บทที่ 3

==========

ความหมายของภาษาการทำโปรแกรม
(Semantics of programming languages)

วัตถุประสงค์

1) เพื่อให้ผู้เรียนเข้าใจถึงความสำคัญของการอธิบายความหมายของภาษาการทำโปรแกรมด้วยวิธีการอย่างเป็นทางการ
2) เพื่อให้ผู้เรียนเข้าใจการอธิบายความหมายของภาษาด้วยไวยากรณ์เชิงลักษณะประจำ
3) เพื่อให้ผู้เรียนเข้าใจการอธิบายความหมายเชิงดำเนินการและสามารถเขียนคำอธิบายความหมายของคำสั่งอย่างง่ายได้
4) เพื่อให้ผู้เรียนเข้าใจการอธิบายความหมายเชิงแทนความและสามารถเขียนคำอธิบายความหมายของคำสั่งอย่างง่ายได้
5) เพื่อให้ผู้เรียนเข้าใจการอธิบายความหมายเชิงพิสูจน์และสามารถเขียนคำอธิบายความหมายของคำสั่งอย่างง่ายได้
6) เพื่อให้ผู้เรียนรู้จักวิธีการพิสูจน์ความถูกต้องของโปรแกรมในเชิงคณิตศาสตร์และสามารถพิสูจน์ความถูกต้องของชุดคำสั่งได้

การอธิบายความหมายของข้อความสั่งต่างๆ ในภาษาคอมพิวเตอร์ เป็นการสื่อสารถึงโปรแกรมเมอร์ที่เป็นผู้ใช้ภาษาเหล่านั้นในฐานะเครื่องมือในการทำโปรแกรม นอกจากนี้ยังเป็นการสื่อสารถึงผู้เขียนคอมไพเลอร์และผู้เขียนระบบดำเนินงาน ให้สามารถสร้างระบบประมวลผลภาษาที่ถูกต้องตามที่ผู้ออกแบบภาษากำหนดไว้ การอธิบายความหมายของภาษานิยมใช้วิธีการทางคณิตศาสตร์เพราะเป็นรูปแบบอย่างเป็นทางการ สามารถพิสูจน์ความถูกต้องได้และช่วยให้การสร้างระบบอัตโนมัติเพื่อการแปลภาษาและการประมวลผลสามารถทำได้

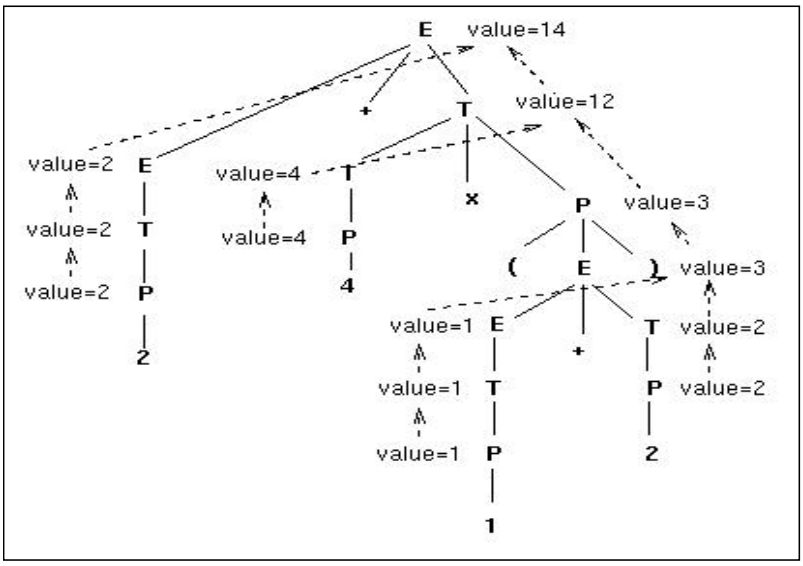
3.1 ไวยากรณ์เชิงลักษณะประจำ

(Attribute grammar)

ความพยายามในการอธิบายความหมายของภาษาการทำโปรแกรม ในยุคแรกเกิดขึ้นจากแนวคิดของ Knuth ในช่วงทศวรรษที่ 1960 โดยการเพิ่มคำอธิบายในลักษณะฟังก์ชันการทำงานเข้าไปในไวยากรณ์บีเอ็นเอฟ และเรียกคำอธิบายนี้ว่า *ลักษณะประจำ* หรือ *แอททริบิวต์* (attribute) ดังตัวอย่างไวยากรณ์นิยามนิพจน์คณิตศาสตร์ต่อไปนี้

โพรดักชัน	แอททริบิวต์
E → E + T	*value* (E_1) = *value* (E_2) + *value* (T)
E → T	*value* (E) = *value* (T)
T → T * P	*value* (T_1) = *value* (T_2) * *value* (P)
T → P	*value* (T) = *value* (P)
P → I	*value* (P) = *value of identifier I*
P → (E)	*value* (P) = *value* (E)

ส่วนแอททริบิวต์ที่เพิ่มขึ้นจะช่วยอธิบายความหมายให้กับแต่ละโพรดักชัน รูปที่ 3.1 แสดงความหมายของการประมวลผลนิพจน์ 2 + 4 * (1 + 2)



รูปที่ 3.1 พาสทรีทีตรวจสอบไวยากรณ์ของนิพจน์และแสดงการคำนวณค่าของนิพจน์

จากจุดเริ่มต้นของการพยายามเพิ่มแอททริบิวต์เข้าไปในไวยากรณ์ของภาษา เพื่ออธิบายหน้าที่การทำงานว่าแต่ละข้อความสั่งของภาษานั้นๆ จะมีขั้นตอนการประมวลผลคำสั่งอย่างไรบ้าง ในระยะหลังได้มีผู้พยายามปรับปรุงและพัฒนารูปแบบการอธิบายความหมายให้เป็นรูปแบบอย่างเป็นทางการมากขึ้น ในปัจจุบันมีวิธีการอธิบายความหมายของข้อความสั่งต่างๆ ในภาษาคอมพิวเตอร์ที่นิยมใช้กันอยู่ 3 วิธีคือ วิธีการอธิบายความหมายเชิงดำเนินการ, วิธีการอธิบายความหมายเชิงแทนความ และวิธีการอธิบายความหมายเชิงพิสูจน์

3.2 การอธิบายความหมายเชิงดำเนินการ
(Operational semantics)

การอธิบายความหมายเชิงดำเนินการของข้อความสั่งต่างๆในโปรแกรม เป็นการอธิบายพฤติกรรมหรือการดำเนินงานของเครื่องเมื่อได้รับคำสั่งนั้นๆ เพียงแต่เครื่องที่ใช้ไม่ใช่เครื่องคอมพิวเตอร์จริง เป็นเครื่องจำลองหรือเครื่องจักรเสมือน วิธีการแบบนี้ถูกนำไปใช้ในการอธิบายความหมายของคำสั่งต่างๆ ในภาษา LISP และเครื่องจำลองที่ใช้เรียกว่า SECD machine

การอธิบายความหมายเชิงดำเนินการ จะเป็นการบรรยายพฤติกรรมการทำงานของเครื่องด้วยการระบุสถานะว่าเมื่อทำแต่ละข้อความสั่งแล้วสถานะของเครื่องเปลี่ยนไปอย่างไร การพิจารณาสถานะของเครื่องจะเป็นการดูสิ่งที่เก็บอยู่ในหน่วยความจำ การอธิบายการเปลี่ยนสถานะจะใช้รูปแบบ

$$\sigma\ (e)\ \Rightarrow\ v$$

เมื่อ σ คือ สถานะหรือสิ่งที่เก็บอยู่ในหน่วยความจำ เช่น σ = {<x,5>, <y,1>, <z,0>} เป็นการอธิบายว่า สถานะปัจจุบันคือหน่วยความจำตำแหน่ง x เก็บค่า 5 ตำแหน่ง y เก็บค่า 1 และตำแหน่ง z เก็บค่า 0

e คือ นิพจน์

v คือ ค่าที่ได้จากการประมวลผลนิพจน์ e เช่น

$$\sigma\ (x)\ \Rightarrow\ 5$$
$$\sigma\ (1)\ \Rightarrow\ 1$$

<u>ความหมายของนิพจน์</u>

การอธิบายความหมายนอกจากอธิบายการเปลี่ยนสถานะด้วยรูปแบบ σ (e) => v แล้ว ยังต้องใช้การอธิบายที่ซับซ้อนขึ้นเรียกว่า กฎกระทำการ (execution rule) ที่ใช้รูปแบบ $\frac{\text{premise}}{\text{conclusion}}$ หรือ $\frac{\text{สิ่งที่เกิดขึ้นก่อน}}{\text{บทสรุปที่ตามมา}}$ เช่น กฎกระทำการที่เกี่ยวกับการบวก

$$\boxed{\frac{\sigma\ (e_1) \Rightarrow v_1 \qquad \sigma\ (e_2) \Rightarrow v_2}{\sigma\ (e_1+e_2) \Rightarrow v_1 + v_2}}$$

