บทที่ 9

==========

การทำโปรแกรมเชิงหน้าที่
(Functional programming)

วัตถุประสงค์

1) เพื่อให้ผู้เรียนสามารถแยกความแตกต่างระหว่างการทำโปรแกรมเชิงกระบวนคำสั่งและการทำโปรแกรมไร้กระบวนคำสั่ง
2) เพื่อให้ผู้เรียน เข้าใจหลักการที่สำคัญของการทำโปรแกรมเชิงหน้าที่
3) เพื่อให้สามารถเชื่อมโยงแนวคิดของฟังก์ชันทางคณิตศาสตร์เข้ากับการทำโปรแกรมคอมพิวเตอร์
4) เพื่อแนะนำผู้เรียนให้รู้จักแคลคูลัสแลมป์ดาที่ใช้เป็นโมเดลพื้นฐานของการทำโปรแกรมเชิงหน้าที่
5) เพื่อให้รู้จักภาษาลิสป์และภาษาสคีมที่เป็นภาษาคอมพิวเตอร์ในกลุ่มของภาษาเพื่อการทำโปรแกรมเชิงหน้าที่
6) เพื่อให้ผู้เรียนเข้าใจถึงวิธีการคำนวณค่าของนิพจน์ วิธีการทำงานกับโครงสร้างข้อมูลลิสต์ และวิธีการนิยามฟังก์ชันทั้งที่เป็นฟังก์ชันปกติและฟังก์ชันแบบเวียนบังเกิด
7) เพื่อแนะนำผู้เรียนให้รู้จักวิธีการใช้ฟังก์ชันอันดับสูง ซึ่งเป็นแนวคิดสำคัญที่ช่วยให้การทำโปรแกรมเชิงหน้าที่มีประสิทธิภาพในการทำงานสูง

แนวทางหลักๆของวิธีการทำโปรแกรมจำแนกได้เป็นสี่แนวทางคือ การทำโปรแกรมเชิงคำสั่ง การทำโปรแกรมเชิงวัตถุ การทำโปรแกรมเชิงหน้าที่ และการทำโปรแกรมเชิงตรรกะ ถ้าพิจารณาจากวิธีการสั่งงานของโปรแกรม การทำโปรแกรมทั้งสี่แนวทางสามารถถูกจัดกลุ่มรวมกันได้เป็นสองกลุ่มคือ กลุ่มการทำโปรแกรมเชิงกระบวนคำสั่ง (procedural programming) และกลุ่มการทำโปรแกรมไร้กระบวนคำสั่ง (non-procedural programming) หรือสามารถเรียกได้อีกชื่อหนึ่งว่า การทำโปรแกรมเชิงประกาศ (declarative programming) การทำโปรแกรมเชิงกระบวนคำสั่ง เช่น การเขียนโปรแกรมด้วยภาษา C หรือภาษา Java จะใช้วิธีระบุขั้นตอนการแปรค่าของตัวแปรหรือของวัตถุไปตามลำดับจนกระทั่งได้ผลลัพธ์สุดท้ายที่ต้องการ ในขณะที่การทำโปรแกรมเชิงประกาศ จะใช้วิธีเขียนข้อความประกาศกำหนดการเชื่อมโยงจากค่าที่เป็นข้อมูลเข้าไปเป็นค่าผลลัพธ์ที่ต้องการ โดยไม่ต้องระบุขั้นตอนการคำนวณโดยละเอียดเหมือนในวิธีทำโปรแกรมเชิงกระบวนคำสั่ง

9.1 การทำโปรแกรมเชิงกระบวนคำสั่งและไร้กระบวนคำสั่ง

(Procedural and non-procedural programming)

การทำโปรแกรมเชิงกระบวนคำสั่ง เป็นวิธีการสั่งงานคอมพิวเตอร์ที่ได้รับอิทธิพลโดยตรงจากสถาปัตยกรรมของเครื่องคอมพิวเตอร์ ที่มักจะเรียกกันว่าเครื่องฟอนนอยมันน์ (von Neumann machine) ที่มีส่วนประกอบหลักคือ หน่วยประมวลผลกลางและหน่วยความจำ

หน่วยความจำทำหน้าที่เก็บชุดคำสั่งของโปรแกรมและข้อมูลที่ต้องการใช้ ในขณะที่การประมวลผลคำสั่งเป็นหน้าที่ของหน่วยประมวลผลกลาง การควบคุมการประมวลผลของโปรแกรมโดยส่วนใหญ่จะใช้วิธีสั่งด้วยข้อความสั่งกำหนดค่า (assignment statement) ดังตัวอย่างซูโดโค้ดต่อไปนี้

```
Program Modulo
(1)  read M1
(2)  read M2
(3)  M1 = M1 - M2
(4)  If M1 >= 0 then goto (3)
(5)  M1 = M1 + M2
(6)  write M1
(7)  halt
```

ข้อความสั่งกำหนดค่า เช่น ข้อความสั่งที่ (3) และ (5) ในโปรแกรม Modulo จะทำให้สถานะของคอมพิวเตอร์เปลี่ยนแปลง เมื่อโปรแกรมทำงานจบ สถานะสุดท้ายของคอมพิวเตอร์จะเป็นตัวบอกผลลัพธ์ของโปรแกรม สถานะของคอมพิวเตอร์จะพิจารณาจากค่าที่ปรากฏในหน่วยความจำ ดังตัวอย่างในรูปที่ 9.1

ลำดับการประมวลผลคำสั่ง (Thread)	สถานะของคอมพิวเตอร์ (State)			
	ข้อมูลเข้า	ตำแหน่งในหน่วยความจำ		ผลลัพธ์
		M1	M2	
	13,4			
(1) read M1		13		
(2) read M2		13	4	
(3) M1 = M1 – M2		9	4	
(4) if M1 >= 0 goto (3)		9	4	
(3) M1 = M1 – M2		5	4	
(4) if M1 >= 0 goto (3)		5	4	
(3) M1 = M1 – M2		1	4	
(4) if M1 >= 0 goto (3)		1	4	
(3) M1 = M1 – M2		-3	4	
(4) if M1 >= 0 goto (3)		-3	4	
(5) M1 = M1 + M2		1	4	
(6) write M1		1	4	1
(7) halt		1	4	

รูปที่ 9.1 สถานะของคอมพิวเตอร์ขณะประมวลผลโปรแกรม Modulo

การทำโปรแกรมเชิงกระบวนคำสั่ง จึงเป็นวิธีการทำโปรแกรมที่ใช้คำสั่งกำหนดค่าเป็นหลักในการควบคุมการทำงานของเครื่องคอมพิวเตอร์ โดยมีคำสั่งประเภทอื่น เช่น คำสั่งวนซ้ำ (for, while, repeat, etc.) คำสั่งมีเงื่อนไข (if, switch, etc.) ช่วยประกอบในการกำหนดลำดับการทำงาน การเขียนโปรแกรมด้วยภาษาเชิงกระบวนคำสั่ง (เช่น ภาษา Pascal, C/C++, FORTRAN, BASIC, etc.) จึงเป็นลักษณะการเขียนโปรแกรมที่มีแนวคิดหลักเกี่ยวกับการเปลี่ยนสถานะ(หรือเปลี่ยนค่าในหน่วยความจำ)ของเครื่องคอมพิวเตอร์ไปตาม ลำดับ จนกระทั่งได้ผลลัพธ์สุดท้ายที่ต้องการซึ่งถือได้ว่าลักษณะทางฮาร์ดแวร์และลักษณะการทำงานของเครื่องคอมพิวเตอร์มีอิทธิพลต่อวิธีการคิดและวิธีการเขียนโปรแกรมของโปรแกรมเมอร์

การทำโปรแกรมแบบไร้กระบวนคำสั่ง เป็นลักษณะของวิธีการทำโปรแกรมที่ถูกออกแบบมาเพื่อลดอิทธิพลของลักษณะการทำงานของเครื่องคอมพิวเตอร์ที่มีต่อแนวคิดในการเขียนโปรแกรม เพื่อให้การเขียนโปรแกรมเป็นอิสระจากฮาร์ดแวร์ เมื่อสถาปัตยกรรมด้านฮาร์ดแวร์เปลี่ยนแปลงไป ตัวโปรแกรมที่เป็น

ส่วนซอฟต์แวร์ไม่จำเป็นต้องเปลี่ยนแปลงตาม ภาษาในกลุ่มไร้กระบวนคำสั่งประกอบด้วย ภาษาเชิงตรรกะและภาษาเชิงหน้าที่ ภาษาเชิงตรรกะใช้ความสัมพันธ์หรือรีเลชันเป็นส่วนประกอบหลักในการเขียนโปรแกรม การทำงานของโปรแกรมคือการพิสูจน์ความจริงของความสัมพันธ์ทั้งหลายที่ระบุอยู่ในโปรแกรม โดยโปรแกรมเมอร์ไม่ต้องระบุลำดับการทำงานให้กับโปรแกรม ในขณะที่ภาษาเชิงหน้าที่ ใช้ฟังก์ชันที่คล้ายกับฟังก์ชันทางคณิตศาสตร์ เป็นส่วนประกอบหลักของโปรแกรม การเขียนโปรแกรมเชิงหน้าที่ จึงเป็นการระบุเชื่อมโยงค่าจากข้อมูลเข้าไปยังผลลัพธ์ที่จะเกิดขึ้น โดยไม่ต้องระบุขั้นตอนการเปลี่ยนแปลงสถานะภายในของคอมพิวเตอร์

9.2 หลักการทำโปรแกรมเชิงหน้าที่

(Principles of functional programming)

ภาษาคอมพิวเตอร์ที่ออกแบบมาเพื่อการทำโปรแกรมเชิงหน้าที่จะมีส่วนประกอบที่สำคัญ 3 ส่วน คือ

(1) ส่วนโครงสร้างข้อมูลพื้นฐาน ได้แก่ โครงสร้างข้อมูลเชิงรายการ หรือ ลิสต์ (list) เพื่อใช้เก็บข้อมูลประเภทต่างๆ ภาษาการทำโปรแกรมเชิงหน้าที่มักจะไม่ใช้โครงสร้างข้อมูลอาร์เรย์เพราะมีข้อจำกัดที่ต้องเก็บข้อมูลประเภทเดียวกัน และไม่ใช้โครงสร้างเรคคอร์ดหรือโครงสร้างอื่นๆที่ซับซ้อนมากเกินไป

(2) ส่วนของฟังก์ชันพื้นฐานเพื่อการทำงานกับลิสต์ เช่น ฟังก์ชันในการสร้างลิสต์ ฟังก์ชันเรียกใช้ข้อมูลจากลิสต์

(3) ส่วนโครงสร้างที่ช่วยสร้างฟังก์ชันจากฟังก์ชันที่มีอยู่ เรียกว่าฟังก์ชันอันดับสูง (higher-order function)

โปรแกรมเชิงหน้าที่เป็นโปรแกรมที่ประกอบด้วยฟังก์ชัน ซึ่งนิยามขึ้นจากนิพจน์ (expression) โดยนิพจน์อาจจะเป็นค่าเดี่ยวๆ หรือประกอบขึ้นจากนิพจน์ย่อยอื่นๆ หรือเป็นนิพจน์แบบมีเงื่อนไข ส่วนประกอบของฟังก์ชันจะมีเพียงนิพจน์ โดยไม่มีข้อความสั่ง (statement) เมื่อต้องการสั่งทำงานซ้ำ การทำโปรแกรมเชิงหน้าที่จะใช้วิธีสร้างฟังก์ชันเวียนบังเกิด (recursive function) ซึ่งก็คือฟังก์ชันที่เรียกตัวเองซ้ำ ดังตัวอย่างเปรียบเทียบในรูปที่ 9.2

```
int fact (int n) {
int i = 1;
for (int j = n; j > 1; --j)
     i = i * j;
return i;
}
```

a) โปรแกรมในภาษา C++

```
fun fact (0) = 1
| fact (n) = n * fact(n-1);
```

(b) โปรแกรมในภาษา ML

รูปที่ 9.2 เปรียบเทียบการทำโปรแกรมเชิงคำสั่งและการทำโปรแกรมเชิงหน้าที่

สังเกตได้ว่าในการทำโปรแกรมเชิงคำสั่ง เมื่อประกาศชื่อฟังก์ชัน fact() ส่วนบอดี้ของฟังก์ชันคือข้อความสั่งต่างๆที่อยู่ภายในเครื่องหมายวงเล็บปีกกา {…} แต่ในการทำโปรแกรมเชิงหน้าที่ เมื่อประกาศชื่อฟังก์ชัน fact() ส่วนบอดี้ของฟังก์ชันคือส่วนที่อยู่ด้านขวามือของเครื่องหมายเท่ากับ (=) ซึ่งมีลักษณะเป็นนิพจน์ไม่ใช่ข้อความสั่ง ซึ่งตัวอย่างในรูปที่ 9.2 นิยามฟังก์ชันเป็นสองกรณีตามค่าพารามิเตอร์ที่รับเข้ามา เมื่อค่าที่รับเข้าเป็นค่าศูนย์ fact() จะให้ผลลัพธ์เป็นนิพจน์ 1 แต่ถ้าค่าที่รับเข้าเป็นค่าที่ไม่ใช่ศูนย์ ฟังก์ชัน fact() จะให้ผลลัพธ์เป็นนิพจน์ n * fact(n-1)

