตั้งแต่ ปี ค.ศ. 1968 จดหมายเปิดผนึก ฉบับที่มีชื่อเสียง เขียนโดย E.W. Dijkstra ได้ตั้งข้อสงสัย เกี่ยวกับ ข้อความสั่ง GOTOs ว่าข้อความสั่งนี้ สามารถนำไปสู่ unreadable "spaghetti" code ได้ง่าย

ตัวอย่าง

```
   IF (X.GT.0) GOT0 10
   IF (X.LT.0) GOT0 20
   X = 1
   GOT0 30
10 X = X + 1
   GOT0 30
20 x = -X
   GOT0 10
30 COTINUE
```

ข้อความสั่ง GOTO ใกล้เคียงมาก กับ รหัสภาษาเครื่องจริง (actual machine code) ตามที่ Dijkstra ชี้ว่า การปล่อยให้ใช้ ข้อความสั่ง GOTOs โดยไม่มีข้อจำกัด สามารถทำให้ การออกแบบภาษา อย่างระมัดระวัง มากที่สุดแล้ว นำไปสู่ <u>โปรแกรมที่ไม่สามารถถอดรหัสได้</u> (undecipherable program)

Dijkstra เสนอว่า การใช้ข้อความสั่ง GOTOs ควรมีการควบคุม หรือมิฉะนั้น ให้ยกเลิกไปเลย สิ่งนี้เป็นการโต้แย้งมากที่สุด ในการเขียนโปรแกรม ซึ่งยังคงรุนแรง (rages) มาจนทุกวันนี้

คนกลุ่มหนึ่ง โต้เถียงว่า จำเป็นต้องมี ข้อความสั่ง GOTO ไว้ เพื่อประสิทธิภาพ และโครงสร้างที่ดี

(One group argues that the GOT0 is indispensable for efficiency and even for good structure.)

คนอีกกลุ่มหนึ่ง โต้เถียงว่า ข้อความสั่ง GOTO มีประโยชน์ แต่ต้องเป็นภายใต้สถานะ-การณ์ ซึ่ง จำกัดและระมัดระวังอย่างเต็มที่

(Another argues that it ca be useful under carefully limited circumstances.)

คนกลุ่มที่สาม โต้เถียงว่า ข้อความสั่ง GOT0 ควรจะ ลบทิ้ง จากภาษาคอมพิวเตอร์ ทุกภาษา

(A third argues that it is **anachonism** that should truly be abolished henceforth from all computer languages.)

เราไม่ต้องการยกข้อโต้เถียงทั้งหมด ที่เกิดขึ้นใน การโต้แย้งนี้ บางที เพียงหนึ่งตัวอย่าง อาจจะเพียงพอกับสถานะการณ์ ซึ่ง มีข้อโต้เถียงมากมาย เรื่องการใช้ ข้อความสั่ง GOTO ใน โครงสร้างซ้อนใน อย่างลึก ส่งกลับ มายัง ระดับนอกสุด (in a deeply nested structure to return to the outermot level)

ตัวอย่าง รหัสถูกต้องในภาษา Pascal

```
if ok then begin
    while not done do begin

        while not found do begin
            ...
            if diaster then got0 99;

        end; (* while not found *)
    end, (* while not done *)
end; (* if ok *)
..
99 :
```

ในกรณีเหล่านี้ มันจะดีที่ตรวจดูว่า มีตัวสร้างอื่นๆ บางตัว หรือไม่ ซึ่งใช้ได้ และให้ผลเหมือนกัน และขจัดบางส่วนทิ้งไป เพื่อให้ ข้อความสั่ง GOTO ทำให้ รหัส ง่ายขึ้น ตัวอย่างเช่น

มีหลายภาษา ซึ่งมี ข้อความสั่ง EXIT และ BREAK ซึ่งยอมให้ย้าย การควบคุม ไปยังตอนจบ ของลำดับข้อความสั่ง จากจุดต่างๆ ภายใน

ตัวอย่าง ภาษา Modula-2

```
LOOP
    .
    IF done THEN
        EXIT:
    END; (* if *)
    . . .
END; (* LOOP *)
```

เมื่อ การควบคุม ถูกย้าย โดย ข้อความสั่ง EXIT ไปยัง ข้อความสั่ง ตามหลัง END ของ LOOP สิ่งนี้ บางทีอาจเป็นสถานะการณ์ที่น่าพอใจ มากที่สุด ซึ่งกำหนดให้ โดยตัวอย่าง โปรแกรม Pascal ก่อนหน้านั้น

อย่างไรก็ตาม ถ้าภาษาหนึ่ง มี สถานะการณ์ จำนวนมาก ซึ่งเห็นชัดเจนว่า ข้อความสั่ง GOTOs มีความจำเป็น สำหรับ การทำให้ รหัส ชัดเจน อาจเป็นไปได้ว่า ภาษานั้น มี ตัวสร้างการควบคุมเชิงโครงสร้าง ไม่เพียงพอ ตัวอย่างเช่น ภาษา FOTRAN ซึ่งจำลองแบบ (simulating) ข้อความสั่ง while การใช้ข้อความสั่ง GOTO เป็นสิ่งจำเป็น เพราะว่า ภาษา FORTRAN77 ไม่มีข้อความสั่ง while ให้ใช้

สำหรับกลุ่มคนชุดที่สอง ซึ่งมองว่า ข้อความสั่ง GOTO ควรเก็บไว้ก่อน (should remain) แต่ ให้มีข้อจำกัด ในการนำมาใช้ ตัวอย่างเช่น ภาษา Pascal มีกฎ ควบคุมการใช้ ข้อความสั่ง GOTO ดังนี้ : ห้ามไม่ให้ใช้ GOTO กระโดดเข้าไปใน ลำดับของ ข้อความสั่ง ซึ่ง ข้อความสั่ง GOTO ไม่ใช่ส่วนหนึ่งของมัน

(a GOT0 **cannot** be used to jump into a sequence of statements that the GOT0 statement is not a part of .)