เป็นการอธิบายว่า กรณีนิพจน์ย่อยสองนิพจน์ (e_1 และ e_2) บวกกัน การทำงานของเครื่องจะเป็นการหาค่าของ e_1 (ได้เป็นค่า v_1) และหาค่าของ e_2 (ได้ v_2) จากนั้นนำค่า $v_1 + v_2$ ก็จะได้เป็นค่าของนิพจน์ $e_1 + e_2$

<u>ความหมายของข้อความสั่งกำหนดค่า</u>

การอธิบายความหมายของข้อความสั่งกำหนดค่า (assignment statement) ที่เขียนอยู่ในรูปแบบ *target = source;* จะอธิบายด้วยกฎกระทำการ ต่อไปนี้

$$\frac{\sigma\,(\text{source}) \Rightarrow v}{\sigma\,(\textit{target} = \textit{source;}) \Rightarrow \sigma \cup \{\langle \text{target}, v\rangle\}}$$

ซึ่งมีความหมายว่า สิ่งที่ต้องเกิดขึ้นก่อน คือ การหาค่า v ของนิพจน์หรือข้อความ source แล้วสิ่งที่จะเป็นบทสรุปตามมาคือ ค่าของตัวแปร (target) จะถูกเปลี่ยนค่าเป็น v ในขณะที่ค่าในตำแหน่งอื่นๆ ของหน่วยความจำยังคงเป็นเช่นเดิม

สัญลักษณ์ $\cup$ ใช้แทนความหมาย overriding union หรือการยูเนียนที่มีการเขียนค่าทับ ตัวอย่างเช่น ถ้า σ_1 คือสถานะดั้งเดิม และ σ_2 คือสถานะส่วนที่เปลี่ยนแปลงไป

$$\sigma_1 = \{\langle x,1\rangle, \langle y,2\rangle, \langle z,3\rangle\}$$
$$\sigma_2 = \{\langle y,9\rangle, \langle w,4\rangle\}$$

เมื่อรวมส่วนที่เปลี่ยนแปลงเข้ากับสถานะดั้งเดิม จะได้สถานะใหม่คือ

$$\sigma_1 \cup \sigma_2 = \{\langle x,1\rangle, \langle y,9\rangle, \langle z,3\rangle, \langle w,4\rangle\}$$

ดังนั้นการเขียนข้อความสั่ง *x = 5; x = x+1;* จึงอธิบายความหมายเชิงดำเนินการได้ดังนี้ (สถานะเริ่มต้น $\{\langle x,\text{undef}\rangle\}$ หมายถึงตัวแปร x ยังไม่ได้ถูกกำหนดค่า)

สถานะ	ความหมายเชิงดำเนินการของข้อความสั่ง
σ = {<x, undef>}	
	$\dfrac{\sigma\,(5) \Rightarrow 5}{\sigma\,(x = 5;) \Rightarrow \{\langle x,\text{undef}\rangle\} \cup \{\langle x,5\rangle\}}$
σ = {< x, 5>}	
	$\dfrac{\dfrac{\sigma\,(5) \Rightarrow 5 \quad \sigma\,(1) \Rightarrow 1}{\sigma\,(x + 1) \Rightarrow 6}}{\sigma\,(x = x+1;) \Rightarrow \{\langle x,5\rangle\} \cup \{\langle x,6\rangle\}}$
σ = {< x, 6>}	

การมีข้อความสั่งกำหนดค่าดำเนินการติดต่อกันหลายคำสั่งเช่น S_1; S_2; S_3; อาจรวมอธิบายความหมายในคราวเดียวกันได้ด้วยรูปแบบ

$$\frac{\sigma\ (S_1;) \Rightarrow \sigma_1 \quad \sigma_1\ (S_2;) \Rightarrow \sigma_2 \quad \sigma_2\ (S;) \Rightarrow \sigma_3}{\sigma\ (S_1;\ S_2;\ S_3;) \Rightarrow \sigma_3}$$

ดังนั้นข้อความสั่ง x = 5; x = x+1; อาจอธิบายความหมายด้วยขั้นตอนที่สั้นลงได้ดังนี้

สถานะ	ความหมายเชิงดำเนินการของข้อความสั่ง
σ = {<x, undef>}	σ (x = 5;) => σ_1 {<x, 5>} σ_1 (x = x+1;) => σ_2 {<x, 6>} σ (x = 5; x = x+1;) => σ_2 {<x, 6>}
σ = {< x, 6>}	

<u>ความหมายของข้อความสั่งแบบมีเงื่อนไข</u>

ในกรณีของข้อความสั่งแบบมีเงื่อนไข

if (test) thenStatement else elseStatement;

การอธิบายความหมายด้วยกฎกระทำการ จะแยกเป็นสองกรณี คือ กรณี test ให้ผลลัพธ์เป็นจริงแล้วกระทำคำสั่งในส่วน thenStatement และกรณี test ให้ผลลัพธ์เป็นเท็จแล้วกระทำคำสั่งในส่วน elseStatement

กรณี test เป็นจริง:

$$\frac{\sigma\ (\text{test}) \Rightarrow \text{true} \quad \sigma\ (\text{thenStatement}) \Rightarrow \sigma_1}{\sigma\ (\textit{if (test) thenStatement else elseStatement;}) \Rightarrow \sigma_1}$$

กรณี test เป็นเท็จ:

$$\frac{\sigma\ (\text{test}) \Rightarrow \text{false} \quad \sigma\ (\text{elseStatement}) \Rightarrow \sigma_1}{\sigma\ (\textit{if (test) thenStatement else elseStatement;}) \Rightarrow \sigma_1}$$

ตัวอย่างเช่น ข้อความสั่ง

```
if  (x < 0) x = x-1;  else   x = x+1;
```

อธิบายความหมายเชิงดำเนินการได้ดังนี้

สถานะ	ความหมายเชิงดำเนินการ
σ = {<x, 6>}	
	σ (x<0) => false σ (x=x+1;) =>{<x,7>} σ (if (x<0) x = x-1; else x = x+1;) => {<x,7>}
σ = {< x, 7>}	

<u>ความหมายของข้อความสั่งวนรอบ</u>

ในกรณีของข้อความสั่งวนรอบ หรือ ลูป (loop)

while (test) body;

การอธิบายความหมายจะแยกเป็นสองกรณีเช่นเดียวกับคำสั่ง if คือ กรณีที่ test ให้ผลลัพธ์เป็นจริง แล้วมีการกระทำคำสั่งในส่วน body และกรณี test ให้ผลลัพธ์ เป็นเท็จ แล้วไม่มีการกระทำคำสั่งใดในลูป

กรณี test เป็นจริง:

$$\frac{\sigma\,(\text{test}) \Rightarrow \text{true} \quad \sigma\,(\text{body;}) \Rightarrow \sigma_1 \quad \sigma_1\,(\text{while (test) body;}) \Rightarrow \sigma_2}{\sigma\,(\textit{while (test) body;}) \Rightarrow \sigma_2}$$

กรณี test เป็นเท็จ:

$$\frac{\sigma\,(\text{test}) \Rightarrow \text{false}}{\sigma\,(\textit{while (test) body;}) \Rightarrow \sigma}$$