การประมวลผลโปรแกรมเชิงหน้าที่ จะใช้กลไกพื้นฐานสองประการคือ การเชื่อมโยง (binding) ระหว่างชื่อกับค่า และการเรียกใช้ (application) ฟังก์ชัน จากตัวอย่างในรูปที่ 9.2 (b) โปรแกรมหาค่าแฟคตอเรียลนี้จะเริ่มประมวลผลเมื่อถูกเรียกใช้ เช่น เมื่อเราใช้คำสั่ง fact(3) ขั้นตอนแรกของการประมวลผลคือเชื่อมโยงค่า 3 เข้ากับชื่อ n ในโปรแกรม ขั้นตอนต่อมาคือการเรียกใช้ฟังก์ชันที่ n มีค่าเป็น 3

```
fact(3) = 3 * fact(3-1);
```

ซึ่งจะหาผลลัพธ์ของนิพจน์ 3 * fact(3-1) ได้ก็ต่อเมื่อเราทราบค่าของ fact(2) ฟังก์ชันจึงถูกเรียกใช้อีกครั้งด้วยการเชื่อมโยงค่า 2 เข้ากับชื่อ n

```
fact(2) = 2 * fact(2-1);
```

และการเรียกใช้ฟังก์ชันจะเกิดขึ้นซ้ำอีกครั้ง แต่ด้วยค่าพารามิเตอร์ที่ต่างจากเดิม

```
fact(1) = 1 * fact(1-1);
```

และสุดท้ายฟังก์ชันในกรณีฐาน (base case) จะถูกเรียกใช้และทำให้การเรียกตัวเองซ้ำสิ้นสุด

```
fact(0) = 1
```

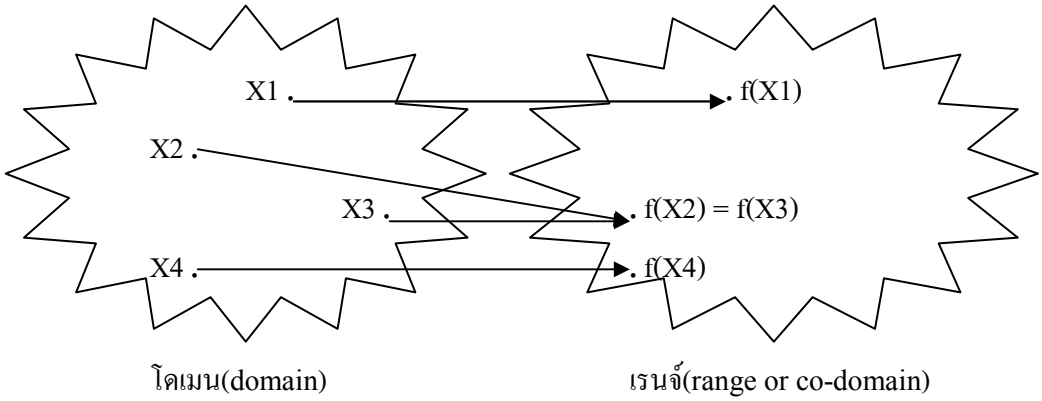
จากนั้นค่า 1 จะถูกส่งย้อนกลับไปให้นิพจน์ที่ยังมีการคำนวณค้างอยู่ เพื่อคำนวณ

```
fact(1) = 1 * fact (0) = 1 * 1 = 1
fact(2) = 2 * fact (1) = 2 * 1 = 2
fact(3) = 3 * fact (2) = 3 * 2 = 6
```

9.3 ฟังก์ชัน

(Function)

ในทางคณิตศาสตร์ฟังก์ชันคือการเชื่อมโยงข้อมูลจากเซตของโดเมนไปยังข้อมูลในเซตของเรนจ์ (หรือโคโดเมน) ดังแสดงด้วยแผนภาพในรูปที่ 9.3



รูปที่ 9.3 ฟังก์ชัน f เชื่อมโยงค่า x จากเซตของโดเมนไปยัง f(x) ในเซตของเรนจ์

การเชื่อมโยงข้อมูล (mapping) ถ้ากระทำบนข้อมูลทุกตัวในเซตของโดเมนจะเรียกฟังก์ชันนั้นว่า ฟังก์ชันสมบูรณ์ (total function) แต่ถ้ามีการเชื่อมโยงข้อมูลเพียงบางตัวในเซตของโดเมน จะเรียกว่า ฟังก์ชันบางส่วน (partial function) แต่ไม่ว่าจะเป็นฟังก์ชันแบบใด ข้อมูลหนึ่งตัวจากเซตของโดเมนจะเชื่อมโยงไปยังข้อมูลเพียงตัวเดียวในเซตของเรนจ์ คุณสมบัตินี้เป็นคุณสมบัติพื้นฐานของทุกฟังก์ชันและช่วยให้การนิยามฟังก์ชันไม่กำกวม

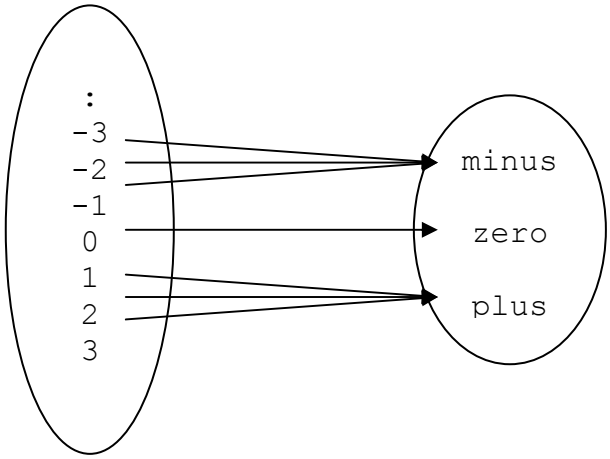
การอธิบายการเชื่อมโยงค่าจากข้อมูลในเซตของโดเมนไปยังข้อมูลในเซตของเรนจ์ทำได้ 3 วิธีคือ

1) โดยการเขียนแจกแจงทุกการเชื่อมโยง
2) โดยการอธิบายด้วยแผนภาพ
3) โดยการอธิบายในรูปของกฎหรือสมการ

รูปที่ 9.4 แสดงตัวอย่างการอธิบายการเชื่อมโยงข้อมูลในทั้งสามรูปแบบ ในตัวอย่างเป็นการเชื่อมโยงจำนวนเลขแต่ละจำนวนไปยังสามค่า คือ ค่าบวก, ค่าลบ, และค่าศูนย์

```
.
.
.
sign(-3) = minus
sign(-2) = minus
sign(-1) = minus
sign(0) = zero
sign(1) = plus
sign(2) = plus
sign(3) = plus
.
.
.
```

(a) การอธิบายการเชื่อมโยงข้อมูลโดยการเขียนแจกแจงทุกการเชื่อมโยง



(b) การอธิบายการเชื่อมโยงข้อมูลโดยใช้แผนภาพ

$$\text{sign}(x) = \begin{cases} \text{minus} & \text{if } x < 0 \\ \text{zero} & \text{if } x = 0 \\ \text{plus} & \text{if } x > 0 \end{cases}$$

(c) การอธิบายการเชื่อมโยงข้อมูลด้วยกฎหรือสมการ

รูปที่ 9.4 รูปแบบต่างๆ ของการอธิบายฟังก์ชัน

การอธิบายการเชื่อมโยงข้อมูลด้วยกฎ(รูปที่ 9.4 c) ชื่อฟังก์ชันคือ sign ตัวแปร x จะเรียกว่าพารามิเตอร์ชนิดฟอร์มอล(formal parameter) ของฟังก์ชัน sign โดย x สามารถแปรเป็นค่าใดในเซตของโดเมนก็ได้ ค่าทางขวามือของกฎ (ได้แก่ minus, zero, plus) คือค่าในเซตของเรนจ์ ดังนั้นฟังก์ชัน sign จัดเป็นฟังก์ชันสมบูรณ์(total function) เพราะมีการเชื่อมโยงทุกค่าในเซตของโดเมนไปยังค่าในเซตของเรนจ์

ถ้ามีการกำหนดฟังก์ชัน sign2 ขึ้นใหม่ดังต่อไปนี้

$$\text{sign2}(x) = \begin{cases} \text{minus} & \text{if } x < 0 \\ \text{plus} & \text{if } x > 0 \end{cases}$$

ฟังก์ชันนี้จะเรียกว่าฟังก์ชันบางส่วน(partial function) เพราะการเชื่อมโยงค่าจากเซตของโดเมนไปยังเซตของเรนจ์ กระทำไม่ครบทุกค่า (ค่า 0 ไม่มีการกำหนดการเชื่อมโยงไว้ในฟังก์ชัน sign2)

ในตัวอย่างของฟังก์ชัน sign(x) ถ้ากำหนดค่า x คือ 6 จากนิยามของฟังก์ชัน ค่า 6 จะเชื่อมโยงไปยังค่า plus ในเซตของเรนจ์ ค่า 6 จะเรียกว่า พารามิเตอร์จริง(actual parameter) เนื่องจากเป็นค่าจริงที่ส่งให้ฟังก์ชัน sign เชื่อมโยงค่า คำว่าพารามิเตอร์ อาจเรียกอีกชื่อได้ว่า อาร์กิวเมนต์(argument) และกระบวนการส่งพารามิเตอร์จริงเพื่อให้ฟังก์ชันเชื่อมโยงค่าไปสู่ค่าในเซตของเรนจ์ เรียกว่า การเรียกใช้ฟังก์ชัน (function application) ตัวอย่างการเรียกใช้ฟังก์ชัน sign ด้วยการส่ง 6 เป็นพารามิเตอร์จริง เขียนได้ดังนี้

```
sign(6)
```

การประมวลผล(evaluate)ของฟังก์ชัน จะเป็นการเชื่อมโยงค่าพารามิเตอร์จริงไปสู่ค่าในเซตของเรนจ์ ตามกฎการเชื่อมโยงที่ได้นิยามเอาไว้ (รูปที่ 9.4 c) สามารถเขียนได้ดังนี้

$$\text{sign}(6) \longrightarrow \text{plus}$$

จากนิยามและลักษณะการประมวลผลของฟังก์ชันจะเห็นได้ว่าตัวแปร เช่น x จะเป็นเพียงชื่อที่ถูกสร้างเตรียมไว้รอการเชื่อมโยงกับค่าที่จะเกิดขึ้นจริงเมื่อมีการเรียกใช้ฟังก์ชัน และเมื่อถูกเชื่อมโยงแล้ว เช่น x มีค่าเป็น 6 ตัวแปรก็จะคงค่านั้นไปตลอดฟังก์ชัน ภาษาเชิงหน้าที่จึงไม่จำเป็นต้องมีคำสั่งกำหนดค่า เพราะตัวแปรจะรับค่าเมื่อฟังก์ชันถูกเรียกใช้และต่อจากนั้นตัวแปรจะไม่ถูกเปลี่ยนแปลงค่าอีก ภาษาเชิงหน้าที่ที่ไม่อนุญาตให้ใช้คำสั่งกำหนดค่า จะเรียกว่า ภาษาเชิงหน้าที่แท้(pure function language) แต่ยังมีภาษาเชิงหน้าที่จำนวนมาก(เช่นภาษา LISP) ที่เตรียมรูปแบบคำสั่งให้ผู้ใช้สามารถกำหนดค่าและเปลี่ยนแปลงค่ากับตัวแปรได้ ทำให้ภาษาเหล่านั้นไม่ใช่ภาษาเชิงหน้าที่แท้