ตัวอย่าง การใช้ ข้อความสั่ง GOT0 ถูกต้อง ใน Pascal

```
1: readln(i);
  if i < j the begin
      search(i, j);
```

```
            if error then got0 1;
        end; (* if *)
```

แต่ รหัส ข้างล่างนี้ ไม่ถูกต้อง

```
        if ok then goto 1
        else begin
            readln(i);
            1: search(i, j);
        end; (* if *)
```

กฎซึ่งเหมือนกันนี้ ประยุกต์ใช้ในภาษา Ada กฎต่างๆ เหล่านี้ถูกสมมติ ให้กับการใช้ข้อความสั่ง GOTOs เพื่อส่งเสริม โครงสร้างของโปรแกรม ไม่ใช่เป็น การทำลายโครงสร้างของโปรแกรม จริงๆ แล้ว ในภาษา ซึ่งไม่มีกลไกควบคุมเชิงโครงสร้าง ให้ใช้ จึงใช้ข้อความสั่ง GOTO เป็นการจำลองแบบ เช่นในการ จำลอง ตัวสร้าง while-loop ของ FORTRAN77

ภาษาโปรแกรมส่วนน้อย เช่น Modula-2 รับเอารูปแบบอย่างมากของ Dijkstra's position มาก และเลิกใช้ ข้อความสั่ง GOTO

7.4 <u>โปรซีเดอร์และพารามิเตอร์</u> (Procedures and Parameters)

โปรซีเดอร์ หมายถึง กลไกอย่างหนึ่งในภาษาโปรแกรมสำหรับการนิยามนามธรรม กลุ่มของการกระทำหรือการคำนวณต่างๆ

(A procedure is a mechanism in a programming language for abstracting a group of actions or computations.)

กลุ่มของการกระทำ เรียกว่า **body** ของ โปรซีเดอร์ และ body ของโปรซีเดอร์ ทั้งหมดนั้น ถูกแทนที่ ด้วย ชื่อของโปรซีเดอร์

โปรซีเดอร์ ถูกประกาศโดยการกำหนดชื่อของมัน พารามิเตอร์ และ body ตัวอย่างเช่น การประกาศโปรซีเดอร์ ของ ภาษา Pascal ข้างล่างนี้

```
    procedure intswap (var x, y : integer);
    var t : integer;
    begin
```

```
        t := x;
        x := y;
        y := t;
    end;
```

ในที่นี้ โปรซีเดอร์ intswap **สลับค่า พารามิเตอร์** x **และ** y **โดยใช้ตัวแปรเฉพาะที่** t

โปรซีเดอร์ ถูกเรียก หรือ ถูกใช้งาน โดย **ชื่อของมัน รวมทั้ง อาร์กิวเมนต์** ให้กับ การเรียก **ซึ่ง สมนัยกับ พารามิเตอร์** :

(A procedure is called or activated by stating its name, together with arguments to the call, which correspond to its parameters :)

```
    intswap(a, b);
```

การเรียกโปรซีเดอร์ เป็นการถ่ายโอน (transfer) การควบคุม ไปยัง จุดเริ่มต้น ของ body ของ โปรซีเดอร์ถูกเรียก (called procedure) หรือ **ผู้ถูกเรียก** (callee) เมื่อการกระทำการมาถึงตอนจบ ของ body การควบคุมจะถูกส่งกลับ ไปยัง **ผู้เรียก** (caller) หรือ โปรซีเดอร์เรียก (calling procedure)

ในบางภาษา การควบคุม อาจถูกส่งกลับไปยังผู้เรียกก่อน จบ body ของผู้ถูกเรียกได้ โดยใช้ข้อความสั่ง **return** ภาษา Pascal ไม่มีข้อความสั่ง return แต่ ภาษา Modula-2 มีข้อความสั่งนี้

ตัวอย่างเช่น

```
PROCEDURE intswap(VAR x, y : INTEGER);
VAR t : INTEGER;
BEGIN
    IF x = y THEN
        RETURN;
    END;
    t := x;
```

```
    x := y;
    y := t;
END intswap;
```

ในบางภาษา เช่น FORTRAN การเรียก โปรซีเดอร์ ต้องมีคำหลัก CALL ด้วย เช่น

```
CALL INTSWAP(A, B)
```

ข้อสังเกต ใน FORTRAN เรียกโปรซีเดอร์ว่า **subroutines** ภาษาโปรแกรม อาจทำความแตกต่างระหว่าง โปรซีเดอร์ซึ่งให้ผลลัพธ์ของการดำเนินการ โดยเปลี่ยนแปลงพารามิเตอร์ หรือ ตัวแปรไม่เฉพาะที่ (nonlocal variables) ของมัน

และ **ฟังก์ชัน (functions)** ซึ่งปรากฏใน นิพจน์ และคำนวณ **ค่าส่งกลับ (returned values.)**

ฟังก์ชัน อาจจะมีผลกระทบ หรือ ไม่มีผลกระทบ กับ พารามิเตอร์ และ ตัวแปรไม่เฉพาะที่ของมัน

(Function may or may not also affect their parameters and nonlocal variables.)

ภาษา Pascal ฟังก์ชันถูกประกาศโดยใช้คำหลัก function แต่สิ่งที่แตกต่าง จาก โปรซีเดอร์คือ ฟังก์ชัน มีค่าส่งกลับ (returned value)

ตัวอย่าง ฟังก์ชัน ภาษา Pascal

```
function intswap(var  x, y : ineteger) : boolean;
var  t : temp;
begin
    t := x;
    x := y;
    y := t;
    intswap := true;
end;
```

การกำหนดค่า ให้กับ intswap ใน ข้อความสั่งสุดท้าย ของ body ของมัน ไม่ใช่ การกำหนดค่า ให้กับ ตัวแปรจริงๆ แต่เป็นการสร้าง ค่าส่งกลับ สำหรับฟังก์ชัน (ในกรณีนี้ ค่าบูลีนคงที่จะเป็นจริงเสมอ) ใน Pascal ข้อความสั่งนี้ จะปรากฏ ณ จุดใดๆ ก็ได้ ใน body ของ ฟังก์ชัน

และไม่ได้หมายความว่า ฟังก์ชัน ส่งกลับ ไปยัง ผู้เรียก ณ จุดนั้น (at that point) ดังนั้น ฟังก์ชัน ข้างล่างนี้ จึงมีความหมาย เหมือนกับ รหัสก่อนหน้า

```
function intswap (var x, y : integer) : boolean;
var t : temp;
begin
    intswap := true;
        t := x;
        x := y;
        y := t;
end;
```

ฟังก์ชัน และ โปรซีเดอร์ ของ Pascal เป็น ตัวสร้าง (construct) ชนิดที่มีทางเข้า หนึ่งทาง และทางออก หนึ่งทาง เหมือนกับ ข้อความสั่งควบคุมเชิงโครงสร้าง ใน Pascal (Pascal functions and procedures are **single-entry,** single-exit constructs, like structured control statements in Pascal) ปกติมัน กระทำการ body ทั้งหมด และส่งกลับ เฉพาะที่ตอนจบเท่านั้น

(they always execute their bodies entirely and return **only** at the end.)

