บทที่ 12

รูปแบบของภาษา

12.1 ภาษาเชิงคำสั่ง

12.2 ภาษาเชิงวัตถุ

12.3 ภาษาเชิงหน้าที่

12.4 ภาษาเชิงตรรกะ

12.5 เอกสารอ้างอิงและเว็บไซต์ที่ควรรู้

วัตถุประสงค์

- แนะนำภาษาในรูปแบบต่างๆ
- ลักษณะที่สำคัญของภาษาแต่ละรูปแบบ
- ตัวอย่างโปรแกรม

แนวคิดพื้นฐานของภาษาโปรแกรมส่วนใหญ่จะมีลักษณะคล้ายคลึงกัน ดังที่ได้กล่าวมาแล้วในบทก่อนๆ นี้ เมื่อนำแนวคิดเหล่านี้มารวมกันก็จะทำให้เกิดเป็นภาษาโปรแกรมที่สมบูรณ์ได้ ภาษาโปรแกรมมีวิวัฒนาการมายาวนาน มีจำนวนมากมายหลายภาษา แต่ละภาษาอาจมีความแตกต่างกันตามวัตถุประสงค์ของการสร้างขึ้นเพื่อการใช้งาน เช่น ภาษา Fortran ที่ออกแบบมาให้เหมาะกับงานทางด้านการคำนวณและวิทยาศาสตร์ ภาษา Cobol ที่ออกแบบมาให้เหมาะกับงานด้านธุรกิจ ภาษา php ที่ออกแบบมาให้เหมาะกับการเขียนโปรแกรมบนเว็บ ภาษาเหล่านี้จะมีโครงสร้างที่แตกต่างกัน โดยทั่วไปเรามักจะแบ่งภาษาออกตามลักษณะสำคัญของภาษา ซึ่งสามารถแบ่งได้เป็น 4 รูปแบบหลัก คือ ภาษาเชิงคำสั่ง ภาษาเชิงวัตถุ ภาษาเชิงฟังก์ชัน ภาษาเชิงตรรกะ ภาษาทั้ง 4 รูปแบบนี้ได้กล่าวไว้โดยสังเขปในบทที่ 8 ในบทนี้เราจะเน้นลักษณะสำคัญของภาษาแต่ละรูปแบบโดยละเอียด พร้อมทั้งยกตัวอย่างโปรแกรมประกอบการอธิบาย

12.1 ภาษาเชิงคำสั่ง

เป็นภาษาที่เก่าแก่ที่สุดและมีการพัฒนาเป็นอย่างดีมาจนถึงปัจจุบัน ถือกำเนิดพร้อมกับคอมพิวเตอร์ในยุคแรกที่มีสถาปัตยกรรมแบบ Von Neumann ซึ่งมีลักษณะที่สำคัญที่เป็นรากฐานของภาษาเชิงคำสั่ง คือ โปรแกรมและข้อมูลจะต้องอยู่ภายในหน่วยความจำหลัก ดังนั้น หัวใจสำคัญของภาษาในกลุ่มนี้ คือ แนวคิดในการกำหนดค่า กล่าวคือ เป็นการเปลี่ยนแปลงค่าในหน่วยความจำหลักนั่นเอง นอกจากนี้ภาษาเชิงคำสั่งยังสนับสนุนการประกาศตัวแปร นิพจน์ คำสั่งแบบมีเงื่อนไข การทำซ้ำอีกด้วย การประกาศตัวแปรเป็นการเชื่อมโยงชื่อเข้ากับตำแหน่งและชนิดข้อมูลของค่าที่เก็บไว้ การตีความของนิพจน์ก็คือการคำนวณโดยดึงค่าของตัวแปรจากหน่วยความจำมาใช้

โดยมากคำสั่งจะทำงานตามลำดับที่ปรากฏในหน่วยความจำ ซึ่งอาจมีการเปลี่ยนแปลงทิศทางการทำงานโดยปกติได้โดยใช้คำสั่งแบบมีเงื่อนไข หรือคำสั่งแบบทำซ้ำ ลักษณะที่สำคัญโดยทั่วไปที่สนับสนุนภาษาหนึ่งๆ ให้เป็นภาษาเชิงคำสั่ง มีดังต่อไปนี้

- โครงสร้างควบคุม
- การรับและแสดงผล
- การจัดการความผิดปกติของโปรแกรม
- Procedural abstraction
- นิพจน์และการกำหนดค่า
- การสนับสนุน Library สำหรับโครงสร้างข้อมูล

