204435 มโนทัศน์ภาษาโปรแกรม	**ภาคปลาย 2550**

การบรรยายที่ 2: ภาษาโปรแกรมเชิงฟังก์ชัน: Lisp และ Scheme

ผู้สอน: จิตร์ทัศน์ ฝักเจริญผล *29 ตุลาคม 2550*

คำประกาศ: *บันทึกคำบรรยายนี้ มิได้ผ่านการตรวจแก้เช่นเดียวกับบทความสำหรับการตีพิมพ์ทั่วไป การนำไปใช้อ้างอิง โปรดตรวจสอบข้อมูลจากแหล่งอื่นเพื่อความถูกต้องเพิ่มเติม*

Lisp เป็นภาษาโปรแกรมที่เก่าแก่ที่สุดภาษาหนึ่งที่ออกแบบโดย John McCarthy เพื่อใช้ประมวลผลลิสต์ (ชื่อเต็มของ Lisp คือ LISt Processor) แม้ว่าจะเป็นภาษาที่เก่าแก่ภาษาหนึ่ง (เป็นรองจาก Fortran) แต่ในปัจจุบันก็ยังมีคนใช้งานอยู่อีกมาก เป้าหมายของ McCarthy ในการออกแบบภาษานี้คือการสร้างภาษาที่เรียบง่าย (minimalism) แนวคิดหลาย ๆ อย่างที่มีใน Lisp ถูกนำมาประยุกต์ใช้ในภาษาสมัยใหม่มากมาย เช่น การจัดการหน่วยความจำที่มีการเก็บกวาดขยะ (garbage collection) หรือโคลเชอร์ (closure)

เนื่องจาก Lisp เป็นภาษาที่มีอายุยาวนาน ทำให้มีการแตกแขนงและปรับปรุงภาษาปรากฏขึ้นเป็นจำนวนมาก ความพยายามที่จะรวมแขนงต่าง ๆ ของ Lisp ทำให้ได้ภาษาโปรแกรม Common Lisp ขึ้น แขนงหนึ่งของ Lisp ที่ยังมีการใช้อยู่อย่างมากในปัจจุบัน แม้ว่าจะมี Common Lisp แล้วก็คือ Scheme

Scheme เป็นภาษาถิ่น (dialect) ของ Lisp ที่ถูกคิดค้นโดย Guy Lewis Steele Jr. และ Gerald Jay Sussman ปรัญชาหลักในการออกแบบคือความพยายามทำให้ภาษามีขนาดเล็ก เราเลือกใช้ Scheme ในการแนะนำการโปรแกรมเชิงฟังก์ชันเนื่องจากเป็นภาษาที่แสดงลักษณะเด่นของ Lisp ได้ครบ เช่น การจัดฟังก์ชันเป็นข้อมูลพื้นฐาน และการจัดการกับลิสต์และสัญลักษณ์

ในบทนี้เราจะศึกษาและทดลองโปรแกรมภาษา Scheme โดยจะเน้นส่วนย่อยของ Scheme ที่รองรับการโปรแกรมเชิงฟังก์ชัน จากนั้นเราจะพิจารณาปรัชญาและแนวคิดในการออกแบบภาษา Lisp ที่นอกจากจะทำให้ Lisp ยังเป็นภาษาที่น่าสนใจอยู่ในปัจจุบันแล้ว ยังมีอิทธิพลต่อการออกแบบภาษาอื่น ๆ รวมถึงภาษาโปรแกรมสมัยใหม่อีกด้วย

2.1 แนะนำภาษา Scheme

เมื่อเข้าไปในระบบของ Scheme โดยทั่วไปแล้ว เราจะอยู่ในวนรอบ read-eval-print นั่นคือ Scheme จะอ่านนิพจน์จากเรา นำไปคำนวณค่า (evaluate) แล้วแสดงผลออกมา

2.1.1 นิพจน์และฟังก์ชัน

2.1.1.1 นิพจน์

องค์ประกอบพื้นฐานของภาษา Scheme คือนิพจน์ (expression) ซึ่งอาจเป็น *ค่า*, *ชื่อ* หรือเป็น *การเรียกใช้ฟังก์ชัน* ตัวอย่างของนิพจน์ใน Scheme เช่น

```
5                    ; output = 5
(* 3 5)              ; output = 15
(+ (* 3 5) 7)        ; output = 22
```

เพื่อความสะดวกในการเขียน บางครั้งเราจะเขียนนิพจน์ในรูปแบบของการเรียกใช้ผ่านทางคอนโซล เช่น ตัวอย่างข้างต้นเขียนได้เป็น

```
> 5
5
> (* 3 5)
15
> (+ (* 3 5) 7)
22
```

เครื่องหมาย “>” แทนเครื่องหมายเตรียมพร้อมของระบบโต้ตอบของ Scheme

ตัวอย่างที่สองและสามเป็นการเรียกใช้ฟังก์ชัน `*` และ `+` ใน Scheme รูปแบบการเรียกฟังก์ชันจะเขียนเป็นรายการหรือลิสต์ (list) โดยทั่วไปแล้วในการคำนวณค่าของนิพจน์ที่เป็นการเรียกฟังก์ชัน เราพิจารณาให้สมาชิกแรกของลิสต์เป็นชื่อของฟังก์ชัน จากนั้นสมาชิกอื่น ๆ ถัดไปจะเป็นอาร์กิวเมนต์ เราจะหาค่าของอาร์กิวเมนต์ทั้งหมดก่อนจากนั้นจึงเรียกใช้ฟังก์ชันโดยส่งค่าเหล่านั้นเป็นอาร์กิวเมนต์