ตัวอย่างข้อความสั่งวนรอบ

```
while (x<50)
      x = 2*x;
```

สามารถอธิบายความหมายเชิงดำเนินการได้ดังนี้

สถานะ	ความหมายเชิงดำเนินการ
$\sigma = \{<x,7>\}$	
	$\dfrac{\sigma\,(x<50) \Rightarrow \text{true} \quad \sigma\,(x=2*x;) \Rightarrow \sigma_1 = \{<x,14>\} \quad \sigma_1\,(\text{while } (x<50)\ x=2*x;) \Rightarrow \sigma_2}{\sigma\,(\text{while } (x<50)\ x=2*x;) \Rightarrow \sigma_2}$
$\sigma = \{<x,14>\}$	
	$\underline{\sigma\,(x<50) \Rightarrow \text{true} \quad \sigma\,(x=2*x;) \Rightarrow \sigma_1 = \{<x,28>\} \quad \sigma_1\,(\text{while } (x<50)\ x=2*x;) \Rightarrow \sigma_2}$ $\sigma\,(\textit{while } (x<50)\ x=2*x;) \Rightarrow \sigma_2$
$\sigma = \{<x,28>\}$	
	$\underline{\sigma\,(x<50) \Rightarrow \text{true} \quad \sigma\,(x=2*x;) \Rightarrow \sigma_1 = \{<x,56>\} \quad \sigma_1\,(\text{while } (x<50)\ x=2*x;) \Rightarrow \sigma_2}$ $\sigma\,(\textit{while } (x<50)\ x=2*x;) \Rightarrow \sigma_2$
$\sigma = \{<x,56>\}$	
	$\underline{\sigma\,(x<50) \Rightarrow \text{false}}$ $\sigma\,(\textit{while } (x<50)\ x=2*x;) \Rightarrow \sigma$
$\sigma = \{<x,56>\}$	

3.3 การอธิบายความหมายเชิงแทนความ

(Denotational semantics)

การอธิบายความหมายของข้อความสั่งต่างๆ ในโปรแกรมด้วยวิธีการอธิบายเชิงแทนความ เป็นการใช้ฟังก์ชันอธิบายคำสั่งว่ามีผลให้สถานะเปลี่ยนไปเป็นสถานะใด การอธิบายในรูปแบบนี้ยังคงคล้ายกับในแบบการอธิบายเชิงดำเนินการ ที่ใช้สถานะเป็นสิ่งบอกความหมายของคำสั่งเพียงแต่รูปแบบการอธิบายเปลี่ยนจากการใช้ premise/conclusion เป็นการอธิบายในลักษณะฟังก์ชันที่เชื่อมโยงค่าจากในเซตของโดเมนไปสู่ค่าในเซตของเรนจ์ ด้วยรูปแบบดังนี้

$$M: \text{Statement} \times \sum \rightarrow \sum$$

โดยที่ M หมายถึง meaning function หรือฟังก์ชันความหมาย ทำหน้าที่เชื่อมโยงจากสถานะหนึ่งไปสู่อีกสถานะหนึ่ง ตามความหมายของ Statement

$\sum$ หมายถึง เซ็ตของสถานะทั้งหมด โดยที่สถานะจะเขียนอยู่ในรูปของคู่ลำดับของตัวแปรและค่าปัจจุบันของตัวแปรนั้น

ถ้าไวยากรณ์ของภาษาระบุรูปแบบการเขียนโปรแกรมและรูปแบบการเขียนข้อความสั่งต่างๆ ด้วยรูปแบบไวยากรณ์ EBNF ต่อไปนี้

```
Program          →  void main ( ) '{' Declarations Statements '}'
Declarations     →  {Declaration}*
Declaration      →  Type Identifiers;
Type             →  int | boolean
Identifiers      →  Identifier {, Identifier}*
Statements       →  {Statement}*
Statement        →  ; | Block | Assignment | IfStatement | WhileStatement
Block            →  '{' Statements '}'
Assignment       →  Identifier = Expression;
IfStatement      →  if  (Expression) Statement [ else Statement ]
WhileStatement   →  while (Expression) Statement
Expression       →  Conjunction { || Conjunction }*
Conjunction      →  Relation { && Relation }*
Relation         →  Addition { [< | <= | > | >= | = = | != | ] Addition}*
Addition         →  Term { [ + | - ] Term}*
Term             →  Negation { [ '*' | / ] Negation}*
Negation         →  [ ! ] Factor
Factor           →  Identifier | Literal | (Expression)
Identifier       →  Letter { [Letter | Digit]}*
Literal          →  true | false | Digit {Digit}*
Letter           →  a | b | … | z | A | B | … | Z
Digit            →  0 | 1 | … | 9
```

ความหมายของทั้งโปรแกรมและความหมายของข้อความสั่งประเภทต่างๆ สามารถอธิบายในเชิงแทนความด้วยฟังก์ชันความหมาย (M) ดังต่อไปนี้

<u>ความหมายของโปรแกรม</u>

$$M: Program \rightarrow \sum$$
$$M(Program\ p) = M(p.body, \{<v_1,undef>, <v_2,undef>,..., <v_n,undef>\})$$

ข้อความบรรทัดแรกเป็นการประกาศโพรโทไทพ์ (prototype) หรือโครงสร้างของฟังก์ชันซึ่งในทางคณิตศาสตร์จะเรียกว่าเป็นการกำหนดลายเซ็น (signature) ของฟังก์ชัน บรรทัดที่สองเป็นการอธิบายความหมายของโปรแกรม ซึ่งระบุว่าความหมายของโปรแกรมก็คือ ความหมายของส่วน body (ตามไวยากรณ์ข้างต้นโปรแกรมจะเริ่มต้นด้วยส่วนประกาศตัวแปร แล้วจึงตามด้วยส่วนชุดคำสั่งที่เรียกว่า body ของโปรแกรม) ที่มีสถานะเริ่มต้นเป็น $\{<v_1,undef>, <v_2,undef>,..., <v_n,undef>\}$ โดย $v_1, v_2,..., v_n$.เป็นตัวแปรที่ได้ประกาศไว้ในส่วนประกาศตัวแปร เมื่อส่วน body ถูกขยายความไปตามลำดับว่าประกอบด้วยคำสั่งใดบ้าง ในที่สุดเราจะทราบสถานะสุดท้ายซึ่งจะทำให้เราตีความได้ว่าโปรแกรมนี้ทำงานจนจบแล้วได้ค่าตัวแปรต่างๆ เป็นค่าใดบ้าง และนั่นจะเป็นการบอกความหมายของโปรแกรม

<u>ความหมายของข้อความสั่งกำหนดค่า</u>

$$M: Assignment \times \sum \rightarrow \sum$$
$$M: (Assignment\ a, State\ \sigma) = \sigma\ \bar{U}\{<a.target>, M(a.source, \sigma)>\}$$

ข้อความสั่งกำหนดค่า หรือ Assignment(a) คือข้อความที่เขียนอยู่ในรูปแบบ

a.target = a.source;

เช่น
```
p = 1;
q = p * 2;
```

ดังนั้นการพิจารณาความหมายของคำสั่งว่า เมื่อทำงานเสร็จสิ้นแล้วจะได้ผลลัพธ์เป็นอย่างไรจึงต้องหาค่าของส่วน *a.source* (นั่นคือนิพจน์ *p*2*) ให้ได้ก่อน เมื่อทราบค่าแล้วเราจะรู้ว่าขั้นตอนต่อไปก็คือ การเปลี่ยนค่า *a.target* (นั่นคือตัวแปร *q*) ให้มีค่าตามค่าของ a.source (คือค่า 2)

ตัวอย่างเช่น ถ้าสถานะเริ่มต้น σ = {<x,2>, <y,-3>, <z,75>} ความหมายข้อความสั่ง *z = x+2*y;* อธิบายได้ดังนี้

ขั้นตอนแรกหาค่าของ a.source

$$M(a.source, \sigma) = M(x+2*y, \{<x,2>, <y,-3>, <z,75>\})$$
$$= -4$$

ขั้นตอนต่อไปกำหนดค่าให้ a.target

$$M(z = x+2*y, \{<x,2>, <y,-3>, <z,75>\}) = \{<x,2>, <y,-3>, <z,75>\}\ \bar{U}\ \{<z,-4>\}$$
$$= \{<x,2>, <y,-3>, <z,-4>\}$$

<u>ความหมายของนิพจน์</u>

จากตัวอย่างก่อนหน้านี้มีการคำนวณค่าของนิพจน์ *x+2*y* เพื่อจะหาความหมายของคำสั่ง *z = x+2*y;* ตามไวยากรณ์ในภาษาตัวอย่างนอกจากนิพจน์คณิตศาสตร์ที่มีการคำนวณ +,-,*,/ แล้วยังมีนิพจน์เปรียบเทียบและนิพจน์ตรรกะ ซึ่งอธิบายการหาค่าของนิพจน์ได้ด้วยฟังก์ชัน ApplyBinary และฟังก์ชัน ApplyUnary