ใน Modula-2 เหมือนกับ Pasal คือ ฟังก์ชัน เป็นเพียงโปรซีเดอร์ ซึ่งมีค่าส่งกลับ และสิ่งนี้ เน้นโดย การประกาศว่า มันเป็นโปรซีเดอร์

ตัวอย่าง

```
PROCEDUE intswap(VAR x, y : INTEGER) : BOOLEAN;
VAR t : INTEGER
BEGIN
    t := x;
    x := y;
    y := t;
    RETURN  TRUE;
END iutswap;
```

อย่างไรก็ตาม ภาษา Modula-2 ค่าส่งกลับ กำหนดโดย ข้อความสั่ง RETURN ซึ่งถ่ายโอน การควบคุม กลับไปยังผู้เรียก ดังนั้น ข้อความสั่ง RETURN ใน โปรแกรมภาษา Modula-2 ซึ่ง**ต้องอยู่ท้ายสุดเสมอ**

ในบางภาษา มีเฉพาะฟังก์ชัน เท่านั้น เช่น **ภาษาเชิงหน้าที่** (Functional languages) จะมีคุณสมบัตินี้ ในบางภาษา เช่น ภาษา C โปรซีเดอร์ แตกต่างจาก ฟังก์ชัน โดยการส่งกลับ ค่า **null** หรือ **void**

(In C, a procedure is distinguished from a function by returning a **null** or **void** value :)

```
void  intswap(int*x, int*y)
/* x and y are pointers to integers */
{int    t = *x;
       *x = *y;
       *y =   t; }
```

ในบางภาษา การประกาศ โปรซีเดอร์ และ การประกาศ ฟังก์ชัน เขียนในรูปแบบ คล้ายกับ การประกาศตัวคงที่ โดยใช้ เครื่องหมายเท่ากับ เช่น การประกาศ โปรซีเดอร์ ของ ภาษา Algol68 ข้างล่างนี้ ซึ่งนิยาม โปรซีเดอร์ intswap อย่างเดียวกับ รหัส Pascal (หรือ C) ชุดเริ่มต้น

```
proc  intswap = (ref  int  x, y) void :
begin
    int  t := x;
        x := y;
        y := t
end;
```

โปรดสังเกต การประกาศ ของ พารามิเตอร์ ในวงเล็บ หลัง เครื่องหมาย "=" และ ค่าส่งกลับ void หลัง วงเล็บ

การใช้ เครื่องหมายเท่ากับ เพื่อประกาศ โปรซีเดอร์ มีเหตุผลเพราะว่า การประกาศโปรซีเดอร์ ให้ ชื่อโปรซีเดอร์ มีความหมายว่า ยังมีตัวคงที่ ระหว่าง การกระทำการ ของโปรแกรม เราอาจพูดว่า การประกาศโปรซีเดอร์ สร้าง (creates) ค่าโปรซีเดอร์คงที่ และเกี่ยวข้องกับ ชื่อเชิงสัญลักษณ์ - ชื่อของโปรซีเดอร์ - กับค่านั้น

ในการอภิปรายต่อจากนี้ เราจะไม่ทำให้ โปรซีเดอร์ และฟังก์ชัน แตกต่างกัน แต่จะ

พิจารณาว่า ทั้งคู่ เป็นกลไกอย่างเดียวกัน ในภาษาโปรแกรม

โปรซีเดอร์ สื่อสาร (communicates) กับ ส่วนที่เหลือ ของ โปรแกรม โดยผ่าน พารา-มิเตอร์ของมัน และผ่านทาง **nonlocal references** นั่นคือ อ้างถึง ตัวแปร ซึ่ง ประกาศ นอก body ของมันเอง

กฎสโคป (scope rules) ซึ่งสร้าง ความหมายของ nonlocal referencce ได้กล่าวไป แล้วในบทที่ 5 ในส่วนที่เหลือของหัวข้อนี้ สิ่งแรกจะทบทวน ความหมาย (semantics) ของบล็อก เน้น ความแตกต่างระหว่าง โปรซีเดอร์บล็อก และ nonprocedure blocks จากนั้น จะศึกษา พารา-มิเตอร์ ของ โปรซีเดอร์ (procedure parameters) ซึ่งเป็นกลไก สำหรับ การสื่อสาร กับส่วนที่เหลือของโปรแกรม

โครงสร้างของ สิ่งแวดล้อม ซึ่ง จำเป็น เพื่อเก็บ (maintain) การสื่อสาร และการโยงการควบคุม (control links) ระหว่าง การเรียกโปรซีเดอร์ จะได้อภิปราย ในหัวข้อถัดไป

7.4.1 **ความหมายของโปรซีเดอร์** (Procedure Semantics)

โปรซีเดอร์ หมายถึง บล็อกซึ่ง การประกาศของมัน ถูกแยกต่างหาก จากการกระทำการของมัน (A procedure is a block whose declaration is separated from its execution.)

ในบทที่ 5 เราได้เห็น ตัวอย่างของบล็อก ในภาษา C และ ภาษา Algol60 แล้วว่า ไม่ใช่ โปรซีเดอร์ บล็อก (procedure blocks) บล็อกเหล่านี้ ปกติ ถูกกระทำการทันที เมื่อ ถูกพบ ตัวอย่างเช่น ใน Algol60 บล็อก A และ B ในรหัสข้างล่างนี้ จะถูกกระทำการ เมื่อมันถูกพบ

```
A : begin
        integer x, y;


          x := y * 10;
    B : begin
        integer i;
        1 := x div 2;


        end B;
end A;
```