ในการคำนวณค่านิพจน์ที่ซับซ้อน เช่น

```
(+ (- (* 3 5) (+ (- 4 2) (+ 1 2))))
```

ลำดับการคำนวณค่าของฟังก์ชันดังกล่าวให้อิสระกับเราในการเริ่มคำนวณได้ในหลายนิพจน์ย่อย เช่น `(* 3 5)` `(- 4 2)` หรือ `(+ 1 2)` อย่างไรก็ตาม ไม่ว่าเราจะเริ่มคำนวณที่ใดก่อน เราจะพบว่าผลลัพธ์ที่ได้นั้นจะเหมือนเดิมเสมอโดยไม่ขึ้นกับลำดับการประมวลผล การไม่ขึ้นต่อลำดับการคำนวณนี้พบในเฉพาะภาษาที่ไม่มีผลกระทบข้างเคียง (side-effect) เช่นภาษาเชิงฟังก์ชัน

2.1.1.2 การนิยาม

เพื่อความสะดวก เราสามารถกำหนดชื่อเข้ากับค่าได้ โดยเขียน

```
(define x 10)
x                    ; output = 10
(* 3 x)              ; output = 30
```

การใช้งานดังกล่าวอาจคล้ายกับการใช้ตัวแปรในภาษาเชิงคำสั่ง อย่างไรก็ตาม ในภาษาเชิงฟังก์ชันจะมองตัวแปรในเชิงคณิตศาสตร์ นั่นคือตัวแปรคือสัญลักษณ์ที่ใช้แทนค่า ดังนั้นตัวแปรหนึ่ง ๆ จะมีค่าได้ค่าเดียวเท่านั้น ผิดกับในภาษาเชิงคำสั่งที่ตัวแปรจะหมายถึงเนื้อที่เก็บข้อมูลในหน่วยความจำซึ่งเปลี่ยนแปลงค่าได้

สังเกตว่าในด้านบน การสั่ง `(define x 10)` นั้นมีรูปแบบเหมือนกับการเรียกใช้ฟังก์ชัน แต่ว่าไม่ใช่ เพราะว่าถ้าเป็นการเรียกใช้ฟังก์ชัน ต่าของอาร์กิวเมนต์จะต้องถูกคำนวณออกมาทั้งหมดเสียก่อน ซึ่งจะทำไม่ได้เนื่องจากในเวลาดังกล่าว ยังไม่มีการกำหนดนิยามให้กับ `x`

ลักษณะของคำสั่งทีดูคล้ายฟังก์ชัน แต่มีวิธีการคำนวณค่าแตกต่างไปเช่น `define` จะเรียกว่า *รูปแบบพิเศษ* (special form)

2.1.1.3 ฟังก์ชัน

ใน Scheme ฟังก์ชันจัดเป็นข้อมูลชนิดหนึ่ง เราสามารถเขียนฟังก์ชันได้โดยใช้รูปแบบพิเศษ `lambda` โดยมีตัวอย่างดังนี้

```
(lambda (x) x)            ; identity function
(lambda (x) (* x x))      ; this is a square function
(lambda (x y) (< x y))    ; this function compares two arguments
```

หลังคำว่า lambda เราจะระบุอาร์กิวเมนต์ของฟังก์ชัน และระบุนิพจน์ของฟังก์ชันนั้นถัดไป คำว่า lambda เป็นชื่อที่มาจาก Lambda Calculus ซึ่งเป็นระบบคำนวณบนฟังก์ชันของ Church ซึ่งเราจะได้เรียนต่อไป

เราใช้การเขียนฟังก์ชันดังกล่าวได้ในนิพจน์โดยตรง เช่น

```
((lambda (x) (* x x)) 20)  ; output = 400
((lambda (x y) (< x y)) 10 30)     ; output = #t (true)
```

ใน Scheme ค่าจริงคือ `#t` ค่าเท็จคือ `#f`

นอกจากนี้ เราสามารถกำหนดชื่อให้กับฟังก์ชัน (หรือจะเรียกว่านิยามฟังก์ชัน) ได้ โดยใช้รูปแบบพิเศษ `define` ตัวอย่างเช่น เราสามารถนิยามฟังก์ชัน `square` ได้ดังนี้

```
(define square (lambda (x) (* x x)))
```

เพื่อความกระทัดรัด Scheme จึงมีอีกรูปแบบหนึ่งในการนิยามฟังก์ชัน ตัวอย่างข้างต้นเขียนใหม่ได้เป็น

```
(define (square x) (* x x))
```

2.1.1.4 ฟังก์ชันที่คืนค่าเป็นฟังก์ชัน

เนื่องจากฟังก์ชันเป็นข้อมูลในภาษาโปรแกรมเชิงฟังก์ชัน เราสามารถนิยามฟังก์ชันที่คืนค่าเป็นฟังก์ชันได้ด้วย พิจารณาฟังก์ชัน `addwith` ต่อไปนี้

```
(define (addwith x)
  (lambda (y) (+ x y)))
```