การที่ภาษาเชิงหน้าที่ไม่มีคำสั่งกำหนดค่า เนื่องจากต้องการให้โปรแกรมเป็นอิสระจากฮาร์ดแวร์และต้องการให้การทำงานของฟังก์ชันไม่มีผลข้างเคียง(side effect) ซึ่งลักษณะเช่นนี้จะต่างจากภาษาเชิงคำสั่งที่นำตัวแปรไปผูกติดกับโครงสร้างตำแหน่งของหน่วยความจำ ทำให้ฟังก์ชันสามารถใช้คำสั่งเปลี่ยนแปลงค่าตัวแปรส่วนกลาง (global variables) แล้วส่งผลกระทบต่อการทำงานของฟังก์ชันอื่น

9.4 แคลคูลัสแลมป์ดา

(Lambda calculus)

แคลคูลัสแลมป์ดาเป็นแคลคูลัสอย่างง่าย คิดขึ้นโดย Alonzo Church ในปี ค.ศ. 1941 เพื่อใช้เป็นโมเดลอธิบายการทำงานของฟังก์ชัน การศึกษาแคลคูลัสแลมป์ดาจึงช่วยให้เข้าใจส่วนประกอบต่างๆ ของการทำโปรแกรมเชิงหน้าที่ และช่วยให้เข้าใจความหมายของฟังก์ชันได้โดยไม่ต้องอ้างอิงกับไวยกรณ์ของภาษา คอมพิวเตอร์ใดๆ

ชื่อของแคลคูลัสแลมป์ดาได้มาจากการใช้อักขระในภาษากรีก λ (lambda) เพื่อแทนฟังก์ชัน วิธีการนิยามฟังก์ชันของแคลคูลัสแลมป์ดาแสดงตัวอย่างได้ดังนี้

```
(λx. x * x)
```

อักขระ λ ใช้ระบุว่าเป็นฟังก์ชัน(โดยไม่มีชื่อฟังก์ชัน) ตัวแปร x เป็นพารามิเตอร์ชนิดฟอร์มอลของฟังก์ชัน เครื่องหมายจุด (.)ใช้แยกส่วนบอดี้ออกจากส่วนหัวของฟังก์ชัน บอดี้ของฟังก์ชันนี้คือ x * x ซึ่งหมายถึงฟังก์ชันยกกำลังสอง(square) นั่นเอง การนิยามฟังก์ชันจะมีวงเล็บเปิด " (" เพื่อบอกขอบเขตเริ่มต้นของฟังก์ชันและมีวงเล็บปิด ") " บอกการสิ้นสุดขอบเขตของฟังก์ชัน

การเรียกใช้ฟังก์ชันในแคลคูลัสแลมป์ดา จะใช้วิธีเขียนค่าที่จะส่งเป็นพารามิเตอร์จริงให้กับฟังก์ชันไว้หลังขอบเขตนิยามฟังก์ชัน ดังตัวอย่างต่อไปนี้

```
((λx. x * x)2)
```

เป็นการส่งค่า 2 เป็นพารามิเตอร์จริง เมื่อฟังก์ชันได้รับค่าจะเชื่อมโยงค่า 2 กับตัวแปร x ได้ผลลัพธ์เป็น 2 * 2

การนิยามฟังก์ชันในแคลคูลัสแลมป์ดา จะประกอบไปด้วยส่วนประกอบสองส่วน คือ ส่วนพารามิเตอร์ และส่วนบอดี้ของฟังก์ชัน ทั้งส่วนพารามิเตอร์และส่วนบอดี้สร้างขึ้นจากนิพจน์

<u>นิพจน์แลมป์ดา</u>

นิพจน์ในแคลคูลัสแลมป์ดามีอยู่ 3 ประเภทคือ

(1) ตัวระบุหรือตัวแปรเดี่ยว(single identifier) เช่น x

(2) นิยามฟังก์ชัน(function definition, or abstraction) ซึ่งเขียนอยู่ในรูปแบบ (λx. M) เมื่อ x คือตัวแปรที่ทำหน้าที่เป็นพารามิเตอร์ชนิดฟอร์มอลของฟังก์ชันและ M คือนิพจน์ที่ทำหน้าที่เป็นส่วนบอดี้ของฟังก์ชัน ส่วนบอดี้สามารถเป็นนิพจน์แลมป์ดาประเภทใดก็ได้

(3) การเรียกใช้ฟังก์ชัน(function application) ซึ่งเขียนอยู่ในรูปแบบ (MN) เมื่อM และ N เป็นนิพจน์แลมป์ดา มีความหมายว่าเรียกใช้ฟังก์ชัน M ด้วยการส่งค่า N ให้กับฟังก์ชัน เช่น ((λx. x * x)2) นิพจน์ M คือ (λx. x * x) และนิพจน์ N คือ 2

ทั้งสามตัวอย่างต่อไปนี้จัดเป็นนิพจน์แลมป์ดา

```
x
(λx. x)
((λx.x) (λy.y))
```

การเขียนนิพจน์แลมป์ดาอาจมีการละเว้นไม่เขียนวงเล็บ เช่น (λx. x) อาจเขียนเป็น λx. x การเรียกใช้ฟังก์ชันจะมีลำดับจากซ้ายไปขวา ดังนั้นถ้ามีการละเว้นเครื่องหมายวงเล็บ นิพจน์ E1 E2 E3 จะหมายถึงการเรียกใช้ฟังก์ชันเป็นลำดับ ((E1 E2) E3) และในกรณีที่นิพจน์มีทั้งการนิยามฟังก์ชันและการเรียกใช้ฟังก์ชัน ส่วนของการเรียกใช้ฟังก์ชันจะมีความสำคัญสูงกว่าการนิยามฟังก์ชัน ดังนั้นนิพจน์ λx. y z จะหมายถึง (λx. (y z))

Alonzo Church สร้างแคลคูลัสแลมป์ดาขึ้นมาให้คำนวณฟังก์ชันที่มีพารามิเตอร์ตัวเดียว เช่น (λ x.x * x) ต่อมานักคณิตศาสตร์ชื่อ Haskell B. Curry ได้เสนอเทคนิคเพื่อปรับปรุงแคลคูลัสแลมป์ดาให้ทำงานกับฟังก์ชันที่มีหลายพารามิเตอร์ได้ และเรียกเทคนิคนี้ตามชื่อผู้เสนอว่า *เคอรีอิ้ง* (currying)

เทคนิคเคอรีอิ้ง ใช้การสร้างฟังก์ชันในรูปแบบแคลคูลัสแลมป์ดาที่เทียบเท่ากับฟังก์ชันหลายพารามิเตอร์ ถ้าฟังก์ชัน f เป็นฟังก์ชันที่มี k พารามิเตอร์ ฟังก์ชัน g ที่รับพารามิเตอร์เดียวจะเป็นฟังก์ชันเทียบเท่า (เรียกว่า curried version ของ f) ถ้า f และ g ให้ค่าที่เท่ากันนั่นคือ

$$f(x_1, x_2, \ldots, x_k) = g\ x_1, x_2, \ldots, x_k$$

ตัวอย่างเช่นฟังก์ชันการคูณ f(x, y) = x * y สามารถคำนวณด้วยแคลคูลัสแลมป์ดาโดยแปลงให้อยู่ในรูปแบบเคอรีได้เป็น (λ x.(λy.x * y))รูปแบบเคอรีทำให้การคำนวณฟังก์ชันทำเพียงบางส่วนได้ เช่น ในฟังก์ชัน f ที่เป็นฟังก์ชันการคูณถ้ารู้ค่าพารามิเตอร์ x เพียงค่าเดียวโดย x มีค่าเป็น 2 ฟังก์ชัน f สามารถแปลงให้อยู่ในรูปแบบเคอรีได้เป็น

```
(λx.(λy.x * y))2 => (λy. 2 * y)
```

ทำให้ได้รูปแบบฟังก์ชันชั่วคราวที่เป็นฟังก์ชันทวีคูณและสามารถหาค่าของฟังก์ชันนี้ได้เมื่อรู้ค่า y

ตัวแปรในแคลคูลัสแลมป์ดามีการกำหนดขอบเขตเป็นตัวแปรส่วนกลาง (global variable) และตัวแปรเฉพาะที่ (local variable) เช่นเดียวกับในภาษาคอมพิวเตอร์ระดับสูง เพียงแต่ในแคลคูลัสแลมป์ดาเรียกตัวแปรเฉพาะที่ซึ่งมีขอบเขตการเรียกใช้อยู่เฉพาะภายในฟังก์ชันว่าตัวแปรถูกยึดเหนี่ยว (bound variable) และเรียกตัวแประส่วนกลางที่การเชื่อมโยงค่าอยู่นอกขอบเขตของฟังก์ชันว่าตัวแปรอิสระ (free variable) ตัวอย่างเช่น นิพจน์แลมป์ดาที่นิยามฟังก์ชันยกกำลัง k

$$(\lambda x.x^k)$$

ตัวแปร x คือตัวแปรถูกยึดเหนี่ยว และตัวแปร k คือตัวแปรอิสระ เนื่องจากตัวแปรถูกยึดเหนี่ยวมีขอบเขตอยู่เพียงภายในฟังก์ชันเท่านั้น เราจึงสามารถเปลี่ยนชื่อตัวแปรถูกยึดเหนี่ยวได้โดยไม่เกิดผลกระทบกับฟังก์ชันอื่น เช่น นิพจน์แลมป์ดาที่นิยามฟังก์ชันยกกำลัง k สามารถเปลี่ยนชื่อตัวแปรเป็น

$$(\lambda y.y^k)$$

<u>*การคำนวณค่านิพจน์แลมป์ดา*</u>

แคลคูลัสแลมป์ดานิยามฟังก์ชันโดยใช้นิพจน์ และเรียกใช้ฟังก์ชันด้วยนิพจน์เช่นเดียวกัน เมื่อมีการเรียกใช้ฟังก์ชันการพิจารณาผลลัพธ์การทำงานของฟังก์ชันจึงเป็นการหาค่าผลลัพธ์ของนิพจน์ วิธีการคำนวณค่าผลลัพธ์ของนิพจน์จะใช้วิธีการแปลงรูปนิพจน์ (rewrite) ไปจนกระทั่งได้รูปแบบปกติ (normal form) นิพจน์ที่เป็นรูปแบบปกติคือนิพจน์ที่ไม่ได้อยู่ในรูปแบบของการเรียกใช้ฟังก์ชัน ตัวอย่างต่อไปนี้แสดงการแปลงรูปนิพจน์ไปสู่รูปแบบปกติ

	$((\lambda y.((\lambda x.xyz)a))b)$	
=>	$((\lambda y.(ayz))b)$	ผลลัพธ์จากการเรียกใช้ฟังก์ชันชั้นในสุดโดยแทน x ด้วย a
=>	(ayz)	ผลลัพธ์จากการเรียกใช้ฟังก์ชันโดยแทน y ด้วย b

การแปลงรูปนิพจน์อาจกระทำจากฟังก์ชันชั้นนอกสุด (แทน y ด้วย b) ซึ่งจะให้ผลลัพธ์ได้เป็นแบบเดียวกันดังนี้

	$((\lambda y.((\lambda x.xyz)a))b)$
=>	$((\lambda x.xbz)a)$
=>	(abz)

จากตัวอย่างจะเห็นว่าถึงแม้จะใช้ลำดับที่ต่างกันในการแปลงรูปนิพจน์ แต่ผลลัพธ์นำไปสู่รูปแบบปกติที่เหมือนกัน คุณสมบัติการมีรูปแบบปกติรูปแบบเดียวของแคลคูลัสแลมป์ดา เรียกว่า มีคุณสมบัติตามทฤษฎีของเชิร์ชและรอสเซอร์ (Church-Rosser theorem)

การคำนวณค่านิพจน์แลมป์ดาจะกระทำตามกฎการแปลงรูปนิพจน์ (rewrite rules) หรือบางครั้งเรียกว่ากฎการลดรูปนิพจน์ (reduction rules) ซึ่งประกอบด้วยกฎที่สำคัญ 2 ข้อดังนี้

<u>*กฎข้อที่1*</u> *:* การเปลี่ยนชื่อตัวแปรหรือเรียกว่า การแปลงแบบอัลฟา (α-conversion)

$$\lambda x.M \;\; {}_{\alpha} \Rightarrow \;\; \lambda y.\{y/x\}M, \text{ y is not free in M}$$

กฎการเปลี่ยนชื่อตัวแปรกำหนดว่าถ้า x เป็นตัวแปรถูกยึดเหนี่ยวเราสามารถเปลี่ยนชื่อตัวแปร x เป็น y แต่ทั้งนี้ชื่อ y จะต้องไม่ปรากฏเป็นตัวแปรอิสระใน M สัญลักษณ์ {y/x} หมายถึงการแทนชื่อ x ด้วย y

