Part A Programming Language Principle

3 Control Flow

เนื้อหาในบทนี้อธิบายรายละเอียด คำศัพท์ และรายละเอียดเกี่ยวกับข้อความสั่ง (statement) ในบางกลุ่ม โดยในภาษาโปรแกรม (programming language) ใด ๆ จะมีข้อความสั่งที่ควบคุมลำดับการดำเนินงานของโปรแกรมในลักษณะต่าง ๆ ได้แก่

- Sequencing
- Selection
- Iteration
- Procedural abstraction
- Recursion

3.1 Expression

ในข้อความสั่งหนึ่ง ๆ จะมีองค์ประกอบพื้นฐาน คือ expression หรือ นิพจน์ ซึ่งเป็นข้อความหรือชุดข้อความที่เมื่อแปลหรือประมวลผลแล้วจะให้ผลลัพธ์เป็นค่าใดค่าหนึ่ง เป็นต้นว่า ค่าคงที่สัญพจน์ (literal constant) คือ ตัวอักษรหรือตัวเลขที่ให้ค่าคงที่ตามรูปที่แสดง เช่น ตัวเลข หรือ ข้อความที่อยู่ระหว่างเครื่องหมายอัญประกาศ (String) ชื่อตัวแปร (Named Variable) ชื่อตัวแปรจะแทนค่าใดค่าหนึ่งในหน่วยความจำที่อ้างไว้ ค่าคงที่ (constant) ซึ่งถูกกำหนดไว้ตั้งแต่ตอนเขียนโปรแกรม การคำนวณทางคณิตศาสตร์และตรรกศาสตร์ (operator กระทำกับ operand) และ อาร์กิวเมนต์ของฟังก์ชัน

ค่าของนิพจน์ดังที่กล่าวมาจะถูกใช้เป็นตัวควบคุมการทำงานของโปรแกรม เช่น นำไปเปลี่ยนค่าในหน่วยความจำที่ตัวแปรอ้างถึง เปลี่ยนสถานะของโปรแกรมตามค่าผลลัพธ์ของนิพจน์อย่างการเป็นเงื่อนไขการทำงาน (branch condition) หรือควบคุมการวนลูปว่าควรหยุดเมื่อใด เป็นต้น

ในแต่ละภาษาจะมีรูปแบบการวางตัวดำเนินการ (operator) เช่น เครื่องหมายบวกหรือลบ เป็นต้น แบ่งได้เป็น 3 รูปแบบ ได้แก่

- Prefix: ก่อนหรือข้างหน้า เช่น + a b
- Infix: กลางหรือระหว่าง เช่น a + b
- Postfix: หลังหรือท้าย เช่น a b +

3.2 Precedence and Associativity

ในการเขียนนิพจน์ที่เป็นการคำนวณทางคณิตศาสตร์ ในกรณีที่นิพจน์มีการใช้ตัวดำเนินการ (operator) มากกว่า 1 ตัว สมมติใช้การเขียนแบบ infix ซึ่งจะมีรูปแบบเหมือนการเขียนตามปกติของการคำนวณทางคณิตศาสตร์ บางครั้งก็จะมีการใส่เครื่องหมายวงเล็บครอบไว้เพื่อกำหนดลำดับการคำนวณ นั่นคือ คอมพิวเตอร์จะทราบได้ว่าการคำนวณจะต้องดำเนินการกับนิพจน์ที่อยู่ในเครื่องหมายวงเล็บก่อน แต่ในก็สามารถที่จะกรณีที่นิพจน์ไม่ได้ใส่เครื่องหมายวงเล็บกำกับไว้การคำนวณก็สามารถจะคำนวณได้โดยการเลือกลำดับใด ๆ มาประมวลผลก่อนหลังขึ้นอยู่กับผู้ที่ทำการคำนวณ ซึ่งทำให้เกิดความกำกวมว่าจะดำเนินการกับคู่ไหนก่อนหรือหลังได้ หากพิจารณานิพจน์ต่อไปนี้

```
a+b*c**d**e/f
```

นิพจน์ข้างต้นนี้เป็นนิพจน์ของภาษา Fortran (** หรือเครื่องหมายดอกจันคู่ หมายถึง ตัวดำเนินการยกกำลัง) หากต้องการจะแปลนิพจน์นี้จะต้องแปลลำดับการดำเนินการอย่างได้อย่างไร จะนำตัวแปร b และ c มาคูณกันก่อนแล้วจึงยกกำลังด้วยตัวแปร d หรือว่าในทางตรงกันข้าม ดังนั้นเพื่อให้ความเข้าใจในการแปลนิพจน์เหล่านี้เป็นไปในทางเดียวกันจึงมีคำศัพธ์ที่เกี่ยวข้อง 2 คำ คือ precedence และ associativity ซึ่งจะเป็นข้อกำหนดของภาษาโปรแกรมเพื่อแปลนิพจน์ในรูปแบบกำกวมนี้ให้ออกมาตรงกัน

ทุกภาษาโปรแกรมจะมีการกำหนด precedence และ associativity ซึ่งเป็นกฎที่สำหรับการแปลลำดับในการประมวลผลของตัวดำเนินการ โดย precedence จะเป็นตัวกำหนดลำดับของตัวดำเนินการ (operator) ว่าเมื่อเจอหลายตัว จะดำเนินการคำนวณ คู่ของตัวดำเนินการใดก่อน เช่น ระหว่างตัวดำเนินการทางคณิตศาสตร์ ได้แก่ บวก ลบ คูณ และหาร เครื่องหมายบวกและลบ จะมีค่า precedence ต่ำกว่า เครื่องหมายคูณและหาร นั่นคือ เมื่อพบนิพจน์ที่เครื่องหมายลบและคูณ อย่าง a-b*c การคำนวณจะทำการคำนวณคู่ของเครื่องหมายคูณก่อนจึงนำผลมากลบทีหลัง นั่นคือ จะมีความหมายเมื่อใส่วงเล็บกำกับ คือ a-(b*c) เป็นต้น

ในขณะที่ Associativity จะเป็นตัวกำหนดทิศทางการเลือกคู่ของตัวดำเนินการในแต่ละระดับของ precedence ว่าจะเลือกจากทิศทางใดก่อน ว่าจะเป็นจากซ้ายไปขวาหรือขวาไปซ้าย ตัวอย่างเช่น หากเรามีนิพจน์ที่มีทั้งเครื่องหมายลบและบวก อย่าง a-b+c ซึ่งทั้งสองเครื่องหมายจะมี precedence เท่ากัน และเมื่อ associativity สำหรับ precedence กลุ่มนี้ถูกกำหนดให้เป็นจากซ้ายไปขวา เครื่องหมายลบก็จะถูกเลือกคำนวณก่อนแล้วจึงนำผลไปบวกต่อ นั่นคือมีความหมายเดียวกับนิพจน์ (a-b)+c เป็นต้น

3.2.1 Precedence

ในบทเรียนภาษาโปรแกรมหรือหนังสือจะมีตารางกำหนด precedence และ associativity ซึ่งโดยทั่วไปแล้วกลุ่มของตัวดำเนินการที่มี precedence สูงซึ่งถูกเลือกดำเนินการก่อนจะอยู่ด้านบนสุดของตาราง ตัวดำเนินการที่อยู่ระดับเดียวกันก็จะมี precedence ระดับเดียวกัน โดยตัวอย่างต่อไปนี้จะแสดงการจัด precendence ของภาษาโปรแกรมต่าง ๆ ได้แก่ Fortran, Pascal, C และ Ada

3.2.1.1 Precedence ของภาษา Fortran

```
Predence Level 1: **
Predence Level 2: *, /
Predence Level 3: +, - (unary and binary)
Predence Level 4: .eq., .ne., .lt., .le., .gt., .ge. (comparisons)
Predence Level 5: .not.
Predence Level 6: .and.
Predence Level 7: .or.
Predence Level 8: .eqv., .neqv. (logical comparisons)
```

3.2.1.2 Precedence ของภาษา Pascal

```
Predence Level 1:  not
Predence Level 2:  *, /, div, mod, and
Predence Level 3:  +, - (unary and binary), or
Predence Level 4:  <, <=, >, >=, =, <>, IN
```

3.2.1.3 Precedence ของภาษา C

```
Predence Level 1:  ++, -- (post-inc., dec.)
Predence Level 2:  ++, -- (pre-inc., dec.), +, - (unary),
                   &, * (address, contents of), !, ~ (logic, bit-wise not)
Predence Level 3:  * (binary), /, % (modulo division)
Predence Level 4:  +, - (binary)
Predence Level 5:  <<, >> (left and right bit shift)
Predence Level 6:  <, <=, >, >= (inequality tests)
Predence Level 7:  ==, != (equality tests)
Predence Level 8:  & (bit-wise and)
Predence Level 9:  ^ (bit-wise exclusive or)
Predence Level 10: | (bit-wise inclusive or)
Predence Level 11: && (logical and)
Predence Level 12: || (logical or)
Predence Level 13: ?: (if…then…else)
Predence Level 14: =, +=, -+, *=, /=,%=, >>=, <<=, &=, ^=, |= (assigment)
Predence Level 15: , (sequencing)
```