ฟังก์ชันดังกล่าวรับค่า `x` แล้วคืนฟังก์ชันที่รับอาร์กิวเมนต์หนึ่งตัว แล้วนำไปบวกกับ `x` ตัวอย่างการทำงานเช่น

```
> ((addwith 10) 20)
30
> (define add15 (addwith 15))
> (add15 30)
45
```

เราจะได้เห็นลักษณะการใช้งานฟังก์ชันแบบนี้อีกเรื่อย ๆ

ในภาษาโปรแกรมทั่วไป การคืนค่าเป็นฟังก์ชันดังกล่าวทำได้ยุ่งยากมาก และในบางภาษาเช่นภาษา C แทบ จะทำไม่ได้เลย

2.1.2 เงื่อนไข

Scheme มีรูปแบบพิเศษในการอธิบายเงื่อนไขด้วยคำสั่ง `if` ซึ่งมีตัวอย่างการใช้งานเป็นดังนี้

```
(define (max x y)
  (if (> x y)
      x
      y))
```

รูปแบบของคำสั่ง `if` คือ: `(if <condition> <if-true> <if-false>)`

ด้วยรูปแบบพิเศษดังกล่าว เราสามารถนิยามฟังก์ชัน `factorial` ได้เป็น

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1)))))
```

สังเกตว่าเรานิยามฟังก์ชันดังกล่าวแบบเรียกตัวเอง ทั้งนี้เนื่องจากการทำซ้ำหรือการวนรอบไม่มีอยู่ในแนวคิดเชิงฟังก์ชัน

ในส่วนของเงื่อนไขนั้น นอกจากการเปรียบเทียบทั่วไปแล้ว Scheme ยังมีฟังก์ชันอื่น ๆ ที่คืนค่าจริง/เท็จด้วย เรามักเรียกฟังก์ชันกลุ่มดังกล่าวว่า *เพรดิเคต* (predicate) ตัวอย่างเช่น ฟังก์ชัน `eq?` และ `equal?` สำหรับตรวจสอบว่าข้อมูลมีค่าเท่ากันหรือไม่

ในกรณีที่มีหลายเงื่อนไข เรามีรูปแบบพิเศษ `cond` ที่มีรูปแบบทั่วไปเป็น

```
(cond (<condition-1> <expression-1>)
      (<condition-2> <expression-2>)
         ...
      (<condition-k> <expression-k>))
```

มีนิยามในการคำนวณหาค่าของนิพจน์ดังนี้: ถ้าเงื่อนไขแรกเป็นจริงนิพจน์ดังกล่าวจะมีค่าเท่ากับนิพจน์แรก ถ้าเงื่อนไขที่ 1 ถึง $i-1$ ไม่เป็นจริง และเงื่อนไขที่ i เป็นจริง นิพจน์ `cond` จะมีค่าเท่ากับนิพจน์ที่ i

2.1.3 ลิสต์

Lisp (และ Scheme ที่เป็นลูกหลานของ Lisp) เป็นภาษาที่ออกแบบมาโดยมีเป้าหมายหนึ่งคือการประมวลผลลิสต์ (หรือรายการ) ดังนั้นโครงสร้างข้อมูลแบบลิสต์จึงเป็นโครงสร้างข้อมูลพื้นฐานของภาษา

ลิสต์ใน Lisp จะเขียนอยู่ในวงเล็บ ยกตัวอย่างเช่น `(1 2 3 4 5)` หรือ `(1 2 (3 4 5) 6)` ที่เป็นลิสต์ที่มีสมาชิกบางตัวเป็นลิสต์

อย่างไรก็ตาม การเขียนลิสต์ในลักษณะดังกล่าวมีรูปแบบเหมือนกับการเรียกใช้ฟังก์ชัน ดังนั้น ถ้าเราป้อนนิพจน์ที่เป็นค่าคงที่แบบลิสต์ในระบบโต้ตอบของ Scheme เราจะได้การแสดงข้อผิดพลาดออกมา เช่น

```
> (1 2 3)
. procedure application: expected procedure, given: 1; arguments were: 2 3
```

ดังนั้น เราจำเป็นจะต้อง “ห้าม” ไม่ให้มีการคำนวณค่าของนิพจน์ดังกล่าว โดยจะระบุด้วยรูปแบบพิเศษ quote ดังนี้