ในบทที่ 5 เราได้เห็นแล้วว่า **สิ่งแวดล้อม** กำหนด การจัดสรร หน่วยความจำ และเก็บ (maintains) ความหมายของ ชื่อต่างๆ ระหว่าง การกระทำการ (the **environment** determines the allocation of memory and maintains the meaning of names during execution.) ในภาษาโครงสร้างแบบบล็อก (block-structured language) ระหว่างการกระทำการ เมื่อพบ บล็อก มัน จะเป็นเหตุ ให้เกิดการจัดสรรหน่วยเก็บ ของ ตัวแปรเฉพาะที่ และวัตถุอื่นๆ ซึ่งสมนัยกับการประกาศของบล็อก หน่วยความจำ ซึ่งถูกจัดสรรให้กับ วัตถุเฉพาะที่ ของบล็อก เรียกว่า **ระเบียนการใช้งาน** ของบล็อก และกล่าวได้ว่า บล็อกจะถูกใช้งาน ขณะที่มัน กระทำการ ภายใต้ การผูกโยง ซึ่งสร้างขึ้นโดย ระเบียนการใช้งานของมัน

(This memory allocated for the local objects of the block is called the **activation record** of the block. and the block is said to be **activated** as it executes under the bindings established by its activation record.j

ขณะที่เข้ามายังบล็อก ระหว่างการกระทำการ การควบคุม ส่งจาก การใช้งานของ บล็อกล้อมรอบ (surrounding block) ไปยัง การใช้งาน ของ บล็อกใน (inner block) เมื่อออกจาก บล็อกใน การควบคุม ส่งกลับไปยัง บล็อกล้อมรอบ และระเบียนการใช้งาน ของบล็อกใน จะถูกปล่อย (is released) กลับไปยัง สิ่งแวดล้อมของระเบียนใช้งาน ของ บล็อกล้อมรอบ

ตัวอย่างเช่น ในรหัส Algol60 ข้างต้น ระหว่างการกระทำการ x และ y ถูกจัดสรรเนื้อที่ ใน ระเบียนใช้งาน ของ บล็อก A :

x
y

ระเบียนใช้งานของ A

เมื่อเข้ามายังบล็อก B เนื้อที่หน่วยความจำ ถูกจัดสรร ในระเบียนใช้งาน ของ B: ดังนี้

x	
y	ระเบียนใช้งาน ของ A
i	ระเบียนใช้งาน ของ B

เมื่อ ออกจาก B สิ่งแวดล้อม ย้อนกลับ ไปยัง ระเบียนใช้งาน ของ A ดังนั้น การใช้งานของ B ยังต้องเก็บ สารสนเทศบางอย่าง เกี่ยวกับ การใช้งาน จากสิ่งซึ่ง มันเข้าไป

ในรหัสข้างต้น บล็อก B จำเป็น ต้องเข้าถึง ตัวแปร x ซึ่งประกาศ ในบล็อก A การอ้างถึง x ภายใน B หมายถึง **nonlocal reference** เพราะว่า x ไม่ได้ถูกจัดสรรเนื้อที่ ในระเบียนใช้งานของ B แต่ถูกจัดสรรเนื้อที่ ในระเบียนใช้งานของ บล็อก A ซึ่งล้อมรอบ ดังนั้น จึงจำเป็นที่ B ต้องเก็บสารสนเทศ เกี่ยวกับ การใช้งานล้อมรอบ ของมัน

ขณะนี้ ให้พิจารณาว่า จะเกิดอะไรขึ้น ถ้า B เป็นโปรซีเดอร์ ถูกเรียกจาก A แทนที่ จะเป็นบล็อกเข้าถึง โดยตรงจาก A สมมติว่า เพื่อให้เป็น ตัวอย่างแบบรูปธรรม จงพิจารณาสถานะการณ์ต่อไปนี้ ในวากยสัมพันธ์ของ Pascal

```
program envex;
var x : integer;

    procedure B;
    var i : integer;
    begin
        i := x div 2;
        .
    end; (* B *)

    procedure A;
    var x, y : integer;
    begin

        x := y * 10;
        B;
    end; (* A *)
begin (* main *)
    A;
end, (* envex *)
```

B ยังคงถูกเข้าถึงจาก A และการใช้งาน ของ B ต้องเก็บ สารสนเทศบางอย่าง เกี่ยวกับการใช้งาน ของ A เพื่อให้การควบคุม สามารถ ส่งกลับไปยัง A ได้ เมื่อออกจาก B แต่ขณะนี้ มีความแตกต่าง ในวิธี แก้ปัญหา nonlocal reference ภายใต้ lexical scoping rule (ดูบทที่ 5) x ใน B เป็น global x ของ โปรแกรม ไม่ใช่ x ซึ่งประกาศ ใน A ในเทอมของ ระเบียนใช้งาน จะมีภาพดังต่อไปนี้

x	สิ่งแวดล้อมส่วนกลาง
x	
y	ระเบียนใช้งาน ของ A
i	ระเบียนใช้งาน ของ B

การใช้งาน ของ B ต้องเก็บสารสนเทศ เกี่ยวกับ สิ่งแวดล้อมส่วนกลาง เพราะว่า nonlocal reference x จะถูกพบที่นั่น แทนที่ จะเป็น ในระเบียนใช้งาน ของ A ที่เป็นเช่นนี้ เพราะว่า สิ่งแวดล้อมส่วนกลาง คือ **สิ่งแวดล้อมการนิยาม** (defining environment) ของ B ขณะที่ ระเบียนใช้งาน ของ A คือ **สิ่งแวดล้อมการเรียก** (calling environment) ของ B (บางครั้ง เราเรียก **สิ่งแวดล้อมการนิยาม** (defining environment) ว่า **สิ่งแวดล้อมแบบคงที่** (static environment) และเรียก**สิ่งแวดล้อมการควบคุม** (control environment) ว่า **สิ่งแวดล้อมแบบพลวัต** (dynamic environment)

สำหรับ บล็อก ซึ่งไม่ใช่ โปรซีเดอร์ บล็อก นั้น สิ่งแวดล้อมการนิยาม และ สิ่งแวดล้อมการเรียก ปกติคือ สิ่งเดียวกัน ซึ่งตรงกันข้าม กับโปรซีเดอร์ บล็อก ที่ว่า สิ่งแวดล้อมการเรียก และ สิ่งแวดล้อมการนิยาม ไม่เหมือนกัน จริงๆ แล้ว ใน โปรซีเดอร์ หนึ่งชุด จะมี สิ่งแวดล้อมการเรียก จำนวน เท่าไหร่ก็ได้ ในขณะที่ มันอาจเก็บ defining environment เดียวกัน