ApplyBinary: Operator × Value × Value → Value
ApplyBinary (Operator op, Value v1, Value v2)

= v1+v2	ถ้า op คือเครื่องหมาย +
= v1-v2	ถ้า op คือเครื่องหมาย –
= v1*v2	ถ้า op คือเครื่องหมาย *
= floor(|v1/v2|)*sign(v1*v2)	ถ้า op คือเครื่องหมาย /
= v1 < v2	ถ้า op คือเครื่องหมาย <
= v1 ≤ v2	ถ้า op คือเครื่องหมาย ≤
= v1 = v2	ถ้า op คือเครื่องหมาย = =
= v1 ≠ v2	ถ้า op คือเครื่องหมาย ! =
= v1 ≥ v2	ถ้า op คือเครื่องหมาย >=
= v1 > v2	ถ้า op คือเครื่องหมาย >
= v1 ∧ v2	ถ้า op คือเครื่องหมาย &&
= v1 ∨ v2	ถ้า op คือเครื่องหมาย ||

ApplyUnary : Operator × Value → Value
ApplyUnary (Operator op, Value v) = ¬ v ถ้า op คือเครื่องหมาย !

ฟังก์ชัน ApplyBinary และ ApplyUnary จะถูกเรียกใช้เพื่อคำนวณหาค่าของนิพจน์ ในกรณีที่นิพจน์ไม่ใช่ค่าคงที่หรือไม่ใช่ตัวแปรเดี่ยว แต่เป็นนิพจน์ที่ประกอบขึ้นจากนิพจน์ย่อยๆ มากระทำกันด้วยตัวกระทำการทางคณิตศาสตร์ (เช่น +, -) หรือตัวกระทำการทางการเปรียบเทียบ (เช่น <, >) หรือทางตรรกะ (เช่น &&, ||)

ฟังก์ชันที่ใช้อธิบายความหมายของนิพจน์ แสดงได้ดังนี้

M: Expression × ∑ → Value
M (Expression e, State σ)

= e	ถ้า e เป็นค่าคงที่
= σ(e)	ถ้า e เป็นตัวแปร
= ApplyBinary (e.op, M(e.term1, σ), M(e.term2, σ))	ถ้า e เป็นนิพจน์ที่มีตัวถูกดำเนินการคู่
= ApplyUnary (e.op, M(e.term, σ)	ถ้า e เป็นนิพจน์ที่มีตัวถูกดำเนินการเดี่ยว

จากตัวอย่างข้อความสั่ง *z = x+2*y;* ที่มีสถานะเริ่มต้นเป็น σ = {<x,2>, <y,–3>, <z,75>} สามารถอธิบายขั้นตอนการคำนวณนิพจน์ *x+2*y* โดยละเอียดได้ดังนี้

M(x+2*y, {<x,2>, <y,–3>, <z,75>}) = ApplyBinary(+, A, B)

โดยที่ A = M(x, {<x,2>, <y,–3>, <z,75>}) = 2

และ B = M(2*y, {<x,2>, <y,–3>, <z,75>})

= ApplyBinary(*, C, D)

โดยที่ C = M(2, {<x,2>, <y,–3>, <z,75>}) = 2

และ D = M(y, {<x,2>, <y,–3>, <z,75>}) = –3

= –6

= –4

<u>*ความหมายของข้อความสั่ง*</u>

ข้อความสั่งตามไวยากรณ์ของภาษาตัวอย่างประกอบด้วย

Statement → ; | Block | Assignment | IfStatement | WhileStatement

นั่นคือข้อความสั่งประกอบด้วย 5 ประเภท

(1) ข้อความสั่งที่ไม่ต้องมีการกระทำใด เรียกว่าเป็นคำสั่ง skip

(2) ข้อความสั่งที่เป็นกลุ่มหรือ บล็อก คือชุดคำสั่งที่เขียนอยู่ภายในวงเล็บปีกกา {...}

(3) ข้อความสั่งกำหนดค่า

(4) ข้อความสั่งแบบมีเงื่อนไข

(5) ข้อความสั่งวนรอบ

ความหมายของข้อความสั่งประเภทต่างๆ อธิบายได้ดังนี้

M:Statement x ∑→∑

M(Statement s, State σ) = M((Skip)s, σ)	ถ้า s คือคำสั่ง skip หรือ ;
= M ((Assignment) s, σ)	ถ้า s คือคำสั่งกำหนดค่า
= M ((Conditional) s, σ)	ถ้า s คือคำสั่งแบบมีเงื่อนไข
= M ((Loop) s, σ)	ถ้า s คือคำสั่งแบบวนรอบ
= M ((Block) s, σ)	ถ้า s คือคำสั่งเป็นกลุ่ม

M(Skip s, State σ) = σ

M (Assignment a, State σ) = σ Ū {<a. target, M(a.source, σ)>}

M (Conditional c, State σ) = M(c.thenbranch, σ)	ถ้า M(c.test, σ) เป็นจริง
= M(c.elsebranch, σ)	ถ้า M(c.test, σ) เป็นเท็จ

M(Loop l, State σ)	= M (l, M(l.body, σ))	ถ้า M(l.test, σ) เป็นจริง
	= σ	ถ้า M(l.test, σ) เป็นเท็จ
M(Block b, State σ)	= σ	ถ้า b เป็นบล็อกว่าง
	= M((Block) $b_{2\ldots n}$, M((Statement) b_1,σ))	ถ้า $b=b_1b_2\ldots b_n$

3.4 การอธิบายความหมายเชิงพิสูจน์
(Axiomatic semantics)

การอธิบายความหมายของโปรแกรมและข้อความสั่งต่างๆ ในโปรแกรมด้วยวิธีการเชิงพิสูจน์จะใช้วิธีการให้เหตุผลเกี่ยวกับกระบวนการทำงานของคำสั่งและโปรแกรม เครื่องมือสำคัญที่ใช้ประกอบการให้เหตุผลคือ *ข้อความยืนยัน* (assertion)

ข้อความยืนยัน เป็นข้อความที่ใช้บอกพฤติกรรมการประมวลผลของเครื่องในลักษณะข้อความเชิงประกาศ (declarative statement) โดยไม่ต้องระบุสถานะโดยละเอียด ว่าค่าของตัวแปรต่างๆ มีค่าอะไรบ้าง ตัวอย่างเช่น ชุดของข้อความสั่งต่อไปนี้เป็นการคำนวณหาค่าที่มากกว่าระหว่างตัวเลขสองจำนวน

```
int Max (int a, int b) {
        int m ;
        if (a>=b)
                m=a ;
        else
                m=b ;
        return m ;
}
```

ข้อความยืนยันที่ใช้อธิบายความหมายของชุดคำสั่งนี้คือ

```
{m = max (a,b) }
```

นั่นคือ เมื่อฟังก์ชัน Max() ข้างต้นทำงานเสร็จ ผลลัพธ์ของการทำงาน (หรือ พฤติกรรมของโปรแกรม) คือ ค่าของตัวแปร m จะต้องเป็นค่ามากจากการเปรียบเทียบค่าของตัวแปร a และ b

การอธิบายความหมายในเชิงการพิสูจน์ของคำสั่งและของทั้งโปรแกรม จะใช้ข้อความยืนยันสภาพหรือพฤติกรรมก่อนหน้าการทำคำสั่ง และใช้ข้อความยืนยันพฤติกรรมหลังจากทำคำสั่งแล้ว ข้อความยืนยันก่อนหน้าจะเรียกว่า *เงื่อนไขก่อน* (precondition) ข้อความยืนยันภายหลังคำสั่งจะเรียกว่า *เงื่อนไขหลัง* (postcondition) ตัวอย่างฟังก์ชัน Max() ข้างต้นสามารถเขียนอธิบายความหมายได้ด้วยเงื่อนไขก่อนและเงื่อนไขหลัง ดังนี้

```
{ true }
        int Max ( int a, int b) {
                int m ;
                if (a >= b) m=a ;
                        else m=b ;
                return m ;
                }
{ m = max(a,b)}
```