```
> (quote (1 2 3))
(1 2 3)
```

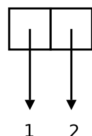
รูปแบบดังกล่าวใช้บ่อยมาก จึงมีวิธีการเขียนแบบย่อได้โดยใช้เครื่องหมายคำพูดเดี่ยว (`'`) ตัวอย่างข้างต้นเขียนได้เป็น `'(1 2 3)`

ชนิดข้อมูลใน Lisp นอกจากจำนวนเต็มแล้วยังมีข้อมูลประเภทสัญลักษณ์ (symbol) ซึ่งเปรียบเสมอสตริงที่ไม่สามารถแบ่งย่อยได้ ชนิดข้อมูลแบบนี้นี่เองที่ทำให้ Lisp เป็นภาษาที่นิยมในงานประยุกต์ด้านปัญญาประดิษฐ์และการคำนวณเชิงสัญลักษณ์ ตัวอย่างของลิสต์ของสัญลักษณ์ เช่น `(hello today is monday)` เป็นต้น

สมาชิกในลิสต์จะเป็นข้อมูลชนิดใดก็ได้ โดยอาจเป็นค่าคงที่พื้นฐาน เช่นจำนวน ค่าคงที่ตรรกศาสตร์ สัญลักษณ์ (symbol) สตริง หรือลิสต์ว่าง (เขียนแทนด้วย `()`) ค่าคงที่พื้นฐานเหล่านี้เรียกว่า *อะตอม* (atom) หรืออาจเป็นข้อมูลประกอบอื่น ๆ เช่นคู่ลำดับหรือลิสต์เองก็ได้

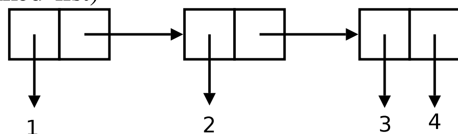
2.1.3.1 โครงสร้างภายในของลิสต์

ลิสต์จะถูกสร้างขึ้นมาจากอะตอม โดยใช้*คู่ลำดับ*เป็นโครงสร้างพื้นฐาน จากอะตอมเราสามารถสร้างคู่ลำดับขึ้นมาได้โดยใช้ฟังก์ชัน `cons` ซึ่งรับข้อมูลสองตัวที่มีชนิดเป็นอะไรก็ได้ แล้วสร้างคู่ลำดับของข้อมูลสองตัวนั้น ตัวอย่างเช่น `(cons 1 2)` เราจะได้โครงสร้างข้อมูลแบบคู่ลำดับแสดงดังรูปด้านล่าง



โครงสร้างข้อมูลดังกล่าวเรียกว่า *คอนส์เซลล์* (cons cell) ช่องแรกของคอนส์เซลล์จะเรียกว่า car ส่วนช่องที่สองจะเรียกว่า cdr (อ่านว่าคูเดอร์) ชื่อทั้งสองนี้มีที่มาจากคำสั่งภาษาเครื่องของ IBM 704 ในการแสดงผลที่ส่วนโต้ตอบของ Scheme คอนส์เซลล์ดังกล่าวจะแสดงเป็น `(1 . 2)`

พลังของการสร้างคู่ลำดับดังกล่าวอยู่ที่ความสามารถในการสร้างคู่ลำดับของข้อมูลใดก็ได้ ยกตัวอย่างเช่น ถ้าเราสั่ง `(cons 1 (cons 2 (cons 3 4)))` เราจะได้โครงสร้างดังรูปด้านล่าง ซึ่งมีลักษณะเหมือนกับลิงก์ลิสต์ (linked list)



ลิสต์ก็มีโครงสร้างภายในไม่ต่างกัน เพียงแต่ว่าสมาชิกสุดท้ายจะเป็นลิสต์ว่าง `()` (ซึ่งเป็นอะตอมแบบหนึ่ง) เราสามารถสร้างลิสต์ `(1 2 3)` ได้โดยใช้ฟังก์ชัน `cons` ดังนี้

```
> (cons 1 (cons 2 (cons 3 () )))
(1 2 3)
```

สังเกตว่าเราต้อง quote ลิสต์ว่างด้วย

ตัวอย่างการใช้งานคำสั่ง `cons` แสดงด้านล่าง ให้สังเกตการใช้เครื่องหมายคำพูดด้วย

```
> (cons 1 '(2 3 4))
(1 2 3 4)
> (cons '(1 2) '(3 4 5))
((1 2) 3 4 5)
> (cons 'hello '(world today))
(hello world today)
```

นอกจากคำสั่ง `cons` แล้ว เรายังสามารถสร้างลิสต์ได้โดยใช้ฟังก์ชัน `list` ด้านล่างแสดงตัวอย่าง

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 1 2 (list 3 4 5) 6)                ; nested list
(1 2 (3 4 5) 6)
```

2.1.3.2 การเข้าถึงข้อมูลในลิสต์

เราสามารถเข้าถึงข้อมูลในลิสต์ได้ด้วยฟังก์ชัน `car` และ `cdr` โดยฟังก์ชันทั้งสองรับคอนส์เซลล์จากนั้นคืนส่วนของ car และส่วนของ cdr ของคอนส์เซลล์ตามลำดับ ตัวอย่างการเรียกใช้งานเช่น

```
> (car '(1 2 3))
1
> (car '((1 2) 3))
(1 2)
> (cdr '(1 2 3 4))
(2 3 4)
> (car '((the student) runs))
(the student)
> (car (cdr '((the student) runs)))
runs
> (cdr (car '((the student) runs)))
(student)
```

ด้วยฟังก์ชันทั้งสองเราสามารถเขียนฟังก์ชันที่ทำงานกับลิสต์ได้ ฟังก์ชันด้านล่างนับจำนวนสมาชิกในลิสต์

```
(define (len lst)
  (if (null? lst)
      0
      (+ 1 (len (cdr lst)))))
```

ในตัวอย่างข้างต้นมีการใช้เพรดิเคต `null?` ที่จะคืนค่าจริงถ้าอาร์กิวเมนต์เป็นลิสต์ว่าง นอกจากนี้ยังมีเพรดิเคต `list?` ที่ใช้ตรวจสอบว่าอาร์กิวเมนต์เป็นลิสต์หรือไม่