Procedure abstraction

ภาษารูปแบบนี้จะมีลักษณะของ อัลกอริทึมร่วมกับโครงสร้างข้อมูล (algorithm plus data structures) อัลกอริทึมจะนำมาพัฒนาเป็นโปรแกรมโดยใช้แนวคิดที่สำคัญ 2 ประการ คือ

- procedure abstraction เป็นกระบวนการที่ทำให้โปรแกรมเมอร์สนใจเฉพาะการติดต่อระหว่างฟังก์ชันกับผลที่ได้รับ โดยไม่จำเป็นต้องสนใจรายละเอียดของการทำงาน เช่น การพัฒนา library ของฟังก์ชันมาตรฐานทางคณิตศาสตร์มาพร้อมกับภาษา ตัวอย่างเช่น ฟังก์ชัน sin, cos, sqrt ทำให้โปรแกรมเมอร์สะดวกในการเรียกใช้

- stepwise refinement เป็นกระบวนการที่ทำให้สามารถนำ procedure abstraction มาใช้ในการพัฒนาอัลกอริทึมจากรูปแบบทั่วไปมาเป็นการนำมาใช้แบบเฉพาะเจาะจง ตัวอย่างเช่น การพัฒนาฟังก์ชัน sorting ที่โปรแกรมต้องการอัลกอริทึมในการเรียกข้อมูลตัวเลขในอาร์เรย์ โดยไม่จำเป็นต้องรู้รายละเอียดว่าทำอย่างไร เช่น `sort(list, len)` โปรแกรมเมอร์รู้เพียงแค่ต้องส่งพารามิเตอร์ที่เป็นอาร์เรย์ของตัวเลขให้กับ list และระบุความยาวของอาร์เรย์ด้วย len ซึ่งแม้ว่าจะมีการเปลี่ยนแปลง หรือปรับปรุงอัลกอริทึมภายในฟังก์ชัน sorting ในภายหลัง โปรแกรมเมอร์ก็ยังคงสามารถเรียกใช้งานเช่นเดิมโดยไม่มีผลกระทบแต่อย่างใด

นิพจน์และการกำหนดค่า

เป็นอีกพื้นฐานที่สำคัญของภาษาเชิงคำสั่ง ซึ่งสามารถเขียนให้อยู่ในรูปทั่วไป คือ

```
target = expression
```

ส่วนเครื่องหมายที่ใช้ในการกำหนดค่าอาจแตกต่างกันไปในแต่ละภาษา เครื่องหมายที่เป็นที่นิยมมีอยู่สองรูปแบบคือ = ซึ่งมีภาษา Fortran เป็นต้นแบบ และ := ซึ่งมีภาษา Algol เป็นต้นแบบ

ความหมายของการกำหนดค่า ก็คือ นำค่าที่ได้จากการประเมินค่าของนิพจน์ไปใส่ไว้ใน target การเขียนนิพจน์จะใช้ตัวดำเนินการทางคณิตศาสตร์และตรรกะ รวมถึงอาจมีการเรียกใช้ฟังก์ชันมาตรฐานของภาษานั้นๆ ด้วย

การสนับสนุน Library สำหรับโครงสร้างข้อมูล

โครงสร้างข้อมูลพื้นฐานที่สำคัญของภาษาเชิงคำสั่ง คือ อาร์เรย์และเรคอร์ด ซึ่งได้กล่าวถึงแล้วในเรื่องชนิดข้อมูล นอกจากนี้โครงสร้างทั้งสองชนิดแล้ว ภาษาโปรแกรมสมัยใหม่หลายๆ ภาษาได้ขยาย library ของฟังก์ชันมาตรฐานที่ช่วยในการพัฒนาโปรแกรมที่มีความซับซ้อน รวมทั้งพัฒนาโครงสร้างข้อมูลที่ซับซ้อนเพิ่มขึ้นด้วย เช่น ลิสต์ เวกเตอร์ สแตก คิว เซต เป็นต้น