3.2.1.4 Precedence ของภาษา Ada

```
Predence Level 1:  abs (absolute value), not, **
Predence Level 2:  *, /, mod, rem
Predence Level 3:  +, - (unary)
Predence Level 4:  +, - (binary), & (concatenation)
Predence Level 5:  =, /=, <, <=, >, >=
Predence Level 6:  and, or, xor (logical operators)
```

3.2.2 Associativity

เป็นการกำหนดลำดับการดำเนินการของตัวดำเนินการในแต่ละระดับของ precedence ว่าจะมีทิศทางเป็นอย่างไร ซึ่งสามารถแตกต่างในแต่ละระดับหรือกลุ่มของตัวดำเนินการได้ ว่าจะเป็นจากซ้ายไปขวา หรือขวาไปซ้าย โดยแต่ละกลุ่มจะถูกกำหนด associativity ดังนี้

- Basic arithmetic จะมีลำดับจากซ้ายไปขวา
- Exponetiation หรือ เลขยกกำลัง จะมีลำดับจากขวาไปซ้าย เช่น 4**3**2 จะมีลำดับการคำนวณเป็น 4**(3**2) เป็นต้น
- Assignment จะมีลำดับจากขวาไปซ้าย เช่น a = b = a + c จะทำการ assign ค่าคู่ขวาสุดก่อนนั่นคือ ให้ b = a + c จากนั้นจึงทำการ assign ค่าตัวแปร a = b ตามลำดับ เป็นต้น

Exercise 3.1 Precedence and Associativity

ให้ใช้ตาราง precedence ในหัวข้อ 3.2.1.1 ถึง 3.2.1.4 และ associativity ในหัวข้อ3.2.2 เพื่อแปลนิพจน์ของแต่ละภาษาที่กำหนดให้ต่อไปนี้ ให้อยู่ในรูปแบบที่ใส่วงเล็บกำกับลำดับการคำนวณ และให้หาผลลัพธ์ที่ได้จากการคำนวณ โดยกำหนดค่าของตัวแปร a ถึง f ดังนี้ a = 1, b=2, c=3, d=2, e = 2 และ f = 3

1. นิพจน์ภาษา Fortran: a + b * c ** d ** e / f
2. นิพจน์ภาษา Pascal: a < b and c < d
3. นิพจน์ภาษา C: a < b && c < d

เฉลย Exercise 3.1 ข้อ 1

นิพจน์ภาษา Fortran: a + b * c ** d ** e / f

คำตอบ: a + ((b * (c ** (d ** e))) / f) result is 55

คำอธิบาย:

จากนิพจน์ จัดลำดับ precedence ของเครื่องหมายคือ ลำดับที่ 1 คือ ** ลำดับที่ 2 คือ * และ / และลำดับที่ 3 คือ +

การคำนวณ precedence ลำดับที่ 1 คือ ** มี 2 เครื่องหมาย เมื่อพิจารณาจาก exponetiation จะมี associativity จากขวาไปซ้าย ดังนั้นลำดับจึงเป็น d**e ก่อน จึงนำผลลัพธ์ไปยกกำลังกับ c ได้ผลลัพธ์เป็น 3**(2**2) = 3**4 = 81

การคำนวณ precedence ลำดับที่ 2 คือ * และ / มี 2 เครื่องหมาย เมื่อพิจารณาจาก basic arithmatic จะมี associativity จากซ้ายไปขวา ดังนั้นจึงการคูณก่อนจากนั้นจึงหาร เป็นดังนี้ (b*81)/f = (2*81)/3 = 162/3 = 54

การคำนวณ precedence ลำดับที่ 3 คือ + มีเครื่องหมายเดียว ได้ผลลัพธ์เป็น a + 54 = 1 + 54 = 55

จบการเฉลย Exercise 3.1 ข้อ 1

เฉลย Exercise 3.1 ข้อ 2

นิพจน์ภาษา Pascal: a < b and c < d

คำตอบ: (a < (b and c)) < d result is static semantic error

จากนิพจน์ จัดลำดับ precedence ของเครื่องหมายคือ ลำดับที่ 1 คือ and ลำดับที่ 2 คือ <

ดังนั้น precedence ลำดับแรกจะวงเล็บคู่ and ได้เป็น a < (b and c) < d

Precendence ลำดับถัดมา มีเครื่องหมาย < 2 ตัว จึงพิจารณาจาก Associativity จากซ้ายไปขวา จึงวงเล็บครอบคู่ซ้ายก่อน ได้เป็น (a < (b and c)) < d

และเมื่อทำการคำนวณตามลำดับ คู่แรก คือ b and c ซึ่ง b มีค่าเป็น 2 และ c มีค่าเป็น 3 เมื่อมาทำการคำนวณกับตัวปฏิบัติการ and จะให้ผลเป็น compilation error เนื่องจากค่าที่ใช้กับ and ต้องเป็นชนิด Boolean ดังนั้น จึงให้ผลลัพธ์เป็น static semantic error นั่นเอง

จบการเฉลย Exercise 3.1 ข้อ 2

เฉลย Exercise 3.1 ข้อ 3

นิพจน์ภาษา C: a < b && c < d

คำตอบ: (a < b) && (c < d) result is F

คำอธิบาย:

จากนิพจน์ จัดลำดับ precedence ของเครื่องหมายคือ ลำดับที่ 1 คือ < ลำดับที่ 2 คือ &&

ดังนั้น precedence ลำดับแรก มีเครื่องหมาย < 2 ตัว จึงพิจารณาจาก Associativity จากซ้ายไปขวา จึงได้วงเล็บจากซ้ายไปขวา จะได้เป็น (a < b) && (c < d)

Precendence ลำดับถัดมา เครื่องหมาย && เนื่องจาก นิพจน์ถูกจัดกลุ่มทั้งด้านซ้ายและขวาของเครื่องหมายเรียบร้อยแล้วจึงไม่ต้องเติมวงเล็บครอบ ซึ่งจะเห็นว่ารูปแบบไม่มีวงเล็บเหมือนจะมีความหมายเดียวกับโจทย์ข้อ 2 แต่เนื่องจาก precedence ที่ต่างกันจึงได้ลำดับการประมวลผลที่แตกต่างกัน

ต่อมาเป็นการคำนวณหาผลลัพธ์ เมื่อแทนค่าตัวแปร จะได้เป็นดังนี้

(1 < 2) && (3 < 2)

เมื่อคำนวณค่าในแต่ละวงเล็บได้ผลลัพธ์เป็น Boolean ดังนี้

T && F

ดังนั้น ได้ผลลัพธ์เป็น F นั่นเอง

จบการเฉลย Exercise 3.1 ข้อ 3

3.2.3 Evaluation Order within Expression

จากหัวข้อก่อนหน้า precedence กับ associativity จะช่วยกำหนดลำดับการประเมินค่า (evaluation order) ภายในนิพจน์ว่าจะเป็นอย่างไร ซึ่งจะบอกลำดับของแต่ละคู่ยังไม่ครอบคลุมถึงลำดับของตัวถูกดำเนินการ (operands) ตัวใดจะถูกคำนวณก่อน ตัวอย่างเช่นนิพจน์ a - f(b) - c * d ถ้าใส่วงเล็บตาม precedence และ associativity จะได้เป็น (a - f(b)) - (c * d) ซึ่งทำให้เกิดการลบ ระหว่าง 2 นิพจน์ย่อย คือ a -f(b) และ c * d คำถามคือค่านิพจน์ฝั่งใดจะถูกคำนวณก่อน เช่นเดียวกันกับนิพจน์เรียกใช้ฟังก์ชัน f(a, g(b), h(c)) ซึ่งประกอบด้วยการส่งค่าอาร์กิวเมนต์ 3 ค่า คือ a, g(b) และ h(c) ซึ่งมีฟังก์ชันปนอยู่ คำถามคือจะคำนวณค่าของอาร์กิวเมนต์ตัวใดก่อน จะซ้ายไปขวาหรือขวาไปซ้ายหรือลำดับใดก็ได้

ลำดับที่เลือกบางครั้งจะกระทบต่อผลลัพธ์ในการประมวลผลนิพจน์ได้ อย่างเช่นหากนิพจน์แรกฟังก์ชัน f(b) มีการเปลี่ยนแปลงค่าของตัวแปร c หรือ d หรือหากนิพจน์หลังฟังก์ชัน g(b) ไปแก้ไขค่าตัวแปร a หรือ c ผลลัพธ์ก็อาจจะแตกต่างกันในลำดับการประมวลผลที่ต่างกัน (ซ้ายไปขวาให้ผลลัพธ์ไม่เหมือนกับขวาไปซ้าย) ดังนั้น การกำหนดลำดับการประมวลผลจึงสำคัญเพราะลำดับมีผลกระทบต้องผลลัพธ์ที่ได้ด้วย

นอกจากนี้ ในหลาย ๆ ภาษาโปรแกรมไม่ได้บังคับลำดับการประมวลผลตัวถูกดำเนินการและปล่อยให้เป็นหน้าที่ของ compiler ในการวิเคราะห์แปลโดยเลือกกำหนดลำดับการประมวลผลที่ทำงานได้ที่ได้เร็วที่สุด ตัวอย่างเช่นในการประมวลผลนิพจน์ a * c + f(c) หากแต่ละผลลัพธ์จะถูกเก็บไว้ใน register แล้วเมื่อเลือกคำนวณ a * b ก่อน สิ่งที่เกิดขึ้น เมื่อคำนวณแล้วจะมี Register ตัวนึงเก็บค่าผลลัพธ์ จากนั้นจึงเกิดการเรียกโปรแกรมย่อย f(c) ทำให้ machine code ที่แปลได้จะต้องเพิ่ม