เงื่อนไขก่อนที่เป็นข้อความ { true } หมายถึง ไม่มีการกำหนดเงื่อนไขใดเป็นพิเศษก่อนหน้าการทำงานของฟังก์ชัน Max()

การอธิบายความหมายของคำสั่งด้วยการระบุเงื่อนไขก่อนและเงื่อนไขหลังทำคำสั่ง นิยมเขียนอยู่ในรูปแบบ

{P} *S* {Q}

เมื่อ P คือ เงื่อนไขก่อน Q คือเงื่อนไขหลัง และ S คือคำสั่ง หรือ Statement การอธิบายความหมายของแต่ละคำสั่งด้วยโครงสร้างสามส่วนนี้ เรียกว่า Hoare triple ซึ่งเป็นการตั้งตามชื่อของ C.A.R. Hoare ที่เป็นผู้เสนอแนวคิดให้ใช้ข้อความยืนยันเพื่ออธิบายความหมายของโปรแกรม การเขียนด้วยรูปแบบ {P} S {Q}มีความหมายว่า ขณะเริ่มต้นที่เงื่อนไข P เป็นจริง การประมวลผลคำสั่ง S เสร็จจะมีผลให้เงื่อนไข Q เป็นจริง แต่ทั้งนี้มีข้อแม้ว่าคำสั่ง S จะต้องทำงานแบบมีจุดสิ้นสุด

ข้อความที่เขียนอยู่ในรูปแบบ {P} *S* {Q} จะเรียกว่า *แอ็กเซียม* (axiom) หมายถึงข้อความที่เป็นจริง ตัวอย่างเช่น

{x = 0} *x = x+1;* {x > 0}

เป็นแอ็กเซียมที่ระบุว่า “ณ สถานะเริ่มต้นที่ตัวแปร x มีค่าเป็นศูนย์ เมื่อทำคำสั่ง *x = x+1;* แล้วจะมีผลให้สถานะเปลี่ยนไปสู่สภาพที่ x มีค่ามากกว่า ศูนย์” ดังนั้นแอ็กเซียมนี้เป็นการสื่อความหมายของคำสั่ง *x = x+1;* ว่าเป็นคำสั่งที่เพิ่มค่าให้กับตัวแปร x

ในกรณีที่มีข้อความอื่นเพิ่มขึ้น เช่น *x = x+1; y = x;* เราสามารถใช้แอ็กเซียมเดิมเป็นพื้นฐานอธิบายความหมายของคำสั่ง *y = x;* ได้ดังนี้

{x = 0} *x = x+1;* {x > 0}
{x > 0} *y = x;* {y > 0}

จะสังเกตได้ว่าเงื่อนไขหลังของคำสั่ง *x = x+1;* จะทำหน้าที่เป็นเงื่อนไขก่อนให้กับคำสั่ง *y = x;* ซึ่งเป็นคำสั่งลำดับถัดมา

การให้เหตุผลเกี่ยวกับพฤติกรรมการทำงานของคำสั่งหนึ่งที่มีผลต่อเนื่องไปสู่คำสั่งอื่นๆ ที่อยู่ในลำดับถัดมา จำเป็นต้องอาศัยกฎการอธิบายที่เรียกว่า *กฎการพิสูจน์* (proof rules) ซึ่งแสดงรายละเอียดได้ดังนี้

ชนิดคำสั่ง	รูปแบบคำสั่ง (s)	กฎการพิสูจน์
Assignment	s.target = s.source;	$\frac{\text{true}}{\{Q[s.target\backslash s.source]\}\ S\ \{Q\}}$
Sequence (Block)	S_1S_2 ;	$\frac{\{P\}\ S_1\{R\} \quad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1S_2\ \{Q\}}$
Conditional	if (s.test) s.thenpart; else s.elsepart;	$\frac{\{s.test \wedge P\}\ s.thenpart\ \{Q\} \quad \{\neg\ s.test \wedge P\}\ s.elsepart\ \{Q\}}{\{P\}S\{Q\}}$
Loop	while (s.test) s.body	$\frac{\{\ s.test \wedge P\}\ s.body\ \{P\}}{\{P\}S\{\neg\ s.test \wedge P\}}$
Rule of consequence		$\frac{P \supset P' \quad \{P'\}\ S\ \{Q'\} \quad Q' \supset Q}{\{P\}S\{\ Q\}}$

กฎการพิสูจน์เป็นกฎที่เขียนอยู่ในรูปแบบ $\frac{\text{premise}}{\text{conclusion}}$ มีความหมายว่า ถ้าส่วน premise เป็นจริง ส่วน conclusion จะเป็นจริงด้วย สัญลักษณ์ Q[a\b] ที่ใช้ในกฎการพิสูจน์คำสั่ง Assignment มีความหมายถึง สภาพ หรือ สถานะที่เป็นผลจากการแทนค่า a ลงใน b ตัวอย่างเช่น ถ้า Q คือ { $x=1 \wedge y = 4$} การเขียน Q[1\x] จะหมายถึง { $1=1 \wedge y=4$} ส่วนสัญลักษณ์ $P \supset P'$ และ $Q' \supset Q$ ใน Rule of consequence หมายถึงสัญลักษณ์ implication (=>) หรือสัญลักษณ์ ถ้า…แล้ว ในทางตรรกศาสตร์ ตัวอย่างเช่น

```
(x > 0) ⊃ (x+1 > 0)
```

มีความหมายว่า ถ้า x มีค่ามากกว่าศูนย์แล้ว การเพิ่มค่า x ขึ้นหนึ่งค่าก็ยังคงทำให้ x มีค่ามากกว่าศูนย์

การให้เหตุผลเกี่ยวกับพฤติกรรมการทำงานของคำสั่งและพฤติกรรมของทั้งโปรแกรม นอกจากสามารถใช้อธิบายความหมายของคำสั่งและความหมายของโปรแกรมแล้ว การใช้ประโยชน์อย่างกว้างขวางในอีกด้านหนึ่งคือ ใช้พิสูจน์ความถูกต้องของโปรแกรม

3.5 การตรวจสอบความถูกต้องของโปรแกรม
(Program verification)

ในการพัฒนาโปรแกรมคอมพิวเตอร์ เกณฑ์ที่สำคัญที่สุดในการพิจารณาคุณภาพของโปรแกรมคือ ความถูกต้องของโปรแกรม (program correctness) โปรแกรมที่ถูกต้อง คือ โปรแกรมที่ทำงานได้ตรงตามข้อกำหนดและทำงานได้ถูกต้องกับทุกข้อมูลเข้าที่เป็นไปได้ ซึ่งในทางปฏิบัตินักพัฒนาซอฟต์แวร์มักจะใช้วิธีการสุ่มทดสอบกับข้อมูลเข้าจำนวนหนึ่ง และเมื่อโปรแกรมทำงานได้โดยไม่มีปัญหากับข้อมูลทดสอบ ผู้พัฒนาก็จะส่งมอบโปรแกรมให้กับลูกค้าหรือผู้ใช้

Dijkstra ได้ให้ความเห็นไว้เมื่อปี ค.ศ 1972 เกี่ยวกับวิธีทดสอบความถูกต้องของโปรแกรมแบบนี้ว่าเป็นการพิสูจน์การมีอยู่ของข้อผิดพลาดไม่ใช่เป็นการพิสูจน์ว่าโปรแกรมไม่มีข้อผิดพลาด(only prove the presence of bugs, never their absence) เพราะการพิสูจน์โปรแกรมว่าปราศจากข้อผิดพลาดจะต้องทดสอบกับทุกข้อมูลเข้าที่เป็นไปได้ Dijkstra ได้ยกตัวอย่างว่าแม้แต่โปรแกรมอย่างง่ายที่เป็นการคำนวณกับตัวเลขสองจำนวน แล้วให้ผลลัพธ์ออกมาเป็นจำนวนเลข ถ้าตัวเลขนั้นมีขนาด 32 บิต ตัวเลขสองจำนวนที่สามารถเป็นข้อมูลเข้าจะมีจำนวนมากถึง 2^{64} จำนวน(ประมาณ 10^{20} จำนวน) และถ้าเครื่องคอมพิวเตอร์ที่ใช้ในการทดสอบสามารถทดสอบข้อมูลเข้าได้หนึ่งร้อยล้านจำนวนต่อวินาที การทดสอบกับข้อมูลเข้าทั้งหมดที่เป็นไปได้จะใช้เวลาถึง 10^5 ปี ดังนั้นการพิสูจน์ความถูกต้องของโปรแกรมด้วยการทดลองประมวลผลจริงกับทุกชุดข้อมูลเข้า (input) จึงเป็นเรื่องที่เป็นไปไม่ได้ในทางปฏิบัติเพราะจำนวนข้อมูลเข้าที่เป็นไปได้มีมากเกินไป