การคำนวณค่าของนิพจน์แลมป์ดาจะเกิดขึ้นเมื่อมีการเรียกใช้ฟังก์ชันและถ้าส่วนนิยามฟังก์ชันมีชื่อตัวแปรถูกยึดเหนี่ยวซ้ำกับชื่อตัวแปรในส่วนที่เรียกใช้ฟังก์ชันจะต้องมีการเปลี่ยนชื่อตัวแปรเพื่อป้องกันการคำนวณค่าผิดพลาดดังตัวอย่างต่อไปนี้ที่นิยามฟังก์ชัน (λxyz.xz(yz)) แล้วเรียกใช้ด้วยการส่งค่า (λx.x) ให้พารามิเตอร์ตัวแรก และค่า (λx.x) ให้พารามิเตอร์ตัวที่สอง

(λxyz.xz(yz)) (λx.x) (λx.x)

ชื่อตัวแปรถูกยึดเหนี่ยว x ใน (λx.x) ทั้งสองเทอมจึงต้องถูกเปลี่ยนชื่อตามกฎการแปลงแบบอัลฟาเพื่อป้องกันไม่ให้เกิดกรณีชื่อซ้ำ ซึ่งจะทำให้เกิดการผิดพลาดในระหว่างการคำนวณค่าของนิพจน์ ถ้าเปลี่ยนชื่อ x ในเทอมแรกเป็น u และในเทอมหลังเป็น v จะได้ผลลัพธ์ดังนี้

(λxyz.xz(yz)) (λu.u) (λv.v)

<u>กฎข้อที่2</u> : การลดรูปนิพจน์หรือเรียกว่า การลดรูปแบบเบต้า (β-reduction)

λx.M $_\beta$ => {N/x}M

กฎการลดรูปกำหนดว่าเมื่อเรียกใช้ฟังก์ชัน (λx.M) ด้วยค่า N ผลลัพธ์ที่เกิดขึ้นคือนิพจน์ M ที่ตัวแปรถูกยึดเหนี่ยว x ถูกแทนที่ด้วย N เช่นจากตัวอย่างก่อนหน้านี้

(λxyz.xz(yz)) (λu.u) (λv.v)

แสดงการคำนวณค่านิพจน์แลมป์ดาด้วยกฎการลดรูปแบบเบต้า เป็นลำดับขั้นตอนได้ดังนี้ (แสดงเทอมที่จะถูกลดรูปด้วยการขีดเส้นใต้)

λ<u>x</u>yz.<u>x</u>z(yz)) <u>(λu.u)</u> (λv.v)
$_\beta$=> (λ<u>y</u>z.(λu.u)z(<u>y</u>z)) <u>(λv.v)</u>
$_\beta$=> (λz.(λu.u)z((λ<u>v</u>.<u>v</u>)<u>z</u>))
$_\beta$=> (λz.(λ<u>u</u>.<u>u</u>)<u>z</u>(z))
$_\beta$=> (λz.(z)(z))
=> (λz.zz)

ตัวอย่างต่อไปนี้แสดงการคำนวณค่านิพจน์แลมป์ดาโดยใช้กฎการลดรูปแบบเบต้า

ตัวอย่างที่1 : (λ<u>v</u>.<u>v</u>)<u>c</u> $_\beta$=> c

ตัวอย่างที่2 : (λ<u>v</u>.x(<u>v</u> c) <u>v</u>)(<u>a y</u>) $_\beta$=> x((a y) c)(a y)

ตัวอย่างที่3 : (λ<u>v</u>.c)(<u>x v</u>) $_\beta$=> c

ตัวอย่างที่4 : (λ<u>v</u>.<u>v</u> c)(<u>λx.x a</u>) $_\beta$=> (λ<u>x</u>.<u>x</u> a)<u>c</u> $_\beta$=> c a

ตัวอย่างที่5.1 : (λx.plus x 1) ((λy.times y y) 3)

```
β=>  plus((λy.times y y) 3) 1
β=>  plus(times 3 3) 1
```

ตัวอย่างที่5.2 : (λx.plus x 1) ((λy.times y y) 3)

```
β=>  (λx.plus x 1) (times 3 3)
β=>  plus(times 3 3) 1
```

ในแคลคูลัสแลมป์ดาแท้ (pure lambda calculus) นิพจน์มีเพียง 3 ประเภทคือ ตัวแปรเดี่ยว, คำนิยามฟังก์ชัน, การเรียกใช้ฟังก์ชัน โดยไม่ได้รวมค่าคงที่และฟังก์ชันการคำนวณคณิตศาสตร์ดังนั้น (times 3 3) จึงปรากฏเป็นผลลัพธ์สุดท้ายในตัวอย่างข้างต้นโดยไม่มีการลดรูปต่อไปเป็น (9) และยิ่งไปกว่านั้นในแคลคูลัสแลมป์ดาแท้ไม่มีการใช้จำนวนเลข เช่น 3 แต่จะกำหนดทุกอย่างอยู่ในรูปของตัวแปรและฟังก์ชัน เช่น ถ้ากำหนดให้ x แทนค่าศูนย์และ f(x) เป็นฟังก์ชันที่เพิ่มค่า x ขึ้นหนึ่งค่า เราสามารถเขียนค่าคงที่เป็นจำนวนเลขในรูปแบบของแคลคูลัสแลมป์ดาแท้ได้ดังนี้

```
0 ≡ λf.λx.x
1 ≡ λf.λx.f(x)
2 ≡ λf.λx.f(f(x))
3 ≡ λf.λx.f(f(f(x)))
.
.
.
n ≡ λf.λx.f^n(x)
```

การเขียนค่าคงที่และฟังก์ชันพื้นฐานทางคณิตศาสตร์ เช่น +, -, * ในรูปแบบของแคลคูลัสแลมป์ดาก็ทำได้ไม่สะดวกจึงมีการปรับปรุงเป็นแคลคูลัสแลมป์ดาประยุกต์ (applied lambda calculus) ที่อนุญาตให้นิพจน์เป็นค่าคงที่และฟังก์ชันพื้นฐานต่างๆ ได้ (เพื่อใช้แทนโอเปอเรเตอร์ เช่น +, *) ดังนั้นกฎการคำนวณค่าของนิพจน์แลมป์ดาจึงต้องมีการกำหนดเพิ่มเติมเพื่อให้สามารถคำนวณค่าคงที่และฟังก์ชันพื้นฐานต่างๆ ได้

<u>กฎข้อที่3</u> : การกำจัดฟังก์ชนซ้ำซ้อนหรือเรียกว่า การแปลงแบบเอทต้า (η-conversion)

```
(λx.E x)  η =>  E , x is not free in E
```

<u>กฎข้อที่4</u> : การลดรูปนิพจน์ที่มีค่าคงที่หรือเรียกว่า การลดรูปแบบเดลต้า (δ-reduction)
เช่น ถ้า if, true, false เป็นค่าคงที่กฎการลดรูปค่าคงที่เหล่านี้คือ

```
if  true  M N  δ => M
if  false M N  δ => M
```

โดยที่ค่าคงที่ true และ false นิยามได้ตามลำดับดังนี้

```
λx.λy. x
λx.λy. y
```

แคลคูลัสแลมป์ดาเป็นรูปแบบอย่างเป็นทางการที่ถูกสร้างขึ้นมาเพื่ออธิบายขั้นตอนการทำงานของฟังก์ชันโดยเป็นการแปลงรูปของสัญลักษณ์ไปสู่รูปแบบปกติซึ่งจะเป็นผลลัพธ์สุดท้ายของฟังก์ชัน ในแคลคูลัสแลมป์ดาถือฟังก์ชันเป็นค่าที่มีความสำคัญสูง (first-class value) โดยฟังก์ชันสามารถเป็นค่าในนิพจน์, เป็นพารามิเตอร์ในฟังก์ชันอื่นและเป็นข้อมูลเหมือนกับข้อมูลพื้นฐานอื่นๆ นอกจากนี้การทำงานกับฟังก์ชันในแคลคูลัสแลมป์ดาจะไม่มีผลข้างเคียงนั่นคือ ลำดับการทำงานกับฟังก์ชันจะเป็นอย่างไรก็ได้โดยผลลัพธ์ของฟังก์ชันจะเป็นเช่นเดิมไม่เปลี่ยนแปลง แคลคูลัสแลมป์ดาจึงเป็นต้นแบบของการพัฒนาภาษาการทำโปรแกรมเชิงหน้าที่ทั้งหลายที่เกิดขึ้นภายหลัง เช่น ภาษา LISP, Scheme, ML, Miranda, Haskell

9.5 ภาษาลิสป์และสคีม

(LISP and Scheme languages)

ภาษา LISP (List and Symbol Processing Language) ถูกสร้างขึ้นโดย John Macarthy ในปี ค.ศ. 1958 ที่สถาบันแห่งรัฐแมสซาชูเสทหรือเอ็มไอที (Massachusetts Institute of Technology, MIT) จึงจัดเป็นภาษาคอมพิวเตอร์ระดับสูงที่มีอายุเก่าแก่เป็นอันดับสองรองจากภาษาฟอร์แทรน ภาษา LISP ถูกสร้างขึ้นมาเพื่อประมวลผลกับข้อความประเภทสัญลักษณ์ต่างๆ จึงเหมาะสมที่จะใช้เป็นภาษาในการทำโปรแกรมในงานปัญญาประดิษฐ์ (artificial intelligence) แต่ถ้าเป็นการประมวลผลกับตัวเลขภาษา LISP จะทำงานได้ช้ากว่าภาษาฟอร์แทรนมาก

ภาษา LISP ที่ถูกสร้างขึ้นครั้งแรกเป็นภาษาเชิงหน้าที่อย่างแท้จริงเพราะการทำงานทุกอย่างกระทำผ่านฟังก์ชันไม่มีการใช้ตัวแปรส่วนกลางและไม่มีการใช้คำสั่งกำหนดค่าให้ตัวแปร ค่าของตัวแปรจะเกิดจากการส่งผ่านค่าระหว่างฟังก์ชันเท่านั้น แต่ภาษา LISP ในรุ่นหลัง เช่น Common LISP อนุญาตให้ใช้คำสั่งกำหนดค่าจึงทำให้ไม่เป็นภาษาเชิงหน้าที่อย่างแท้จริง ภาษา LISP จัดเป็นภาษาแรกที่ริเริ่มให้มีการใช้ฟังก์ชันเรียกตัวเองซ้ำ (recursion), ใช้ฟังก์ชันเป็น first-class value นั่นคือฟังก์ชันปรากฏในนิพจน์ได้ ถูกส่งเป็นพารามิเตอร์ได้ และใช้เป็นข้อมูลได้, และเป็นภาษาแรกที่กำหนดวิธีการเรียกคืนหน่วยความจำที่ไม่ได้ถูกใช้งานแล้ว (garbage collection)

ลักษณะเด่นของภาษา LISP คือรูปแบบของข้อมูลและรูปแบบของโปรแกรมมีลัษณะเหมือนกันคือเขียนอยู่ภายในวงเล็บด้วยรูปแบบนิพจน์พรีฟิกซ์ โปรแกรมในภาษา LISP ประกอบด้วยฟังก์ชันโดยการนิยามฟังก์ชันจะได้พื้นฐานมาจากนิพจน์แลมป์ดา เช่น ฟังก์ชันการบวกในแคลคูลัสแลมป์ดา

```
(λx.λy. x + y)
```

จะมีความหมายเหมือนกับโปรแกรมในภาษา LISP ต่อไปนี้

```
(LAMBDA (x y) (PLUS  x y)
```

การเรียกใช้ฟังก์ชันด้วยการส่งค่า 2 และ 3 เป็นพารามิเตอร์ก็จะมีลักษณะเช่นเดียวกับแคลคูลัสแลมป์ดา

```
((LAMBDA (x y) (PLUS  x y)) 2 3)
```

ฟังก์ชันในภาษา LISP สามารถถูกตั้งชื่อได้ด้วยการใช้คำสั่ง (หรือฟังก์ชัน) DEFINE ดังนี้

```
(DEFINE (ADD (LAMBDA (x y) (PLUS  x y) )))
```

ซึ่งจะช่วยให้เราสามารถเรียกใช้ฟังก์ชันสะดวกขึ้นดังนี้

```
(ADD 2 3)
```

ภาษา Scheme เป็นภาษาที่มีรากฐานมาจากภาษา LISP แต่ปรับปรุงโครงสร้างของภาษาให้มีขนาดเล็กลง และใช้การพิจารณาขอบเขตของตัวแปรแบบคงตัว (static scoping) แทนที่จะเป็นแบบพลวัต (dynamic scoping) เหมือนในภาษา LISP (หมายเหตุตัวแปลภาษาสคีมสามารถดาวน์โหลดได้จากเว็บไซต์ของสถาบัน MIT ที่ URL: Swissnet.ai.mit.edu/projects/scheme/index.html) การแนะนำวิธีการทำโปรแกรมเชิงหน้าที่จะใช้รูปแบบของภาษา Scheme ตัวแปลภาษาจะมีลักษณะของอินเตอร์พรีตเตอร์โดยการสั่งงานจะเป็นลักษณะโต้ตอบเช่น

```
>
   (+ 4 5)
9
```