บรรทัดของการ save register ผลลัพธ์ลงใน bookkeeping ของ f(c) และบรรทัดในการคืนค่า register เมื่อ f(c) ทำงานเสร็จสิ้นด้วย ซึ่งต่างจากการเลือกทำ f(c) ก่อน ซึ่งจะไม่เกิดบรรทัดในการ save และ restore ค่า register ผลลัพธ์จำนวนคำสั่งที่ประมวลผลจึงน้อยกว่าทำให้ทำงานได้เร็วกว่า ดังนั้น หากต้องการให้การทำงานของโปรแกรมมีสมรรถนะ (performance) ที่ดีที่สุด compiler ก็จะต้องเลือกลำดับที่จะคำนวณ f(c) ก่อน

แต่ก็ในบางภาษาโปรแกรมก็กำหนดลำดับตายตัวเอาไว้ อย่างภาษา Java และ C# กำหนดให้ใช้ลำดับจากซ้ายไปขวาเท่านั้น นั่นคือ นิพจน์ (a - f(b)) - (c * d) ลำดับที่จะถูกประมวลก่อนคือ a – f(b) ซึ่งทำให้ความหมายของนิพจน์มีความชัดเจนสำหรับโปรแกรมเมอร์ แต่อาจต้องแลกกับสรรถนะที่ลดลงจากการที่บางครั้งลำดับจากขวามาซ้ายทำได้ดีกว่า

จากประเด็นปัญหาในเรื่องลำดับการเลือกประมวลของตัวถูกดำเนินการหรืออาร์กิวเมนต์ ซึ่งส่งผลในด้านผลกระทบต่อผลลัพธ์และสมรรถนะการทำงาน โปรแกรมเมอร์จำเป็นต้องเข้าใจและคำนึงถึงลำดับที่แต่ละภาษาโปรแกรมจะใช้ในการประมวลผล และหลีกเลี่ยงผลที่ไม่ถูกต้องโดยการบังคับลำดับการประมวลผลให้ทำงานตามที่ต้องการจริง เช่น การจัดกลุ่มนิพจน์ด้วยวงเล็บ เป็นต้น

Exercise 3.2 Precedence, Associativity, Evaluation Order

กำหนดให้ใช้ precedence table และ associativity rule ในหัวข้อก่อนหน้านี้ โดยกำหนด evaluation order เป็นแบบ left-to-right ให้แสดงผลลัพธ์ทางหน้าจอของโปรแกรมภาษา C เมื่อกระทำการบรรทัดที่ 5 จากโค้ดต่อไปนี้

```
1:  int give2() { printf("two\n"); return 2; }
2:  int give3() { printf("three\n"); return 3; }
3:  int give4() { printf("four\n"); return 4; }
4:  int main() {
5:    printf("%d\n", give4() + give2() * give3() - give4() / give2());
6:    return 0;
7:  }
```

เฉลย Exercise 3.2

คำสั่ง printf เป็นการสั่งพิมพ์ผลลัพธ์ของนิพจน์ give4() + give2() * give3() – give4() / give2() เมื่อทำการจัดกลุ่มตาม precedence และ associativity แล้วจะได้ เป็นดังนี้

(give4() + (give2() * give3())) – (give4() / give2())

และเมื่อทำการคำนวณด้วย evaluation order แบบ left-to-right จะได้ เป็นการเลือกทำนิพจน์กลุ่มซ้ายก่อน คือ give4() + (give2() * give3()) ซื่อเมื่อถูก Evaluate ก็จะเลือกตัวด้านซ้ายก่อน คือ give4() จากนั้นจึงไปทำนิพจน์ด้านขวา คือ give2() และ give3() ตามลำดับ เช่นเดียวกันนิพจน์กลุ่มด้านขวา ก็จะถูกทำซ้ายไปขวา คือ give4 และ give2() ดังนั้น ลำดับการเรียกฟังก์ชันย่อยจะเป็นดังนี้

give4(), give2(), give3(), give4(), give2()

เมื่อพิจารณาแต่ละฟังก์ชันเมื่อถูกเรียกจะทำการพิมพ์ค่าตัวเลขออกมาเป็นตัวอักษรและคืนค่าของตัวเลขของฟังก์ชันนั้น เช่น give4() เมื่อถูกเรียก จะพิมพ์ข้อความ four ออกทางหน้าจอ และให้ค่าผลลัพธ์เป็นเลข 4 ดังนั้นเมื่อแต่ละฟังก์เมื่อถูกเรียก

แล้วจะได้ค่าตัวเลขแทนที่ในนิพจน์ เป็น (4 + (2 * 3)) – (4 / 2) ได้ผลลัพธ์เท่ากับ 8 และเมื่อโค้ดบรรทัดที่ 5 กระทำการเสร็จสิ้นจะได้ผลลัพธ์ดังนี้

four

two

three

four

two

8

จบการเฉลย Exercise 3.2

3.3 Assignment

Assignment เป็นตัวดำเนินการที่สำคัญของภาษาโปรแกรมเชิงคำสั่ง เพื่อใช้ในการเปลี่ยนแปลงค่าข้อมูลที่เก็บลงในหน่วยความจำที่ถูก bind ไว้กับตัวแปร เมื่อเกิด assignment คือ การนำค่าข้อมูลที่ต้องการไปเก็บไว้ยังหน่วยความจำที่ตัวแปรนั้นได้จองเอาไว้

Assignment เป็นตัวดำเนินการแบบมีตัวถูกดำเนินการ (operand) 2 ตัว ตัวหนึ่งจะเป็นค่าที่ต้องการจัดเก็บลงในหน่วยความจำ และอีกตัวเป็น reference ซึ่งอ้างอิงไปยังหน่วยความจำ เช่น ตัวแปร เป็นต้น

Assignment มีบทบาทสำคัญในภาษาโปรแกรมเนื่องจากเป็นการเปลี่ยนแปลงข้อมูลในหน่วยความจำ จึงก่อให้เกิดผลข้างเคียงในโปรแกรมทำให้ข้อความสั่งที่ถัดจาก assignment มีพฤติกรรมที่เปลี่ยนไปได้ พิจารณาจากตัวอย่างโค้ดต่อไปนี้

```
1:  //C
2:  int max(int x, int y) {
3:        if (x > y) {return x;} else {return y;}
4:  }
5:  int main() {
6:        int a, b;
7:        a = 1; b = 2;
8:        printf("max is %d\n", max(a, b)); //max is 2
9:        a = 3;
10:       printf("max is %d\n", max(a, b)); //max is 3
11:       return 0;
12: }
```

จากโค้ดจะเห็นว่าในบรรทัดที่ 8 และ 10 เป็นข้อความสั่งเดียวกันคือพิมพ์ค่า max ระหว่างตัวแปร a และ b ออกทางหน้าจอ แต่ผลลัพธ์ของทั้งสองบรรทัดแตกต่างกัน คือ พิมพ์ค่า 2 และ 3 ออกมาตามลำดับ เหตุผลที่ผลลัพธ์ max(a,b) ได้ไม่เท่ากันเพราะผลกระทบจากการเปลี่ยนแปลงค่าหน่วยความจำโดยข้อความสั่ง assignment ในบรรทัดที่ 9 ทำให้ค่าที่นำมาเปรียบเทียบในฟังก์ชัน max เปลี่ยน ซึ่งเป็นสาเหตุที่ทำให้พฤติกรรมให้คำสั่งที่เขียนเหมือนกันมีพฤติกรรมเปลี่ยนไป

แต่ในสำหรับบางภาษาเครื่องหมายเท่ากับไม่ได้ทำหน้าที่เป็นตัวดำเนินการ assignment ก็อาจจะไม่ได้รับผลกระทบอย่างที่ได้กล่าวมา อย่างภาษา Haskell เครื่องหมายเท่ากับถูกใช้ในความหมายในการประกาศค่าให้กับตัวแปร ซึ่งจะเป็นการประกาศครั้งเดียวและไม่สามารถเปลี่ยนแปลงค่าได้ ทำให้คำสั่ง max(a, b) สามารถคงพฤติกรรมเดียวกันได้ ไม่ว่าจะถูกเรียกซ้ำกี่ครั้งก็ตาม ตัวอย่างโค้ดตัวอย่างที่แล้วในภาษา Haskell เป็นดังนี้

```
1: --Haskell has no assignment and no side effect
2: a, b :: Int
3: a = 1
4: b = 2
5: max a b
6: --2
7: max a b
8: --2
```

3.3.1 Semantics of Assignment

การแปลความหมายของ assignment ขึ้นอยู่กับภาษาโปรแกรมว่ามีวิธีการอ้างอิงของ binding อย่างไร มีอยู่ 2 แบบ คือ value model และ reference model