△**แบบฝึกหัดย่อย.** ฟังก์ชัน `len` ข้างต้นจะนับจำนวนสมาชิกของลิสต์เท่านั้น เช่น การเรียก `(len '((1 2) 3 (4 (5 6))))` จะคืนค่า 3 จงเขียนฟังก์ชัน `len-all` ที่นับรวมสมาชิกของลิสต์ที่อยู่ในลิสต์ดังกล่าวด้วย (เช่น ถ้ารับตัวอย่างลิสต์ข้างต้นฟังก์ชันจะคืนค่า 6)

2.1.4 ตัวแปรท้องถิ่น (Local variables)

สำหรับการนิยามฟังก์ชันที่ซับซ้อน เราสามารถตั้งชื่อให้กับผลการคำนวณของส่วนย่อย ๆ ของฟังก์ชันได้ ด้วยคำสั่ง `let` ที่สร้างตัวแปรภายใน ดังตัวอย่างดังต่อไปนี้

```
> (let ((x 1) (y 1)) (+ x y))
2
```

หรือฟังก์ชันด้านล่างที่รับลิสต์ `lst` แล้วบวกทุก ๆ ค่าในลิสต์ด้วย `x`

```
(define (add-to-all lst x)
  (if (null? lst)
      ()
      (let ((head (car lst))        ; local variables: head and tail
            (tail (cdr lst)))
        (cons (+ x head)
              (add-to-all tail)))))
```

2.1.5 การเรียกตัวเองแบบหาง (Tail recursion)

ฟังก์ชันที่มีการเรียกตัวเองแบบหางคือฟังก์ชันเรียกตัวเองที่ผลลัพธ์ที่ได้จากการเรียกตัวเองจะเป็นคำตอบของฟังก์ชันนั้นด้วย ยกตัวอย่างเช่นฟังก์ชัน `factorial-t` ที่เขียนด้านล่างนี้ สังเกตว่าในการเรียกใช้ฟังก์ชันดังกล่าวจะต้องเพิ่มอาร์กิวเมนต์ `a` ที่กำหนดค่าเริ่มต้นให้เป็น 1 เข้าไปด้วย เช่น `(factorial-t 1 10)`

```
(define (factorial-t a x)
  (if (eq? x 0)
      a
      (factorial-t (* a x) (- x 1))))
```

ในการคำนวณค่าของฟังก์ชันลักษณะนี้ เรามีวิธีการคำนวณค่าแบบพิเศษที่ทำให้ประสิทธิภาพในการทำงานไม่แตกต่างจากการเขียนในรูปแบบวนรอบ ข้อกำหนดของภาษา Scheme ระบุว่าทุก ๆ implementation ของภาษา Scheme จะต้องจัดการในส่วนนี้ได้ด้วย อย่างไรก็ตามไม่ใช่ทุกภาษาโปรแกรมจะทำการจัดการดังกล่าวให้ (เช่น Common Lisp ก็ไม่ได้ระบุไว้ในข้อกำหนด)

2.2 การโปรแกรมเชิงฟังก์ชัน (Functional programming)

2.2.1 ฟังก์ชันชั้นสูง (Higher-order function)

เราเรียกฟังก์ชันที่รับอาร์กิวเมนต์เป็นฟังก์ชัน หรือฟังก์ชันที่มีการคืนค่าเป็นฟังก์ชันว่า *ฟังก์ชันชั้นสูง* (higher-order function)

ใน Scheme การใช้งานดังกล่าวทำได้สะดวก ยกตัวอย่างเช่นฟังก์ชัน `maplist` ด้านล่างนี้ที่รับลิสต์และฟังก์ชัน `f` จากนั้นคือลิสต์ที่ได้จากการเรียกฟังก์ชันดังกล่าวบนทุก ๆ สมาชิกในลิสต์

```
(define (maplist lst f)
  (if (null? lst)
      ()
      (cons (f (car lst))
            (maplist (cdr lst) f))))
```

ตัวอย่างการใช้งานเช่น

```
> (maplist '(1 2 3) (lambda (x) (* x x)))    ; squaring all elements
(1 4 9)
```

หรือในตัวอย่างด้านล่างที่สร้างฟังก์ชัน `compose` ที่รับฟังก์ชัน f และ g จากนั้นคืนฟังก์ชัน $f \circ g$ (ทบทวน: $(f \circ g)(x) = f(g(x))$)

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

ตัวอย่างการใช้งานเช่น

```
> ((compose (lambda (x) (+ x 1))
            (lambda (x) (* x 2))) 10)
21
> (define f (compose (lambda (x) (* x x))
                     (lambda (x) (+ x 1))))
> (f 3)
16
```

ความสามารถในการใช้และจัดการฟังก์ชันในลักษณะเดียวกับข้อมูลอื่น ๆ ช่วยเพิ่มระดับในการสร้างภาพนามธรรม (abstraction) ของโปรแกรมขึ้น นอกจากนี้ลักษณะการประมวลผลหลาย ๆ อย่างสามารถอธิบายในอยู่ในรูปของกระบวนการพื้นฐานได้ ซึ่งถูกทำเป็นฟังก์ชันมาตรฐานไว้แล้ว เช่น `map` เป็นต้น เราจะได้พิจารณาตัวอย่างเพิ่มเติมเมื่อเราศึกษาภาษา ML