เป็นการสั่งบวกค่า 4 และ 5 ผลลัพธ์ที่ได้คือ 9 สัญญลักาณ์ '> ' เป็นสัญลักษณ์การพร้อมรับคำสั่งเมื่อจะเลิกใช้งานให้ใช้คำสั่ง

```
> (quit)
```

หรือ

```
> (exit)
```

หรือกดแป้น control-D

9.6 โปรแกรมและนิพจน์

(Program and expression)

โปรแกรมในภาษาสคีมและภาษาลิสป์ จะประกอบขึ้นจากนิพจน์ซึ่งเขียนอยู่ภายในวงเล็บด้วยรูปแบบนิพจน์พรีฟิกซ์ เช่น

```
(+ 2 3)
(- 3 2)
(* 5 (+4 6))
(max 2 3 17)
```

สัญลักษณ์ตัวแรกที่ปรากฏภายในนิพจน์ (เช่น +, -, *, max) จะเป็นโอเปอเรเตอร์หรือชื่อฟังก์ชัน สัญลักษณ์ที่ปรากฏตามมาจะเป็นโอเปอแรนด์ การใช้รูปแบบนิพจน์พรีฟิกซ์จะช่วยให้สามารถเขียนนิพจน์ที่มีความยืดหยุ่นทำงานกับโอเปอแรนด์จำนวนเท่าใดก็ได้ดังตัวอย่างต่อไปนี้ (เครื่องหมาย "; " ที่ปรากฏในตัวอย่างคือ เครื่องหมายบอกข้อความหมายเหตุหรือ comment ในภาษาสคีม)

```
(+)             ; evaluates to 0
(+ 5)           ; evaluates to 5
(+ 5 4 3 2 1)   ; evaluates to 15
(*)             ; evaluates to 1
(* 5)           ; evaluates to 5
(* 1 2 3 4 5)   ; evaluates to 120
```

จากตัวอย่างข้างต้นค่าที่ปรากฏภายในนิพจน์ เช่น (+ 5) จะเป็นค่าเฉพาะที่ภายในนิพจน์นั้นเท่านั้น ถ้าต้องการให้ค่าหรือตัวแปรเป็นส่วนกลางสามารถเรียกใช้ได้จากนิพจน์อื่นๆ ให้ใช้ฟังก์ชัน define เช่น

```
(define f 120)
```

เป็นการกำหนดชื่อตัวแปร f ให้เชื่อมโยงกับค่า 120 และชื่อ f สามารถถูกเรียกใช้ได้ในนิพจน์อื่นๆ ดังนี้

```
f               ; evaluates to 120
(+ f  5)        ; evaluates to 125
```

นิพจน์ในภาษาสคีมจำแนกได้เป็น 4 ประเภทคือ นิพจน์คณิตศาสตร์, นิพจน์ตรรกะ, นิพจน์แบบมีเงื่อนไข และนิพจน์แลมป์ดา

นิพจน์คณิตศาสตร์ (arithmetic expressions) เป็นนิพจน์ที่ให้ค่าเป็นจำนวนเลข เช่น

```
7          ; has the value 7 (7 is called an atom)
(+ f  3)   ; has the value 123
```

นิพจน์ตรรกะ (Boolean expressions) เป็นนิพจน์ที่ให้ค่าเป็นจริง (แทนด้วย #t) และเท็จ (แทนด้วย #f) เช่น

```
(< 1 2)                    ; has the value #t
(>= 3 4)                   ; has the value #f
(= 4 4)                    ; has the value #t
(not(> 5 6))               ; has the value #t
(and(< 3 4) (= 2 3))       ; has the value #f
(or(< 3 4) (= 2 3))        ; has the value #t
```

นิพจน์แบบมีเงื่อนไข (conditional expressions) เป็นนิพจน์เลือกทำงานโดยการเลือกขึ้นอยู่กับเงื่อนไขที่ระบุ โอเปอเรเตอร์ที่ใช้มี 2 ประเภทคือ if และ cond โดยมีรูปแบบการใช้คือ

```
(if <test> <true-expression> <false-expression>)

(cond
     (<tes-exp1> <expression1>)
     (<tes-exp2> <expression2>)
     ...
     (else <else-exp>))
```

ตัวอย่างเช่น

```
(if (> 5 4) 42 99)        ; has the value 42
(if (< 5 4) 42 99)        ; has the value 99
(if (> 5 0) 33)           ; has the value 33
(if (> 5 0) 33)           ; has the value #f

(cond
     ((> 3 2)  "greater ")
     ((< 3 2)  "less "))
                          ; has the value "greater"

(cond
     ((> 3 3)  "greater")
     ((< 3 3)  "less"))
     (else     "equal"))
                          ; has the value "equal")

(define  n  -5)
(cond
     ((< n 0)  "negative")
     ((> n 0)  "positive"))
     (else     "zero"))
                          ; has the value "negative"
```

นิพจน์แลมป์ดา (lambda expressions) เป็นนิพจน์ที่ใช้นิยามฟังก์ชัน ตัวอย่างเช่น

```
(lambda (x) (+ x 1))      ; has the value 'the
                          ; function that adds one
                          ; to its parameter'
((lambda (x) (+ x 1)) 3) ; has the value 4
```

การตั้งชื่อให้กับฟังก์ชันสามารถทำได้ดังนี้

```
(define add (lambda (x) (+ x 1)))
```

หรือใช้รูปย่อที่ไม่ต้องมีคำว่า lambda

```
(define (add x) (+ x 1))
```

และการเรียกใช้ฟังก์ชันที่ได้รับการตั้งชื่อทำได้ดังนี้

```
(add 3)              ; has the value 4
```

ภาษาสคีมมีฟังก์ชันที่ได้รับการนิยามไว้แล้ว (built-in functions) ให้ใช้งานได้ดังต่อไปนี้

- max, min
- +, *, -, /
- quotient, modulo, remainder
- ceiling, floor, abs, magnitude, round, truncate
- gcd, lcm
- exp, log, sqrt
- sin, cos, tan
- <, >, = , <=, >=
- real?, number?, complex?, rational?, integer?
- zero?, positive?, negative?, odd?, even?, exact?
- and, or, not
- equal?, boolean?
- char?, char=?, char<=?, char>=?

ตัวอย่างการใช้งาน

```
(define x 3)              ; x has the value 3
(if (even? x) 7 (+ x 5)) ; expression has the value 8
```

โปรแกรมและข้อมูลในภาษาสคีม (และภาษาลิสป์) จะใช้โครงสร้างเดียวกันคือโครงสร้างของลิสต์ ซึ่งเขียนอยู่ในรูปแบบ

```
(a b c d)
```

ตัวแปลภาษาสคีมจะประมวลผลโปรแกรมด้วยการตีค่า a ว่าคือชื่อฟังก์ชัน และ b, c, d คืออาร์กิวเมนต์ของฟังก์ชัน เช่น

```
(+ 2 3 4)
```

เครื่องหมาย + คือฟังก์ชันการบวก อาร์กิวเมนต์คือ 2, 3, 4 การบวกจะกระทำแบบเรียกตัวเองซ้ำนั่นคือ (+ 2 3) ได้ผลลัพธ์เป็น 5 จากนั้นทำ (+ 5 4) ได้ผลลัพธ์เป็น 9 แต่ ถ้าผู้ใช้เขียนคำสั่ง

```
(2 3 4)
```

ตัวประมวลผลจะพยายามตีค่าว่า 2 คือชื่อฟังก์ชันซึ่งไม่ใช่ ดังนั้นตัวประมวลผลจะส่งข้อความบอกความผิดพลาดขึ้นที่จอภาพ

ถ้าผู้ใช้ต้องการให้ (2 3 4) เป็นรูปแบบของข้อมูล สามารถทำได้ด้วยการใช้คำสั่ง quote เพื่อบังคับตัวประมวลผลให้ละเว้นการประมวลผลโปรแกรมที่พยายามตีค่าว่า 2 คือชื่อฟังก์ชัน การใช้คำสั่ง quote เพื่อระบุว่าส่วนที่ตามมาเป็นข้อมูลทำได้ดังนี้

```
(quote (2 3 4))
```

หรือเขียนในรูปย่อได้ดังนี้

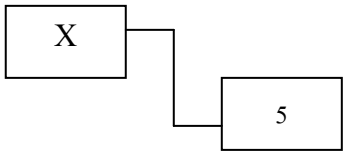
```
'(2 3 4)
```

ตัวอย่างต่อไปนี้แสดงการประมวลผลนิพจน์, ตัวแปรและค่าคงที่ โดยตัวแปรและค่าคงที่ถือเป็นอะตอมที่ไม่ต้องเขียนวงเล็บครอบ

```
(define f 120)        ; binds indentifier f to the
                      ; value 120
f                     ; evaluates to 120
(+ f 5)               ; evaluates to 125
(+)                   ; calls + with no arguments
(+ 5)                 ; calls + with 1 argument
(+ 5 4 3 2 1)         ; calls + with 5 argument
(+ (5 4 3 2 1))       ; error: tries to evaluate
                      ; 5 as a function
(f)                   ; error : f is not a function
5                     ; evaluates to 5
nil                   ; evaluates to nil, predefined
#t                    ; evaluates to true, predefined
(define x f)          ; defines x to be 120
                      ; (value of f)
(define x 'f)         ; defines x to be the symbol f
(define acolor 'red); defines acolor to be red
(define acolor red) ; an error, symbol red not
                      ; defined
(define colors (quote (red yellow green))) ; defines
                      ; colors to be a list of red,
                      ; yellow, green
(define colors '(red yellow green)) ; defines colors
                      ; to be a list of red, yellow,
                      ; green
```

9.7 ชนิดของข้อมูลลิสต์ และการประมวลผลลิสต์
(List and list processing)

โปรแกรมในภาษาการทำโปรแกรมเชิงหน้าที่สามารถใช้ตัวแปรได้ แต่แนวคิดเกี่ยวกับตัวแปรจะต่างจากการทำโปรแกรมเชิงคำสั่งที่เชื่อมโยงชื่อตัวแปรกับตำแหน่งที่ใช้เก็บค่าในหน่วยความจำ ซึ่งทำให้ผู้ใช้สามารถเปลี่ยนแปลงค่าในหน่วยความจำไปอย่างไรก็ได้ผ่านการสั่งกำหนดค่าตัวแปร การทำโปรแกรมเชิงหน้าที่มีการเชื่อมโยงค่ากับตัวแปรโดยไม่มีการเกี่ยวข้องลงไปถึงตำแหน่งในหน่วยความจำ ตัวอย่างเช่น (`define x 5`) เป็นการเชื่อมโยงชื่อ x กับค่า 5 ดังแสดงเป็นแผนภาพได้ตามรูปที่ 9.5 โดยไม่มีการระบุความสัมพันธ์ต่อเนื่องลงไปถึงหน่วยความจำ ดังนั้นเมื่อระบุชื่อ x สิ่งที่ได้จะคือ 5 ซึ่งเป็นค่าที่เชื่อมโยงอยู่กับ x โดยไม่สามารถจะเปลี่ยนแปลงค่าให้เพื่อขึ้นหรือลดลงได้ ทำได้เพียงเปลี่ยนการเชื่อมโยงให้ x ไปเชื่อมโยงกับค่าอื่น



รูปที่ 9.5 ความสัมพันธ์ที่เกิดขึ้นจากการใช้คำสั่ง (`define x 5`)

ตามตัวอย่างในรูปที่ 9.5 ค่า 5 ถือเป็นชนิดข้อมูลพื้นฐาน (primitive data types) ในภาษาสคีมกำหนดชนิดข้อมูลพื้นฐานไว้ 7 ประเภทคือ อักขระ (characters), ข้อความ (strings), ค่าตรรกะ (booleans), เลขจำนวนเต็ม (integers), จำนวนตรรกยะ (rational numbers), เลขจำนวนจริง (real numbers), และเลขจำนวนเชิงซ้อน (complex numbers)

ชนิดข้อมูลคอมโพสิต (composite data type) ในภาษาสคีมมีเพียงประเภทเดียวคือ ลิสต์ (lists)