สำหรับ value model นั้น คือ การกำหนดให้ตัวแปรแต่ละตัวเป็น container ที่อิสระจากกัน หมายถึง แต่ละตัวแปรจะอ้างตำแหน่งในหน่วยความจำคนละตำแหน่งกัน แต่ละตัวแปรเมื่อถูกใช้ในข้อความสั่งกำหนดค่าก็จะนำค่าจริงไปเก็บไว้ยังตำแหน่งหน่วยความจำที่ตัวแปรนั้น ๆ อ้างถึงโดยตรง โดยตัวแปรที่ถูกใช้ใน assignment จะถูกแปลความค่าได้ 2 แบบ คือ l-value กับ r-value

l-value เป็นค่าที่อยู่ในหน่วยความจำที่ถูกจองให้กับตัวแปรนั้น ๆ ส่วน r-value คือ ค่าข้อมูลที่ถูกจัดเก็บไว้ ณ หน่วยความที่ตัวแปรนั้นจองไว้ พิจารณาจากโค้ดแสดงข้อความสั่งกำหนดค่าภาษา C ดังต่อไปนี้

```
1: // C
2: b = 2;          // l-value of b is used
3: c = b;          // r-value of b is used, l-value of c is used
4: a = b+c;        // r-values of b and c are used, l-value of a is used
```

จากโค้ดแสดงให้เห็นว่าการแปลความว่าตัวแปรหนึ่งในข้อความสั่งกำหนดค่า จะถูกแปลเป็น l-value หรือ r-value นั้นขึ้นอยู่กับตำแหน่งของตัวแปรว่าอยู่ด้านไหนของเครื่องหมายเท่ากับ นั่นคือ ถ้าตัวแปรอยู่ทางด้านซ้าย (left-hand-side) ของเครื่องหมายจะถูกแปลเป็น l-value ซึ่งหมายถึงตำแหน่งที่จะนำค่าไปเก็บ แต่ถ้าตัวแปรอยู่ทางด้านขวา (right-hand-side) ของเครื่องหมายจะถูกแปลเป็น r-value ซึ่งคือค่าที่ตัวแปรนั้นเก็บอยู่ โดยจากโค้ดสามารถแสดงสถานะตัวแปรและหน่วยความจำได้ดังนี้

```
Data Segment:
```

a = 4, b = 2, c = 2

ในการทำงานของ assignment แต่ละบรรทัดจากโค้ดข้างต้นจึงสามารถแปลได้เป็น บรรทัดที่ 2 ตัวแปร b อยู่ทางด้านซ้ายของเครื่องหมายเท่ากับจึงใช้ l-value ดังนั้นจึงเป็นการนำค่า 2 ไปเก็บยังหน่วยความจำ ณ ตำแหน่งที่ 2 ดังแสดงใน Data Segment ข้างต้น บรรทัด 3 ตัวแปร c อยู่ทางด้านซ้ายของเครื่องหมายใช้ l-value และตัวแปร b อยู่ทางขวาของเครื่องหมาย

ใช้ r-value ดังนี้ จึงเป็นการนำค่าที่อยู่ในหน่วยความจำของ b คือ ค่า 2 ไปเก็บยังหน่วยความจำ ณ ตำแหน่งที่ 3 และบรรทัดที่ 4 ตัวแปร a ใช้ l-value กับตัวแปร b และ c ใช้ r-value เป็นการบวกค่า r-value คือ 2 + 2 ได้เท่ากับ 4 นำไปเก็บในหน่วยความจำที่ตำแหน่งที่ 1 หากแสดงสถานะของหน่วยความจำในแต่ละบรรทัดของโปรแกรมจะเป็นดังนี้

```
1:              // Data Segement: 0, 0, 0
2: b = 2;       // Data Segement: 0, 2, 0
3: c = b;       // Data Segement: 2, 2, 2
4: a = b+c;     // Data Segement: 4, 2, 2
```

สำหรับส่วนของ assignment แบบ reference model นั้น การแปลความหมายของตัวแปรจะเปลี่ยนไปใช้ ค่า l-value ทั้งหมด แต่เมื่อตัวแปรนั้นปรากฏอยู่ในตำแหน่งของ r-value โปรแกรมจะใช้การ dereference เพื่อให้ได้ค่าจริงมาแทน ซึ่งจะถูกทำโดยอัตโนมัติในภาษาโปรแกรมส่วนมาก พิจารณาจากโค้ดต่อไปนี้

```
1: %Clu
2: b := 2;      % l-value of b is used
3: c := b;      % l-value of b is dereferenced, l-value of c is used
4: a := b+c;    % l-values of b and c are dereferenced, l-value of a is used
5:              % 2 and 4 are immutable values at some locations to which any
variables can refer
```

จาก code เป็นตัวอย่างภาษา Clu ใช้เครื่องหมาย := เป็นเครื่องหมาย assignment ในบรรทัดที่ 2 คือ กำหนดค่า 2 ให้กับตัวแปร b และด้วย reference model ค่า 2 จึงถูกนำไปเก็บในหน่วยความจำก่อน เช่น เก็บไว้ใน heap ตำแหน่งที่ 1 จึงนำตำแหน่งที่บันทึกค่านี้ไปปรับปรุงค่า l-value ของตัวแปร b โดยสถานะหน่วยความเมื่อโปรแกรมทำงานทั้ง 3 ข้อความสั่งแล้วจะได้ดังนี้

```
Binding Table (binding = heap address):
a = 2, b = 1, c = 1
Heap:
2, 4
```

ด้วยระบบของภาษา Clu จะมองค่า 2 และ 4 ใน heap เป็นค่าคงที่ (Constant) ไม่สามารถเปลี่ยนแปลงค่า (immutable value) ได้แล้ว นั่นคือ เขียนค่าลงไปได้แค่ครั้งเดียว

ตัวแปรใน reference model ทุกตัวจะถูกมองเป็น l-value หมด เพราะฉะนั้นจากเมื่อไหร่ก็ตามที่ตัวแปรถูกวางไว้ด้านขวาของ assignment เช่น บรรทัดที่ 3 ตัวแปร b เมื่ออยู่ทางด้านขวาของ assignment ตัวแปรนั้นจะถูก dereference ก่อน เพื่อจะเอาค่าที่มันอ้างถึงเอาไปใช้ กล่าวคือ เมื่อเราเอา b ไป assign ให้ c ต้องทำการ dereference ก่อนว่า b นี้ reference ไปยังค่าอะไรแล้วเอาค่านั้นมา assign ให้กับ c อีกทีหนึ่งซึ่งพอ assign ให้กับ c ก็เหมือนกับว่าเราให้ c reference ไปยังค่า 2

จาก model ที่กล่าวมาการนั้นจะถูกนำไปใช้อย่างไรก็ขึ้นอยู่กับแต่ละภาษาโปรแกรมขึ้นอยู่กับปัจจัยต่าง ๆ เช่น สมรรถนะการทำงาน หรือเพื่อการบริหารการใช้ทรัพยากรระบบ เป็นต้น การใช้งานไม่จำเป็นจะต้องใช้เฉพาะวิธีการอย่างใดอย่างหนึ่ง สามารถใช้ร่วมกันได้อย่างเช่น ภาษา Java มีการใช้ value mode กับ build-in type เช่น int, double, float,

long เป็นต้น และใช้ reference model สำหรับ user-defined type ได้แก่ คลาสต่าง ๆ ดังตัวอย่างการประกาศตัวแปรต่อไปนี้

```
1: int a;
2: Dog d= new Dog();
```

โดยสถานของหน่วยความจำจะเป็นดังนี้

Data Segment:

a = 0

```
Binding Table(binding = heap address):
d = 1
Heap:
```

dog object

3.4 Short-Circuit Evaluation

ในบางภาษาโปรแกรมจะมีการออกแบบการประมวลผลที่เรียกว่า short-circuit evaluation สำหรับการประมวลผลนิพจน์ตรรกะ (logic expression) ซึ่งจะให้ผลลัพธ์เป็น true หรือ false ดังตัวอย่างต่อไปนี้

```
(a < b) and (b < c)
(a > b) or (b > c)
```

จากนิพจน์ข้างต้นในการทำงานของ short-circuit จะให้ผลลัพธ์ทันทีที่รู้คำตอบของนิพจน์ เช่น นิพจน์แรกถ้า a < b เป็น false ก็จะให้คำตอบทันโดยไม่ต้องดูของนิพจน์ย่อยอื่น เช่นเดียวกันกับนิพจน์คำตอบก็จะได้ทันทีหาก a > b เป็น true เนื่องจากอื่นจะได้ผลลัพธ์เป็นอย่างก็ไม่ทำให้ผลลัพธ์เปลี่ยน ด้วยวิธีการนี้จะทำให้การประมวลผลทำได้เร็วขึ้นเนื่องจากไม่ต้องทำการประมวลผลทั้งนิพจน์

3.4.1 Short-Circuit Changes Semantics of Boolean Expressions

ต่อไปนี้เป็นตัวอย่างของโปรแกรมเพื่อค้นหาภายใน list โดยจะแสดงเปรียบเทียบภาษาโปรแกรมที่ใช้และไม่ใช้ short-circuit ภาษาโปรแกรมแรกเป็นแบบที่ใช้ short-circuit คือ ภาษา C ดังนี้

```
1: //C
2: p = my_list;
3: while (p && p->key != val)
4:    p = p->next;
```