การทดสอบความถูกต้องของโปรแกรม ที่ได้รับการยอมรับกันโดยทั่วไปในกลุ่มนักคอมพิวเตอร์ คือ การทดสอบความถูกต้องในเชิงการพิสูจน์ทางคณิตศาสตร์ ด้วยการให้เหตุผลที่สามารถพิสูจน์ได้ว่าแต่ละข้อความสั่งในโปรแกรมทำงานได้ตรงตามวัตถุประสงค์กับทุกข้อมูลเข้า และการให้เหตุผลเช่นที่ว่านี้จะใช่แอ็กเซียมประกอบกับกฎการพิสูจน์เพื่อพิสูจน์ย้อนกลับตั้งแต่เงื่อนไขหลังของโปรแกรม ย้อนขึ้นไปทีละคำสั่งจนกระทั่งถึงต้นโปรแกรมเพื่อยืนยันว่า จากข้อมูลตั้งต้นที่เป็นเงื่อนไขก่อนการทำโปรแกรมเมื่อผ่านกระบวนการทำงานของโปรแกรมจะได้ผลลัพธ์สุดท้ายที่ตรงตามข้อกำหนด (หรือ specification) ของโปรแกรม

การพิสูจน์ความถูกต้องของโปรแกรมด้วยการใช้แอ็กเซียมนี้ เป็นการพิสูจน์ความถูกต้องเพียงบางส่วน (partial correctness) ไม่ใช่เป็นการพิสูจน์ความถูกต้องอย่างสมบูรณ์ (complete correctness) เนื่องจากในการพิสูจน์จะเลือกใช้เพียงบางค่าของตัวแปร ที่จะมีผลให้โปรแกรมทำงานได้จบอย่างสมบูรณ์ ทั้งนี้เนื่องจากค่าของตัวแปรบางค่าที่ไม่ถูกเลือกใช้นั้น อาจส่งผลให้โปรแกรมทำงานอย่างไม่รู้จบ หรือเป็นค่าที่เกินขอบเขตที่โปรแกรมจะสามารถทำงานได้ตามปกติ (เช่น การหาค่าแฟคตอเรียลของ 536)

วิธีการพิสูจน์ความถูกต้องของโปรแกรมโดยการใช้แอ็กเซียม จะเป็นการพิจารณาจากเงื่อนไขหลังร่วมกับการทำงานของข้อความสั่ง เพื่อสร้างเงื่อนไขก่อน เช่น ถ้ามีข้อความสั่งและเงื่อนไขหลังให้ดังต่อไปนี้

$$x = y + 1;$$
$$\{x > 0\}$$

ซึ่งมีความหมายว่า เมื่อกำหนดค่า y + 1 ให้กับตัวแปร x แล้ว ค่าของตัวแปร x จะมีค่ามากกว่าศูนย์ ดังนั้น เงื่อนไขก่อน หรือสภาวะก่อนหน้าที่จะทำข้อความสั่ง ก็คือ ค่าของตัวแปร y จะต้องไม่เป็นค่าติดลบ ซึ่งเป็นไปได้หลายสภาวะ เช่น y=0, y=1, y=2, y=3, .. รวมถึง y ≥ 0 สภาวะที่เราจะเลือกใช้เป็นเงื่อนไขก่อนจะเป็นสภาวะที่มีข้อจำกัด หรือมีความเจาะจงน้อยที่สุด เรียกว่า *เงื่อนไขก่อนที่อ่อนที่สุด* (weakest precondition) นั่นคือเงื่อนไข y ≥ 0 เพราะมีความเจาะจงน้อยที่สุด และจากเงื่อนไขก่อนที่แปลง (derive) ขึ้นมาจากข้อความสั่งและเงื่อนไขหลังเราจะได้รูปแบบ Hoare triple ที่เป็นแอ็กเซียมดังนี้

```
{ y ≥ 0 }
                x = y + 1;
{ x > 0}
```

แอ็กเซียมนี้อธิบายว่า เมื่อข้อมูลเข้า *y* เป็นตัวเลขที่มีค่ามากกว่าหรือเท่ากับศูนย์ ข้อความสั่ง *x = y + 1;* จะทำให้ได้ผลลัพธ์เป็นข้อมูลตัวเลขที่มีค่ามากกว่าศูนย์ นอกจากนี้เงื่อนไขก่อนที่ได้จะทำหน้าที่เป็นเงื่อนไขหลังให้กับข้อความสั่งก่อนหน้า ไปตามลำดับจนกระทั่งได้เงื่อนไขก่อนที่เป็นเงื่อนไขเริ่มต้นของโปรแกรม ซึ่งถ้าตรงกับข้อกำหนด (specification) ของโปรแกรม แสดงว่าโปรแกรมทำงานได้ถูกต้อง การพิสูจน์ความถูกต้องแบบนี้จึงเป็นการพิสูจน์ย้อนหลังจากผลลัพธ์ (output) ย้อนกลับไปหาข้อมูลเข้า (input) หรือสถานะเริ่มต้น ตัวอย่างการพิสูจน์ความถูกต้องของโปรแกรมแสดงได้ดังต่อไปนี้

<u>*การพิสูจน์ความถูกต้องของโปรแกรม Max*</u>

```
{true}
        int Max ( int a, int b ) {
                int m ;
                if ( a >= b)
                        m = a ;
                else
                        m = b ;
                return m ;
                }
{ m=max(a,b)}
```

การพิสูจน์ความถูกต้องของโปรแกรมนี้เป็นการพิสูจน์ว่าโปรแกรมส่งกลับค่า m ที่เป็นค่าที่มากกว่าระหว่างคู่ตัวเลขจำนวนเต็ม a, b ใดๆ ซึ่งมีขั้นตอนการพิสูจน์เพียงขั้นตอนเดียว คือพิสูจน์ความถูกต้องของคำสั่ง

if (a >= b) m = a ; else m = b ;

คำสั่ง *return m;* และคำสั่ง *int m;* เป็นคำสั่งส่งค่ากลับและคำสั่งประกาศตัวแปร ซึ่งจะไม่มีผลเปลี่ยนแปลงพฤติกรรมการทำงานของโปรแกรม (สถานะของโปรแกรมไม่มีการเปลี่ยนแปลง)

จากกฎการพิสูจน์คำสั่งแบบมีเงื่อนไข

$$\frac{\{s.test \wedge P\}\ s.thenpart\ \{Q\} \qquad \{\neg\ s.test \wedge P\} s.elsepart\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

จะเห็นว่าส่วน premise ของกฎการพิสูจน์ จะแยกเป็นสองกรณีคือ กรณีที่เงื่อนไขของคำสั่ง if เป็นจริง (s.test = true) จะทำที่ s.thenpart และกรณีที่เงื่อนไขเป็นเท็จ {¬ s.test } จะทำที่ s.elsepart การพิสูจน์ด้วยแอ็กเซียมจึงต้องแยกพิจารณาเป็นสองกรณี

กรณีที่ 1: กรณีที่เงื่อนไขของคำสั่ง if เป็นจริง นั่นคือ (a >= b) เป็นจริง

```
{P}
                    m=a;
{Q} = {m=max(a,b)}
```

จากกฎการพิสูจน์ของคำสั่งกำหนดค่า (s.target = s.source)

$$\frac{true}{\{Q[s.target\backslash s.source]\}\ S\ \{Q\}}$$

ช่วยให้เราหาเงื่อนไขก่อนที่อ่อนที่สุด (weakest precondition) หรือ {P} ได้ดังนี้

```
{P} = {Q[s.target\s.source]}
    = {Q[m\a]}
    = {a=max(a,b)}
```

เมื่อนำไปแทนใน premise ส่วน s.thenpart ของกฎการพิสูจน์แบบมีเงื่อนไข จะได้แอ็กเซียมดังนี้

```
  {s.test ∧ P}              s.thenpart     {Q}
= {a ≥ b ∧ a = max(a,b)}    m = a;         {m=max(a,b)}
```