โครงสร้างจริง ของ สิ่งแวดล้อม ซึ่งเก็บ track ของ สิ่งแวดล้อมการนิยาม และ สิ่งแวดล้อมการเรียก จะอภิปราย ในหัวข้อถัดไป สิ่งที่เราสนใจ ในหัวข้อนี้ คือ วิธีการใช้งาน ของ บล็อก **<u>สื่อสาร</u>** (communicates) กับ ส่วนที่เหลือของโปรแกรม

จะเห็นชัดเจนว่า nonprocedure block สื่อสารกับ บล็อกล้อมรอบ ของมัน ผ่านทาง nonlocal references : lexical soping ยอมให้มันเข้าถึงตัวแปรทั้งหมด ในบล็อกล้อมรอบ ซึ่ง ไม่ได้มีการประกาศใหม่ ของมันเอง

ในทางตรงกันข้าม ภายใต้ lexical scoping, โปรซีเดอร์ บล็อก สามารถสื่อสารได้ เฉพาะกับ **บล็อกการนิยาม** (defining block) ของมันเท่านั้น โดยผ่านทาง nonlocal variables ไม่มีวิธีใดเลยที่จะเข้าถึงตัวแปรใน สิ่งแวดล้อมการเรียก (calling environment) ของมัน โดยตรง ในตัวอย่าง

รหัส Pascal, โปรซีเดอร์ B ไม่สามารถเข้าถึง ตัวแปร เฉพาะที่ x ของ โปรซีเดอร์ A ได้โดยตรง ไม่เพียงเท่านั้น ยังไม่สามารถเอาค่า x ใน A มาทำการคำนวณ ใน B ได้

วิธีสื่อสาร ของ โปรซีเดอร์ กับ สิ่งแวดล้อมการเรียก (calling environment) ของมัน คือ ผ่านทาง **พารามิเตอร์ (parameters)**

รายการพารามิเตอร์ (parameter list) ถูกประกาศ พร้อมกับ บทนิยาม ของ โปรซีเดอร์ ดังนี้

```
function gcd(u, v : integer) : integer;
begin
    if v = 0 then gcd := u
    else gcd := gcd(v, u mod v)
end;
```

ในที่นี้ u และ v เป็น พารามิเตอร์ ของ ฟังก์ชัน gcd พารามิเตอร์ ทั้งสองตัวนี้ จะไม่มีค่าใดๆ ทั้งสิ้น จนกว่า gcd จะถูกเรียก เมื่อนั้น มันจะถูกแทนที่โดย **อาร์กิวเมนต์** (arguments) จากสิ่งแวดล้อมการเรียก ดังนี้

```
z := gcd(x + y, 10);
```

การเรียก gcd นี้ พารามิเตอร์ u ถูก แทนที่ด้วย อาร์กิวเมนต์ x + y และพารามิเตอร์ v ถูกแทนที่ด้วย 10 เพื่อเน้น ความจริงที่ว่า พารามิเตอร์ จะยังไม่มีค่าใดๆ ทั้งสิ้น จนกระทั่งมันถูกแทนที่ด้วย อาร์กิวเมนต์ ดังนั้น บางครั้ง เราจึงเรียก พารามิเตอร์ ว่า **พารามิเตอร์ทางการ (formal parameters)** และเรียก อาร์กิวเมนต์ว่า **พารามิเตอร์จริง (actual parameters)**

การเรียก โปรซีเดอร์ เช่น gcd ผูกโยง (binds) อาร์กิวเมนต์ กับ พารามิเตอร์ ของ การประกาศโปรซีเดอร์ ในขณะที่ ณ เวลาเดียวกัน มีการโอนย้ายการควบคุม ไปยัง body ของ โปรซีเดอร์ การผูกโยง เหล่านี้ ตีความหมายอย่างไรนั้นขึ้นอยู่กับ **กลไกการส่งพารามิเตอร์** (parameter passing mechanism) ซึ่งใช้ สำหรับเรียก เราจะอภิปราย กลไกการส่งพารามิเตอร์ที่สำคัญที่สุด สี่ ชนิด ในส่วนที่เหลือของหัวข้อนี้ คือ ส่งโดยค่า ส่งโดยอ้างอิง ส่งโดย ค่า-ผลลัพธ์ และส่งโดย ชื่อ

CT 314 (1)

7.4.2 <u>กลไกการส่ง พารามิเตอร์</u>

(Parameter Passing Mechanisms)

<u>ส่งโดยค่า</u> (Pass by Value) กลไกนี้ อาร์กิวเมนต์ ซึ่งเป็นนิพจน์ ถูกประเมินผล ณ เวลา ของการเรียก และค่าของมัน จะกลายเป็น ค่าของ พารามิเตอร์ ระหว่าง การกระทำการ ของโปร-ซีเดอร์ (In this mechanism, the arguments are expressions that are evaluated at the time of the call, and their values become the values of the parameters during the execution of the procedure.) รูปแบบที่ง่ายที่สุด คือ ค่าของ พารามิเตอร์ เป็นค่าคงที่ ระหว่าง การกระทำการ ของ โปรซีเดอร์ และเราอาจตีความหมายว่า **การส่งโดยค่า** คือ การแทนที่ พารามิเตอร์ ทั้งหมด ใน body ของ โปรซีเดอร์ ด้วย ค่าของ อาร์กิวเมนต์ของมัน

<u>ตัวอย่างเช่น</u> เรียก gcd(10, 2 + 3) ของ ฟังก์ชัน gcd ข้างต้น ขณะกระทำการ body ของ gcd นั้น u จะถูกแทนที่ด้วย 10 และ v จะถูกแทนที่ด้วย 5 ดังนี้

```
if 5 = 0 then gcd := 10
else gcd := gcd(5, 10 mod 5)
```

รูปแบบ ของ การส่งโดยค่านี้ คือการ default ใน Ada (พารามิเตอร์เช่นนี้ อาจจะ ประกาศ ชัดแจ้ง เป็น in parameters)

การส่งโดยค่า เป็น กลไกอัตโนมัติ ใน Pascal และ Modula-2 และ เป็นกลไก การส่ง พารามิเตอร์ ที่สำคัญที่มีอยู่อย่างเดียว ในภาษา C และ Algol68 อย่างไรก็ตาม ในภาษาเหล่านี้ มีการตีความหมาย แตกต่างกันเล็กน้อย เมื่อใช้การส่งโดยค่า กล่าวคือ :

พารามิเตอร์ ถูกมองว่าเป็น ตัวแปรเฉพาะที่ ของ โปรซีเดอร์ ที่มีค่าแรก กำหนดโดย ค่า ของอาร์กิวเมนต์ ใน การเรียก

(The parameters are viewed as local variables of the procedure, with initial values given by the values of the arguments in the call.)