อีกภาษาไม่ใช้ short-circuit คือ ภาษา Pascal ดังนี้

```
1: (* Pascal *)
2: p := my_list;
3: while (p <> nil) and (p^.key <> val) do (*ouch!*)
4:    p := p^.next;
```

จากตัวอย่างโปรแกรมทั้ง 2 ภาษาข้างต้นเขียนออกมาให้ความหมายเดียวกัน แต่เนื่องจากความแตกต่างในการใช้กับไม่ใช้ short-circuit ทำให้การทำงานในบรรทัดที่ 3 ซึ่งมีนิพจน์ตรรกะอยู่ ทำให้การทำงานของโปรแกรมให้ผลลัพธ์แตกต่างกันไป กล่าวคือ กรณีที่ my_list ไม่เคยถูกกำหนดค่าใด ๆ มาก่อน ทำให้ p ไม่มีค่า สำหรับการประมวลผลเมื่อตัวแปรที่ไม่มีค่าจะถูกแปลค่าเป็น false ดังนี้ ในภาษา C จึงไม่กระทำการภายใน block ของ while และโปรแกรมทำงานได้ไม่เกิดข้อผิดพลาดใด ๆ

แต่ในขณะที่ภาษา Pascal ไม่ได้มี short-circuit จึงจำเป็นต้องทำการตรวจสอบนิพจน์อีกฝั่งของตัวดำเนินการ and ซึ่งคือ p^.key <> val ซึ่งในกรณีที่ p ไม่ได้ถูกกำหนดค่า หรือมีค่าเป็น nil จะทำให้โปรแกรมเกิดข้อผิดพลาดขึ้นเนื่องจากหาข้อมูลที่อ้างถึงไม่ได้ ก็จะทำให้โปรแกรมจบด้วยการเกิดข้อผิดพลาดนั่นเอง ซึ่งหากต้องการแก้ไขให้โปรแกรมทำงานได้อาจจะต้องเขียนโปรแกรมที่ยาวและซับซ้อนขึ้น เช่น

```
1: (* Pascal *)
2: p := my_list;
3: if (p <> nil) then
4:    while (p^.key <> val) do
5:      p := p^.next;
```

ดังนั้น short-circuit ให้ประโยชน์ทั้งด้านการลดเวลาในการประมวลผลของโปรแกรมและการลดขนาดของโปรแกรมที่ถูกเขียนขึ้นด้วย แต่การเขียนจะเป็นรูปแบบใดนั้นโปรแกรมเมอร์เองจำเป็นจะต้องทราบด้วยว่าภาษาโปรแกรมที่ใช้เขียนนั้นมีการใช้ short-circuit ด้วยหรือเปล่า เพื่อให้สามารถเขียนโปรแกรมที่ทำงานได้ถูกต้องนั่นเอง

Exercise 3.3 Short-Circuit

จากโค้ดที่กำหนดให้ต่อไปนี้ให้แสดงวิธีใช้ประโยชน์จาก short-circuit ในการตรวจสอบเงื่อนไขของ logical expression ในบรรทัดที่ 4 และ 6 เพื่อเพิ่มความปลอดภัยของโปรแกรม

```
1: const int MAX = 10;
2: int A[MAX];
3: ...
4: if (A[i] > foo) ...
5: ...
6: if (n/d < threshold)...
```

เฉลย Exercise 3.3

ในบรรทัดที่ 4 การเรียก array มีโอกาสเกิด array index out-of-bound ดังนั้นควรตรวจสอบค่าของตัวแปร i ว่าอยู่ภายในขอบเขตของ array หรือเปล่าซึ่งประกาศไว้ในค่าคงที่ชื่อ MAX ดังนั้นขอบเขตคือ 0 ถึง MAX – 1 หรือในกรณีนี้คือ 9 ดังนั้น การแก้ไขก็จะได้เป็นดังนี้

```
if ((i >=0) && (i < MAX) && A[i] > foo) ...
```

ซึ่งในการประมวลผล A[i] > foo จะไม่ถูกคำนวณหากค่าของตัวแปร i ไม่ได้อยู่ในขอบเขตของ array นั่นคือ หาก i >= 0 เป็น false ก็จะไม่ทำงานข้อความสั่งภายใน if ทันที หรือหากเป็น true แต่ i < MAX เป็น false ก็จะได้ผลเช่นเดียวกัน ทำให้โปรแกรมไม่มีโอกาสเกิด array index out-of-bound ได้แล้ว

ต่อมาในบรรทัดที่ 6 ตัวแปรที่มีความเสี่ยงที่จะทำให้โปรแกรมทำงานผิดพลาด คือ ตัวแปร d ซึ่งจะเกิด Error หากมีค่าเป็น 0 (divide by zero)ดังนั้น ควรเพิ่มการตรวจสอบว่าหากมีค่าเป็น 0 ไม่ให้ตรวจสอบด้วยการหาร จะได้เป็นดังนี้

```
if (d != 0 && n/d < threshold) ...
```

จบการเฉลย Exercise 3.3

3.5 Sequencing

เป็นลักษณะของการประมวลผลข้อความสั่งแบบเรียงตามลำดับ โดยแต่ละข้อความสั่งก็สามารถจะก่อให้เกิดผลข้างเคียงต่อโปรแกรม โดยเมื่อมีข้อความสั่งเรียงกันในมุมมองของโค้ดบรรทัดบนจะถูกกระทำการก่อน โดยชุดของ sequening สามารถถูกห่อหุ้มเป็น compound statement (block) เช่น begin ... end หรือ {...}

3.6 Selection

Selection จะอยู่ในรูปของ if ... then ... else ... ซึ่งแต่โปรแกรมภาษาก็จะกำหนดวากยสัมพันธ์ (syntax) ที่แตกต่างกันไป โดยรูปแบบทั่วไป ก็จะเป็นดังต่อไปนี้

```
if condition then statement
else if condition then statement
else if condition then statement
...
else statement
```

3.6.1 Short-Circuited Condition in Selection

ในโปรแกรมภาษาที่ใช้ short-circuit เมื่อถูกใช้ใน condition ของ if-then-else จะช่วยให้ compiler สามารถสร้างโค้ดที่ควบคุมการกระโดดไปทำงานในตำแหน่งต่าง ๆ ตาม เงื่อนไขได้โดยไม่ต้องมีการบันทึกผลลัพธ์การเปรียบเทียบทางตรรกะ (Boolean value) เก็บไว้ในรีจิสเตอร์ พิจารณาจากตัวอย่างต่อไปนี้

```
1: if ((A > B) and (C > D)) or (E ≠ F) then
2:     then_clause
3: else
4:     else_clause
```

จากโค้ดจะสามารถเกิด short-circuit ได้ 2 จุ คือ กรณีแรก A > B เป็น false (ไม่ต้องทำ C > D) และกรณีที่สอง A > B และ C > D เป็น true (ไม่ต้องทำ E ≠ F) ดังนั้น จากคุณสมบัติเหล่านี้โค้ดที่ complier สร้างขึ้นจะได้เป็นดังนี้

```
1:        r1 := A
2:        r2 := B
3:        if r1 <= r2 goto L4
4:        r1 := C
5:        r2 := D
6:        if r1 > r2 goto L1
7: L4:    r1 := E
8:        r2 := F
9:        if r1 = r2 goto L2
```

```
10: L1:    then_clause
11:        goto L3
12: L2:    else_clause
13: L3:
```

จากโค้ดที่ถูกสร้างขึ้นจะสังเกตได้ว่า ณ บรรทัดที่ 3 จะกระโดดไป L4 เมื่อ A > B เป็น false คือ A <= B (r1 <= r2) เป็น true ซึ่งจะข้ามการทำงานส่วนเปรียบเทียบ C > D ไป และ ในบรรทัดที่ 6 เมื่อ C > D (r1 > r2) เป็น true จะกระโดดไป L1 ซึ่งจะข้ามการเปรียบเทียบ E ≠ F ทำให้ code บางส่วนของการเปรียบเทียบตรรกะจะไม่ถูกกระทำการ นอกจากกนี้ จะสังเกตได้ว่าในโค้ดที่ถูกสร้างขึ้นมานั้นใช้รีจิสเตอร์เพียง 2 ตัว คือ r1 และ r2 นั่นคือไม่ต้องมีการเก็บผลลัพธ์ค่า Boolean ผลลัพธ์ ซึ่งหากไม่ได้ใช้ short-circuit จะต้องใช้รีจิสเตอร์อย่างน้อยอีก 1 ตัวเพื่อเก็บผลลัพธ์นำไปเปรียบเทียบต่อไป

3.6.2 Nested If

ลักษณะของ Nested if จะเป็น selection ที่มีหลาย ๆ เงื่อนไขเพื่อจะสั่งให้ทำงานเมื่อสถานะของโปรแกรมตรงตามเงื่อนไข โดยการตรวจสอบจะทำการตรวจสอบตามลำดับของเงื่อนไขไปทีละตัวเมื่อตรงตามเงื่อนจึงเข้าทำงานใน block นั้นแล้วจบการตรวจสอบชุดของ selection นั้น ๆ พิจารณาจากโค้ดดังต่อไปนี้

```
1:  --Ada
2:  i := …  -- calculate tested expression
3:  if i = 1 then
4:    clause_A
5:  elsif i = 2 or i = 7 then
6:    clause_B
7:  elsif i in 3..5 then
8:    clause_C
9:  elsif i = 10 then
10:   clause_D
11: else
12:   clause_E
13: end if;
```