12.2 ภาษาเชิงวัตถุ

เป็นภาษาที่มีวิวัฒนาการมาจากภาษาเชิงคำสั่ง บางตำราอาจจัดภาษาเชิงวัตถุว่าเป็นกลุ่มย่อยของภาษาเชิงคำสั่ง โดยเริ่มต้นจากการพัฒนา Procedure abstraction มาเป็นแนวคิดแบบ Abstract Data Types (ADTs) ซึ่งเป็นการสร้างโดยใช้โครงสร้างเดิมที่มีในภาษาเชิงคำสั่ง นั่นคือ การนำชนิดข้อมูลที่สร้างขึ้นโดยผู้เขียนโปรแกรม (user defined data type) มาใช้งานให้ปลอดภัย โดยมีลักษณะดังนี้

- รายละเอียดของการเก็บข้อมูลต้องไม่มีความสำคัญต่อการใช้งานชนิดข้อมูล
- ข้อมูลภายในจะไม่ถูกอ้างถึงได้โดยตรง แต่ผ่านทาง operations ที่กำหนดไว้

ถ้าหากมองให้เข้าใจได้ง่ายเมื่อเทียบกับภาษาเชิงคำสั่ง ชนิดข้อมูลที่มีมาพร้อมกับภาษาโปรแกรม หรือ Build-in type ก็คือ ADTs นั่นเอง ตัวอย่างเช่น ชนิดข้อมูล ***int*** ในภาษา C ที่โปรแกรมเมอร์จะมองไม่เห็นลักษณะการแทนข้อมูล แต่สามารถเรียกใช้ตัวดำเนินการ (operation) ทุกตัวที่กำหนดมาพร้อมกับชนิดข้อมูลนี้ได้ การใช้

งานทำได้โดยประกาศตัวแปรเป็นชนิดข้อมูล ***int*** หากเราสร้าง user-defined ADTs ก็จะต้องทำในลักษณะแบบเดียวกับ build-in ADTs นั่นเอง

ภาษาที่มีลักษณะแบบนี้เรียกว่าเป็นภาษาแบบ object-based ตัวอย่างเช่น ภาษา Modula-2, Ada ซึ่งภาษาที่สร้าง ADTs ได้ต้องมีกลไก 2 อย่าง คือ

- *Encapsulation* เป็นการนำข้อมูลกับ operations มาผูกติดกันเป็นหน่วยหนึ่ง
- *Information hiding* เป็นการกำหนดขอบเขตในการอ้างถึงข้อมูลหรือ operations ของชนิดข้อมูลหนึ่งๆ

ข้อดีของภาษาแบบ object-based มีหลายประการ ประการแรกทำให้ง่ายต่อการจัดการโครงสร้างของโปรแกรม สามารถแก้ไขเปลี่ยนแปลงได้สะดวก คอมไพล์แยกส่วนได้ สามารถเปลี่ยนการแทนข้อมูลได้ โดยไม่มีผลกระทบต่อผู้ใช้ นอกจากนี้ยังมี Reliability สูงเนื่องจากมีการซ่อนการแทนข้อมูล ทำให้ผู้ใช้ไม่สามารถเข้าถึงข้อมูลในวัตถุได้โดยตรง

ต่อมาจึงได้แนวคิดของ ADTs ได้พัฒนาจนกลายเป็นภาษาเชิงวัตถุ (object-oriented programming หรือ OOP) เต็มรูปแบบ

กลุ่มของภาษาเชิงวัตถุ

ภาษาเชิงวัตถุสามารถแบ่งกลุ่มออกเป็น 3 ประเภท ดังต่อไปนี้

- สนับสนุน OOP โดยเป็นส่วนเพิ่มเติมในภาษาที่อยู่แล้ว
    - C++ (สนับสนุน procedural และ data-oriented programming ด้วย)
    - Ada95 (สนับสนุน procedural และ data-oriented programming ด้วย)
    - CLOS (สนับสนุน funcitional programming ด้วย)
    - Scheme (สนับสนุน funcitional programming ด้วย)
- สนับสนุน OOP แต่ยังคงใช้โครงสร้างพื้นฐานแบบภาษา imperative
    - Eiffel (เป็นภาษาใหม่ไม่มีรากฐานมาจากภาษาใด)
    - Java (รากฐานจากภาษา C++)
- ภาษา OOP อย่างเดียว (Pure OOP) เช่น ภาษา Smalltalk