2.3 ลักษณะเด่นอื่น ๆ ของ Lisp

2.3.1 โปรแกรม/ข้อมูล

ใน Lisp ข้อมูลและโปรแกรมถูกจัดเก็บอยู่ในรูปแบบเดียวกัน คือเป็นลิสต์ ดังนั้นเราจึงสามารถจัดการกับโปรแกรมได้เหมือนกับข้อมูลได้ Lisp และ Scheme มีฟังก์ชัน `eval` ที่คำนวณหาค่าของลิสต์โดยพิจารณาว่าลิสต์ดังกล่าวเป็นโปรแกรม ยกตัวอย่างเช่น

```
> (eval '(+ (* 2 3) 5))
11
> (eval (cons '+ '(1 2 3 4)))
10
```

สังเกตว่าฟังก์ชัน + สามารถรับอาร์กิวเมนต์ได้หลายตัว แล้วนำมาบวกกันทั้งหมด

รูปแบบดังกล่าวทำให้เราเขียนฟังก์ชัน `sum` ได้อีกแบบหนึ่ง โดยการสร้างคำสั่งบวกขึ้นมา แล้วเรียก `eval`

```
(define (sum lst)
  (eval (cons '+ lst)))
```

ในขณะที่มีการออกแบบภาษา Lisp นั้น แนวคิดของ Universal Turing machine ที่กล่าวว่ามี Turing machine ที่สามารถจำลองการทำงานของ Turing machine อื่น ๆ ได้ทั้งหมดนั้นจัดว่าเป็นแนวคิดที่ทรงพลังมาก ทำให้ในการออกแบบ Lisp นั้น McCarthy ได้เขียนฟังก์ชันในภาษา Lisp ที่ใช้คำนวณค่าของนิพจน์ของ Lisp ทั้งหมด ซึ่งนั่นก็คือฟังก์ชัน `eval` นั่นเอง

2.3.2 การเก็บกวาดขยะ (Garbage collection)

สังเกตว่าในการเขียนภาษา Lisp เราสามารถสร้างคอนส์เซลล์ได้เรื่อย ๆ โดยไม่มีความจำเป็นต้องจัดการเก็บกวาดคอนส์เซลล์ที่ไม่ได้ใช้ ลักษณะดังกล่าวอาจไม่ใช่เรื่องใหม่สำหรับผู้ที่เขียนโปรแกรมด้วยภาษาสมัยใหม่ เช่น Java หรือ C# แต่ในสมัยที่มีการออกแบบภาษา Lisp การยกภาระดังกล่าวให้กับระบบภาษา นับเป็นเรื่องหนึ่งที่ปฏิวัติวงการภาษาโปรแกรมเลยทีเดียว เนื่องจากข้อโต้เถียงต่าง ๆ เช่น ความซับซ้อนของระบบคอมไพเลอร์ หรือความไร้ประสิทธิภาพของการเก็บกวาด

เรานิยาม *ขยะ* ดังนี้ "ในช่วงเวลาใด ๆ ของการทำงานของโปรแกรม หน่วยความจำที่ตำแหน่ง m จะเป็นขยะถ้าโปรแกรมดังกล่าว ไม่มีทางกลับมาใช้งานหน่วยความจำที่ m ได้อีก"

สังเกตว่า ถ้าเราสามารถหาได้ว่าในขณะหนึ่ง ๆ มีหน่วยความจำใดบ้างที่โปรแกรมสามารถเข้าเรียกใช้งานได้ เราก็สามารถระบุหน่วยความจำที่ไม่ใช่ขยะได้ทั้งหมด อัลกอริทึมสำหรับจัดการเก็บกวาดขยะที่ง่ายที่สุดอันหนึ่งที่ชื่อว่า Mark-and-sweep เริ่มทำงานโดยการกำหนดเครื่องหมาย 0 ให้กับทุก ๆ ชิ้นของหน่วยความจำที่ถูกจองไว้ จากนั้นไล่พิจารณาหน่วยความจำที่เข้าถึงได้ทั้งหมดแล้วเปลี่ยนเครื่องหมายให้เป็น 1 สุดท้ายชิ้นของหน่วยความจำที่โปรแกรมไม่สามารถเข้าถึงได้จะเหลืออยู่โดยมีเครื่องหมายเป็น 0

$\triangle$**แบบฝึกหัดย่อย.** พิจารณาการทำงานใน Scheme ต่อไปนี้

```
> (define (gen-list n)
    (if (= n 0)
        ()
        (cons n (gen-list (- n 1)))))
> (define x (cdr (cdr (cdr (gen-list 5)))))
> x
(2 1)
```

ให้ระบุว่าหลังการทำงาน มีคอนส์เซลล์ในระบบเป็นอย่างไร และคอนส์เซลล์ใดบ้างที่เป็นขยะ

2.4 ผลกระทบข้างเคียง (Side effect)

ในการเขียนโปรแกรมทั่วไป เช่นในภาษา C ตัวแปรจะใช้แทนเนื้อที่ในหน่วยความจำสำหรับเก็บข้อมูลซึ่งเปลี่ยนแปลงได้ พิจารณาส่วนของโปรแกรมต่อไปนี้

```
int d;
int div_by_d(int x) { return x/d;}
```