จากโค้ดเมื่อ compiler สร้างโค้ดแล้วจะได้เป็นดังนี้

```
1:        r1 := ...  - calculate tested expression
2:        if r1 ≠ 1 goto L1
3:        clause_A
4:        goto L6
5: L1:    if r1 = 2 goto L2
6:        if r1 ≠ 7 goto L3
7: L2:    clause_B
8:        goto L6
9: L3:    if r1 < 3 goto L4
10:       if r1 > 5 goto L4
```

```
11:       clause_C
12:       goto L6
13: L4:   if r1 ≠ 10 goto L5
14:       clause_C
15:       goto L6
16: L5:   clause_E
17: L6:
```

จากโค้ดที่ถูกสร้างขึ้นมาข้างต้นนั้นจะเห็นว่า การกระโดดในช่วงของการตรวจสอบเงื่อนไขในกรณีที่ไม่ตรงตามเงื่อนไขจะค่อย ๆ กระโดดไปตามลำดับ Label ของ if clause เป็นลำดับ ๆ ไป นั่น คือ กระโดไป L1 จากบรรทัดที่ 2 L3 จากบรรทัดที่ 6 และ L4 จากบรรทัดที่ 9 และ 10 ตามลำดับไป

3.6.3 Case/Switch Statements

นอกจาก nested if แล้วภาษาโปรแกรมยังได้มี selection อีกแบบหนึ่ง คือ case/switch ซึ่งจะมีลักษณที่ มี label บอกเงื่อนไขไว้ด้านซ้าย เช่น case ตามด้วย condition เป็นต้น โดยถ้าสถานะของโปรแกรมตรงตามเงื่อนไขในแต่ละเคสก็จะถูกกระทำการ โดยโครงสร้างการเขียนโปรแกรมในลักษณะ case/switch มีวัตถุประสงค์หลักเพื่อให้การสร้างโค้ดที่มีประสิทธิภาพทำได้โดยง่าย นั่นคือ ผลการสร้างโค้ด (code generation) จะได้ผลลัพธ์ที่แตกต่างไปจาก nested if ที่สามารถเขียนให้มีตรรกะเทียบเท่ากันได้ ตัวอย่างโค้ด case/switch เป็นดังนี้

```
1: --Ada with case labels and arms
2: case …  --calculate tested expression
3: is
4:        when 1      => clause_A
5:        when 2 | 7  => clause_B
6:        when 3..5   => clause_C
7:        when 10     => clause_D
8:        when others => clause_E
9: end case;
```

จากโค้ดส่วนที่เป็น case label คือ ส่วนที่เริ่มตั้งแต่ when ถึง ก่อนเครื่องหมาย => คือ arm โดยโค้ดที่สร้างขึ้นโดย compiler จะเป็นดังนี้

```
1: -- General form
2:
3:        goto L6  --jump to code to compute address
4: L1:    clause_A
5:        goto L7
6: L2:    clause_B
7:        goto L7
8: L3:    clause_C
9:        goto L7
10:       ...
```

```
11: L4:    clause_D
12:        goto L7
13: L5:    clause_E
14:        goto L7
15: L6:    r1 := ...  --computed target of branch
16:        goto *r1
17: L7:
```

จากโค้ดที่ถูกสร้าง จะเห็นว่าส่วนของ arm จะถูกสร้างและมี label กำกับไว้ก่อน (L1 – L5) โดยจะรวมส่วนของการคำนวณ Label เป้าหมายว่าจะเริ่มต้นที่ arm ไหน ไว้ที่เดียว โดยรวมส่วนของเงื่อนไข case ทั้งหมดเข้าด้วยกัน คือ ตั้งแต่ L6 (บรรทัดที่ 15 เป็นต้นไป) ซึ่งเมื่อโปรแกรมเริ่มต้นก็จะกระโดดมาทำงานที่บรรทัดนี้ก่อน โดยผลลัพธ์จะได้เป็นที่อยู่ของปลายทางที่จะกระโดดไป ดังที่แสดงในบรรทัดที่ 16

โดยเป้าหมายการออกแบบของ case/switch คือ โค้ดที่ถูกสร้างขึ้นได้จะมีลักษณะที่จะคำนวณหาค่าเพื่อเข้าถึงที่อยู่ของ arm โดยตรง ในขณะที่ nested if จะค่อย ๆ ไล่ไปทีละเงื่อนไขว่าตรงกันหรือไม่ไปตามลำดับทำให้ code ที่ได้จะมีลักษณะของการปนกันในการตรวจสอบกับส่วนของเป้าหมายที่ต้องการประมวลผล ซึ่งจะมีการกระโดดไปมาที่ซับซ้อนกว่า

3.6.4 Case/Switch Implementation Example

ต่อไปนี้จะเป็นตัวอย่างวิธีการคำนวณหา address ของ arm ที่จะสร้างขึ้นโดยคอมไพล์ โดยขยายจากตัวอย่างในหัวข้อที่แล้ว โดยมีการเปลี่ยนแปลงตั้งแต่บรรทัดที่ 15 ดังนี้

```
15: T:     &L1    -- tested expression = 1
16:        &L2
17:        &L3 → r1 = 0
18:        &L3 → r1 = 1
19:        &L3 → r1 = 2
20:        &L5
21:        &L2
22:        &L5
23:        &L5
24:        &L4    -- tested expression = 10
25: L6:    r1 := ...
26:        if r1 < 1 goto L5
27:        if r1 > 10 goto L5
28:        r1 -:= 1
29:        r1 := T1[r1]
30:        goto *r1
31: L7:
```

```
1:         r1 := ...  - calculate tested expression
2:         if r1 ≠ 1 goto L1
3:         clause_A
4:         goto L6
5:  L1:    if r1 = 2 goto L2
6:         if r1 ≠ 7 goto L3
7:  L2:    clause_B
8:         goto L6
9:  L3:    if r1 < 3 goto L4
10:        if r1 > 5 goto L4
11:        clause_C
12:        goto L6
13: L4:    if r1 ≠ 10 goto L5
14:        clause_C
15:        goto L6
16: L5:    clause_E
17: L6:
```

จากโค้ดข้างต้นประกอบด้วยส่วนต่าง ๆ คือ บรรทัดที่ 15 ซึ่งถูกกำกับด้วย label ชื่อว่า T ซึ่งเป็นการประกาศเป็น Array ที่เก็บที่อยู่ของบรรทัดของแต่ละ arm โดยใช้ชื่อ label ต่อท้ายเครื่องหมาย & เช่น &L1 หมายโค้ดบรรทัดที่ 4 เป็นต้น โดย T

เป็น array ขนาด 10 ที่ใส่ข้อมูลที่อยู่ของ arm ในแต่ละ condition ของค่า r1 เช่น ถ้าค่า r1 เท่ากับ 1 ก็จะให้ค่าที่อยู่ &L1 และหากเท่ากับ 10 ก็จะให้ค่าที่อยู่ &L4 เป็นต้น ส่วนทำไมขนาด array ทำไมต้องเป็น 10 นั้น อธิบายได้ว่า compiler ได้ทำการวิเคราะห์ส่วนของ case ทั้งหมด ว่าอยู่ในช่วงไหน เช่น กรณีนี้จะเป็น 1-10 เป็นต้น ดังนั้นค่าที่สนใจจึงมี 10 ค่าจึงได้สร้าง array ซึ่งเรียกว่า jump table เพื่อใช้บอกค่าในช่วงที่ case สนใจในแต่ละค่าว่าจะถูกพากระโดดไปทำงาน ณ จุดใดนั่นเอง โดยจะสังเกตได้ว่าค่าที่อยู่จะสอดคล้องกับค่าเงื่อนไขใน case เช่น when 2 | 7 ซึ่ง arm หรือ clause_B นั้นถูกสร้างไว้ที่ label ชื่อ L2 ดังนี้ค่าใน array ที่ index เท่ากับ 1 และ 6 (บรรทัดที่ 16 และ 21)จะมีค่าเป็น &L2 เป็นต้น

ตอนนี้ก็ทราบถึงตัวเลขที่อยู่ในช่วงขอบเขตของ case ซึ่งจะถูกนำไปสร้างเป็น jump table แล้ว ส่วนที่นอกขอบเขตก็ต้องมีการจัดการด้วย ซึ่งจะมีปรากฏในบรรทัดที่ 26 ค่าที่ต่ำกว่า 1 และบรรทัดที่ 27 ค่าที่มากกว่า 10 เหล่านี้ให้ทำในส่วนของ others case ส่วนค่า r1 ถ้าอยู่ในช่วงก็จะถูกนำมาใช้เป็น index ดึงค่าที่อยู่ที่ต้องกระโดดไป โดยจะมีการปรับ offset ให้ตรงกัน นั่นคือลบด้วย 1 เนื่องจาก array มีค่า index เริ่มต้นที่ 0 จากนั้นจึงได้ที่อยู่ที่ต้องการตามทำงานในบรรทัดที่ 29 และจึงกระโดดไปยังที่อยู่ผลลัพธ์ที่ได้มานั้นดังแสดงในบรรทัดที่ 30 เช่นถ้า r1 มีค่าเป็น 3 โปรแกรมก็จะโดดทำงานยังบรรทัดที่ 8 นั่นเอง ซึ่งจะเห็นว่าจะใช้ค่าที่สนใจเพื่อเลือกที่อยู่ที่จะกระโดดไปทำงาน ณ บรรทัดที่ต้องการได้เลย โดยไม่ต้องคอยตรวจสอบทีละ case ว่าค่าที่สนใจตรงกับเงื่อนไขหรือเปล่าทำให้สามารถทำงานได้เร็วกว่าเมื่อเทียบกับ nested if

Exercise 3.4 Case/Switch Implementation

คำถาม 1: จากตัวอย่างการสร้างโค้ด case/switch โดยใช้ jump table จะสามารถเกิดปัญหาอะไรขึ้นได้บ้าง?