แนวคิดพื้นฐานของภาษาเชิงวัตถุ

แนวคิดที่สำคัญของภาษาเชิงวัตถุ คือ การมองปัญหาอยู่ในลักษณะของกลุ่มวัตถุ (Object) ซึ่งโดยทั่วไปจะเก็บสถานะ (state) ที่เป็นข้อมูลของตัวเองได้ และสามารถทำกิจกรรมบางอย่างได้ เช่น คำนวณ เปลี่ยนแปลงข้อมูล ติดต่อหรือโต้ตอบกับวัตถุอื่นได้ ตัวอย่างเช่น การเขียนโปรแกรมที่เกี่ยวกับระบบทะเบียนนักเรียน ก็จะมองปัญหาว่าประกอบด้วยนักเรียน ซึ่งเป็นวัตถุ ที่มีข้อมูล คือ ชื่อ ที่อยู่ ผลการเรียน และสามารถทำกิจกรรมที่เกี่ยวข้องกับตัวเองได้ เช่น ตอบคำถามเกี่ยวกับตัวเอง การลงทะเบียนเรียน การติดต่อเพื่อดูผลสอบ จะเห็นได้ว่าแนวคิดเชิงวัตถุนี้แตกต่างจากตัวแปรในภาษาเชิงคำสั่ง ซึ่งไม่สามารถจำลองพฤติกรรมและการทำงานของวัตถุเช่นนี้ได้

กลไกสำหรับกำหนดโครงสร้างและพฤติกรรมของวัตถุ ได้แก่ *Class* โดยมี *Instance* ที่ถูกสร้างขึ้นเพื่อจำลองการทำงานของวัตถุใน class นั้น instance จะมีพื้นที่หน่วยความจำสำหรับเก็บข้อมูลของวัตถุ มี method สำหรับกระทำต่อข้อมูล หรือโต้ตอบกับ instance อื่น สามารถสร้าง instance ของ class ได้หลาย instance โดยแต่ละตัวจะมีโครงสร้างเหมือนกัน แต่มีข้อมูลที่ต่างกันไป โดยเมื่อสร้าง instance แล้ว มันจะรออยู่เฉย ๆ จนกว่าจะมีใครเรียกใช้ หากมองเปรียบเทียบกับภาษาเชิงคำสั่งแล้ว Class อาจเทียบได้กับชนิดข้อมูล ส่วน Instance ก็จะคล้ายกับตัวแปร

ตัวอย่างเช่น รูปทรง (shape) สามารถจำแนกออกเป็นรูปทรงย่อยๆ ได้หลากหลาย เช่น กล่อง วงรี เส้น ตัวอักษร ซึ่งเราสามารถมองเป็น class hierarchy ได้ว่ารูปทรงคือ class หลักโดยมีรูปทรงย่อยเป็น subclass โดย Class จะเป็นตัวระบุคุณสมบัติต่าง ๆ ของ object เมื่อมีการกำหนดพื้นที่สำหรับ object หนึ่งจะมีการสร้าง instance ขึ้น Object แต่ละตัวจะติดต่อกันโดยการรับส่ง *message* เมื่อ object ได้รับ message จะ execute method เพื่อตอบสนอง

Class ประกอบด้วยสมาชิก 2 ประเภท คือ

- data member คือสมาชิกที่เป็นข้อมูล อาจเป็นค่าคงที่ ตัวแปรของชนิดข้อมูลพื้นฐานหรือตัวแปรที่เป็น object
- method member คือสมาชิกที่เป็นฟังก์ชัน ซึ่งตอบสนองต่อ message เรียกสั้น ๆ ว่า *method*

เราสามารถสร้าง class ใหม่ได้ โดยนำ class ที่ถูกสร้างและใช้งานอยู่แล้วมาปรับปรุงเปลี่ยนแปลงให้เหมาะสมกับหน้าที่ใหม่ที่ต้องการ โดย class เดิมจะยังคงใช้งานได้เช่นเดิม เรียกว่าเป็น *superclass* ของ class ใหม่ class ใหม่จะได้สมาชิกของ class เดิมมาพร้อมกับมีการเพิ่มสมาชิกใหม่หรือเปลี่ยนแปลงสมาชิกที่มีอยู่เดิมให้ต่างไป จะเรียกว่าเป็น *subclass* ของ class เดิม กลไกการสร้าง class แบบนี้เรียกว่า การสืบทอดคุณสมบัติ หรือ *inheritance* ดังตัวอย่างโปรแกรมในภาษา C++ ต่อไปนี้

```
class A  {
public:
      int x;
      char f( );
       A( );
};
class D : public A {
      int x;
      int g( );
};
```