เมื่อเราเห็นโปรแกรมนี้ เราไม่มีทางทราบได้เลยว่าโปรแกรมดังกล่าวจะทำงานแล้วจะเกิดความผิดพลาดจากการหารด้วยศูนย์หรือไม่ ทั้งนี้เนื่องจากเราไม่ทราบว่าค่าของตัวแปร d เมื่อเรียกใช้จะเป็นเท่าใด ถ้ามีส่วนของโปรแกรมอื่น ๆ ที่เปลี่ยนค่า d จนมีค่าเป็น 0 เราก็อาจเกิดปัญหาได้ ลักษณะดังกล่าว เรียกว่าผลกระทบข้างเคียง ซึ่งเป็นสิ่งปกติและต้องระวัง ในการเขียนโปรแกรมในภาษาที่มีสถานะ เช่นภาษาเชิงกระบวนการ หรือภาษาเชิงวัตถุ

แม้ว่าภาษาโปรแกรมเชิงฟังก์ชัน เช่น Scheme จะพยายามหลีกเลี่ยงผลกระทบข้างเคียงเพียงใด การทำงานหลาย ๆ อย่างก็มีเป้าหมายเพื่อสร้างผลกระทบข้างเคียง เช่น การรับและอ่านข้อมูล[1] หรือบางครั้งการเขียนโปรแกรมแบบที่มีผลกระทบข้างเคียงก็ทำให้ได้โปรแกรมที่มีประสิทธิภาพมากกว่า

2.4.1 การรับข้อมูลและแสดงผลข้อมูล

ใน Scheme ฟังก์ชัน read จะอ่านข้อมูลจากผู้ใช้แล้วคืนค่าออกมา ตัวอย่างการทำงานเช่น

```
> (+ (read) 30)
10                              ; this is what the user typed
40
> (cons (read) '(world))
hello                           ; this is what the user typed
(hello world)
```

ส่วนการแสดงผล จะมีฟังก์ชัน display ที่แสดงผลข้อมูล เช่น

```
> (display '(hello world))
(hello world)
> (display (gen-list 5))
(5 4 3 2 1)
> (display (+ 10 5))
15
```

นอกจากนี้มีฟังก์ชัน newline สำหรับขึ้นบรรทัดใหม่

ในหลาย ๆ ครั้ง เราต้องการประมวลผลด้วย และแสดงผลด้วย ใช้รูปแบบพิเศษ begin ในการระบุให้มีการคำนวณไล่ไปตามลำดับ และคืนผลเป็นการคำนวณสุดท้าย ยกตัวอย่างเช่น ถ้าเราสั่ง (begin 1 (read) (display 10) 5) จะมีช่องให้เราป้อนค่า มีการแสดงค่า 10 และสุดท้ายคืนค่าเป็น 5 เป็นต้น

ตัวอย่างของโปรแกรมพิมพ์สูตรคูณเป็นดังนี้

```
(define (table m n)
  (if (= n 0)
      ()
      (begin (table m (- n 1))
             (display (list n (* n m)))
             (newline))))
```

[1] ในบางภาษา เช่น Haskell มีวิธีการจัดการกับการรับและอ่านข้อมูลโดยไม่จำเป็นต้องขึ้นกับผลกระทบข้างเคียง

2.4.2 ตัวแปรแบบภาษาทั่วไป

2.5 สรุปแนวคิดและนวัตกรรมใหม่ในการออกแบบภาษา Lisp

2.6 ตัวอย่างโปรแกรม

ในส่วนนี้เราจะเขียนฟังก์ชันสำหรับจัดการกับต้นไม้ค้นหาแบบทวิภาค (binary search tree)

2.6.1 โครงสร้างแบบต้นไม้

```
(define test-tree
  '(5 (3 () (4 () ())) (6 () ())))
```

2.6.2 การแสดงผลต้นไม้

```
(define (left-child tree)
  (car (cdr tree)))

(define (right-child tree)
  (car (cdr (cdr tree))))

(define (indent level)
  (if (= level 0)
      ()
      (begin (display "   ")
             (indent (- level 1)))))

(define (display-tree level tree)
  (if (null? tree)
      ()
      (begin (display-tree (+ level 1) (left-child tree))
             (indent level)
             (display (car tree))
             (newline)
             (display-tree (+ level 1) (right-child tree)))))
```

2.6.3 การแทรกข้อมูลเข้าไปในต้นไม้

```
(define (new-node x)
  (list x () ()))

(define (insert tree x)
  (if (null? tree)
      (new-node x)
      (let ((root (car tree))
            (left (left-child tree))
            (right (right-child tree)))
        (cond ((= root x) tree)
              ((> root x) (list root (insert left x) right))
              (else (list root left (insert right x)))))))
```

2.7 แบบฝึกหัด

1. ให้วาดรูปแสดงคอนส์เซลล์ต่าง ๆ ที่ได้จากการคำนวณค่าของนิพจน์ `(cons 'A (cons (cons 'B 'C) 'D))`

2. เขียนฟังก์ชัน `middle` ที่รับลิสต์ที่มีความยาวเป็นจำนวนคี่ จากนั้นคืนค่าสมาชิกตัวที่อยู่ตรงกลาง

```
> (middle '(1 2 3 4 5))
3
```

3. เขียนฟังก์ชัน `last` ที่รับลิสต์ จากนั้นคืนข้อมูลตัวสุดท้าย ไม่ต้องสนใจกรณีที่ได้รับลิสต์ว่าง

```
> (last '(1 2 3 4 5))
5
```