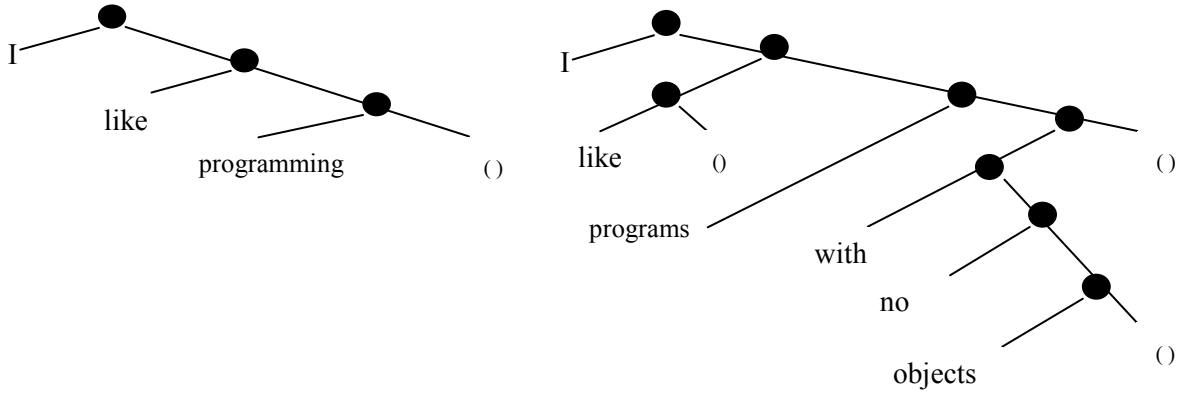
ลิสต์เป็นโครงสร้างข้อมูลเชิงประกอบที่ภายในสามารถมีสมาชิกได้หลายจำนวน สมาชิกที่ปรากฏเป็นตัวแรกในลิสต์จะเรียกว่าหัว (head) และสมาชิกส่วนที่เหลือในลิสต์จะเรียกว่าหาง (tail) เช่น '(a b c) เป็นลิสต์ที่ประกอบขึ้นจากสมาชิกสามตัวส่วนหัวคือ a และส่วนหางคือ ลิสต์ที่เหลือซึ่งได้แก่ '(b c)

ลิสต์คือกลุ่มของข้อมูลที่มีการเรียงลำดับตามตำแหน่งที่ปรากฏภายในลิสต์ จำนวนสมาชิกภายในลิสต์มีได้ตั้งแต่ศูนย์ตัวขึ้นไป ลิสต์ที่ไม่มีสมาชิกเลย เรียกว่าลิสต์ว่าง (empty list, or null list) และเขียนแทนด้วย () หรือใช้ค่าคงที่ nil สมาชิกที่ปรากฏภายในลิสต์ สามารถเป็นค่าประเภทใดก็ได้ เช่น ค่าตรรกะ, ตัวเลข, ตัวแปร, สัญลักษณ์, อักขระ, ข้อความ, ลิสต์, ฟังก์ชัน ตัวอย่างต่อไปนี้แสดงโครงสร้างข้อมูลลิสต์

```
(I like programming)
(I (like) programs (with no objects))
```

ตัวอย่างแรกเป็นลิสต์ที่มีจำนวนสมาชิก 3 ตัว สมาชิกแต่ละตัวเขียนแยกจากกันด้วยการเว้นช่องว่างอย่างน้อยหนึ่งช่องว่าง ดังนั้นสมาชิกในลิสต์นี้จะได้แก่สัญลักษณ์ (symbol) สามตัวคือ I, like,

programming ในตัวอย่างที่สองเป็นลิสต์ที่มีจำนวนสมาชิก 4 ตัว สมาชิกภายในมีทั้งสัญลักษณ์และลิสต์ ได้แก่ I, (like), programs, (with no objects) สมาชิกตัวที่สองและสี่เป็นลิสต์โครงสร้างของลิสต์ตามตัวอย่างข้างต้นทั้งสองตัวอย่างสามารถแสดงในรูปของโครงสร้างทรีได้ดังรูปที่ 9.6



(I like programming) (I (like) programs (with no objects))

รูปที่ 9.6 โครงสร้างลิสต์ที่แสดงในลักษณะของแผนภูมิต้นไม้

จากรูปแผนภูมิต้นไม้จะสังเกตได้ว่าลิสต์ทุกลิสต์จะมีสัญลักษณ์ลิสต์ว่าง (แทนด้วยเครื่องหมาย ()) ปิดท้ายอยู่เสมอ สัญลักษณ์นี้จะแฝงอยู่เพื่อบอกการสิ้นสุดลิสต์ สัญลักษณ์ () ที่ปิดท้ายลิสต์จะเป็นประโยชน์ในการทำงานกับลิสต์แบบเวียนบังเกิด (recursive)

ฟังก์ชันพื้นฐานที่ใช้ในการสร้างลิสต์ได้แก่ฟังก์ชัน list และ quote (ซึ่งเทียบเท่ากับการใช้สัญลักษณ์ ') ตัวอย่างเช่น

```
(list 1 2 (+ 1 2))        ; has value (1 2 3)
(quote (1 2 (+ 1 2)))     ; has value (1 2 3)
'(1 2 (+ 1 2))            ; has value (1 2 3)
```

ฟังก์ชันที่ใช้ประมวลผลข้อมูลลิสต์ประกอบด้วย

(null? x)	; ใช้ตรวจสอบว่า x ป็นลิสต์ว่างหรือไม่ ถ้าใช่จะให้ค่าเป็นจริง
	; แต่ถ้าไม่ใช่จะให้ค่าเป็นเท็จ
(cons a x)	; ใช้เพิ่มสมาชิกใหม่เข้าไปในลิสต์ โดย a จะถูกแทรกเข้าไป
	; เป็นตำแหน่งแรกในลิสต์
(append x y)	; ใช้เชื่อมต่อลิสต์สองลิสต์ทำให้เกิดลิสต์ใหม่ที่สมาชิกส่วนต้น
	; ของลิสต์มาจากลิสต์ x เดิม และสมาชิกส่วนท้ายของลิสต์ใหม่
	; มาจากลิสต์ y เดิม
(car x)	; เป็นฟังก์ชันที่ส่งค่ากลับเป็นสมาชิกตำแหน่งแรกในลิสต์ x

(cdr x) ; เป็นฟังก์ชันที่ส่งค่ากลับเป็นลิสต์ที่ทุกอย่างเหมือนลิสต์ x
; ยกเว้นสมาชิกตัวแรกของ x ที่จะไม่ปรากฏในลิสต์
(length x) ; เป็นฟังก์ชันที่ส่งค่ากลับเป็นจำนวนสมาชิกในลิสต์ x
(reverse x) ; เป็นฟังก์ชันที่ส่งค่ากลับเป็นลิสต์ที่มีสมาชิกเหมือน x
; แต่ลำดับถูกย้อนกลับ

ตัวอย่างการใช้ฟังก์ชันทำงานกับลิสต์แสดงได้ดังนี้

นิพจน์	**ค่าที่ได้**
(define mylist (list 1 2 3 4 5))	mylist
mylist	(1 2 3 4 5)
(length mylist)	5
(reverse mylist)	(5 4 3 2 1)
(cons a mylist)	(a 1 2 3 4 5)
(append '(a b) mylist)	(a b 1 2 3 4 5)
(null? Mylist)	#f
(car mylist)	1
(cdr mylist)	(2 3 4 5)
(car (cdr mylist))	2
(cdr (cdr mylist))	(3 4 5)
(car (cdr (cdr mylist)))	3
(equal? '(1 2) '(1 2))	#t
(equal? 5 5)	#t
(equal? 5 '(5))	#f
(equal? '(1 2) '(2 1))	#f
(equal? () ())	#t

9.8 ฟังก์ชันเวียนบังเกิด

(Recursive function)

การสั่งทำงานซ้ำในการทำโปรแกรมเชิงหน้าที่จะใช้วิธีให้ฟังก์ชันเรียกตัวเองซ้ำ หรือเรียกว่า ฟังก์ชันเวียนบังเกิด ดังตัวอย่างฟังก์ชันการคำนวณค่าแฟคตอเรียลต่อไปนี้

```
(define (factorial n )
     (if (< n 1) 1
          (* n ( factorial (n - 1)))))
```

ลักษณะการเรียกตัวเองซ้ำ เมื่อมีการคำนวณค่า 4! แสดงเป็นขั้นตอนได้ดังนี้

```
(factorial 4)  = (* 4(factorial 3 ))
               = (* 4(* 3(factorial 2 )))
               = (* 4(* 3(* 2 (factorial 1 ))))
               = (* 4(* 3(* 2(* 1(factorial 0 )))))
               = (* 4(* 3(* 2(* 1 1 ))))
               = (* 4(* 3(* 2 1 )))
               = (* 4(* 3 2 ))
               = (* 4 6)
               = 24
```

ตัวแปลภาษาสคีมบางเวอร์ชันมีฟังก์ชัน trace ให้ใช้ติดตามการทำงานของฟังก์ชัน ซึ่งจะแสดงลำดับการเรียกฟังก์ชัน ดังนี้

```
> (trace factorial)
> (factorial 4)
Trace: (factorial 4)
Trace: (factorial 3)
Trace: (factorial 2)
Trace: (factorial 1)
Trace: (factorial 0)
Trace: Value = 1
Trace: Value = 1
Trace: Value = 2
Trace: Value = 6
Trace: Value = 24
24
> (untrace factorial)
> (factorial 4)
24
```

การสร้างฟังก์ชันเวียนบังเกิด จะเริ่มต้นด้วยการกำหนดกรณีพื้นฐาน(base case) ซึ่งเป็นกรณีที่สามารถคำนวณค่าได้ทันทีโดยไม่ต้องเรียกตัวเองซ้ำ เช่น กรณีที่ n มีค่าน้อยกว่า 1 จะให้คำตอบได้ทันทีว่า n! คือ 1 กรณีอื่นที่ไม่ใช่กรณีพื้นฐาน จะคำนวณค่าด้วยการเรียกตัวเองซ้ำ(recursive step) และการเรียกตัวเอง

ซ้ำนั้นจะต้องมีจุดสิ้นสุด จากตัวอย่างการกำหนดค่าแฟคตอเรียลรับประกันการสิ้นสุด เนื่องจากในแต่ละครั้งของการเรียก factorial ค่า n จะถูกลดลงหนึ่งค่า ดังนั้นในที่สุด n จะถูกลดค่าจนกระทั่งเป็นศูนย์และเข้าสู่กรณีพื้นฐาน

จากตัวอย่างฟังก์ชัน sum ต่อไปนี้แสดงการนิยามฟังก์ชันในแบบเรียกตัวเองซ้ำ ฟังก์ชัน sum รับข้อมูลเข้าเป็นลิสต์ของจำนวนเลข เช่น (1 2 3 4) แล้วบวกค่าจำนวนเลขทั้งหมดในลิสต์ ส่งค่ากลับเป็นค่าผลรวมเป็น 10

```
( define (sum alist)
     (if (null? alist) 0
          (+ (car alist) (sum (cdr alist)))))
```

ในตัวอย่างของฟังก์ชัน sum กรณีพื้นฐานคือ เมื่อลิสต์เป็นลิสต์ว่าง จะให้ผลรวมของค่าเป็นศูนย์ แต่ถ้าลิสต์นั้นไม่ใช่ลิสต์ว่าง จะดึงข้อมูลตัวแรกจากลิสต์เพื่อรอบวกกับข้อมูลที่จะดึงจากลิสต์ที่เหลือ ซึ่งแสดงลำดับการทำงานกับลิสต์ (1 2 3 4) ได้ดังนี้

```
(sum '(1 2 3 4)) = (+ 1 (sum '(2 3 4)))
                 = (+ 1 (+ 2 (sum '(3 4))))
                 = (+ 1 (+ 2 (+ 3 (sum '(4)))))
                 = (+ 1 (+ 2 (+ 3 (+ 4 (sum '()))))
                 = (+ 1 (+ 2 (+ 3 (+ 4 0))))
                 = (+ 1 (+ 2 (+ 3 4)))
                 = (+ 1 (+ 2 7))
                 = (+ 1 9)
                 = 10
```