ดังนั้น ในภาษา Pascal และ Modula-2, value parameters อาจถูกกำหนดค่า ให้ เช่น เดียวกับ ตัวแปรเฉพาะที่ (แต่ไม่มีการเปลี่ยนแปลง ค่าใดๆ นอกโปรซีเดอร์) ในขณะที่ Ada in parameters จะกำหนดค่าให้ไม่ได้

<u>ส่งโดยอ้างอิง</u> (Pass by Reference)

ในที่นี้ อาร์กิวเมนต์ ต้องเป็นตัวแปร ที่มีการจัดสรรตำแหน่งแล้ว แทนที่จะเป็น การส่งค่า ของตัวแปร การส่งโดยอ้างอิง ส่งตำแหน่งของตัวแปร เพื่อให้ พารามิเตอร์ เป็น <u>สมนาม</u> สำหรับ

อาร์กิวเมนต์ และการเปลี่ยนแปลงใดๆซึ่งกระทำกับ พารามิเตอร์ จะเกิดขึ้นใน อาร์กิวเมนต์ เช่นเดียวกัน

(Here the arguments must be variables with allocated locations, Instead of passing the value of a variable, pass by reference passes **the** location of the variable, so mat the parameter becomes an **alias** for **the** argument and any changes made to the parameter occur to the argument as well.)

ภาษา FORTRAN ส่งโดยอ้างอิง เป็นกลไกการส่งพารามิเตอร์ เพียงอย่างเดียวเท่านั้นที่มีอยู่

(In FORTRAN, pass by reference is the only parameter passing mechanism.)

ภาษา Pascal และ Modula-2 ส่งโดยอ้างอิง จะทำได้สำเร็จ ด้วยการใช้ คำหลัก VAR (**โปรดสังเกตว่า** สิ่งนี้ ไม่ใช่รูปแบบอย่างเดียวกับ การใช้ VAR สำหรับ สิ่งของ สองสิ่ง ซึ่งไม่เกี่ยวข้องกัน คือ การประกาศตัวแปร และ การส่งโดยอ้างอิง) :

```
procedure refer(var x : integer):
begin
    x := x + 1;
end;
```

หลังจากเรียก refer(a) ค่าของ a จะเพิ่มขึ้นอีก 1 ดังนั้น ผลกระทบ (side effect) เกิดขึ้น สมนามหลายอย่าง อาจเป็นไปได้ เช่น ตัวอย่างข้างล่างนี้ (Multiple aliasing is also possible, such as in the code.)

```
var a: integer;
    '
procedure demo(var x, y : integer);
begin
    x := 2;
    y := 3;
    a := 4;
end,

demo(a, a);
```

ภายใน procedure demo หลังจากเรียก ไอเดนติไฟเออร์ x, y และ a ทั้งหมดนี้ อ้างถึงตัวแปรตัวเดียวกัน ชื่อ ตัวแปร a (Inside procedure demo after the call, the identifiers x. y and a all refer to the same variable, namely, the variable a.)

ภาษา C และ Algol68 จะทำการส่งโดยอ้างอิง ด้วย การส่ง เลขที่อยู่ หรือการส่ง ตำแหน่ง ชัดแจ้ง ภาษา C ใช้ตัวปฏิบัติการ "&" เพื่อแสดงตำแหน่ง ของ ตัวแปร และ ตัวปฏิบัติการ "*" เพื่อ dereference a pointer ดังนั้น

```
void refer(int * x)
{* x += 1; /* adds 1 to *x */}


int a;
    . . .
refer(&a);
```

ได้ผลลัพธ์อย่างเดียวกับ รหัส Pascal ข้างต้น อย่างไรก็ตาม แถวลำดับ ปกติ ส่งโดยอ้างอิง (แถวลำดับ หมายถึง "pointer constants") :

```
void p(int x[ ])
{x[0] = 1;}


int a[10];
    . . .
p(a);
```

มีผลอย่างเดียวกับ กำหนดค่า 1 ให้กับ a[0]

ภาษา Algol68 **ตัวชี้** (pointers) แสดงด้วยข้อมูล ชนิด ref (สำหรับอ้างอิง) ดังนั้น พิจารณารหัสต่อไปนี้

```
proc refer = (ref int x) int :
begin
    x := x + 1
end;
```

```
begin
    int x := 1;
    refer (x)
end
```

ในที่นี้ x จบด้วยค่า 2 เพราะว่า x มีชนิดเป็น ref int และค่าเท่ากับ ตำแหน่งของมัน ดังนั้น ตำแหน่งของ x ถูกส่งไปยัง โปรซีเดอร์ refer ในทางตรงกันข้าม รหัสต่อไปนี้ ผิดใน Algol68 เพราะว่าพารามิเตอร์ x มีเพียง 1 ค่า และ ไม่สามารถถูกกำหนดค่า :

```
proc p = (int x) int :
begin
    x := x + 1
end;
```

ส่งโดยค่า-ผลลัพธ์ (Pass by Value-Result)

กลไกนี้ กระทำสำเร็จ ได้ผลลัพธ์เหมือนกับ ส่ง โดยอ้างอิง ยกเว้น ไม่มีการสร้าง สมนาม จริง : ค่าของ อาร์กิวเมนต์ ถูกทำสำเนา และใช้ในโปรซีเดอร์ และหลังจากนั้น ค่าสุดท้าย ของ พารามิเตอร์ จะถูกทำสำเนา กลับไปยัง ตำแหน่งของ อาร์กิวเมนต์ เมื่อออกจาก โปรซีเดอร์

(This mechanism achieves a similar result to pass by reference, except that no actual alias is established : the value of the argument is copied and used in the procedure. and then the final value of the parameter is copied back out to the location of the argument when the procedure exits.)

ดังนั้น วิธีนี้ บางครั้งเรียกว่า copy-in, copy-out หรือ copy-restore สิ่งนี้คือ กลไก ของภาษา Ada in-out parameter (Ada มี out parameter อย่างง่าย ซึ่งไม่มี ค่าแรกส่งเข้า สิ่งนี้ เรียกว่า **pass by result**

(Ada also has simply an **out** parameter, which has no initial value passed in; this could be called **pass by result**.)