4. เขียนฟังก์ชัน `myappend` ที่รับลิสต์สองลิสต์ จากนั้นคืนค่าเป็นลิสต์สองอันต่อกัน

```
> (myappend '(1 2) '(3 4 5))
(1 2 3 4 5)
```

5. เขียนฟังก์ชัน `filter` ที่รับลิสต์ และเพรดิเคต `fn` จากไล่พิจารณาข้อมูลทุกตัวในลิสต์ แล้วคืนลิสต์ย่อยที่ประกอบด้วยข้อมูลที่เพรดิเคต `fn` คืนค่าจริง

```
> (filter '(1 2 3 4 5) (lambda (x) (>= x 3)))
(3 4 5)
```

6. เขียนฟังก์ชัน `reverse` ที่รับลิสต์ จากนั้นคืนค่าลิสต์ที่เรียงลำดับย้อนกลับ

```
> (reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

7. เขียนฟังก์ชัน `split` ที่รับลิสต์หนึ่งลิสต์ จากนั้นแบ่งลิสต์ออกเป็นสองส่วน (อย่างไรก็ได้) ให้จำนวนสมาชิกในลิสต์ผลลัพธ์ทั้งสองต่างกันไม่เกิน 1 ตัว (วิธีการหนึ่งที่ทำได้ก็คือพิจารณาข้อมูลทีละสองจำนวน แล้วแยกใส่ลิสต์ผลลัพธ์ลิสต์ละตัว)

```
> (split '(1 2 3 4 5 6 7))
((1 3 5 7) (2 4 6))
```

8. เขียนฟังก์ชัน `merge` ที่รับลิสต์สองลิสต์ โดยที่ข้อมูลในแต่ละลิสต์เรียงจากน้อยไปมากแล้ว แล้วให้ `merge` คืนค่าลิสต์ที่ได้จากการรวมลิสต์ทั้งสองเข้าด้วยกัน โดยให้สมาชิกในลิสต์ผลลัพธ์เรียงจากน้อยไปหามากด้วยเช่นกัน

```
> (merge '(1 2 4 6) '(3 5 7 8 9))
(1 2 3 4 5 6 7 8 9)
```

9. * เขียนฟังก์ชัน `merge-sort` ที่รับลิสต์ แล้วจัดเรียงข้อมูลจากน้อยไปหามาก ด้วยวิธีแบบ merge sort (ใช้ฟังก์ชัน split และ merge ได้)

10. นิพจน์แบบเงื่อนไข. เพื่อนของคุณเขียนฟังก์ชัน `myif` ขึ้นมา ดังด้านล่าง

```
(define (myif c exp1 exp2)
  (if c
      exp1
      exp2))
```

เขาได้ทดลองกับนิพจน์ `(myif (> 2 3) 10 20)` และพบว่าทำงานได้ถูกต้อง ให้อธิบายและยกตัวอย่างปัญหาที่อาจเกิดขึ้น ถ้านำฟังก์ชันดังกล่าวไปใช้งานจริง ๆ

11. การเก็บกวาดขยะโดยการนับการอ้างถึง (reference count) เป็นวิธีการเก็บกวาดขยะอีกแบบหนึ่ง ที่มีหลักการทำงานคร่าว ๆ ดังนี้

    ทุก ๆ หน่วยของหน่วยความจำที่มีการจอง (เช่นคอนส์เซลล์) จะมีเนื้อที่กันไว้เป็นตัวนับจำนวนการถูกอ้างถึง ในตอนแรกเมื่อมีการจองหน่วยความจำนั้นขึ้น ตัวนับดังกล่าวจะถูกกำหนดให้เป็น 0 จากนั้นถ้ามีตัวแปรอื่น ๆ อ้างมายังหน่วยความจำนี้ ตัวนับดังกล่าวก็จะถูกเพิ่มค่าขึ้น ถ้าตัวแปรที่อ้างมายังหน่วยความจำนี้ถูกทำลาย ตัวนับดังกล่าวก็จะถูกลดลง เมื่อตัวนับเป็น 0 แสดงว่าไม่มีใครอ้างถึงหน่วยความจำนี้อีกแล้ว ทำให้สามารถทำลายหน่วยความจำนี้ทิ้งไปได้เช่นเดียวกัน

    (a) จงแสดงรูปคอนส์เซลล์รวมทั้งตัวนับการอ้างถึงของแต่ละคอนส์เซลล์ตลอดการทำงานในการหาค่านิพจน์ต่อไปนี้: `(car (cdr (cdr (cons 1 (cons 2 (cons 3 4))))))` ให้ระบุด้วยว่าคอนส์เซลล์ใดกลายเป็นขยะแล้ว

    (b) สำหรับ Lisp ที่มีคำสั่งเปลี่ยนค่าในคอนส์เซลล์: `rplaca` และ `rplacd` (หรือคำสั่ง `set!`, `set-car!` และ `set-cdr!` ใน Scheme) เราสามารถสร้างข้อมูลที่เป็นขยะ (ตามนิยามในเอกสาร) แต่จะไม่ถูกเก็บกวาดโดยวิธีการเก็บกวาดที่ใช้การนับการอ้างถึง จงยกตัวอย่างข้อมูลดังกล่าว (วาดรูป) พร้อมทั้งเขียนคำสั่งที่สร้างขึ้นมา

2.8 เอกสารอ้างอิง

แบบฝึกหัดเกี่ยวกับ reference count นำมาจาก Mitchell ข้อ 3.6