จากตัวอย่างข้างต้น เป็นการแสดงคุณสมบัติการสืบทอดและ โดย class D จะเป็น subclass ของ class A สำหรับ Object ของ class D จะมีสมาชิก 4 ตัวคือ

- x เป็น data member ที่สืบทอด (inherit) มาจาก class A (A::x)
- f เป็น method ที่สืบทอดมาจาก class A
- x เป็น data member ที่ประกาศเพิ่มใน class D
- g เป็น method ที่ประกาศเพิ่มใน class D

Method *overriding* เป็นการกำหนดชื่อของ method ใน subclass ให้มีชื่อเหมือนกับ method ใน parent class เพื่อเปลี่ยนแปลงการทำงาน method ใน parent class ให้ทำหน้าที่อื่นใน subclass การ binding ว่าชื่อ method ที่ instance ส่งมาเป็น method ตัวใด จะทำแบบ dynamic binding แนวคิดในการจัดการกับ instance ของ class ที่ต่างกันด้วย method เดียวกันเรียกว่า *polymorphism* แนวคิดนี้เป็นการสนับสนุนการออกแบบโปรแกรมเพื่อให้ใช้งานร่วมกัน หรือการนำกลับมาใช้ได้อีก ดังตัวอย่างโปรแกรมในภาษา C++ ต่อไปนี้

```
class Shape {
public : Shape *draw (Shape *)
  {   return this;  }
};
class Ellipse : public Shape {
public : Shape * draw (Shape *)
  {   return this;  }
};
class Circle : public Shape {
public : Shape * draw (Shape *)
  {   return this;  }
};

Shape * s;
s = new Ellipse;
Shape *p =  ...;
s->draw(p);      // Ellipse::draw(p)
s = new Circle;
s->draw(p);      // Circle::draw(p)
```

จากตัวอย่างจะเห็นว่า object ชื่อ s เป็น instance ของ class Shape โดย s สามารถเรียกใช้ method ชื่อ draw ได้ แต่การทำงานของ method draw แต่ละครั้งอาจแตกต่างกัน ขึ้นอยู่กับว่า s ทำการ binding กับ class อะไร

Method overloading เป็นการเรียก method หนึ่งด้วย argument ที่แตกต่างกันไปในการเรียกแต่ละครั้ง และจะต้องมีหลาย ๆ method สำหรับการเรียกแต่ละแบบ เช่น method ในการบวกอาจประกอบด้วย 3 method ซึ่งมี argument ต่างกัน คือ

function add(x:integer; y:integer)

function add(x:real; y:real);

function add(x:integer; y:real);

ในการสืบทอดคุณสมบัติ เราสามารถทำ Multiple Inheritance ได้ คือเป็นการสร้าง class ใหม่จาก class แม่มากกว่าหนึ่ง class

ข้อดีของภาษาเชิงวัตถุ อาจกล่าวได้ว่าเป็นภาษาเข้าใจง่าย เพราะการทำงานเปรียบเสมือนการจำลองเหมือนในโลกจริง โดยอาศัยการมองทุกอย่างเป็น object ซึ่งแต่ละตัวมีหน้าที่และความหมายในตัว การบำรุงรักษา และแก้ไขโปรแกรมทำได้ง่าย เป็นภาษาที่มีความปลอดภัยสูง เพราะจัดการกับข้อผิดพลาดได้ดี และ

สนับสนุนการนำกลับมาใช้ใหม่ได้ (reusability) ทำให้ลดขั้นตอนในการเขียนโปรแกรม โดยมากโปรแกรมที่เขียนด้วยภาษาเชิงวัตถุมักจะมีคุณภาพสูง ใช้ได้หลายแพลตฟอร์ม

ส่วนข้อเสียก็มีเช่นกัน กล่าวคือ เป็นภาษาที่เข้าใจยาก สำหรับผู้เริ่มต้นเขียนโปรแกรม หรือถนัดเขียนโปรแกรมด้วยภาษาเชิงคำสั่ง ภาษาเชิงวัตถุ เช่น Java มักทำงานได้ช้ากว่าภาษาโปรแกรมอื่น นอกจากนี้ภาษายังมีความกำกวมเกิดขึ้นได้ ถ้ามีลักษณะ multiple inheritance