จากตัวอย่างฟังก์ชัน member ต่อไปนี้ ทำหน้าที่ตรวจสอบว่าค่าที่ระบุเป็นสมาชิกอยู่ในลิสต์หรือไม่ ถ้าไม่ใช่ฟังก์ชันจะส่งค่ากลับเป็นลิสต์ว่าง แต่ถ้าใช่ฟังก์ชันจะส่งกลับลิสต์ที่มีค่าที่ระบุเป็นสมาชิกตัวแรก การเรียกใช้ฟังก์ชันแสดงตัวอย่างได้ดังนี้

```
(member 4 '(0 2 4 6 8))        ; return (4 6 8)
(member 1 '(0 2 4 6 8))        ; return ()
(member 2 '((1 2) 3 (4 5)))    ; return ()
(member '(1 2)'((1 2)3 (4 5)); return ((1 2)3(4 5))
```

รายละเอียดคำสั่งในฟังก์ชัน member แสดงได้ดังต่อไปนี้

```
(define (member elm alist)
     (if (null? alist) '()
          (if (equal? elm (car alist)) alist
               (member elm (cdr alist))
          )))
```

การนิยามฟังก์ชันในแบบเวียนบังเกิดเป็นวิธีสั่งทำงานซ้ำที่มีรูปแบบสั้น แต่การใช้เนื้อที่ในหน่วยความจำจะสิ้นเปลืองมากเพราะต้องใช้เนื้อที่สแตกในแต่ละครั้งของการเรียกฟังก์ชัน วิธีปรับปรุง

ประสิทธิภาพการใช้เนื้อที่หน่วยความจำให้ดีขึ้น คือ พยายามเปลี่ยนรูปแบบฟังก์ชันเวียนบังเกิดให้เป็น tail recursion นั่นคือปรับให้ขั้นตอนเรียกตัวเองซ้ำเกิดขึ้นท้ายสุด วิธีนี้จะทำให้ใช้เนื้อที่สแตกน้อยลง

ตัวอย่างต่อไปนี้แสดงการนิยามฟังก์ชัน number-interval ที่ทำหน้าที่สร้างข้อมูลเป็นช่วง เช่น ถ้าผู้ใช้ระบุ (number-interval 1 5) ผลลัพธ์ที่ได้จะเป็น (1 2 3 4 5) ซึ่งเป็นตัวเลขที่อยู่ในช่วง 1 ถึง 5 การนิยามฟังก์ชันจะทำในสองรูปแบบคือ แบบฟังก์ชันเวียนบังเกิดตามปกติ และแบบฟังก์ชันเวียนบังเกิดที่ถูกปรับปรุงให้เป็น tail recursion

<u>Recursive function</u>

```
(define (number-interval f t)
     (if (> f t) '()
          (cons f (number-interval (+ f 1) t))
))
```

<u>Tail-recursive function</u>

```
(define (number-interval-tail f t)
     (reverse (number-interval-help f t '())))
(define (number-interval-help f t result)
     (if (> f t) result
          (number-interval-help (+ f 1) t
           (cons f result))
))
```

สังเกตได้ว่าการปรับปรุงฟังก์ชันเวียนบังเกิดให้เป็น tail recursion จะต้องใช้ฟังก์ชันช่วยงานเพิ่มขึ้นคือ ฟังก์ชัน number-interval-help วิธีการเรียกใช้งานฟังก์ชันแสดงตัวอย่างได้ดังนี้

```
> (number-interval 1 10)
(1 2 3 4 5 6 7 8 9 10)
>(number-interval-tail 10 20)
(10 11 12 13 14 15 16 17 18 19 20)
>(number-interval-tail 20 10)
()
```

9.9 ฟังก์ชันอันดับสูง

(Higher-order function)

ความสามารถที่สำคัญที่สุดของภาษาการทำโปรแกรมเชิงหน้าที่คือ การนิยามและการใช้ฟังก์ชันอันดับสูง ฟังก์ชันจะถูกจัดว่าเป็นฟังก์ชันอันดับสูง ถ้าฟังก์ชันนั้นสามารถรับฟังก์ชันอื่นเป็นพารามิเตอร์และสามารถส่งค่ากลับเป็นฟังก์ชัน

รูปที่ 9.7 แสดงตัวอย่างการนิยามฟังก์ชัน flip ในสองรูปแบบคือ รูปแบบเต็ม ซึ่งเป็นรูปแบบเดียวกับแคลคูลัสแลมป์ดา และรูปแบบย่อที่ละเว้นคำว่า lambda ในคำนิยามฟังก์ชัน

`(define flip (lambda (f)`
`  (lambda (x y)`
`  (f y x))))`
 	
(a) รูปแบบเต็มของฟังก์ชัน
	`(define (flip f)`
	`  (lambda (x y)`
	`  (f y x)))`
	
	(b) รูปแบบย่อ

รูปที่ 9.7 การนิยามฟังก์ชัน flip ที่เป็นฟังก์ชันอันดับสูง

ฟังก์ชัน flip รับพารามิเตอร์ f ที่เป็นฟังก์ชันมีสองพารามิเตอร์ คือ x และ y (สังเกตได้จากข้อความ lambda(x y)) จากนั้นฟังก์ชัน flip ทำคำสั่ง (f y x) ซึ่งเป็นการส่งค่ากลับเป็นฟังก์ชัน f ที่มีพารามิเตอร์ตัวแรกเป็น y และตัวที่สองเป็น x ดังนั้นฟังก์ชัน flip จึงเป็นฟังก์ชันอันดับสูงที่ทำหน้าที่รับฟังก์ชันสองพารามิเตอร์ แล้วส่งฟังก์ชันกลับโดยสลับลำดับของพารามิเตอร์

ตัวอย่างฟังก์ชันอันดับสูงอย่างง่ายอีกตัวอย่างคือฟังก์ชัน negate ที่ทำหน้าที่รับฟังก์ชันอื่นแล้วประมวลผลว่าฟังก์ชันนั้นให้ค่าเป็นจริงหรือเท็จ ถ้าให้ค่าเป็นจริงฟังก์ชัน negate จะทำนิเสธให้กลับเป็นเท็จ และถ้าให้ค่าเป็นเท็จฟังก์ชัน negate จะเปลี่ยนเป็นจริง ในตัวอย่างนี้สิ่งที่ส่งกลับเป็นค่าจริง/เท็จ ไม่ใช่การส่งกลับฟังก์ชันเหมือนในตัวอย่างแรก

รายละเอียดคำสั่งในฟังก์ชัน negate แสดงได้ดังต่อไปนี้

```
(define (negate p)
     (lambda (x)
          (if (p x) #f #t)))
```

ฟังก์ชันคอมโพสิตในทางคณิตศาสตร์ที่เขียนอยู่ในรูปแบบ $f \circ g$ สามาถเขียนอยู่ในรูปแบบของภาษาสคีม (และภาษาลิสป์) ได้ดังนี้

```
(define (compos f g)
     (lambda (x)
          (if (g x))))
```

ภาษาสคีมมักจะมีการใช้ฟังก์ชัน map เป็นฟังก์ชันอันดับสูงรับพารามิเตอร์เป็นฟังก์ชันและใช้ฟังก์ชันนั้นบนข้อมูลทุกตัวในลิสต์ ดังตัวอย่างต่อไปนี้

นิพจน์	ค่าที่ได้
(map square '(1 2 3 4 5))	(1 4 9 16 25)
(map string? List 1 'en "en" 2 'to "to"))	(#f #f #t #f #f #t)
(map (lambda (x) (* 2 x)) (list 10 20 30 40))	(20 40 60 80)

ฟังก์ชัน map ตามตัวอย่างการใช้งานข้างต้น สามารถนิยามได้ดังนี้

```
(define (map f x)
     (cond ((null? x) '())
          (else (cons (f (car x)) (map f(cdr x))))))
```

ฟังก์ชัน filter เป็นฟังก์ชันอันดับสูงอีกฟังก์ชันหนึ่งที่ถูกนำมาใช้งานเมื่อต้องการกรองเฉพาะข้อมูลที่มีคุณสมบัติตรงตามเงื่อนไข เช่น

```
(filter even? '(1 2 3 4 5))
```

เป็นการกรองไว้เฉพาะข้อมูลที่เป็นเลขคู่ ซึ่งผลลัพธ์ที่ได้จะเป็นลิสต์ (2 4) หรือคำสั่งต่อไปนี้เป็นการกรองเฉพาะเลขคี่

```
(filter (negate even?) '(1 2 3 4 5))
```

ผลลัพธ์ที่ได้จะเป็น (1 3 5)

```
(define (filter predicate alist)
   (reverse (filter-help predicate alist '() )))
(define (filter-help predicate alist result)
   (cond ((null? alist) result)
      ((predicate (car alist))
         (filter-help predicate(cdr alist)
            (cons (car alist) result)))
      (else
         (filter-help predicate (cdr alist) result))
))
```

9.10 การทำโปรแกรมเชิงหน้าที่ในภาษาลิสป์และสคีม

(Functional programming in LISP and Scheme languages)

ตัวอย่างต่อไปนี้เป็นการทำโปรแกรมในภาษา LISP เพื่อบวกค่าตัวเลขทั้งหมดที่เก็บอยู่ในอาร์เรย์ ซึ่งในภาษา LISP จะใช้โครงสร้างลิสต์แทนอาร์เรย์ ผลลัพธ์ที่ได้จะเป็นข้อความ "Sum = …" แล้วตามด้วยค่าผลบวกของตัวเลข

```
 1 %lisp
 2 >; Store values as a list of characters
 3 >(define (SumNext V)
 4           (cond ((null V) (progn (print “Sum”) 0))
 5           (T (+ (SumNext (cdr V)) (car V)) ) ))
 6 SUMNEXT
 7 >; Create vector of input values
 8 (defun GetInput(f c)
 9           (cond ((eq c 0) nil)
10           (T (cons (read f) (GetInput f (- c 1))))))
11 GETINPUT
12 >(defun DoIt()
13           (progn
14           (setq infile (open '"lisp.data"))
15           (setq array (GetInput infile (read infile)))
16           (print array)
17           (print (SumNext array))))
18 DOIT
19 >(DoIt)
20
21 (1 2 3 4)
22 “Sum=”
23 10
24 10
```

ภาษา LISP, Scheme และภาษาการทำโปรแกรมเชิงหน้าที่อื่นๆ ถูกออกแบบมาให้ทำงานได้กับการประมวลผลสัญลักษณ์ (symbol processing) ซึ่งต่างจากภาษา FORTRAN ที่ถูกออกแบบมาเพื่อการประมวลผลกับจำนวนเลข (numerical processing) การคำนวณเชิงสัญลักษณ์เช่น การคำนวณสมการแคลคูลัสพื้นฐาน (ตามสูตรการคำนวณในรูปที่ 9.8) จึงทำได้ง่ายในภาษา Scheme โปรแกรมภาษา Scheme เพื่อคำนวณสมการดิฟเฟอเรนเชียล แสดงได้ดังรูปที่ 9.9

$$\frac{d}{dx}(c) = 0 \qquad c \text{ is a constant}$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx} \qquad u \text{ and } v \text{ are functions of } x$$

$$\frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right)\Big/ v^2$$

รูปที่ 9.8 การคำนวณค่าดิฟเฟอเรนเชียล

```
(define (diff x expr)
   (if (atom? expr)
      (if (equal? X expr) 1 0)
      (let ((u (cadr expr)) (v (caddr expr)))
         (case (car expr)
         ((+) (list '+ (diff x u) (diff x v)))
         ((-) (list '- (diff x u) (diff x v)))
         ((*) (list '+ (list '* u (diff x v))
                       (list '* v (diff x u))))
         ((/) (list '/ (list '- (list '* v (diff x u))
                               (list '* u (diff x v))
                               (list '* v v) ))
))))
```

รูปที่ 9.9 โปรแกรมคำนวณค่าสมการดิฟเฟอเรนเชียลในรูปแบบภาษา Scheme

จากโปรแกรมในรูปที่ 9.9 มีการใช้คำสั่ง let ในบรรทัดที่ 4

```
(let ((u (cadr expr)) (v (caddr expr)))
...
)
```

คำสั่ง let ใช้ในการตั้งชื่อให้กับนิพจน์ที่จะถูกอ้างถึงอีกหลายครั้งในฟังก์ชัน เช่น ในตัวอย่างข้างต้น นิพจน์ (cadr expr) จะถูกอ้างถึงด้วยชื่อ u และนิพจน์ (caddr expr) ถูกอ้างถึงด้วยชื่อ v [หมายเหตุ ฟังก์ชัน cadr เป็นรูปแบบย่อของ ฟังก์ชัน car(cdr …) และฟังก์ชัน caddr เป็นรูปแบบย่อของ car(cdr(cdr …))]