ส่งโดย ค่า-ผลลัพธ์ แตกต่างจาก การส่งโดยอ้างอิง เฉพาะ เรื่องการมี สมนามเท่านั้น (pass by value-result is only distinguishable from pass by reference in the presence of aliasing.)

ตัวอย่าง จงพิจารณารหัสต่อไปนี้

```
procedure p(x, y : integer);
begin
    x := x + 1;
    y := y + 1;
end;
begin
    a := 1;
    p(a, a);
end.
```

ถ้าเป็นการส่งโดยอ้างอิง หลังจากเรียก p แล้ว a มีค่าเป็น 3 ในขณะที่ ถ้าเป็นการส่งโดยค่า-ผลลัพธ์ a มีค่าเป็น 2

ภาษา Ada โดยบทนิยามของภาษากล่าวว่า inout parameters อาจจะถูกทำให้เกิดผลเป็นจริงด้วย ส่งโดยอ้างอิง และการคำนวณใดๆ ซึ่งอาจแตกต่างกัน ภายใต้กลไกทั้งสอง (ดังนั้น เกี่ยวข้องกับ alias) คือ ข้อผิดพลาด

ส่งโดยชื่อ (Pass by Name)

วิธีนี้ เป็นกลไกการส่งพารามิเตอร์ ซึ่ง เข้าใจยากที่สุด ใช้ในภาษา Algol60 แต่หลังจากนั้น ไม่มีการนำมาใช้อีกเลย โดยเฉพาะ หลังจาก การโต้ตอบ ซึ่งซับซ้อน ด้วยตัวสร้างภาษาอื่นๆ โดยเฉพาะ แถวลำดับ และการกำหนดค่า ถูกค้นพบเมื่อเร็วๆ นี้ ส่งโดยชื่อ ได้ถูกนำกลับมาสนใจอีกในภาษาเชิงหน้าที่ (functional languages) ซึ่งเรียกว่า **การประเมินผลแบบหน่วง (delayed evaluation)**

ความคิดของ ส่งโดยชื่อ คือ อาร์กิวเมนต์ จะไม่ถูกประเมินผล จนกระทั่ง มันถูกใช้จริง (เป็นพารามิเตอร์) ใน โปรแกรมถูกเรียก

(The idea of pass by name is that the argument is not evaluated until its actual use (as a parameter) in the called program.)

ดังนั้น ชื่อของอาร์กิวเมนต์ หรือ การแทนที่ บริบทของมัน ณ จุดของการเรียก แทน (replace) ชื่อ ของ พารามิเตอร์ ซึ่งสมนัยกับมัน

ตัวอย่าง

```
procedure   p(x);;
begin
    x := x + 1;
end ,
```

ถ้าเรียก p(a[i]) ผลลัพธ์คือ การประเมินผล

a[i] := a[i] + 1

ดังนั้น ถ้า i ถูกเปลี่ยนแปลง ก่อน ใช้ x ภายใน p ผลลัพธ์จะแตกต่างจาก การเรียกโดยอ้างอิง หรือการเรียกโดย ค่า-ผลลัพธ์

ตัวอย่าง

```
var i : integer;
    a : array[l . . 10] of integer;

    procedure   p(x)
    begin
        i:=i+  1;
        x := x + 1;
    end;

begin
    i := 1;
    a[l]  :=  1;
    a[2] := 2;
    p(a[i]);
end.
```

ผลลัพธ์ คือ a[2] = 3 และ a[1] ไม่เปลี่ยนแปลง

การตีความหมาย ของ การส่งโดยชื่อ (pass by name) เป็นดังนี้ :
text ของ อาร์กิวเมนต์ ณ จุดเรียก ถูกมองเป็น ฟังชัน ในตัวมันเอง ซึ่งจะถูกประเมินผล ทุกครั้ง ที่สมนัยกับ ชื่อพารามิเตอร์ เมื่อพบใน รหัสของ โปรซีเดอร์ถูกเรียก อย่างไรก็ตาม อาร์กิวเมนต์ปกติจะถูกประเมินผล ใน สิ่งแวดล้อมของผู้เรียก (caller) ในขณะที่ โปรซีเดอร์ จะถูกกระทำการ ใน สิ่งแวดล้อม การนิยามของมันเอง (in its defining environment)

ตัวอย่าง ภาษา Pascal

```
Program test;
var   i : integer;

    function p(y : mteger) : integer;
    var j : integer;
    begin
        j := y;
        i := i + 1:
        p := j + y;
    end: (* p *)

    procedure  q;
    var j : integer;
    begin
        i := 0;
        j := 2;
        writeln(p(i + j));
    end; (* q *)

begin (* main *)
    q;
end.
```

ในที่นี้ อาร์กิวเมนต์ i + j เรียกฟังก์ชัน p จาก โปรซีเดอร์ q จะถูกประเมินผล ทุกครั้งเมื่อพบ พารามิเตอร์ y ภายใน p อย่างไรก็ตาม นิพจน์ i + j ถูกประเมินผล ขณะที่มันยังอยู่ใน q ดังนั้น ข้อความสั่งบรรทัดแรก ของ p จะให้ผลลัพธ์ เท่ากับ 2 จากนั้น ในข้อความสั่งบรรทัดที่ 3 เนื่องจากขณะนี้ i ค่าเท่ากับ 1 มันจึงให้ ผลลัพธ์เป็น 3 (j ซึ่งอยู่ในนิพจน์ i + j คือ j ใน q เพราะฉะนั้น มันจะ ไม่เปลี่ยนแปลงค่า ถึงแม้ว่า j ภายใน p มีการเปลี่ยนแปลง)

ดังนั้น ถ้า ส่งโดยชื่อ สำหรับกรณี พารามิเตอร์ y ของ p ใน โปรแกรม ตัวโปรแกรมจะ<u>พิมพ์ 5</u>

ในอดีตนั้น การตีความหมายของ การส่งโดยชื่อ อาร์กิวเมนต์เป็นฟังก์ชัน ซึ่งจะถูกประเมินผล ระหว่างการกระทำการของ โปรซีเดอร์ถูกเรียก

<u>**ตัวอย่าง**</u> Jensen's device ซึ่งใช้ การส่งโดยชื่อ ประยุกต์ กับ การดำเนินการ กับ แถวลำดับทั้งหมด