และจาก rule of consequence ที่ระบุว่า

$$\frac{P \supset P' \quad (P'\}\ S\ \{Q'\} \quad Q' \supset Q}{\{P\}\ S\ \{Q\}}$$

และจากความจริงที่ข้อความ {a ≥ b ∧ a = max(a,b)} สามารถถูกตีความ (imply) ได้จาก {a ≥ b} ทำให้เราสามารถเขียนแอ็กเซียมของคำสั่งส่วน s.thenpart ได้ใหม่ดังนี้

$$\frac{(a \geq b) \supset (a = max(a,b)) \quad \{(a \geq b) \wedge (a = max(a,b))\}\ m=a;\ \{m=max(a,b)\}}{\{a \geq b\}\ m = a;\ \{m=max(a,b)\}}$$

กรณีที่ 2: กรณีเงื่อนไขของคำสั่ง if เป็นเท็จ นั่นคือ a < b

```
{P} = {Q[m\b]} = {b=max(a,b)}
                                m = b;
{Q} = {m=max(a,b)}
```

เราสามารถเขียนแอ็กเซียมของคำสั่งส่วน s.elsepart ได้ดังนี้

$$\frac{(a < b) \supset (b = \max(a,b)) \quad \{a < b \wedge b = \max(a,b)\}\ m{=}b;\ \{m{=}\max(a,b)\}}{\{a < b\}\ m{=}b;\ \{m{=}\max(a,b)\}}$$

จากการพิสูจน์ premise ทั้งส่วน s.thenpart และส่วน s.elsepart ได้ข้อสรุปดังนี้

```
{a ≥ b} m = a; {m = max(a,b)}
{a < b} m = b; {m = max(a,b)}
```

จากทั้งสองกรณีได้เงื่อนไขหลังการทำคำสั่งที่ตรงกัน คือ `{m = max(a,b)}` ทำให้เราสามารถสรุปได้ว่าคำสั่ง if ทำงานแล้วได้ค่าที่เป็นค่า maximum ของการเปรียบเทียบค่าระหว่าง a และ b

```
{true}
            if (a >= b)     m = a;
              else          m = b;
{ m=max(a,b)}
```

<u>*การพิสูจน์ความถูกต้องของโปรแกรม Factorial*</u>

```
int Factorial (int n) {
        int f = 1;
        int i = 1;
        while (i < n) {
                i = i + 1;
                f = f * i;
        }
        return f;
    }
```

เงื่อนไขก่อนของโปรแกรม หรือ ข้อกำหนดเกี่ยวกับข้อมูลเข้า คือ `1 ≤ n` และเงื่อนไขหลังของโปรแกรม หรือ ข้อกำหนดเกี่ยวกับผลลัพธ์ของโปรแกรม คือ `f = n!`

โปรแกรม Factorial มีการใช้คำสั่งวนรอบ (while) ซึ่งมีกฎการพิสูจน์ดังนี้

$$\frac{\{s.test \wedge P\}\ s.body\ \{P\}}{\{P\}\ S\ \{\neg\, s.test \wedge P\}}$$

ส่วน premise ของกฎเป็นการระบุว่า เงื่อนไขก่อนที่จะเข้าไปทำคำสั่งในส่วน body ของลูปก็คือส่วน test ของคำสั่งลูปเป็นจริง ในส่วน conclusion ของกฎระบุว่าผลที่เกิดตามมาของการทำคำสั่งลูปก็คือ เมื่อทำคำสั่งภายในลูปเสร็จแล้วส่วน test จะกลายเป็นเท็จ (เพราะถ้า test เป็นจริงคำสั่งลูปจะยังไม่จบ ยังต้องวนกลับไปทำคำสั่งในส่วน body จนกว่า test จะเป็นเท็จ) จากกฎการพิสูจน์คำสั่ง while สามารถเขียนชุดคำสั่งให้สอดคล้องกับกฎได้ดังนี้

```
{Q}
                initialization
{P}
                while (test) {
                        loobBody
                    }
{¬test ∧ P}
                finalization
{R}
```

เมื่อแบ่งส่วนชุดคำสั่งแล้ว {Q} จะเป็นเงื่อนไขก่อน หรือสถานะเริ่มต้นของโปรแกรม และ {R} จะเป็นเงื่อนไขหลัง หรือสถานะสุดท้ายที่จะบอกผลลัพธ์ของโปรแกรม {P} จะเรียกว่า *ส่วนไม่แปรผันของลูป* (loop invariant) ส่วนไม่แปรผันของลูป คือ ข้อความยืนยันที่จะเป็นจริงในทุกรอบการทำงานของลูป ส่วนไม่แปรผันนี้จะกำหนดจากความหมายของลูป ในโปรแกรมแฟคตอเรียลนี้ส่วนไม่แปรผันคือ

```
{ (1 ≤ i) ∧ (i ≤ n) ∧ ( f = i! ) }
```

และเราสามารถเขียนชุดคำสั่งของโปรแกรมแฟคตอเรียลที่มีข้อความยืนยันปรากฏตามส่วนต่างๆ ได้ดังต่อไปนี้

```
{1≤n}
                f = 1;
                i = 1;
{1≤i ∧ i≤n ∧ (f=i!)}
                while (i < n) {
                        i = i + 1;
                        f = f *i;
                        }
{i≥n ∧ 1≤i ∧ i≤n ∧ (f=i!)}
                return f;
{f = n!}
```

การแยกพิจารณาโปรแกรมเป็นสามส่วนคือ ส่วน initialization, ส่วน loop, ส่วน finalization ช่วยให้เราสามารถพิสูจน์ความถูกต้องของโปรแกรมแยกเป็นส่วนๆได้ โดยเริ่มจากส่วนง่ายก่อนนั่นคือส่วน finalization

การพิสูจน์ส่วน finalization

$\{(i \geq n) \wedge (1 \leq i) \wedge (i \leq n) \wedge (f=i!)\}$

return f;

$\{f = n!\}$

คำสั่ง *return f;* ไม่มีผลต่อการเปลี่ยนแปลงค่าใดๆ ในโปรแกรม การพิสูจน์ความถูกต้องจึงเป็นเพียงการเชื่อมโยงด้วยการให้เหตุผลว่าเงื่อนไขก่อนสามารถตีความไปสู่เงื่อนไขหลัง จากข้อความในเงื่อนไขก่อนสังเกตได้ว่า $i \geq n \wedge i \leq n$ สามารถตีความได้ว่า $i=n$ ดังนั้น

$$(i \geq n) \wedge (1 \leq i) \wedge (i \leq n) \wedge (f = i!)$$
$$\supset (i = n) \wedge (f = i!)$$
$$\supset (f = n!)$$

การพิสูจน์ส่วน initialization

ส่วน initialization ประกอบด้วยสองคำสั่ง *f = 1; i = 1;* ซึ่งสามารถระบุเงื่อนไขก่อนและ หลังได้ดังนี้

$\{1 \leq n\}$

f = 1;

$\{R'\}$

i = 1;

$\{1 \leq i \wedge i \leq n \wedge (f = i!)\}$

ซึ่ง $\{R'\}$จะมีค่าเป็น $\{1 \leq 1 \wedge 1 \leq n \wedge (f=1!)\}$ และเมื่อนำ $\{R'\}$พิจารณาร่วมกับคำสั่ง *f=1;* เราจะได้เงื่อนไขก่อน ของคำสั่ง *f=1;* คือ $\{1 \leq 1 \wedge 1 \leq n \wedge (1=1!)\}$ซึ่งมีความหมายเดียวกับ $\{1 \leq n\}$ ทำให้สรุปการพิสูจน์ส่วน initialization ได้ดังนี้

$$\frac{\{1 \leq n\}\ f=1;\ \{1 \leq 1 \wedge 1 \leq n \wedge (f=1!)\} \quad \{1 \leq 1 \wedge 1 \leq n \wedge (f=1!)\}\ i=1;\ \{1 \leq i \wedge i \leq n \wedge (f=i!)\}}{\{1 \leq n\} \quad f=1;\ i=1;\ \{1 \leq i \wedge i \leq n \wedge (f=i!)\}}$$