คำถาม 2: จากคำตอบของคำถาม 1 จงอธิบายว่า compiler ความมีวิธีแก้ปัญหานั้นอย่างไร?

เฉลย Exercise 3.4

เฉลยคำถาม 1:

เนื่องจาก jump table จะสร้างขึ้นตามช่วงข้อมูลที่สนใจคิดอย่างง่าย คือ จะค่า min และ max ของ case เช่น จากตัวอย่างในหัวข้อที่แล้วค่าจะอยู่ในช่วง 1 – 10 ขนาดของ jump table จึงเป็น 10 แต่ถ้าหากช่วงค่าที่สนใจใน case มีช่วงที่กว้างมากขึ้น เช่น 1 – 200 ขนาดของ jump table ก็จะมีขนาดใหญ่ขึ้นตามไปอย่างกรณีนี้ก็จะเปลี่ยนเป็น 200 ดังนั้นส่วนของหน่วยความจำส่วนของโปรแกรม (text segment) ก็จะมีขนาดใหญ่ขึ้นตามความกว้างของช่วง case ซึ่งกรณีนี้จะไม่เป็นปัญหาหาก case สนใจทุก ๆ ค่า คือ มี arm สำหรับทุกตัว แต่บางครั้งช่วงที่เกิดขึ้นอาจจะมีค่าที่สนใจเพียงไม่กี่ค่า เช่น สนใจแค่ค่าขอบ อย่าง 1 และ 200 หรือ ค่าที่สนใจกระจัดกระจายกันไปไม่ได้สนใจทั้งหมด ซึ่งหากเป็นกรณีหลังก็จะเกิดการสิ้นเปลืองเนื้อที่หน่วยความจำ

เฉลยคำถาม 2:

จากปัญหาการสิ้นเปลืองเนื้อที่หน่วยความจำที่กล่าวมา กรณีที่ case มีจำนวนน้อยสามารถแก้ได้ด้วยการเปลี่ยนไปใช้งาน nested if ซึ่งไม่มีการสร้าง jump table แทน

สำหรับกรณีที่ case กระจัดกระจายก็อาจจะเปลี่ยนวิธีการสร้าง jump table ของ compiler เช่น เปลี่ยนเป็น binary search tree ซึ่งเก็บเฉพาะค่าที่สนใจ และการคำนวณที่อยู่จากค่าก็ใช้วิธีการค้นหาใน jump table แทนการคำนวณค่า index ก็จะทำได้เร็วที่จะใช้ nested if เป็นต้น ซึ่ง compiler อาจจะต้องมีกระบวนการในการวิเคราะห์เพิ่มเติมเพื่อเลือกสร้างโค้ดให้เหมาะสมกับ case ที่ถูกเขียนขึ้น

จบการเฉลย Exercise 3.4

3.6.5 Switch Statement with Fall-Through

Swith แบบ Fall-Through จะพบได้ในภาษาโปรแกรม อย่างเช่น ภาษา C และสืบทอดต่อยังภาษาลูก เช่น C++ และ Java เป็นต้น โดยตัวอย่างโค้ดเป็นดังนี้

```
1:  switch (… /*tested expression */) {
2:  case 1: clause_A;
3:        break;
4:  case 2:
5:  case 7: clause_B;
6:        break;
7:  case 3:
8:  case 4:
9:  case 5: clause_C;
10:       break;
11: case 10: clause_D;
12:       break;
13: default: clause_E;
14:       break;
```

ลักษณะของการเขียนโปรแกรมในวิธีการนี้ แต่ละค่าที่สนใจจะต้องถูกเขียนเป็น label แยกกัน ไม่สามารถเขียนเป็นช่วงค่าหรือเป็นนิพจน์บูเลียนได้ แต่ในแต่ละ label สามารที่จะไม่ต้องกำหนด arm หรือ clause ได้ โดยหาก case ใด ๆ มีค่าตรงค่าที่สนใจก็จะมีการ fall-through ไปยังทุก case ที่ถัดจาก case นั้น ซึ่งจะทำให้ทุก arm ถูกประมวลผลตามไปด้วยทั้งหมด เว้นเสียแต่ว่ามีการเขียนข้อความสั่ง break ซึ่งเป็นคำสั่ง exit ออกจาก block ซึ่งเมื่อเกิดใน block ของ swithc ก็จะเป็ฯการสั่งหยุดการ fall-through และออกจาก block ของ switch ไปทำงานข้อควมสั่งที่ถัดต่อไปจาก block

ดังนั้นหากจะกำหนดค่าที่สนใจเป็นช่วงก็สามารถทำได้โดยกระจายค่า และใส่ arm ไว้ใน label สุดท้ายที่เหลือละส่วน arm ไว้ เช่น ในบรรทัดที่ 7 – 9 เทียบได้กับนิพจน์ภาษา Ada 3..5 เป็นต้น

3.7 Iteration

การวนซ้ำ (Iteration) เป็นลักษณะการควบคุมลำดับการประมวลผลของโปรแกรมอีกอย่างหนึ่ง ซึ่งมีอยู่หลายลักษณะดังจะอธิบายต่อไปนี้

การวนลูปแบบ enumeration controlled เป็นการวนลูปในจำนวนของค่าที่มีการจำกัดขอบเขตไว้ เช่น มีการกำหนดค่าเริ่มต้นและสิ้นสุด

การวนลูปแบบ logically controlled เป็นการวนลูปที่ขึ้นอยู่กับสถานะของค่าบูเลียนค่าหนึ่งจนกว่าจะมีการเปลี่ยนแปลง

การวนซ้ำที่วนไปตามสมาชิกในข้อมูลชนิด collection อย่าง Set หรือ List เป็นต้น โดยวนไปเรื่อย ๆ จนครบตามจำนวนสมาชิก เพื่อทำการประมวลผลบางอย่างกับสมาชิกภายใน collection

3.7.1 Enumeration-Controlled Loop

พิจารณาจากตัวอย่างโค้ดจากภาษาโปรแกรม 2 ภาษา ดังต่อไปนี้

ภาษา modula-2:

```
1:  (* Modula-2: enumeration-controlled *)
2:  FOR i := first TO last BY step DO
3:  ...
4:  END
```

ภาษา C:

```
1:  /* C: combination of enumeration- and logically-controlled */
2:  for (i = first; i <= last; i += step) {
3:          ...
4:  }
```

ตัวอย่างในภาษา Modular-2 เป็นแบบ enumeratation-controlled คือ มีการกำหนดขอบเขตของ loop ได้เลย ดังนิพจน์ first TO last BY step คือ เริ่มต้นจาก first ถึง last โดยให้เพิ่มค่ารอบละ step

แต่ในตัวอย่างภาษา C จะเป็นแบบผสม enumeration-controlled กับ logically-controlled คือ การสิ้นสุดลูปจะตรวจสอบด้วยนิพจน์บูเลียน คือ i <= last ซึ่งจะถูกตรวจสอบทุกครั้งก่อนเข้าประมวลผลใน block ถ้าไม่ตรงเงื่อนไขหรือได้ค่าเป็น false ก็จะจบการวนซ้ำลง โดยจะโค้ดตัวอย่างเมื่อถูก compiler สร้างโค้ดจะได้เป็นดังนี้

```
1:          r1 := first
2:          r2 := step
3:          r3 := last
4:          goto L2
5:  L1:     ...
6:          r1 := r1 + r2
7:  L2:     if r1 <= r3 goto L1
```

จากโค้ดจะเห็นเริ่มต้นจะมีกระโดดไปย L2 ก่อนเพื่อตรวจสอบว่าค่า r1 ยังอยู่ในขอบเขตหรือไม่ ถ้าอยู่จึงกระโดดไปทำที่ L2 โดยในคำสั่งท้ายสุดจะเพิ่มค่า r1 เท่ากับจำนวน step ซึ่งถูกเก็บไว้ใน r2 ด้วย แล้วจึงเริ่มตรวจสอบว่ารอบถัดไปยังอยู่ในขอบเขตหรือไม่ แต่ถ้าไม่อยู่ในขอบเขตแล้วก็จะจบการทำงานลง

3.7.2 Logically Controlled Loop

ลักษณะการควบคุมการวนซ้ำด้วยค่าสถานะบูเลียนหนึ่งอาจจะเป็นค่าของตัวแปรหรืออยู่ในรูปแบบนิพจน์เปรียบเทียบก็ได้ โดยจะมีอยู่หลายรูปแบบ ได้แก่