12.3 ภาษาเชิงฟังก์ชัน

เป็นภาษาที่ใช้หลักการของฟังก์ชันทางคณิตศาสตร์ คือเป็นการ mapping สมาชิกของเซตหนึ่ง เรียกว่า domain set หรือ input ไปยังสมาชิกของอีกเซ็ตหนึ่งเรียกว่า range set หรือ output โดยใช้รูปแบบของ lambda expression ในการระบุพารามิเตอร์ที่ใช้ และฟังก์ชันการ mapping ดังตัวอย่างของฟังก์ชัน cube (x) = x * x * x ซึ่งเขียนอยู่ในรูปแบบ lambda expression ได้ดังนี้

```
λ(x) x * x * x
```

Lambda expressions เป็นการอธิบายฟังก์ชันแบบไม่มีชื่อ การใช้งานทำได้โดยการแทนที่ค่าพารามิเตอร์ในนิพจน์ ตัวอย่างเช่น `(λ(x) x * x * x)(3)` มีค่าเท่ากับ 27

ภาษาเชิงฟังก์ชันจะเป็นใช้การเรียกฟังก์ชันมาทำงาน แทนการระบุในรูปคำสั่ง เรียกว่าเป็นลักษณะการทำงานตามหน้าที่ (Functional) มีองค์ประกอบของภาษาดังนี้

- ฟังก์ชันเบื้องต้น (Primitive Function
- รูปแบบของฟังก์ชัน (Functional Forms)
- วิธีดำเนินการ (Application Operation)
- ออบเจกต์ (Object)

ตัวอย่างของภาษาเชิงฟังก์ชันคือ ภาษา FP ซึ่งมี John Backus เป็นผู้คิดค้นขึ้นในปี ค.ศ.1970 เป็นภาษาที่มีการทำงานไม่ขึ้นกับสถาปัตยกรรมของเครื่อง ประกอบด้วยออบเจกต์ n ตัว คือ x1, x2 ,…, xn เขียนแทนด้วย < x1, x2 ,…, xn > สร้างขึ้นโดยใช้หลักการของฟังก์ชันทางคณิตศาสตร์ ตัวอย่างของฟังก์ชันเบื้องต้น เช่น

```
ROTR : < x1 , x2 ,…, xn > ≡ < xn , x1 ,…, xn-1 >
LENGTH : < x1 , x2 ,…, xn > ≡  n
+ : < x , y >  ≡   x  +  y
- : < x , y >  ≡   x  -  y
```

ตัวอย่างรูปแบบฟังก์ชัน เช่น

$h \equiv f \circ g$ หมายถึง $h(x) \equiv f(g(x))$

ตัวอย่างการดำเนินการ เช่น กำหนดให้ $f(x) \equiv x * x * x$ และ $g(x) \equiv x + 3$,

$h \equiv f \circ g$ จะได้ผลลัพธ์คือ $(x + 3)* (x + 3)* (x + 3)$

ข้อดีของภาษาเชิงฟังก์ชันเมื่อเทียบกับภาษาเชิงคำสั่ง คือ มี syntax และ semantic ที่เข้าใจได้ง่าย ซึ่งแตกต่างจากภาษาเชิงคำสั่งที่มักจะมีรูปแบบที่ซับซ้อน เข้าใจได้ยาก นอกจากนี้โปรแกรมที่เขียนด้วยภาษาเชิงฟังก์ชันสามารถทำงานแบบพร้อมกัน (concurrent) ได้อัตโนมัติ ในขณะที่ภาษาเชิงคำสั่งโปรแกรมเมอร์จะต้องเป็นผู้เขียนให้ทำงานแบบพร้อมกันได้เอง

ตัวอย่างการใช้งาน ได้แก่ ภาษา LISP ที่มักจะใช้ในงานทางด้านปัญญาประดิษฐ์ เช่น การแทนความรู้ (Knowledge representation) การเรียนรู้ด้วยเครื่อง (Machine learning) การประมวลผลภาษาธรรมชาติ (Natural language processing) เป็นต้น หรือภาษา Scheme ที่มักใช้ในการสอนวิชาการเขียนโปรแกรมเบื้องต้น เพราะเป็นภาษาที่รูปแบบการเขียนง่าย ไม่ซับซ้อน