การพิสูจน์ส่วน loopBody

ส่วนที่สามที่ต้องพิสูจน์คือส่วน loop ซึ่งเป็นการพิสูจน์ loop invariant ว่าการทำงานของคำสั่งภายในลูป จะยังทำให้ loop invariant เป็นจริงทุกรอบของการทำงาน นั่นคือเป็นการพิสูจน์ข้อความต่อไปนี้

$\{i<n \wedge 1 \leq i \wedge i \leq n \wedge (f=i!)\}$

i = i+1;
*f = f*i;*

$\{1 \leq i \wedge i \leq n \wedge (f=i!)\}$

ซึ่งแยกข้อความ $i = i+1; f = f*i;$ ออกจากกันได้เป็น

$\{i<n \wedge 1 \le i \wedge i \le n \wedge (f=i!)\}$

$i = i+1;$

$\{R'\}$

$f = f*i;$

$\{1 \le i \wedge i \le n \wedge (f=i!)\}$

ด้วยการแทนย้อนกลับค่า f ด้วย $f*i$ เราจะได้ {R'} เป็น $\{1 \le i \wedge i \le n \wedge (f*i=i!)\}$ และ {R'} จะเป็นเงื่อนไขหลังให้กับคำสั่ง $i = i+1;$ ช่วยให้เราสามารถหาเงื่อนไขก่อนของคำสั่ง $i = i+1;$ ด้วยการแทน i ด้วย $i+1$ ใน R' จะได้

$\{(1 \le i+1) \wedge (i+1 \le n) \wedge (f*(i+1)=(i+1)!)\}$

ซึ่งถ้าเราสามารถให้เหตุผลเชื่อมโยงได้ว่า

$\{(i<n) \wedge (1 \le i) \wedge (i \le n) \wedge (f=i!)\}$
$\supset \{(1 \le i+1) \wedge (i+1 \le n) \wedge (f*(i+1)=(i+1)!)\}$

เราจะสามารถจบการพิสูจน์โปรแกรมแฟคตอเรียลนี้ได้โดยสมบูรณ์

การจะพิสูจน์การเชื่อมโยงดังกล่าว เราจะใช้ความรู้พื้นฐานทางตรรกศาสตร์ต่อไปนี้

$$\frac{p \supset q \quad p \supset r}{p \supset q \wedge r}$$

นั่นคือเราจะแยกการให้เหตุผลเชื่อมโยงข้างต้นออกเป็นสามส่วน

ส่วนที่ 1: $\{i<n \wedge 1 \le i \wedge i \le n \wedge (f=i!)\} \supset \{1 \le i+1\}$

ส่วนที่ 2: $\{i<n \wedge 1 \le i \wedge i \le n \wedge (f=i!)\} \supset \{i+1 \le n\}$

ส่วนที่ 3: $\{i<n \wedge 1 \le i \wedge i \le n \wedge (f=i!)\} \supset \{f*(i+1)=(i+1)!\}$

ส่วนที่ 1: เป็นจริงเพราะ $1 \le i$ ย่อมหมายถึง $1 \le i+1$ ด้วย

ส่วนที่ 2: เป็นจริงเพราะ $i < n$ ย่อมหมายถึง $i+1 \le n$ ด้วย

ส่วนที่ 3: เนื่องจาก $1 \le i$ ดังนั้นถ้าเราหารทั้งสองข้างของ $f*(i+1)=(i+1)!$ ด้วย $i+1$ เราจะได้

$\{(i<n) \wedge (1 \le i) \wedge (i \le n) \wedge (f = i!)\}$

$\supset \{f = i!\}$

ดังนั้นการพิสูจน์ความถูกต้องของโปรแกรมแฟคตอเรียลเสร็จสมบูรณ์

3.6 สรุป

การอธิบายความหมายของโปรแกรม เป็นความพยายามที่จะหาวิธีที่เป็นทางการและเป็นสากลมาอธิบายความหมายหรือการทำงานที่จะเกิดขึ้นเมื่อใช้คำสั่งแต่ละคำสั่งของภาษาคอมพิวเตอร์ มาประกอบกันเป็นโปรแกรม รูปแบบอย่างเป็นทางการของการอธิบายความหมายรูปแบบแรกเสนอโดยนักคอมพิวเตอร์ชื่อ Knuth เสนอให้ขยายความไวยากรณ์บีเอ็นเอฟของแต่ละคำสั่งด้วยฟังก์ชันที่อธิบายการทำงานของคำสั่งนั้น รูปแบบนี้เรียกว่า ไวยากรณ์เชิงลักษณะประจำ หรือ attribute grammar

ในการอธิบายความหมายของคำสั่งในระยะหลังนิยมใช้รูปแบบเชิงดำเนินการ, รูปแบบเชิงแทนความและรูปแบบเชิงพิสูจน์ ที่อธิบายคำสั่งได้ครอบคลุมมากกว่าไวยากรณ์เชิงลักษณะประจำซึ่งอธิบายได้เฉพาะคำสั่งกำหนดค่า

การอธิบายความหมายเชิงดำเนินการ (operational semantics) เป็นการอธิบายความหมายของคำสั่งในภาษาระดับสูง ด้วยการอธิบายพฤติกรรมการทำงานเมื่อกระทำคำสั่งบนเครื่องคอมพิวเตอร์จำลองหรือเครื่องคอมพิวเตอร์เสมือน การอธิบายด้วยพฤติกรรมจะเป็นการบรรยายสถานะของเครื่องคอมพิวเตอร์แต่ละขณะของการทำงาน โดยสถานะหรือ state จะระบุจากค่าของตัวแปรต่างๆ

การอธิบายความหมายเชิงแทนความ (denotational semantics) เป็นการใช้ฟังก์ชันสื่อแทน (denote) ความหมายของแต่ละคำสั่ง ฟังก์ชันที่ใช้เรียกว่าฟังก์ชันความหมาย หรือ meaning function ทำหน้าที่เชื่อมโยงจากสถานะเริ่มต้นไปสู่สถานะใหม่หลังจากทำแต่ละคำสั่งแล้ว คำสั่งทุกประเภทของภาษาจะมีฟังก์ชันความหมายอธิบายการทำงานของคำสั่ง

การอธิบายความหมายเชิงพิสูจน์ (axiomatic semantics) เป็นการอธิบายความหมายทางอ้อม โดยพิจารณาว่าโปรแกรมคือ ทฤษฎี (theory) การพยายามเข้าใจความหมายที่เกิดจากการทำงานของโปรแกรมคือ การพิสูจน์ทฤษฎี กระบวนการพิสูจน์จะใช้แอ็กเซียม (axiom) ร่วมกับกฎการพิสูจน์ (proof rules) เพื่อให้เหตุผลเกี่ยวกับการเปลี่ยนสถานะของโปรแกรม วิธีการอธิบายความหมายแบบนี้มีหลักคิดพื้นฐานมาจากตรรกศาสตร์ และจากพื้นฐานแนวคิดที่เอื้อต่อการพิสูจน์ จึงนิยมใช้ axiomatic semantics ในการพิสูจน์ความถูกต้องของโปรแกรม

แบบฝึกหัดท้ายบทที่ 3

คำถามอัตนัย

1. กำหนดโปรแกรมให้ดังนี้

```
void  main( ) {
        int  x, y, z;
        x = 10;
        y = x * 2;
        z = x + y;
}
```

ให้อธิบายความหมายเชิงดำเนินการ (operational semantics) ของแต่ละข้อความสั่ง พร้อมทั้งบอกสถานะเริ่มต้น, สถานะระหว่างกลาง, และสถานะสุดท้ายของโปรแกรม

สถานะ	ความหมายเชิงดำเนินการ

2. จากโปรแกรมในข้อ 1 ให้อธิบายความหมายเชิงแทนความ (denotational semantics) ของโปรแกรมและของแต่ละข้อความสั่ง
3. ให้พิสูจน์ความถูกต้องของโปรแกรมในข้อ 1 ว่าเมื่อทำงานเสร็จสิ้นจะต้องได้ค่า z ที่มีค่ามากกว่า x และมีค่ามากกว่า y โดยให้เขียนเงื่อนไขก่อน (precondition) และเงื่อนไขหลัง (post-condition) ของแต่ละคำสั่ง

เงื่อนไขก่อน (precondition)	คำสั่ง (statement)	เงื่อนไขหลัง (post-condition)
	void main () {	
	int x, y, z;	
	x = 10;	
	y = x * 2;	
	z = x + y;	
	}	