3.7.2.1 Pre-test

จะตรวจสอบค่าสถานะควบคุมการวนซ้ำก่อนที่จะประมวลผลบล็อกที่วนซ้ำ ซึ่งบล็อกนั้นมีโอกาสที่จะไม่ถูกประมวลผลได้ คำสั่งจะมีรูปแบบดังนี้

```
while condition do statement
```

3.7.2.2 Post-test

จะตรวจสอบค่าสถานะควบคุมการวนซ้ำหลังจากการประมวลผลบล็อกที่วนซ้ำไปแล้ว 1 รอบ คำสั่งจะมีรูปแบบดังนี้

```
1:  //C
2:  do {
3:          line = read_line(stdin);
4:  } while line[0] != '$';
```

3.7.2.3 Mid-test

เป็นรูปแบบที่มีข้อความสั่งพิเศษซึ่งถูกเขียนไว้ภายในบล็อกที่วนซ้ำ เพื่อตรวจสอบสถานะเพื่อควบคุมการจบการทำงานของการวนซ้ำ ตัวอย่างโค้ดเป็นดังนี้

```
1:  //C
2:  for (;;) {
3:          line = read_line(stdin);
4:          if (all_blanks(line)) break;
5:          consume_line(line);
6:  }
```

3.7.3 Iterator

เป็นการวนซ้ำใน collection ของข้อมูล โดยลักษณะของ iterator จะมี 2 แบบ คือ true iterator และเป็นอ็อบเจกต์แยกต่างหาก

โดย true iterator หรือ iterator แท้ ก็คือตัว collection นั้นเป็น iterator ด้วยสามารถประมวลผลวนซ้ำแบบ iteration ได้โดยตรง ซึ่งตัวอย่างการใช้ iterator แบบนี้ คือ ภาษา Python โดยตัวอย่างแรกเป็นดังนี้

```
1:  #Python
2:  #Iterator goes unseen as it is implicitly used
3:  for i in [1, 2, 3]:
4:      print (i)
```

เมื่อนำไปกระทำการผลลัพธ์จะพิมพ์ออกทางหน้าจอจะได้เป็น

```
1
2
3
```

จากตัวอย่างแสดงให้เห็นว่า array ในภาษา Python จะเป็น iterator ด้วย ดังนั้นการทำคำสั่งวนซ้ำใน iterator สามารถทำได้กับ array โดยตรงด้วยคำสั่ง in ในบรรทัดที่ 3 ซึ่งจะเป็นการวนพิมพ์ค่าใน array ออกมาทีละตัวจึงได้ผลลัพธ์เป็น 1 2 และ 3 ออกทางหน้าจอตามลำดับ

ตัวอย่างที่ 2 เป็นดังนี้

```
1:  #Python
2:  #range (first, last, step) is a built-in iterator.
3:  #It yields integers in the range in increments of step, but not including
last.
4:  #It is a function but, when called each time, continues where it last left
off, giving next integer.
```

```
5:  my_list = [‘one’, ‘two’, ‘three’, ‘four’, ‘five’]
6:  my_list_len = len (my_list)
7:  for i in range (0, my_list_len, 2):
8:        print (my_list[i])
```

เมื่อนำไปกระทำการผลลัพธ์จะพิมพ์ออกทางหน้าจอจะได้เป็น

```
one
three
five
```

จากตัวอย่างจะเห็นว่าลูป for ทำการวนผ่านผลลัพธ์จากฟังก์ชัน range(first, last, step) ซึ่งเป็นคำสั่งสร้าง iterator ที่มีสมาชิกตั้งแต่ first และเพิ่มไปเท่ากับขนาด step ไปจนกระทั่งก่อน last (ไม่รวม last) ดังนั้น จากข้อความสั่งในบรรทัดที่ 7 จะได้ iteration ที่เป็น collection ของค่า 0, 2 และ 4 จึงได้พิมพ์ผลลัพธ์ออกมากเป็น one, three และ five ตามลำดับ (array ใน python มี index เริ่มต้นที่ 0)

สำหรับในบางภาษาโปรแกรม collection ไม่ได้เป็น iterator ในตัวในการทำงานจำเป็นต้องสร้างเป็น object แยกออกมาต่างหาก ตัวอย่างเช่น ภาษา Java ดังนี้

```
1: //Java
2: ArrayList al = new ArrayList();
3: //add elements to the array list
4: al.add("C");
5: al.add("A");
6: al.add("E");
7: //use iterator to display contents of al
8: System.out.print("Contents of al: ");
9: Iterator itr = al.iterator();
10:while(itr.hasNext()) {
11:       Object element = itr.next();
12:       System.out.print(element + " ");
13:}
```

เมื่อนำไปกระทำการผลลัพธ์จะพิมพ์ออกทางหน้าจอจะได้เป็น

```
Contents of al: C A E
```

จากตัวอย่างบรรทัดที่ 9 มีการสร้าง Iterator สำหรับ ArrayList ขึ้นมาโดยจะถูกนำมาใช้วนซ้ำเพื่อพิมพ์ออกทางหน้าจอ

*** TA Note: สำหรับภาษา Java ตั้งแต่เวอร์ชัน 5 ได้มีการฝังคุณสมบัติ iterator ให้กับ object ชนิด Collection โดยใช่ร่วมกับคำสั่ง foreach ซึ่งเป็นคำสั่งวนซ้ำกับ object ที่มีคุณสมบัติ Iterable โดยจากตัวอย่างสามารถเขียนใหม่ได้ดังนี้

```
1: //Java 5 or later
2: ArrayList<String> al = new ArrayList<String>();
3: //add elements to the array list
4: al.add("C");
```

```
5: al.add("A");
6: al.add("E");
7: System.out.print("Contents of al: ");
8: for (String element : al){
9:        System.out.print(element + " ");
10:}
```

3.7.4 Recursion and Iteration

Recursion เป็นการเขียนโปรแกรมแบบที่มีการเรียกฟังก์ชันตัวเองซ้อนกัน ไม่ว่าจะเป็นแบบเรียกใช้ตัวเองโดยตรงหรือเรียกไปฟังก์ชันอื่นก่อน แล้วฟังก์ชันอื่นเรียกกลับมาอีกที นั่นคือ สถานะใน Stack จะมีเฟรมของฟังก์ชันซ้ำกันอยู่การกว่า 1 เฟรม

โดยอัลกอริทึมแบบ iteration ใด ๆ สามารถเขียนกลับไปมาระหว่างอัลกอริทึมแบบ Recursion ได้ ตัวอย่างเช่นการเขียนโปรแกรมเพื่อหาค่าหารร่วมมาก โดยมีอัลกอริทึมดังนี้

ให้ gcd(a, b) เป็นฟังก์ชันหา ห.ร.ม. ระหว่างจำนวนเต็มบวก a และ b

โดยกรณี a = b ให้คำตอบเป็น a กรณี a > b ให้คำตอบเป็น gcd(a-b, b) และกรณี b > a ให้คำตอบเป็น gcd(a, b-a)

$$\underset{\text{positive intergers, a, b}}{\gcd(\text{a, b})} = \begin{cases} a, & a = b \\ \gcd(a-b, b), & a > b \\ \gcd(a, b-a), & b > a \end{cases}$$

ซึ่งจะสามารถโปรแกรมแบบ iteration ได้ดังนี้

```
1: //C
2: //Iteration, assume a, b > 0
3: int gcd(int a, int b) {
4:        while (a != b) {
5:               if (a > b) a = a-b;
6:               else b = b-a;
7:        }
8:        return a;
9: }
```

และสามารถเขียนแบบ Recursion ได้ดังนี้

```
1: //C
2: //Recursion, assume a, b > 0
3: int gcd(int a, int b) {
4:        if (a == b) return a;   //base case
5:        else if (a > b) return gcd (a-b, b);
6:        else return gcd(a, b-a);
7: }
```

จากโค้ดตัวอย่างทั้ง 2 แบบข้างต้นเมื่อนำไปกระทำการให้ผลลัพธ์เดียวกัน แต่ความแตกต่างจะอยู่ที่การใช้ทรัพยากรของระบบ เมื่อพิจารณาการเขียนโค้ดแบบ iteration จะเห็นว่าการตรวจสอบจะกระทำกับตัวแปร 2 ตัวจนกว่าสถานะจนพบผลลัพธ์ ซึ่งเป็นการกระโดดไปมาภายในโค้ดและเปลี่ยนค่าตัวแปรในหน่วยความจำที่ได้จองไว้ แต่สำหรับ recursion นั้นจะมีการเรียกฟังก์ชันย่อยซ้อนกันมีผลต่อหน่วยความจำส่วน Stack ที่ต้องรับภาระในการจัดเก็บข้อมูลสถานะของการประมวลผล ซึ่งถือเป็น Overhead เมื่อเทียบกับแบบ iteration แต่การเลือกใช้งานก็อาจขึ้นอยู่กับประเภทของภาษาด้วย เช่น แบบ iteration จะง่ายต่อการเขียนในภาษาเชิงคำสั่ง (imperative language) ในขณะที่แบบ recursion จะง่ายในการเขียนด้วยภาษาเชิงฟังก์ชัน (functional language) เป็นต้น