12.4 ภาษาเชิงตรรกะ

เป็นภาษาที่ใช้หลักการทางตรรกศาสตร์ในการแก้ปัญหา การเขียนโปรแกรมไม่จำเป็นต้องระบุขั้นตอนในการแก้ปัญหาโดยละเอียดแบบภาษาเชิงคำสั่ง แต่ใช้การประกาศข้อเท็จจริง ร่วมกับกฎ หรือข้อบังคับของปัญหาแทน จึงนิยมใช้ในการแก้ปัญหาทางด้านสัญลักษณ์ โดยใช้พื้นฐานของตรรกะเพรดิเคต ซึ่งสามารถจัดการเกี่ยวกับความสัมพันธ์ของสัญลักษณ์ต่างๆ ได้ดี จัดเป็นภาษาเชิงประกาศ (declarative) หรือเชิงพรรณนา (descriptive)

องค์ประกอบของภาษา ประกอบด้วย

- ข้อเท็จจริง (fact) ที่ใช้อธิบายข้อเท็จจริงในรูปความสัมพันธ์ของวัตถุ ซึ่งเขียนอยู่รูปแบบดังต่อไปนี้

```
relation-name (list of object names)
```

ตัวอย่างเช่น

```
student(john).
likes(kate, dos).
likes(nick, windows).
likes(july, linux).
```

- ข้อคำถาม (query) ใช้ในการตั้งคำถามเพื่อให้โปรแกรมหาคำตอบได้ มีรูปแบบเหมือนกับข้อเท็จจริง โปรแกรมจะให้คำตอบว่าจริง หรือ เท็จ โดยค้นจากข้อเท็จจริงที่ระบุในโปรแกรม

ตัวอย่างข้อคำถาม ที่โปรแกรมก็จะค้นหาคำตอบว่า no

```
likes(kate, linux)
```

- ตัวแปร (variable) สามารถระบุตัวแปรในข้อคำถามได้ เพื่อให้โปรแกรมหาคำตอบได้เช่นกัน

ตัวอย่างการใช้ตัวแปร x ในข้อคำถาม ที่โปรแกรมก็จะค้นหาคำตอบว่า x คือ july

```
likes(X, linux)
```

- ตัวเชื่อม (conjunction) ใช้ในการเชื่อมข้อคำถาม หรือข้อเท็จจริง หรือ ส่วนของกฎเข้าด้วยกัน
- กฎ (rules) ที่ใช้อธิบาย Object หรือ สิ่งที่เราสนใจ กับ Relation ที่ใช้อธิบายความสัมพันธ์ของ object โดยจะใช้กฎในการเขียนความสัมพันธ์ที่เกี่ยวข้องกับโดเมน

ตัวอย่างกฎ ที่อธิบายความสัมพันธ์ของ grandparent

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y)
```

ตัวอย่างโปรแกรมในภาษา Prolog ในการหาค่า factorial

```
factorial(0, 1).
factorial(N, Result) :-
      N > 0,
      M is N - 1,
      factorial(M, SubRes),
      Result is N * SubRes.
```

เนื่องจากภาษาเชิงตรรกะมีรากฐานมาจากตรรกศาสตร์ การเขียนโปรแกรมและการจัดการได้ถูกต้องตามหลักตรรกวิทยาจึงทำได้ง่าย โปรแกรมมีขนาดกะทัดรัด ทำให้ใช้เวลาในการพัฒนาไม่มากนัก นอกจากนี้การประมวลผลยังสามารถทำแบบขนาน (Parallel) ได้อีกด้วย เช่นภภาษา Prolog ที่สามารถใช้ประโยชน์ของเครื่องแบบ multiprocessor ได้อย่างเต็มที่

ตัวอย่างการใช้งาน ได้แก่ ภาษา Prolog ที่มักจะใช้ในงานทางด้านปัญญาประดิษฐ์ เช่น การประมวลผลภาษาธรรมชาติ (Natural language processing) หรือระบบผู้เชี่ยวชาญ (Expert system) เป็นต้น นอกจากนี้ยังนิยมนำไปใช้งานระบบจัดการฐานข้อมูลเชิงสัมพันธ์อีกด้วย

12.5 เอกสารอ้างอิงและเว็บไซต์ที่ควรรู้

Allen B. Tucker and Robert E. Noonan. Programming Languages – Principles and Paradigms.

Robert W. Sebesta. Concepts of Programming Languages.

สุจิตรา อดุลย์เกษม. เอกสารประกอบการสอนองค์ประกอบคอมพิวเตอร์และภาษา.

บุญเสริจ กิจศิริกุล. เอกสารคำสอนวิชา 2110654 ปัญญาประดิษฐ์.