ตัวอย่างการคำนวณดิฟเฟอเรนเชียลของฟังก์ชัน 2·x+1 เมื่อเทียบกับ x มีขั้นตอนการคำนวณดังนี้

$$
\begin{aligned}
\frac{d(2\cdot x+1)}{dx} &= \frac{d(2\cdot x)}{dx} + \frac{d1}{dx} \\
&= 2\cdot\frac{dx}{dx} + x\cdot\frac{dx}{dx} + 0 \\
&= 2\cdot 1 + x\cdot 0 + 0 \\
&= 2
\end{aligned}
$$

ซึ่งผลลัพธ์ที่ได้จะตรงกับผลการทำงานของฟังก์ชัน diff ที่แสดงขั้นตอนการทำงานได้ดังนี้

```
(diff 'x '(+ (* 2 x) 1))
  = (list '+ (diff 'x '(* 2 x)) (diff 'x 1))
  = (list '+ (list '+ (list '* 2 (diff 'x 'x))
                      (list '* x (diff 'x 2)))
             (diff 'x 1))
  = (list '+ (list '+ (list '* 2 1)
                      (list '* x (diff 'x 2)))
             (diff 'x 1))

  = (list '+ (list '+ '(* 2 1)
             (list '* x (diff 'x 2))) (diff 'x 1))

  = (list '+ (list '+ '(* 2 1)(list '* x 0))
             (diff 'x 1))

  = (list '+ (list '+ '(* 2 1) '(* x 0))(diff 'x 1))
  = (list '+ (+ '(* 2 1) '(* x 0))(diff 'x 1))
  = (list '+ (+ '(* 2 1) '(* x 0)) 0)
```

ผลลัพธ์สุดท้ายคือลิสต์ของ (+ (+ (* 2 1) (* x 0)) 0) ซึ่งคำนวณได้เป็นค่า 2

9.11 สรุป

การทำโปรแกรมเชิงหน้าที่เป็นวิธีการทำโปรแกรมที่ใช้หลักการของฟังก์ชันทางคณิตศาสตร์ซึ่งเป็นการอธิบายการเชื่อมโยงค่าจากข้อมูลเข้าไปยังผลลัพธ์ วิธีการทำโปรแกรมในแนวทางนี้จะเป็นอิสระจากฮาร์ดแวร์ของระบบคอมพิวเตอร์ หลักการคิดเพื่อออกแบบโปรแกรมจะเน้นแพทเทิร์นการเชื่อมโยงจากข้อมูลเข้าไปสู่ผลลัพธ์เท่านั้นโดยไม่ต้องเกี่ยวข้องกับแนวคิดเกี่ยวกับโครงสร้างภายในหน่วยความจำและการเปลี่ยนสถานะของหน่วยความจำ วิธีการทำโปรแกรมเชิงหน้าที่จึงมีความเป็นนามธรรมสูงกว่าวิธีการทำโปรแกรมเชิงคำสั่ง

ภาษาคอมพิวเตอร์ที่ใช้ในการทำโปรแกรมเชิงหน้าที่ภาษาแรกคือภาษา LISP และในระยะต่อมามีการพัฒนาภาษาอื่นๆ ในกลุ่มนี้ขึ้นใช้งานเช่นภาษา common LISP, Scheme, ML, Miranda, Haskell ภาษาเพื่อการทำโปรแกรมเชิงหน้าที่ทั้งหลาย มีพื้นฐานมาจากภาษาคณิตศาสตร์ที่ถูกออกแบบมาเพื่ออธิบายการทำงานของฟังก์ชันนั่นคือภาษาของแคลคูลัสแลมป์ดา ฟังก์ชันในแคลคูลัสแลมป์ดาจะเขียนอยู่ในลักษณะของนิพจน์ที่ประกอบด้วยโอเปอเรเตอร์และโอเปอแรนด์ แคลคูลัสแลมป์ดาไม่มีการใช้คำสั่งกำหนดค่า มีเพียงนิพจน์ เช่น นิพจน์คณิตศาสตร์ นิพจน์ตรรกะ นิพจน์แบบมีเงื่อนไข การทำงานซ้ำในแคลคูลัสแลมป์ดา จะใช้วิธีนิยามฟังก์ชันเวียนบังเกิด หรือฟังก์ชันที่เรียกตัวเองซ้ำ

ฟังก์ชันในแคลคูลัสแลมป์ดาและในภาษาการทำโปรแกรมเชิงหน้าที่ จัดเป็นค่าที่มีความสำคัญอันดับหนึ่ง (first-class value) เพราะฟังก์ชันสามารถใช้เป็นค่าในนิพจน์ ใช้เป็นค่าในพารามิเตอร์ และใช้เป็นค่าที่ส่งกลับ ฟังก์ชันที่สามารถรับฟังก์ชันอื่นเป็นพารามิเตอร์และสามารถส่งค่ากลับเป็นฟังก์ชันจะเรียกว่าฟังก์ชันอันดับสูง (higher-order function) การใช้ฟังก์ชันอันดับสูงทำให้การทำโปรแกรมเชิงหน้าที่เป็นวิธีการทำโปรแกรมที่มีประสิทธิภาพสูงมาก

แบบฝึกหัดท้ายบทที่ 9

คำถามอัตนัย

1. จงคำนวณผลลัพธ์ของนิพจน์แลมป์ดาต่อไปนี้

   (a) ((λx.x * x) 5)

   (b) ((λy.((λx.x + y + z) 3)) 2)

   (c) (λv.(λw.w) ((λx.x)y(λz.z)))

2. ให้ใช้อินเตอร์พรีตเตอร์ของสคีมหาค่าผลลัพธ์ของนิพจน์ต่อไปนี้

   (a) (null? ())
   (b) (null? '(a b c d e))
   (c) (car '(a (b c) d e))
   (d) (der '(a (b c) d e))
   (e) (cadr '(a (b c) d e))

3. กำหนดฟังก์ชัน Sum ให้ดังต่อไปนี้

```
(define (sum alist)
(if (null? alist) 0
        (+ (car alist) (sum (cdr alist)))
 ))
```

   ถ้าเรียกใช้ฟังก์ชัน sum ด้วยนิพจน์ (sum '(1 2 3 4 5)) ให้แสดงขั้นตอนการทำงานของฟังก์ชันจนกระทั่งได้ผลลัพธ์สุดท้าย

4. ให้แสดงขั้นตอนการทำงานของฟังก์ชัน diff เมื่อกำหนดนิพจน์ให้เป็น x + 2x + 1

5. ให้เขียนฟังก์ชันในภาษาสคีมชื่อฟังก์ชัน elements ฟังก์ชันนี้ทำหน้าที่นับจำนวนสมาชิกทั้งหมดในลิสต์รวมทั้งในลิสต์ย่อย เช่น เมื่อเรียกใช้ฟังก์ชัน

   (elements '(1 (2 (3) 4) 5 6))

   ผลลัพธ์ที่ได้คือ 6

คำถามปรนัย: ให้เลือกคำตอบที่ถูกต้องที่สุด

1. โครงสร้างข้อมูลพื้นฐาน (primitive data structure) ของภาษา Scheme และ LISP คืออะไร ?

ก. atom และ list
ข. integer และ object
ค. function และ list
ง. element และ atom

2. ข้อใดอธิบายรูปแบบไวยากรณ์ของภาษา Scheme และ LISP ได้ถูกต้องที่สุด ?

ก. การนิยามฟังก์ชันต้องใช้คำสั่ง lambda
ข. โปรแกรมและข้อมูลใช้รูปแบบของ list
ค. การใช้ข้อมูลจะต้องมีการกำหนด type
ง. การเรียกใช้ฟังก์ชันจะต้องใช้รูปแบบ recursive

3. คำสั่ง (cdr '(A B C D E)) จะ return ค่าใด ?

ก. '(A)
ข. A
ค. (A B C D E)
ง. (B C D E)

4. ข้อใดต่อไปนี้ใช้ช่วยในการนิยามฟังก์ชันที่ไม่มีชื่อ ?

ก. Functional ambiguity
ข. car function
ค. Lambda notation
ง. Function signature

5. ฟังก์ชัน (Null? '()) จะ return ค่าใด ?

ก. ()
ข. #T
ค. #F
ง. Error

6. ข้อใดต่อไปนี้กล่าวถูกต้อง ?

ก. ภาษา Scheme เป็นภาษาที่สมบูรณ์กว่าภาษา LISP
ข. ภาษา Scheme เป็นภาษาที่พัฒนาขึ้นโดยทีมงานแห่งมหาวิทยาลัยแคลิฟอร์เนีย
ค. ภาษา Scheme เป็นเวอร์ชั่นหนึ่งของภาษา LISP
ง. ภาษา Scheme เรียกอีกชื่อหนึ่งว่า Common LISP

7. ใครเป็นผู้สร้างภาษา LISP ?

ก. John McCarthy
ข. Daniel Russell
ค. Linda Edwards
ง. Maurice Haskell

8. ภาษา Scheme ใช้คำสั่งใดในกรณีที่ต้องเลือกทำจากหลายทางเลือก ?

ก. IF THEN
ข. cond
ค. cons
ง. cdr

พิจารณาคำสั่งภาษา Scheme ต่อไปนี้ แล้วตอบคำถามข้อ 9-15

((Lambda (L) (car (cdr L))) ‘(A B C D))

9. ในคำสั่งข้างต้นส่วนใดเป็นฟังก์ชัน ?

ก. ((Lambda (L) (car (cdr L))) ‘(A B C D))

ข. (Lambda (L) (car (cdr L)))

ค. (car (cdr L))

ง. (Lambda (L))

10. ในคำสั่งข้างต้นส่วนใดเป็นการเรียกใช้ฟังก์ชัน ?

ก. ((Lambda (L) (car (cdr L))) ‘(A B C D))

ข. (Lambda (L) (car (cdr L)))

ค. (car (cdr L))

ง. (Lambda (L))

11. ในคำสั่งข้างต้นส่วนใดเป็น body ของฟังก์ชัน ?

ก. ((Lambda (L) (car (cdr L))) ‘(A B C D))

ข. (Lambda (L) (car (cdr L)))

ค. (car (cdr L))

ง. (Lambda (L))

12. ในคำสั่งข้างต้นส่วนใดเป็นส่วนหัว (header) ของฟังก์ชัน ?

ก. ((Lambda (L) (car (cdr L))) ‘(A B C D))

ข. (Lambda (L) (car (cdr L)))

ค. (car (cdr L))

ง. (Lambda (L))

13. การประมวลผลคำสั่งข้างต้นจะให้ผลลัพธ์ใด ?

ก. A ข. B

ค. (B C D) ง. (C D)

14. ฟังก์ชันนี้ชื่ออะไร ?

ก. Lambda ข. Lambda (L)

ค. car ง. ไม่มีชื่อ

15. ฟังก์ชันนี้รับ input parameter จำนวนเท่าใด ?

ก. 1 parameter
ข. 2 parameters
ค. 3 parameters
ง. ไม่รับพารามิเตอร์

16. ข้อใดไม่ใช่ Functional language ?

ก. Simula
ข. Scheme
ค. Haskell
ง. ML

17. ข้อใดเป็นคำกล่าวที่ผิดเกี่ยวกับ Fnctional language ?

ก. ใช้สนับสนุนงานวิจัยด้านปัญญาประดิษฐ์
ข. ถูกออกแบบให้ประมวลผลกับตัวเลขได้เร็ว
ค. สนับสนุนการเขียนฟังก์ชันแบบ recursive
ง. มีความสามารถในการทำ garbage collection

18. ข้อใดกล่าวได้ถูกต้องเกี่ยวกับ pure functional language ?

ก. มีการแยกความแตกต่างระหว่าง function และ procedure
ข. สามารถทำ recursion และ iteration ในลักษณะ loop
ค. สามารถใช้ assignment statement เพื่อเปลี่ยนค่าตัวแปร
ง. การทำงานวนหลายรอบจะใช้ recursive function

19. ลักษณะใดที่มักจะไม่พบใน pure functional language ?

ก. การใช้คำสั่ง recursion
ข. การจัดสรรหน่วยความจำอัตโนมัติ
ค. การใช้คำสั่ง assignment
ง. การใช้ฟังก์ชันเป็นพารามิเตอร์

20. ภาษา Scheme และ LISP กำหนดลำดับความสำคัญ(precedence) ของเครื่องหมาย +, -, *, / ไว้อย่างไร ?

ก. กำหนดให้ * และ / สำคัญกว่า +, -
ข. +, -, *, / มีความสำคัญเท่ากัน
ค. ไม่มีการกำหนด ให้ใช้ลำดับการทำงานจากซ้ายไปขวา
ง. ไม่มีการกำหนด ให้ใช้วงเล็บ ( ) ระบุลำดับก่อน-หลัง