(Jensen's device uses pass by name to apply an operation to an entire array, as in the following example, in Pascal syntax :)

```
function  sum(a, index, lower, upper : integer) : integer;
var  temp : integer;
begin
    temp := 0
    for  index := lower  to  upper  do
    temp := temp + a;
    sum  := temp;
end;
```

ถ้า a และ index ส่งโดย พารามิเตอร์ชื่อ ดังนั้น รหัสข้างล่างนี้

```
var  x : array[1 . . 10] of integer
     i , xtotal :  integer;
        . . .
        xtotal := sum(x[i], i, 1, 10);
```

การเรียก sum จะคำนวณ ผลรวมของ อีลิเมนต์ทุกตัว x[1] จนถึง x[10]

7.4.3 การตรวจสอบชนิดของ พารามิเตอร์

(Type Checking of Parameters)

ใน strongly typed languages การเรียกโปรซีเดอร์ ต้องถูกตรวจสอบ เพื่อให้ ชนิด และจำนวน ของอาร์กิวเมนต์ ตรงกันกับ พารามิเตอร์ ของโปรซีเดอร์ สิ่งนี้หมายความว่า อันดับแรก โปรซีเดอร์นั้น อาจจะไม่มี **จำนวนตัวแปร** (variable number) ของ พารามิเตอร์ และกฎนั้น ต้องถูกกล่าวไว้สำหรับ **ความเข้ากันได้ของชนิด** (type compatability) ระหว่าง พารามิเตอร์ และอาร์กิวเมนต์ ในกรณีของ การส่งโดยอ้างอิง ปกติ พารามิเตอร์ ต้องมีชนิดเดียวกัน แต่ในกรณีของส่งโดยค่า สิ่งนี้ จะเป็น assignment compatibility ซึ่งกระทำในภาษา Modula-2, Pascal และ Ada

แบบฝึกหัด

1. ภาษา Pascal เสนอแนะว่า ลำดับของข้อความสั่งต่างๆ ในข้อความสั่งเชิงโครงสร้าง เช่น ข้อความสั่ง while นั้น ให้ปิดล้อมด้วยคู่ begin-end ดังนี้

   <while-stmt> ::= while <cond> do <statement>

   <statement> ::= <simple-stmt> | <compound-stmt>

   <simple-stmt> ::= <while-stmt> I . . .

   <compound-stmt> ::= begin <stmt-sequence> end

   <stmt-sequence> ::= <stmt-sequence> ';' <statement> I ∈

   (สัญลักษณ์ ∈ ในกฎไวยากรณ์ ข้อสุดท้าย หมายถึง สายอักขระว่าง)

   สมมติว่า ต้องการ ขจัดคู่ begin-end ใน ข้อความสั่งประกอบ ออกไป และเขียนไวยากรณ์ ดังนี้

   <while-stmt> ::= while <cond> do <stmt-sequence>

   <stmt-sequence> ::= <stmt-sequence> ';' <statements ∈

   <statement> ::= <while-stmt> I . . .

   จงแสดงให้เห็นว่า ไวยากรณ์นี้ กำกวม และจะสามารถ แก้ไขให้ถูกต้อง ได้อย่างไร โดย ไม่กลับไปใช้ข้อตกลงของภาษา Pascal

2. จงแสดงให้เห็นว่า จะทำสำเนา ข้อความสั่ง while ใน Pascal ด้วย ข้อความสั่ง repeat ได้อย่างไร

   (Show how to imitate a while-statement in Pascal with a repeat-statement.)

3. จงแสดงให้เห็นว่า จะเขียนข้อความสั่ง repeat และข้อความสั่ง loop-exit ในภาษา FORTRAN โดยใช้ข้อความสั่ง GOTO อย่างไร

4. จงเปรียบเทียบ วากยสัมพันธ์ ที่เหมือนกัน และที่แตกต่างกัน ของ ข้อความสั่ง case ในภาษา Pascal, Modula-2 และ Ada

5. จงใช้ตัวอย่าง ข้างล่างนี้ พิสูจน์ว่า ข้อความสั่ง GOTO มีความสำคัญ เพื่อทำให้ รหัสชัดเจนและรวบรัดขึ้น (to clear and concise code)

```
for i := 1 to n do begin
    for j := 1 to n do
        if x[i, j] <> 0 then goto reject;
    writeln ('First all-zero row is :', i)
    break;
reject :
end;
```

โปรแกรมนี้ สมมติว่า ให้หา แถวที่เป็นศูนย์ แถวแรก (the first zero row) ใน เมทริกซ์ x[1 .. n, 1 .. n], เมื่อ n >= 1

(a) จงเขียน โปรแกรมนี้ใหม่ โดย ใช้เฉพาะ LOOP-EXIT เช่น ใน ภาษา Modula-2

(b) จงเขียน โปรแกรมนี้ใหม่ ใน ภาษา Pascal และให้ใช้เฉพาะ ข้อความสั่ง while

(c) ท่านเห็นด้วยกับคำถามข้อนี้ หรือไม่ ให้อธิบาย

6. Give the output of the following program (written in Pascal syntax) using the four parameter passing methods discussed in Section 7.4

```
program params;
var i : integer;
    a : array [1 2] of integer;
procedure p(x, y : integer);
begin
    x := x + 1;
    i := i + 1;
    y := y + 1;
end,
begin { main }
    a[1] := 1;
    a[2] := 1;
        i := 1;
```

```
    p(a[i], a[i]);
    writeln(a[1]);
    writeln(a[2]);
end.
```

คำตอบ

The output of the program using each parameter passing mechanism is as follows :

pass by value :	1	1
pass by reference :	3	1
pass by value-result :	2	1
pass by name :	2	2

7. Give the output of the following program using the four parameter passing methods of Section 7.4

```
program partwo;
var i : integer;
    a : array [0 . . 2] of integer;
procedure swap(x, y : integer);
begin
    x := x + y;
    y := x - y;
    x := x - y;
end;
begin { main }
    i := 1;
    a[0] := 2;
    a[1] := 1;
```

```
    a[2] := 0;
    swap(i, a[i]);
    writeln(i);
    writeln(a[0]);
    writeln(a[1]);
    writeln(a[2]);
    swap(a[i], a[i]);
    writeln(a[0]);
    writeln(a[1]);
    writeln(a[2]);
end.
```